algorithms

MDPI

Article

LR Parsing for LCFRS

Laura Kallmeyer * and Wolfgang Maier

Department for Computational Linguistics, Institute for Language and Information, Heinrich-Heine Universität Düsseldorf, Universitätsstr. 1, 40225 Düsseldorf, Germany; wolfgang.maier@gmail.com
* Correspondence: kallmeyer@phil.hhu.de; Tel.: +49-211-8113-899

Academic Editor: Henning Fernau
Received: 21 March 2016; Accepted: 18 August 2016; Published: 27 August 2016

Abstract: LR parsing is a popular parsing strategy for variants of Context-Free Grammar (CFG). It has also been used for mildly context-sensitive formalisms, such as Tree-Adjoining Grammar. In this paper, we present the first LR-style parsing algorithm for Linear Context-Free Rewriting Systems (LCFRS), a mildly context-sensitive extension of CFG which has received considerable attention in the last years in the context of natural language processing.

Keywords: parsing; automata; LCFRS

1. Introduction

In computational linguistics, the modeling of discontinuous structures in natural language, i.e., of structures that span two or more non-adjacent portions of the input string, is an important issue. In recent years, Linear Context-Free Rewriting System (LCFRS) [1] has emerged as a formalism which is useful for this task. LCFRS is a mildly context-sensitive [2] extension of CFG in which a single non-terminal can cover $k \in \mathbb{N}$ continuous blocks of terminals. CFG is a special case of LCFRS where $k = 1$. In CFG, only embedded structures can be modelled. In LCFRS, in contrast, yields can be intertwined. The schematically depicted derivation trees in Figure 1 illustrate the different domains of locality of CFG and LCFRS. As can be seen, in the schematic CFG derivation, X contributes a continuous component β to the full yield. In the schematic LCFRS derivation, however, X contributes three non-adjacent components to the full yield. Intuitively speaking, the difference between LCFRS and CFG productions is that the former allow for *arguments* on their non-terminals. Each argument spans a continuous part of the input string, i.e., an argument boundary denotes a discontinuity in the yield of the non-terminal. The production itself specifies how the lefthand side non-terminal yield is built from the yields of the righthand side non-terminals. In the example, the X node in the schematic LCFRS tree would be derived from a production which, on its left hand side, has a terminal $X(x_1, x_2, x_3)$. Note that CFG non-terminals can consequently be seen as LCFRS non-terminals with an implicit single argument.

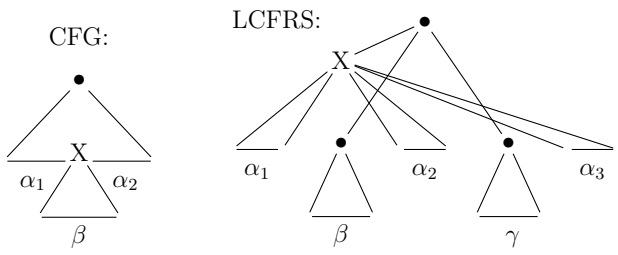


Figure 1. Locality in CFG vs. LCFRS derivation.

LCFRS has been employed in various subareas of computational linguistics. Among others, it has been used for syntactic modeling. In syntax, discontinuous structures arise when a displacement occurs within the sentence, such as, e.g., with topicalization. Constructions causing discontinuities are frequent: In the German TiGer treebank [3], about a fourth of all sentences contains at least one instance of discontinuity [4]; in the English Penn Treebank (PTB) [5], this holds for about a fifth of all sentences [6]. Below, three examples for discontinuities are shown. In (1), the discontinuity is due to the fact that a part of the VP is moved to the front. In (2), the discontinuity comes from the topicalization of the pronoun *Darüber*. (3), finally, is an example for a discontinuity which has several parts. It is due to a topicalization of the prepositional phrase *Ohne internationalen Schaden*, and *scrambling* [7] of elements after the verb.

(1) *Selbst besucht* hat er *ihn nie*
Personally visited has he *him never*
"He has never visited him personally"

(2) *Darüber* muss *nachgedacht* werden
About it must *thought* be
"One must think about that"

(3) *Ohne internationalen Schaden* könne *sich* Bonn *von dem Denkmal* nicht *distanzieren*
Without international damage could *itself* Bonn *from the monument* not *distance*
"Bonn could not distance itself from the monument without international damage."

The syntactic annotation in treebanks must account for discontinuity. In a constituency framework, this can be done by allowing crossing branches and grouping all parts of a discontinuous constituent under a single node. Similarly in dependency syntax [8], words must be connected such that edges end up crossing. As an example, Figure 2 shows both the constituency and dependency annotations of (2). While the discontinuous yields inhibit an interpretation of either structure as a CFG derivation tree, both can immediately been modeled as the derivation of an LCFRS [4,9].

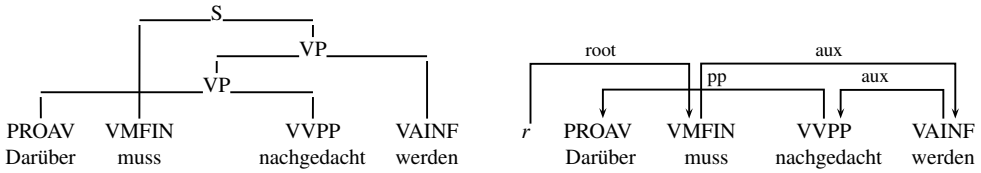


Figure 2. Discontinuity and non-projectivity in NeGra annotation (see (2)) [10].

In grammar engineering, LCFRS and related formalisms have been employed in two ways. *Grammatical Framework (GF)* is actively used for multilingual grammar development and allows for an easy treatment of discontinuities, see [11] for details. GF is an extension of Parallel Multiple Context-Free Grammar, which itself is a generalization of LCFRS [12]. TuLiPA [13] is a multi-formalism parser used in a development environment for variants of Tree Adjoining Grammar (TAG). It exploits the fact that LCFRS is a mildly context-sensitive formalism with high expressivity. LCFRS acts as a pivot formalism, i.e., instead of parsing directly with a TAG instance, TuLiPA parses with its equivalent LCFRS, obtained through a suitable grammar transformation [14].

LCFRS has also been used for the modeling of non-concatenative morphology, i.e., for the description of discontinuous phenomena below word level, such as stem derivation in Semitic languages. In such languages, words are derived by combining a discontinuous root with a discontinuous template. In Arabic, for instance, combining the root *k-t-b* with the template *i-a* results in *kitab* ("book"), combining it with *a-i* results in *katib* ("writer"), and combining it with *ma-∅-a* results in *maktab* ("desk"). Authors of [15,16] use LCFRS for the modeling of such processes. Finally, in machine translation, suitable versions of LCFRS, resp. equivalent formalisms, have been proposed

for the modeling of translational equivalence [17,18]. They offer the advantage that certain alignment configurations that cannot be modeled with synchronous variants of CFG can be induced by LCFRS [19]. As an example, see Figure 3, which shows an English sentence with its French translation, along with a grammar that models the alignment between both.

(4) Je ne veux plus jouer
I do not want to play anymore

$\langle X(jouer) \to \varepsilon, X(toplay) \to \varepsilon \rangle$
$\langle X(veux) \to \varepsilon, X(do, want) \to \varepsilon \rangle$
$\langle X(ne\ x_1\ plus\ x_2) \to X_{\boxed{1}}(x_1) X_{\boxed{2}}(x_2),$
$X(x_1\ not\ x_2 x_3\ anymore) \to X_{\boxed{1}}(x_1, x_2) X_{\boxed{2}}(x_3) \rangle$
$\ldots$

Figure 3. LCFRS for machine translation alignments.

Parsing is a key task since it serves as a backend in many practical applications such as the ones mentioned above. The parsing of LCFRS has received attention both on the symbolic and on the probabilistic side. Symbolic parsing strategies, such as CYK and Earley variants have been presented [20–22], as well as automaton-based parsing [23] and data-driven probabilistic parsing techniques [24–26]. To our knowledge, however, no LR strategy for LCFRS has so far been presented in the literature. LR parsing is an incremental shift-reduce parsing strategy in which the transitions between parser states are guided by an automaton which is compiled offline. LR parsers were first introduced for deterministic context-free languages [27] and later generalized to context-free languages [28,29] and tree-adjoining languages [30,31]. Also for conjunctive grammars and boolean grammars, LR parsing algorithms exist [32–34]. In this paper, we present an LR-style parser for LCFRS, extending our earlier work [35]. Our parsing strategy is based on the incremental parsing strategy implemented by Thread Automata [23].

The remainder of the article is structured as follows. In the following section, we introduce LCFRS and thread automata. Section 3 introduces LR parsing for the context-free case. Section 4 presents the LR algorithm for LCFRS along with an example. In particular, Section 4.1 gives the intuition behind the algorithm, Section 4.2 introduces the algorithms for automaton and parse table constructions, and Section 4.3 presents the parsing algorithm. Section 5 concludes the article.

2. Preliminaries

2.1. Linear Context-Free Rewriting Systems

We now introduce Linear Context-Free Rewriting Systems (LCFRS). In LCFRS, a single non-terminal can span $k \geq 1$ continuous blocks of a string. A CFG is simply a special case of an LCFRS in which $k = 1$. We notate LCFRS with the syntax of Simple Range Concatenation Grammars (SRCG) [36], a formalism equivalent to LCFRS. Note that we restrict ourselves to the commonly used string rewriting version of LCFRS and omit the more general definition of LCFRS of [37]. Furthermore, we assume that our LCFRSs are *monotone* (see condition 3 below) and *ε-free* (we can make this assumption without loss of generality see [20,38]).

Definition 1 (LCFRS). *An Linear Context-Free Rewriting System (LCFRS) [1,20] is a tuple* $G = (N, T, V, P, S)$ *where*

- N *is a finite set of non-terminals with a function dim:* $N \to \mathbb{N}$ *determining the* fan-out *of each* $A \in N$*;*
- T *and* V *are disjoint finite sets of terminals and variables;*
- $S \in N$ *is the start symbol with* $dim(S) = 1$*; and*
- P *is a finite set of rewriting rules (or productions). For all* $\gamma \in P$*, the following holds.*

1. γ *has the form*

$$A(\alpha_0, \ldots, \alpha_{dim(A)-1}) \to A_1(x_0^{(1)}, \ldots, x_{dim(A_1)-1}^{(1)}) \cdots A_m(x_0^{(m)}, \ldots, x_{dim(A_m)-1}^{(m)})$$

where $A, A_1, \ldots, A_m \in N$, $m \geq 0$ *being the rank of* γ; $x_j^{(l)} \in V$ *for* $1 \leq l \leq m, 0 \leq j < dim(A_i)$; *and* $\alpha_i \in (V \cup T)^+$ *for* $0 \leq i < dim(A)$; *all* α_i *and* $x_j^{(l)}$ *are called arguments or components.* γ *may be abridged as* $A(\vec{\alpha}) \rightarrow A_1(\vec{x_1}) \cdots A_m(\vec{x}_m)$; *and we define* $rank(\gamma) = m$.

2. *For all* $x \in V$ *there is at most one unique occurrence of* x *on each lefthand side and righthand side of* γ; *furthermore, there exists an lefthand side occurrence iff there exists a righthand side occurrence.*
3. *Variable occurrences in* γ *are ordered by a strict total order* $\prec$ *which is such that for all* x_1, x_2 *with occurrences in* γ, *if* x_1 *precedes* x_2 *in any* $\vec{x}_i$, $1 \leq i \leq m$, *then* x_1 *also precedes* x_2 *in* α.

The rank of G *is given by* $\max(\{rank(\gamma) \mid \gamma \in P\})$ *and written as* $rank(G)$. G *is of* fan-out k *if* $k \geq \max(\{dim(V) \mid V \in N\})$ *and is written as* $dim(G)$. *The set* $\mathcal{C}_G$ *is defined as* $\mathcal{C}_G = \bigcup_{\gamma \in P} \mathcal{C}_\gamma$ *(subscript* G *can be omitted if clear from the context).*

The LCFRS derivation is based on the *instantiation* of rules, the replacement of variable occurrences by strings of terminals.

Definition 2 (Instantiation). *An instantiation of a production* $\gamma \in P$ *with* $\gamma = A(\vec{\alpha}) \rightarrow A_1(\vec{\alpha}_1) \cdots A_m(\vec{\alpha}_m)$ *is given by a function* $\sigma_\gamma : V_\gamma \rightarrow T^*$, *where* $V_\gamma = \{x \mid \exists\, i, j$ *such that* $rhs(\gamma, i, j) = x\}$ *the set of all* $x \in V$ *occurring in* γ. γ *is instantiated if all variable occurrences* x *have been replaced by* $\sigma_\gamma(x)$. *A non-terminal in an instantiated production is an instantiated non-terminal.*

We now describe the LCFRS derivation process. A *(partial) derivation* is specified as a pair $[D : A(\vec{\alpha})]$ called *item*. Thereby, $A(\vec{\alpha})$ specifies the yield and D is the derivation tree. We use a bracketed representation of trees, e.g., $\gamma_0(\gamma_1\gamma_2)$ is a tree with a root node labeled γ_0 that has two daughters, a left daughter labeled γ_1 and a right daughter labeled γ_2.

Definition 3 (Derivation). *Let* $G = (N, T, V, P, S)$ *be an LCFRS. The set* Δ_G *of derivable items with respect to* G *contains all and only those items that can be deduced with the following deduction rules (the subscript* G *on* Δ_G *may be omitted if clear from the context).*

1. $\dfrac{}{[\gamma : A(\vec{\alpha})]}$ $\gamma : A(\vec{\alpha}) \rightarrow \varepsilon \in P$ and $L(v) = \gamma$
2. $\dfrac{[D_1 : A_1(\vec{\alpha}_1)] \cdots [D_m : A_m(\vec{\alpha}_m)]}{[\gamma(D_1 \ldots D_m) : A(\vec{\alpha})]}$ $A(\vec{\alpha}) \rightarrow A_1(\vec{\alpha}_1) \cdots A_m(\vec{\alpha}_m)$ is an instantiation of $\gamma \in P$

We can now define the string language of an LCFRS G on the basis of the *yield* of a non-terminal.

Definition 4 (Yield, String Language, and Tree Language). *Let* $G = (N, T, V, P, S)$ *be an LCFRS.*

1. *For every* $A \in N$ *we define* $yield(A) = \{\vec{\alpha} \mid \exists D$ *such that* $[D : A(\vec{\alpha})] \in \Delta_G\}$.
2. *The string language of* G *is defined as* $L_S(G) = \{w \mid \langle w \rangle \in yield(S)\}$.
3. *The derivation tree language of* G *is defined as* $L_D(G) = \{D \mid \exists w$ *such that* $[D : S(\langle w \rangle)] \in \Delta_G\}$.

Note that this definition of a tree language captures the set of derivation trees, not the derived trees with crossing branches along the lines of Figures 1 and 2.

Example 1 (LCFRS Derivation). *Figure 4 shows an LCFRS that generates the language* $a^n b^m a^n b^m a^n$, $n, m \in \mathbb{N}$, *together with a derivation of the string aabaabaa. The derivation works as follows. First, the leaf nodes of the derivation tree are produced using the first deduction rule with the productions* γ_4 *and* γ_5. *They introduce the first three as and the two bs. In order to insert three more as, we use the second deduction rule with production* γ_2. *This leads to a derivation tree with a root labeled* γ_2 *that has the* γ_4 *node as its single daughter. Finally the as and the bs are combined with the second deduction rule and production* γ_1. *This leads to a derivation tree with root label* γ_1 *that has two daughters, the first one is the* γ_2*-derivation tree and the second one the* γ_5 *node. The order of the daughters reflects the order of elements in the righthand sides of productions.*

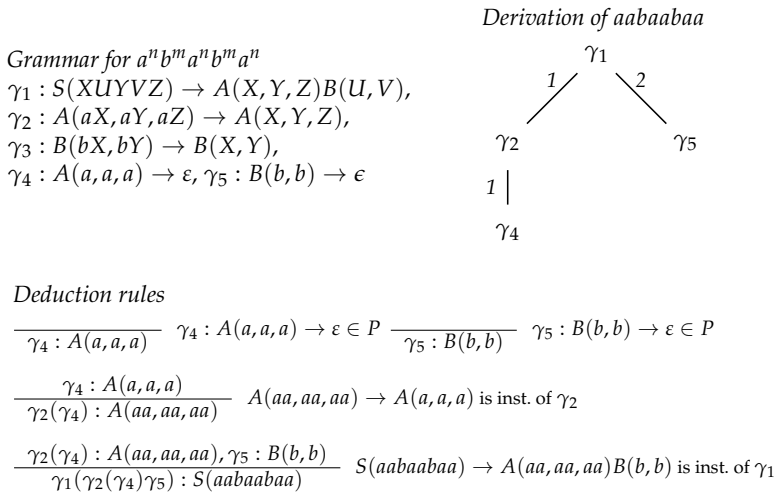


Figure 4. LCFRS example derivation of *aabaabaa* with grammar for $a^nb^ma^nb^ma^n$.

Finally, we define some additional notation that we will need later: Let $\gamma = A(\alpha_0, \ldots, \alpha_{dim(A)-1}) \to A_1(x_0^{(1)}, \ldots, x_{dim(A_1)-1}^{(1)}) \cdots A_m(x_0^{(m)}, \ldots, x_{dim(A_m)-1}^{(m)})$ be a production.

- $lhs(\gamma)$ gives A, $lhs(\gamma, i)$ gives α_i and $lhs(\gamma, i, j)$ the jth symbol of α_i; $rhs(\gamma, l)$ gives A_l, and $rhs(\gamma, l, k)$ gives the kth component of the lth RHS element. Thereby, i, j and k start with index 0, and l starts with index 1. These functions have value $\perp$ whenever there is no such element.
- In the sense of dotted productions, we define a set of symbols $\mathcal{C}_\gamma$ denoting computation points of γ as $\mathcal{C}_\gamma = \{\gamma_{i.j} \mid 0 \leq i < dim_A, 0 \leq j \leq |\alpha_i|\}$.

2.2. Thread Automata

Thread automata TA [23] are a generic automaton model which can be parametrized to recognize different mildly context-sensitive languages. The TA for LCFRS (LCFRS-TA) implements a prefix-valid top-down incremental parsing strategy similar to the ones of [21,22]. In this paper, we restrict ourselves to introducing this type of TA.

As an example, let us consider the derivation of the word *aaba* with the LCFRS from Figures 5 and 6 shows a sample run of a corresponding TA where only the successful configurations are given. We have one thread for each rule instantiation that is used in a parse. In our case, the derivation is $S(aaba) \overset{\alpha}{\Rightarrow} A(aa, ba) \overset{\beta}{\Rightarrow} A(a, b) \overset{\gamma}{\Rightarrow} \varepsilon$, we therefore have one thread for the α-rule, one for β and one for γ. The development of these threads is given in the three columns of Figure 6.

Each thread contains at each moment of the TA run only a single symbol, either a non-terminal or a dotted production. The TA starts with the α-thread containing the start symbol. In a predict step, this is replaced with the dotted production corresponding to starting the α-rule. Now the dot precedes the first variable of the non-terminal A in the righthand side. We therefore start a new A-thread (we "call" a new A). In this thread, rule β is predicted, and the dot is moved over the terminal a in a scan step, leading to $A(a \bullet x, ya) \to A(x, y)$ as content of the thread. Now, since the dot again precedes the first component of a righthand side non-terminal A, we call a new A-thread, which predicts γ this time. A scan allows to complete the first component of the γ-rule (thread content $A(a\bullet, b) \to \varepsilon$). Here, since the dot is at the end of a component, i.e., precedes a gap, the γ-thread is supended and its mother thread, i.e. the one that has called the γ-thread, becomes the active thread with the dot now moved over the variable for the firsts γ-component ($A(ax\bullet, ya) \to A(x, y)$). Similarly, a second suspend operation leads to the α-thread becoming the active thread again with the dot moved over the finished component of the righthand side A ($S(x \bullet y) \to A(x, y)$). The dot now precedes the variable of the second component of this A, therefore the β-thread is resumed with the dot moving on to the second component ($A(ax, \bullet ya) \to A(x, y)$). Similarly, a second resume operation leads to the γ-thread

becoming active with content $A(a, \bullet b) \rightarrow \varepsilon$. The subsequent scan yields a completed A-item that can be published, which means that the mother thread becomes active with the dot moved over the variable for the last righthand side component. After another scan, again, we have a completed item and the α-thread becomes active with a content $S(xy\bullet) \rightarrow A(x, y)$ that signifies that we have found a complete S-yield. Since the entire input has been processed, the TA run was successful and the input is part of the language accepted by the TA.

$$\alpha : S(xy) \rightarrow A(x, y) \qquad \gamma : A(a, b) \rightarrow \varepsilon$$
$$\beta : A(ax, ya) \rightarrow A(x, y)$$

Figure 5. LCFRS for $\{a^n aba^n \mid n \geq 0\}$.

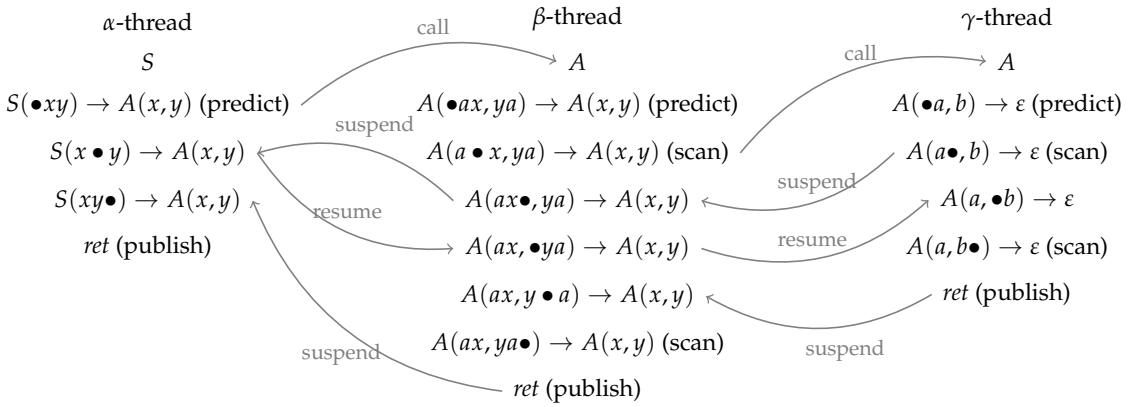


Figure 6. Sample TA-run for $w = aaba$.

More generally, an LCFRS-TA for some LCFRS $G = (N, T, V, P, S)$ works as follows. As we have seen, the processing of a single instantiated rule is handled by a single *thread* which will traverse the lefthand side arguments of the rule. A thread is given by a pair $p : X$, where $p \in \{1, \ldots, m\}^*$ with m the rank of G is the *address*, and $X \in N \cup \{ret\} \cup \mathcal{C}$ where $ret \notin N$ is the *content* of the thread. In our example in Figure 6, addresses were left aside. They are, however, crucial for determining the mother thread to go back to in a suspend step. In Figure 6, the α-thread has address ε, the β-thread address 1 and the γ-thread address 11, i.e., the address of the mother thread is a prefix of the address of the child thread, respectively. This is why a thread knows where to go back to in a *suspend* step. An automaton state is given by a tuple $\langle i, p, \mathcal{T} \rangle$ where $\mathcal{T}$ is a set of threads, the *thread store*, p is the address of the active thread, and $i \geq 0$ indicates that i tokens have been recognized. We use $\langle 0, \varepsilon, \{\varepsilon : S\} \rangle$ as start state.

Definition 5 (LCFRS-TA). *Let $G = (N, T, V, P, S)$ be a LCFRS. The* LCFRS thread automaton for *G is a tuple $\langle N', T, S, ret, \delta, \Theta \rangle$ with*

- $N' = N \cup \mathcal{C}_G \cup \{ret\}$;
- *δ is a function from $\mathcal{C}$ to $\{1, \ldots, m\} \cup \{\bot\}$ such that $\delta(\gamma_{k,i}) = j$ if there is a l such that $lhs(\gamma, k, i) = rhs(\gamma, j-1, l)$, and $\delta(\gamma_{k,i}) = \bot$ if $lhs(\gamma, k, i) \in T \cup \{\bot\}$; and*
- *Θ is a finite set of transitions as defined below.*

Intuitively, a δ value j tells us that the next symbol to process is a variable that is an argument of the j-th righthand side non-terminal.

Definition 6 (Transitions of a LCFRS-TA). *The transition set Θ of a LCFRS-TA as in Definition 5 contains the following transitions:*

- ***Call**: $\gamma_{k,i} \rightarrow [\gamma_{k,i}]A \in \Theta$ if $A = rhs(\gamma, j-1)$ and $lhs(\gamma, k, i) = rhs(\gamma, j-1, 0)$ where $j = \delta(\gamma_{k,i})$.*

- ***Predict***: $A \to \gamma_{0,0} \in \Theta$ *if* $A = lhs(\gamma)$.
- ***Scan***: $\gamma_{k,i} \overset{lhs(\gamma,k,i)}{\to} \gamma_{k,i+1} \in \Theta$ *if* $lhs(\gamma,k,i) \in T$.
- ***Publish***: $\gamma_{k,j} \to ret \in \Theta$ *if* $dim(lhs(\gamma)) = k+1$ *and* $j = |lhs(\gamma,k)|$.
- ***Suspend***:

  (i) $[\gamma_{k,i}]ret \to \gamma_{k,i+1} \in \Theta$ *if* $lhs(\gamma,k,i) = rhs(\gamma,\delta(\gamma_{k,i})-1,dim(rhs(\delta(\gamma_{k,i})-1))-1)$.
  (ii) $[\gamma_{k,i}]\beta_{l,j} \to \gamma_{k,i+1}[\beta_{l,j}] \in \Theta$ *if* $dim(lhs(\beta)) > l+1$, $|lhs(\beta,l)| = j$, $lhs(\gamma,k,i) = rhs(\gamma,\delta(\gamma_{k,i})-1,l)$ *and* $rhs(\gamma,\delta(\gamma_{k,i})-1) = lhs(\beta)$.

- ***Resume***: $\gamma_{k,i}[\beta_{l,j}] \to [\gamma_{k,i}]\beta_{l+1,0} \in \Theta$ *if* $lhs(\gamma,k,i) = rhs(\gamma,\delta(\gamma_{k,i})-1,l+1)$, $rhs(\gamma,\delta(\gamma_{k,i})-1) = lhs(\beta)$ *and* $\beta_{l,j+1} \notin \mathcal{C}_G$.

These are all thransitions in Θ.

A transition $\alpha \overset{a}{\to} \beta$ roughly indicates that in the current thread store, α can be replaced with β while scanning a. Square brackets in α and β indicate parts that do not belong to the active thread. Call transitions start a new thread for a daughter non-terminal, i.e., a righthand side non-terminal. They move down in the parse tree. Predict transitions predict a new rule for a non-terminal A. Scan transitions read a lefthand side terminal while scanning the next input symbol. Publish marks the completion of a production, i.e., its full recognition. Suspend transitions suspend a daughter thread and resume the parent. i.e., move up in the parse tree. There are two cases: Either (i) the daughter is completely recognized (thread content *ret*) or (ii) the daughter is not yet completely recognized, we have only finished one of its components. Resume transitions resume an already present daughter thread, i.e., move down into some daughter that has already been partly recognized.

Figure 7 shows the transitions for our sample grammar. Let us explain the semantics of these rules with these examples before moving to the general deduction rules for TA configurations. As already mentioned, a rule $X_1 \overset{t}{\to} X_2$ with $X_1, X_2 \in N'$ and $t \in T \cup \{\varepsilon\}$ expresses that with X_1 being the content of the active thread, we can replace X_1 with X_2 while scanning t. The predict, scan and publish rules are instances of this rule type. $X \in N'$ in square brackets to the left characterize the content of the mother of the active thread while $X \in N'$ in square brackets to the right characterize the content of a daughter thread where the address of the daughter depends on the one of the active thread and the δ-value of the content of the active thread. Consider for instance the last suspend rule. It signifies that if the content of the active thread is $\gamma_{0,1}$ and its mother thread has content $\beta_{0,1}$, this mother thread becomes the new active thread with its content changed to $\beta_{0,2}$ while the daughter thread does not change content. The resume rules, in contrast to this, move from the active thread to one of its daughters. Take for instance the last rule. It expresses that, if the active thread has content $\beta_{1,0}$ and if this thread has a $\delta(\beta_{1,0})$th daughter thread with content $\gamma_{0,1}$, we make this daughter thread the active one and change its content to $\gamma_{1,0}$, i.e., move its dot over a gap.

Call:	$\alpha_{0,0} \to [\alpha_{0,0}]A$	$\beta_{0,1} \to [\beta_{0,1}]A$		
Predict:	$S \to \alpha_{0,0}$	$A \to \beta_{0,0}$	$A \to \gamma_{0,0}$	
Scan:	$\beta_{0,0} \overset{a}{\to} \beta_{0,1}$	$\beta_{1,1} \overset{a}{\to} \beta_{1,2}$	$\gamma_{0,0} \overset{a}{\to} \gamma_{0,1}$	$\gamma_{1,0} \overset{b}{\to} \gamma_{1,1}$
Publish:	$\alpha_{0,2} \to ret$	$\beta_{1,2} \to ret$	$\gamma_{1,1} \to ret$	
Suspend:	$[\alpha_{0,1}]ret \to \alpha_{0,2}$	$[\alpha_{0,0}]\beta_{0,2} \to \alpha_{0,1}[\beta_{0,2}]$	$[\beta_{0,1}]\beta_{0,2} \to \beta_{0,2}[\beta_{0,2}]$	
	$[\beta_{1,0}]ret \to \beta_{1,1}$	$[\alpha_{0,0}]\gamma_{0,1} \to \alpha_{0,1}[\gamma_{0,1}]$	$[\beta_{0,1}]\gamma_{0,1} \to \beta_{0,2}[\gamma_{0,1}]$	
Resume:	$\alpha_{0,1}[\beta_{0,2}] \to [\alpha_{0,1}]\beta_{1,0}$	$\alpha_{0,1}[\gamma_{0,1}] \to [\alpha_{0,1}]\gamma_{1,0}$		
	$\beta_{1,0}[\beta_{0,2}] \to [\beta_{1,0}]\beta_{1,0}$	$\beta_{1,0}[\gamma_{0,1}] \to [\beta_{1,0}]\gamma_{1,0}$		

(Rules: $\alpha : S(xy) \to A(x,y)$, $\beta : A(ax,ya) \to A(x,y)$, $\gamma : A(a,b) \to \varepsilon$)

Figure 7. TA transitions for the LCFRS from Figure 5.

This is not exactly the TA for LCFRS proposed in [23] but rather the one from [39], which is close to the Earley parser from [21].

The set of configurations for a given input $w \in T^*$ is then defined by the deduction rules in Figure 8 (the use of set union $\mathcal{S}_1 \cup \mathcal{S}_2$ in these rules assumes that $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$). The accepting state of the automaton for some input w is $\langle |w|, \varepsilon, \{\varepsilon : ret\}\rangle$.

$$\text{Initial configuration: } \frac{}{\langle 0, \varepsilon, \{\varepsilon : S\}\rangle} \qquad \text{Call: } \frac{\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,i}\}\rangle}{\langle i, pj, \mathcal{S} \cup \{p : \gamma_{k,i}\} \cup \{pj : A\}\rangle} \quad \begin{array}{l}\gamma_{k,i} \to [\gamma_{k,i}]A \in \Theta,\\ A \in N, \delta(\gamma_{k,i}) = j+1\end{array}$$

$$\text{Predict: } \frac{\langle i, p, \mathcal{S} \cup \{p : A\}\rangle}{\langle i, p, \mathcal{S} \cup \{p : \gamma_{0,0}\}\rangle} \quad \begin{array}{l}A \in N,\\ A \to \gamma_{1,0} \in \Theta\end{array}$$

$$\text{Scan: } \frac{\langle j, p, \mathcal{S} \cup \{p : \gamma_{k,i}\}\rangle}{\langle j+1, p, \mathcal{S} \cup \{p : \gamma_{k,i+1}\}\rangle} \quad \gamma_{k,i} \overset{w_{j+1}}{\to} \gamma_{k,i+1} \in \Theta \qquad \text{Publish: } \frac{\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,j}\}\rangle}{\langle i, p, \mathcal{S} \cup \{p : ret\}\rangle} \quad \gamma_{k,j} \to ret \in \Theta$$

$$\text{Suspend 1: } \frac{\langle i, pj, \mathcal{S} \cup \{p : \gamma_{k,i}, pj : ret\}\rangle}{\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,i+1}\}\rangle} \quad [\gamma_{k,i}]ret \to \gamma_{k,i+1} \in \Theta$$

$$\text{Suspend 2: } \frac{\langle i, pj, \mathcal{S} \cup \{p : \gamma_{k,i}, pj : \beta_{l,m}\}\rangle}{\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,i+1}, pj : \beta_{l,m}\}\rangle} \quad [\gamma_{k,i}]\beta_{l,m} \to \gamma_{k,i+1}[\beta_{l,m}] \in \Theta$$

$$\text{Resume: } \frac{\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,i}, p\delta(\gamma_{k,i}) : \beta_{l,j}\}\rangle}{\langle i, p\delta(\gamma_{k,i}), \mathcal{S} \cup \{p : \gamma_{k,i}, p\delta(\gamma_{k,i}) : \beta_{l+1,0}\}\rangle} \quad \gamma_{k,i}[\beta_{l,j}] \to [\gamma_{k,i}]\beta_{l+1,0} \in \Theta$$

Goal item: $\langle |w|, \varepsilon, \{\varepsilon : ret\}\rangle$.

Figure 8. Deduction rules for TA configurations.

3. LR Parsing

In an LR parser (see [40] chapter 9), the parser actions are guided by an automaton, resp. a *parse table* which is compiled offline. Consider the context-free case. An LR parser for CFG is a guided shift-reduce parser, in which we first build the LR automaton. Its states are sets of dotted productions closed under prediction, and its transitions correspond to having recognized a part of the input, e.g., to moving the dot over a righthand side element after having scanned a terminal or recognized a non-terminal.

Consider for instance the CFG $G = \langle \{S\}, \{a, b, c\}, \{S \to aSb \,|\, c\}, S\rangle$. Its LR automaton is given in Figure 9. The construction of this automaton starts from the prediction closure of $\{S' \to \bullet S\}$, which contains the two S-productions with the dot at the beginning of the righthand side. The prediction closure of a set of dotted productions q is defined as follows: if $A \to \alpha \bullet B\beta \in q$, then $B \to \bullet\gamma \in q$ for all B-productions $B \to \gamma$ in the grammar. From a state q, a new state can be obtained by computing all items that can be reached by moving the dot over a certain $x \in N \cup T$ in any of the dotted productions in q and then, again, building the prediction closure. The new state can be reached from the old one by an edge labeled with x. This process is repeated until all possible states have been constructed. In our example, when being in q_0, we can for instance reach q_1 by moving the dot over an a, and we can reach q_5 (the acceptance state) by moving the dot over an S.

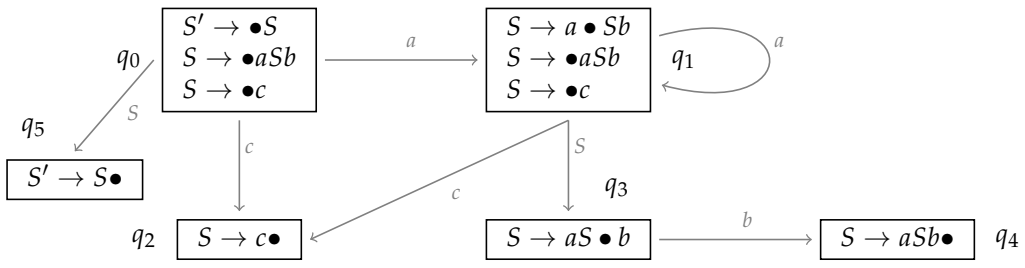


Figure 9. Sample LR automaton for a CFG.

This automaton guides the shift-reduce parsing in the following way: At every moment, the parser is in one of the states. We can *shift* an $a \in T$ if there is an outgoing a-transition. The state changes according to the transition.We can *reduce* if the state contains a completed $A \to \alpha\bullet$. We then reduce

with $A \to \alpha$. The new state is obtained by following the A-transition, starting from the state reached before processing that rule.

The information about possible shift and reduce operations in a state is precompiled into a parse table: Given an LR automaton with n states, we build a *parse table* with n rows. Each row i, $0 \leq i < n$, describes the possible parser actions associated with the state q_i, i.e., for each state and each possible shift or reduce operation, it tells us in which state to go after the operation. The table has $|T| + |N| + 1$ columns where the first are indexed with the terminals, then follows a column for reduce operations and then a column for each of the non-terminals. For every pair of states q_i, q_j that are linked by an a-edge for some $a \in T$, the field of q_i and a contains sj, which indicates that in q_i, one can shift an a and then move to q_j. For every state q_i that contains rule number j with a dot at the end of the righthand side, we add rj to the reduce field of q_i. Finally, for every pair of states q_i, q_j that are linked by an A-edge for some $A \in N$, the field of q_i and A contains j, which indicates that in q_i, after having completed an A, we can move to q_j. On the left of Figure 10, we see the parse table resulting from the automaton in Figure 9. The first $|T| + 1$ columns are the so-called *action*-part of the table while the last $|N|$ columns form the *goto*-part.

	a	b	c		S
q_0	s1		s2		5
q_1	s1		s2		3
q_2				r2	
q_3		s4			
q_4				r1	
q_5				acc	

Stack	remaining input	operation
q_0	acb	initial configuration
$q_0 a q_1$	cb	s1 (shift and move to q_1)
$q_0 a q_1 c q_2$	b	s2
$q_0 a q_1 S q_3$	b	r2 (reduce with rule 2)
$q_0 a q_1 S q_3 b q_4$	ε	s4
$q_0 S q_5$	ε	r1

Figure 10. Parse table and sample parse (rule 1 = $S \to aSb$, rule 2 = $S \to c$).

The parse table determines the possible reductions and shifts we can do at every moment. We start parsing with a stack containing only q_0. In a shift, we push the new terminal followed by the state indicated in the table on the stack. In a reduction, we use the production indicated in the table. We pop its righthand side (in reverse order) and push its lefthand side onto the stack. The new state is the state indicated in the parse table field for the state preceding the lefthand side of this production on the stack and the lefthand side category of the production. On the right of Figure 10, we see a sample trace of a parse according to our LR parse table. Note that this example allowed for deterministic LR parsing, which is not the case in general. There might be shift-reduce or reduce-reduce conflicts.

4. LR for LCFRS

4.1. Intuition

We will now adapt the ideas underlying LR parsing for CFG to the case of LCFRS. The starting point is the set of possible threads that can occur in a TA for a given LCFRS. Our LR automaton for LCFRS precompiles predict, call and resume steps into single states, i.e., states are closed under these three operations.

As an example, consider the TA from Section 2.2. Figure 11 sketches some of the states one obtains for this TA.

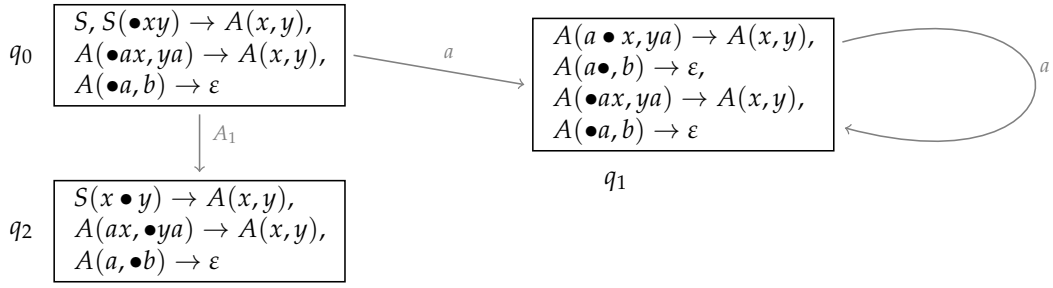


Figure 11. Sample LR states for the TA in Figure 6.

As in the CFG case, we start with a state for the start symbol S and, in this state, add all predictions of S-rules with the dot at the left of the lefthand side component. Furthermore, for every item in some state q where the dot precedes a variable: If this variable is the ith component of a righthand side non-terminal A, we add to q all A-rules with the dot at the beginning of the ith component in the lefthand side. For $i = 1$, this is a call-and-predict operation, for $i > 1$ a resume operation.

Moving the dot over a terminal symbol a leads to a transition labeled with a that yields a new state. And moving the dot over the variable that is the ith component of some righthand side symbol A leads to a transition labeled with A_i.

Our parser then has shift operations and two kinds of reduce operations (the two types of suspend operations in the TA). Either we reduce the ith component of some A where $i < dim(A)$ or we reduce the last component of some A. In the first case, we have to keep track of the components we have already found, in particular of their derivation tree addresses. This is why we have to extend the states sketched in Figure 11 to using dotted productions with addresses.

Our predict/resume closure operations then have to add the index of the daughter that gets predicted/resumed to the address. A problem here is that this can lead to states containing infinitely many items. An example is q_2 in Figure 11. The dotted production $A(ax, \bullet ya) \to A(x,y)$ leads, via the resume closure, to new instances of $A(ax, \bullet ya) \to A(x,y)$ and $A(a, \bullet b) \to \varepsilon$, this time with a different address since these are daughters of the first. Consequently, q_2 would be the set $\{\varepsilon : S(x \bullet y) \to A(x,y), 1 : A(ax, \bullet ya) \to A(x,y), 1 : A(a, \bullet b) \to \varepsilon, 11 : A(ax, \bullet ya) \to A(x,y), 11 : A(a, \bullet b) \to \varepsilon, 111 : A(ax, \bullet ya) \to A(x,y), 111 : A(a, \bullet b) \to \varepsilon, \dots\}$. Such a problem of an infinite loop of predict or resume steps occurs whenever there is a left recursion in one of the components of a rule. We do not want to exclude such recursions in general. Therefore, in order to keep the states finite nevertheless, we allow the addresses to be regular expressions. Then q_2 for instance becomes $\{\varepsilon : S(x \bullet y) \to A(x,y), 1^+ : A(ax, \bullet ya) \to A(x,y), 1^+ : A(a, \bullet b) \to \varepsilon\}$. Concerning the transitions, here as well, we add the addresses of the item(s) that have triggered that transition. See Figure 12 for an example. A new state always starts with an item with address ε.

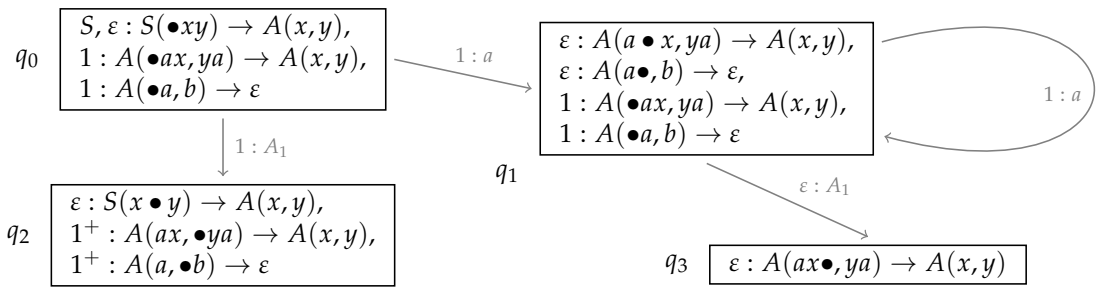


Figure 12. Sample LR states for the TA in Figure 6 including addresses.

Once the automaton is constructed, the shift-reduce parser proceeds as follows: A configuration of the parser consists of a stack, a set of completed components and the remaining input. The completed

components are of the form $r\boxed{k}\gamma_i$ where r is a regular expression, $\boxed{k}$ is a pointer to the derivation tree node for this use of the rule $\gamma : A(a, b) \to \varepsilon$ and i is the component index. The stack has the form $\Gamma_1 x_1 \Gamma_2 \dots x_{n-1}\Gamma_n$ where Γ_i is a regular expression followed by a state and $x_i \in T \cup \{\boxed{j}_k \mid \boxed{j}$ points at a derivation tree node for a rule γ with $1 \leq k \leq dim(lhs(\gamma))\}$.

For the input *aaba* for instance, we would start with a stack εq_0 and then, following the corresponding transition in the automaton, we can shift an a and move to q_1. Since the shift transition tells us that this shift refers to the daughter with address 1, this address is concatenated to the preceding address. The new stack would be $\varepsilon q_0\ a\ 1q_1$ and, after a second shift, $\varepsilon q_0\ a\ 1q_1\ a\ 11q_1$. In q_1 we can not only shift as but we can also reduce since there is an item with the dot at the end of the first A-component, $A(a\bullet, b) \to \varepsilon$. Such a reduce means that we remove this component (here a single a) from the stack, then create a node $11\boxed{0}\gamma_1$ as part of the set of completed components, and then push its pointer and index (here $\boxed{0}_1$). The transition from the state preceding the a that is labeled $lhs(\gamma)_1$ gives the new state and address. The stack is then $\varepsilon q_0\ a\ 1q_1\ \boxed{0}_1\ 1q_3$, and the set of completed components is $\{11\boxed{0}\gamma_1\}$. q_3 also tells us that a first component of a rule, this time $A(ax, ya) \to A(x, y)$, has been completed. It consists of a followed by a first A-component, therefore we have to remove a and $\boxed{0}_1$ (in reverse order) from the stack and then follow the A_1 transition from q_0. The result is a stack $\varepsilon q_0\ \boxed{1}_1\ 1q_2$ and a completed components set $\{1\boxed{1}\beta_1, 11\boxed{0}\gamma_1\}$. The completed components are, furthermore, equipped with an immediate dominance relation. In this example, $\boxed{1}$ immediately dominates $\boxed{0}$. Whenever a reduce state concerns a component that is not the first one, we check in our set of completed components that we have already found a preceding component with the same address or (if it is a regular expression denoting a set) a compatible address. In this case, we increment the component index of the corresponding derivation tree node. In the end, in case of a successful parse, we have a stack $\varepsilon q_0\boxed{i}_1\varepsilon q_{acc}$ and a component set giving the entire derivation tree with root node $\boxed{i}$.

4.2. Automaton and Parse Table Construction

The states of the LR-automaton are sets of pairs $r : X$ where r is a regular expression over $\{1, \dots, rank(G)\}$, and $X \in \mathcal{C}_G \cup \{S\}$. They represent predict and resume closures.

Definition 7. *We define the* predict/resume closure $\overline{q}$ *of some set* $q \subseteq \{\varepsilon : X \mid X \in \mathcal{C}_G \cup \{S\}\}$ *by the following deduction rules:*

- *Initial predict:* $\dfrac{\varepsilon : S}{\varepsilon : \alpha_{0,0}}\ lhs(\alpha) = S$
- *Predict/resume:* $\dfrac{p : \gamma_{i,j}}{pk : \gamma'_{l,0}}\ lhs(\gamma, i, j) = rhs(\gamma, k-1, l), rhs(\gamma, k) = lhs(\gamma')$

This closure is not always finite. However, if it is not, we obtain a set of items that can be represented by a finite set of pairs $r : \gamma_{i,j}$ plus eventually $\varepsilon : S$ such that r is a regular expression denoting a set of possible addresses. As an example for such a case, see q_2 in Figure 12. Let Reg_G be the set of all regular expressions over $\{1, \dots, rank(G)\}$.

Lemma 8. *For every* $q \subseteq \{\varepsilon : X \mid X \in \mathcal{C}_G \cup \{S\}\}$, *it holds that if* $\overline{q}$ *is infinite, then there is a finite set* $\overline{q}' \subset \{r : X \mid r \in Reg_G, X \in \mathcal{C}_G \cup \{S\}\}$ *such that* $\overline{q} = \{p : X \mid$ *there is an* $r : X \in \overline{q}'$ *with* $p \in L(r)\}$.

Proof. $\mathcal{C}_G$ is finite. For each $\gamma_{i,j} \in \mathcal{C}_G$, the set of possible addresses it might be combined with in a state that is the closure of $\{\varepsilon : X_1, \varepsilon : X_2, \dots, \varepsilon : X_n\}$ is generated by the CFG $\langle N, T, P, S_{new}\rangle$ with

- $N = \mathcal{C}_G \cup \{S\} \cup \{S_{new}\}$,
- $T = \{1, \dots, m\}$, and
- P contains the following rules:

- $S_{new} \to X_i \in P$ for all $1 \leq i \leq n$,
- $X \to Yk \in P$ for all instances $\frac{p:X}{pk:Y}$ of deduction rules,
- and $\gamma_{i,j} \to \varepsilon$.

This is a regular grammar, its string language can thus be characterized by a regular expression. □

The construction of the set of states starts with $q_0 = \overline{\{\varepsilon : S\}}$. From this state, follwing a-transitions or A_i-transitions as sketched above, we generate new states. We first formalize these transitions:

Definition 9. *Let $G = (N, T, V, P, S)$ be a LCFRS. Let $q \subseteq \{\varepsilon : S\} \cup \{r : X \mid r \in Reg_G, X \in \mathcal{C}_G\}$.*

- *For every $A \in N$, every $1 \leq i \leq dim(A)$ and every $r \in Reg_G$, we define*
  - $read(q, A_i, r) = \{\varepsilon : \gamma_{j,k+1} \mid r : \gamma_{j,k} \in q$ *and there is some l such that* $rhs(\gamma, l) = A$ *and* $lhs(\gamma, j, k) = \underline{rhs(\gamma, l, i-1)}\}$ *and*
  - $\overline{read}(q, A_i, r) = read(q, A_i, r)$.
- *For every $a \in T$ and every $r \in Reg_G$, we define*
  - $\underline{read}(q, a, r) = \{\varepsilon : \gamma_{j,k+1} \mid r : \gamma_{j,k} \in q$ *and* $lhs(\gamma, j, k) = a\}$ *and*
  - $\overline{read}(q, a, r) = read(q, a, r)$.

The set of states of our automaton is then the closure of $\{q_0\}$ under the application of the $\overline{read}$-functions. The edges in our automaton correspond to $\overline{read}$-transitions, where each edge is labeled with the corresponding pair $r : A_i$ or $r : a$ respectively. Furthermore, we add a state $\{\varepsilon : S\bullet\}$ and an edge labeled $\varepsilon : S_1$ from the start state to this state, which is the acceptance state.

The complete automaton we obtain for the grammar in Figure 5 is shown in Figure 13. For the sake of readability, we use dotted productions instead of the computation points $\gamma_{i,j}$.

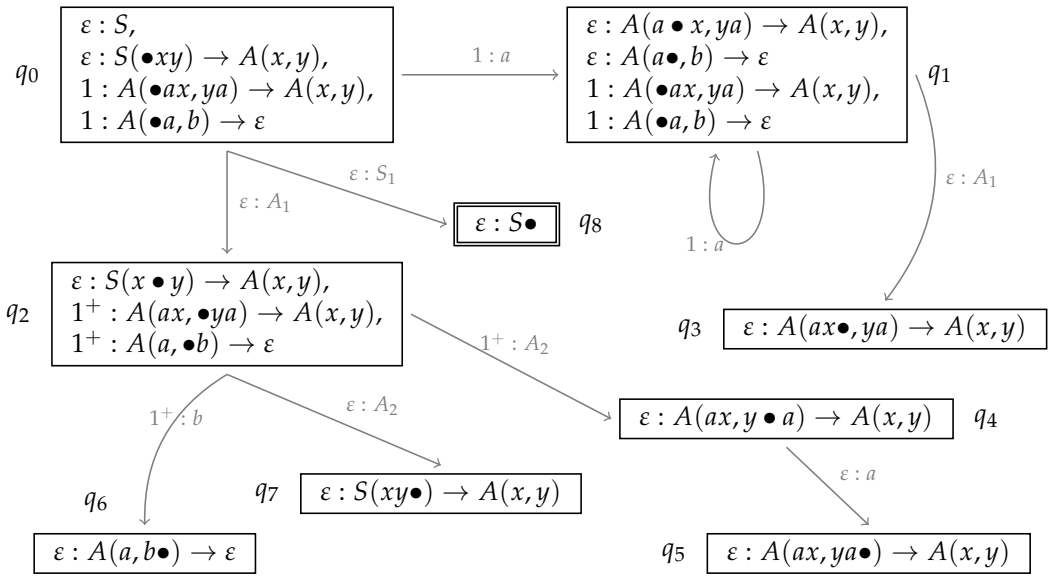


Figure 13. The automaton for the LCFRS in Figure 5.

The number of possible states is necessarily finite since each state is the closure of some set containing only items with address ε. There are only finitely many such sets.

In the parse table, our operations are $s(p, q)$ for shifting some terminal a followed by the old address concatenated with p and state q and $r(\alpha, i)$ for reducing the ith component of rule α. The two

reduce operations can be distinguished by the component indices. These operations are entered into the first $|T|+1$ columns of the table, the so-called *action* part. Furthermore, the *goto*-part of the table tells where to go when traversing a component edge and which address to add then.

The parse table can be read off the automaton as follows:

Definition 10. *Let $G = \langle N, T, P, S\rangle$ be a LCFRS, M its LR-automaton with state set Q. Then we define the corresponding LR parse table as a pair $\langle action, goto\rangle$ with action a $|Q| \times (|T|+1)$ table and goto a $|Q| \times |\{A_i \mid A \in N, 1 \leq i \leq dim(A)\}|$ table. The content of these tables is as follows:*

- $s(p,q') \in action(q,a)$ *iff* $\overline{read}(q,a,p) = q'$;
- $r(\gamma,i) \in action(q,-)$ *iff there is some* $p : \gamma_{i-1,k} \in q$ *such that* $k = |lhs(\gamma, i-1)|$.
- $\langle p,q'\rangle \in goto(q,A_i)$ *iff* $\overline{read}(q,A_i,p) = q'$.
- *Nothing else is in these tables.*

Figure 14 shows the parse table for our example.

	action			*goto*		
	a	b		A_1	A_2	S_1
0	$s(1,1)$			$\langle\varepsilon,2\rangle$		$\langle\varepsilon,8\rangle$
1	$s(1,1)$		$r(\gamma,1)$	$\langle\varepsilon,3\rangle$		
2		$s(1^+,6)$			$\langle 1^+,4\rangle, \langle\varepsilon,7\rangle$	
3			$r(\beta,1)$			
4	$s(\varepsilon,5)$					
5			$r(\beta,2)$			
6			$r(\gamma,2)$			
7			$r(\alpha,1)$			
8			acc			

rules: $\alpha : S(xy) \to A(x,y)$, $\beta : A(ax,ya) \to A(x,y)$, $\gamma : A(a,b) \to \varepsilon$

Figure 14. The parse table.

4.3. Parsing Algorithm

The parser has slightly more complex configurations than the CFG shift-reduce parser. Let Q be the set of states of the LR automaton for a given LCFRS G, q_0 the state containing $\varepsilon : S$ and q_{acc} the *accepting* state containing $\varepsilon : S\bullet$. Furthermore, let $\mathcal{N} = \{\boxed{0}, \boxed{1}, \boxed{2}, \dots\}$ be the set of possible pointers to derivation tree nodes. The parser configurations are then triples $\langle \Gamma, C, i\rangle$ consisting of

- a stack $\Gamma \in Reg_G \circ Q \circ ((T \cup \{\boxed{i}_k \mid \boxed{i} \in \mathcal{N}, k \leq max_{A\in N} dim(A)\}) \circ Reg_G \circ Q)^*$,
- a set of constraints C that are either of the form $r\boxed{i}\gamma_j$ with $r \in Reg_G$, $\boxed{i} \in \mathcal{N}$, $\gamma \in P$ and $1 \leq j \leq dim(lhs(\gamma))$ or of the form $\boxed{k} \lhd_d \boxed{l}$ with $\boxed{k}, \boxed{l} \in \mathcal{N}$, $1 \leq d \leq rank(G)$.
- and the index j up to which the input w has been recognized, $0 \leq j \leq |w|$.

Concerning the constraint set C, the first type of constraints describes nodes in the derivation tree while the second describes immediate dominance relations between them, more precisely the fact that one node is the dth daughter of another node. There is an immediate relation between the two: if $\boxed{i_1} \lhd_d \boxed{i_2}$, then the address of $\boxed{i_2}$ must be the regular expression of $\boxed{i_1}$ concatenated with the daughter index d of $\boxed{i_2}$. Let $\mathcal{N}_C$ be the set of indices from $\mathcal{N}$ occurring in C.

The initial configuration is $\langle \varepsilon q_0, \emptyset, 0\rangle$. From a current configuration, a new one can be obtained by one of the following operations:

- **Shift**: Whenever we have pq on top of the stack, the next input symbol is a and in our parse table we have $action(q,a) = s(p',q')$, we push a followed by $pp'q'$ onto the stack and increment the index j.

- **Reduce**: Whenever the top of the stack is $p_1 q$ and we have a parse table entry $r(\gamma, i) \in action(q, -)$, we can reduce the ith component of γ, i.e, suspend γ. Concretely, this means the following:
  - Concerning the constraint set C, if $i = 1$, we add $p_1 \boxed{k} \gamma_1$ to C where $\boxed{k}$ is a new pointer.
  - If $i \geq 1$, we check whether there is a $p_2 \boxed{k} \gamma_{i-1}$ in C such that the intersection $L(p_1) \cap L(p_2)$ of the languages denoted by p_1 and p_2, $L(p_1)$ and $L(p_2)$, is not empty. We then replace $p_2 \boxed{k} \gamma_{i-1}$ in C with $p \boxed{k} \gamma_i$ where p is a regular expression denoting $L(p_1) \cap L(p_2)$.
  - Concerning the stack Γ, we remove $3|lhs(\gamma, i)|$ elements (i.e., terminals/component pointers and their regular expressions and states) from Γ. Suppose the topmost state on the stack is now $p'q'$. If $goto(q', lhs(\gamma)_i) = \langle p'', q'' \rangle$, we push $\boxed{k}_i$ followed by $p'p''q''$ on the stack.
  - Concerning the dominance relations, for every $\boxed{i}_j$ among the symbols we removed from the stack, we add $\boxed{k} \lhd_d \boxed{i}$ where d is the corresponding daughter index.
  - It can happen that, thanks to the new dominance relations, we can further restrict the regular expressions characterizing the addresses in C. For instance, if we had $1^+\boxed{i} \ldots, 1^+\boxed{k} \ldots$ and $\boxed{i} \lhd_1 \boxed{k}$ in our set, we would replace the address of $\boxed{k}$ with 1^+1.
  - The index j remains unchanged in reduce steps.

Note that the intersection of two deterministic finite state automata is quadratic in the size of the two automata. In LCFRS without left recursion in any of the components, the intersection is trivial since the regular expressions denote only a single path each.

As an example, we run the parser from Section 4.2 with an input $w = aaba$. The trace is shown in Figure 15. In this example, the immediate dominance constraints in C are depicted as small trees. We start in q_0, and shift two as, which leads to q_1. We have then fully recognized the first components of γ and β: We suspend them and keep them in the set of completed components, which takes us to q_2. Shifting the b takes us to q_6, from where we can reduce, which finally takes us to q_4. From there, we can shift the remaining a (to q_5), with which we have fully recognized β. We can now reduce both β and with that, α, which takes us to the accepting state q_8.

Stack	**Constraints on Derivation Tree**	**Remaining Input**	**Operation**
εq_0	–	$aaba$	initial state
$\varepsilon q_0\ a\ 1q_1$	–	aba	shift $a, 1$
$\varepsilon q_0\ a\ 1q_1\ a\ 11q_1$	–	ba	shift $a, 1$
$\varepsilon q_0\ a\ 1q_1\ \boxed{0}_1\ 1q_3$	$11\boxed{0}\gamma_1$	ba	reduce γ_1
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon q_2$	$1\boxed{1}\beta_1$ $\vert$ $11\boxed{0}\gamma_1$	ba	reduce β_1
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon q_2\ b\ 1^+q_6$	$1\boxed{1}\beta_1$ $\vert$ $11\boxed{0}\gamma_1$	a	shift $b, 1^+$
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon q_2\ \boxed{0}_2\ 1^+q_4$	$1\boxed{1}\beta_1$ $\vert$ $11\boxed{0}\gamma_2$	a	reduce γ_2
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon q_2\ \boxed{0}_2\ 1^+q_4\ a\ 1^+q_5$	$1\boxed{1}\beta_1$ $\vert$ $11\boxed{0}\gamma_2$	ε	shift a, ε
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon q_2\ \boxed{1}_2\ \varepsilon q_7$	$1\boxed{1}\beta_2$	ε	reduce β_2
$\varepsilon q_0\ \boxed{2}_1\ \varepsilon q_8$	$\varepsilon\boxed{2}\alpha_1$	ε	reduce α_1

Figure 15. Sample LR parser run with $w = aaba$.

The way the different parsing operations can lead to new configurations can be described by a corresponding parsing schema, i.e., a set of deduction rules and a goal item specification. This is

given in the following definition (the use of set union $C_1 \cup C_2$ in these rules assumes $C_1 \cap C_2 = \emptyset$) (comparing the configurations of a LCFRS-TA to the configurations of our parser, it becomes clear that one problem of the TA is that, because of possible call-predict loops, we can have infinitely many configurations in a TA. [23] avoids this problem by a dynamic programming implementation of TAs. In our automata, this problem does not occur because of the regular expressions we use as addresses).

Definition 11 (LR Parsing: Deduction Rules). *Let $G = \langle N, T, P, S\rangle$ be a LCFRS, $\langle action, goto\rangle$ its LR parse table, and q_0 and q_{acc} as introduced above.*

The set of possible configurations for the LR parser is then defined by the following deduction rules:

- *Initial configuration:* $\dfrac{}{\langle \varepsilon q_0, \emptyset, 0\rangle}$
- *Shift:*

$$\frac{\langle \Gamma pq, C, j\rangle}{\langle \Gamma pqw_{j+1}pp'q', C, j+1\rangle}\ action(q, w_{j+1}) = s(p', q')$$

Reduce 1:

$$\frac{\langle \Gamma p_k q_k X_k \dots p_1 q_1 X_1 pq, C, j\rangle}{\langle \Gamma p_k q_k \boxed{l}_1 p_k p'q', f_\triangleleft(C \cup \{p\boxed{l}\gamma_1\} \cup C_\triangleleft), j\rangle} \quad \begin{array}{l} r(\gamma,1) \in action(q,-), k = |lhs(\gamma,0)|, \\ goto(q_k, lhs(\gamma)_1) = \langle p', q'\rangle, \boxed{l} \notin \mathcal{N}_C \end{array}$$

Reduce 2:

$$\frac{\langle \Gamma p_k q_k X_k \dots p_1 q_1 X_1 pq, C \cup \{p''\boxed{l}\gamma_{i-1}\}, j\rangle}{\langle \Gamma p_k : q_k \boxed{l}_i p_k p'q', f_\triangleleft(C \cup \{p_{new}\boxed{l}\gamma_i\} \cup C_\triangleleft), j\rangle} \quad \begin{array}{l} r(\gamma,i) \in action(q,-), 1 < i, k = |lhs(\gamma, i-1)|, \\ A = lhs(\gamma), goto(q_k, A_i) = \langle p', q'\rangle, \\ L(p_{new}) = L(p'') \cap L(p_k p') \neq \emptyset \end{array}$$

where in both reduce operations,

1. $C_\triangleleft = \{\boxed{l} \lhd_d \boxed{m} \mid \boxed{m}_n \in \{X_1, \dots, X_k\}$ *for some* $n \geq 1$, $\boxed{m}_n$ *stands for the nth component of the dth righthand side element of* γ *for some* $n \geq 1$, *and* $\boxed{l} \lhd_d \boxed{m} \notin C\}$.
2. $f_\triangleleft(C)$ *is defined as the closure of* C *under application of the following rules:*
   (a) *Compilation of address information:*
   $$\frac{C \cup \{p_1\boxed{i}x, p_2\boxed{j}y, \boxed{i} \lhd_d \boxed{j}\}}{C \cup \{p'_1\boxed{i}x, p'_1 d\boxed{j}y, \boxed{i} \lhd_d \boxed{j}\}}\ L(p'_1 d) = L(p_1 d) \cap L(p_2)$$
   (b) *Removal of daughters of completed rules:*
   $$\frac{C \cup \{p\boxed{i_1}\gamma_k, pd\boxed{i_2}x, \boxed{i_1} \lhd_d \boxed{i_2}\}}{C \cup \{p\boxed{i_1}\gamma_k\}}\ k = dim(lhs(\gamma)), 1 \leq d \leq |rhs(\gamma)|$$
- *Goal items:* $\langle \varepsilon q_0\ \boxed{i}_1\ \varepsilon q_{acc}, \{\varepsilon\boxed{i}\alpha_1\}, |w|\rangle$ *for some* $\boxed{i} \in \mathcal{N}$ *and* $\alpha \in P$ *with* $lhs(\alpha) = S$.

4.4. An Example with Crossing Dependencies

The running example we have considered so far generates only a context-free language. It was chosen for its simplicity and for the fact that it has a non-terminal with a fan-out > 1 and a left recursion in one of the components, which requires the use of regular expressions. Let us now consider the LCFRS in Figure 16, an LCFRS for a language that is not context-free with a cross-serial dependency structure. We have again left recursions, this time in all components of the rules β_a and β_b. And the rank of the grammar is 2, not 1 as in the running example we have considered so far. This gives us more interesting derivation trees.

$$\begin{array}{ll} \alpha : S(xyzu) \to A(x,z)B(y,u) & \\ \beta_a : A(xa, ya) \to A(x,y) & \gamma_a : A(a,a) \to \varepsilon \\ \beta_b : B(xb, yb) \to B(x,y) & \gamma_b : B(b,b) \to \varepsilon \end{array}$$

Figure 16. A LCFRS for $\{a^n b^m a^n b^m \mid n, m \geq 1\}$.

The LR automaton for this grammar is given in Figure 17. Since we have a left recursion in both components of the β rules, the predict and resume closures of the items that start the traversal of these components contain each an infinite number of addresses in the derivation tree, encoded in a regular expression (1^+ or 2^+). This automaton leads then to the parse table in Figure 18. As can be seen, the grammar does not allow for a deterministic parsing since, when being in q_0 for instance and having completed the first component of an A, this can lead to q_1 or q_5, consequently there are several entries in the corresponding *goto*-field.

Figure 19 gives a sample parse (only successful configurations) for the input *aabaab*. When reducing the components of the A yields (nodes $\boxed{0}$ and $\boxed{1}$) and of the components of the B yields ($\boxed{2}$), we do not know the exact address of these elements in the derivation tree. Only in the last reduce step, it becomes clear that the $\boxed{1}$ and $\boxed{2}$ nodes are immediate daughters of the root of the derivation tree, and their addresses are resolved to 1 and 2 respectively.

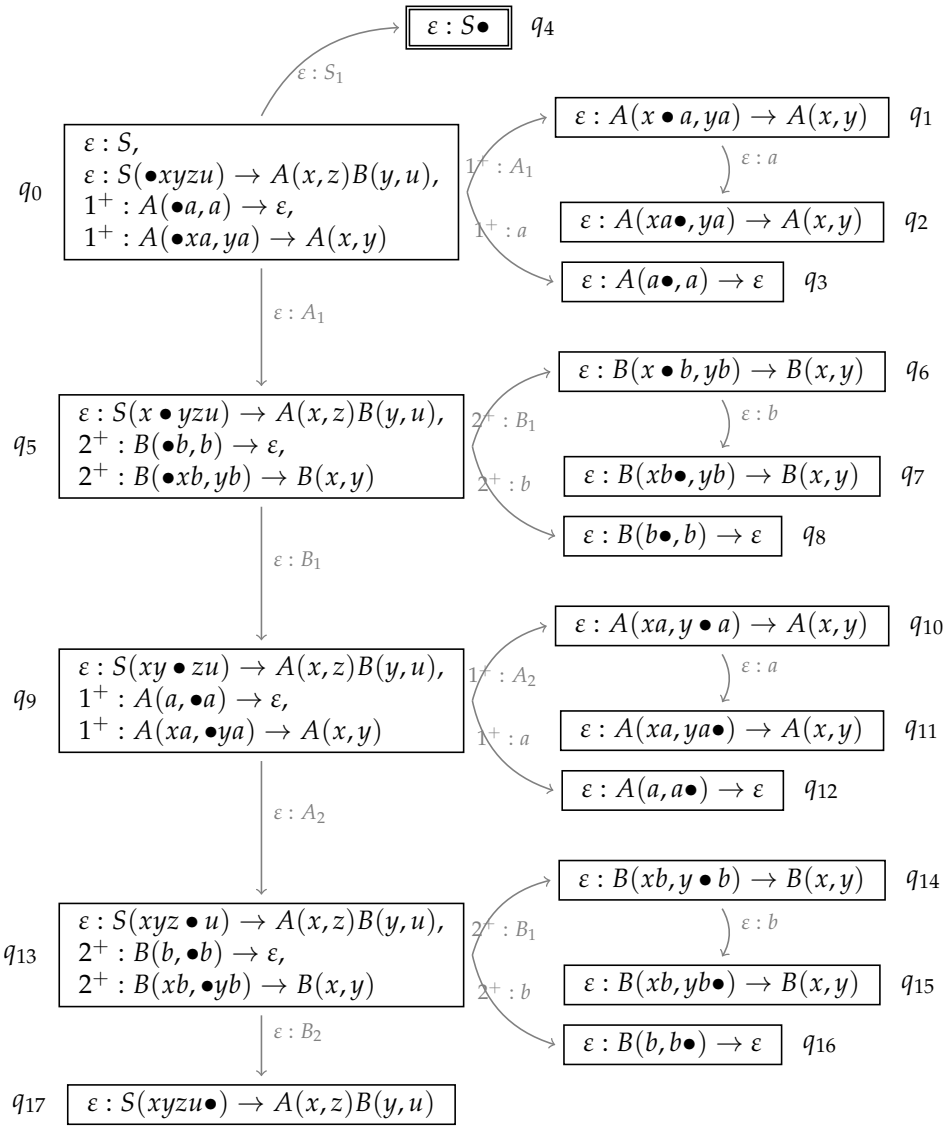


Figure 17. LR automaton for $\{a^n b^m a^n b^m \mid n, m \geq 1\}$.

	a	b		A_1	A_2	B_1	B_2	S_1
0	$s(1^+, 3)$			$\langle 1^+, 1\rangle$, $\langle \varepsilon, 5\rangle$				$\langle \varepsilon, 4\rangle$
1	$s(\varepsilon, 2)$							
2			$r(\beta_a, 1)$					
3			$r(\gamma_a, 1)$					
4			acc					
5		$s(2^+, 8)$				$\langle 2^+, 6\rangle$, $\langle \varepsilon, 9\rangle$		
6		$s(\varepsilon, 7)$						
7			$r(\beta_b, 1)$					
8			$r(\gamma_b, 1)$					
9	$s(1^+, 12)$				$\langle 1^+, 10\rangle$, $\langle \varepsilon, 13\rangle$			
10	$s(\varepsilon, 11)$							
11			$r(\beta_a, 2)$					
12			$r(\gamma_a, 2)$					
13		$s(2^+, 16)$					$\langle 2^+, 14\rangle$, $\langle \varepsilon, 17\rangle$	
14		$s(\varepsilon, 15)$						
15			$r(\beta_b, 2)$					
16			$r(\gamma_b, 2)$					
17			$r(\alpha_1, 1)$					

Figure 18. Parse table obtained from the LR automaton for $\{a^n b^m a^n b^m \mid n, m \geq 1\}$.

Stack	**Constraints on Derivation Tree**	**Remaining Input**	**Operation**
εq_0	–	*aabaab*	initial state
$\varepsilon q_0\ a\ 1^+ q_3$	–	*abaab*	shift $a, 1^+$
$\varepsilon q_0\ \boxed{0}_1\ 1^+ q_1$	$1^+\boxed{0}\gamma_{a1}$	*abaab*	reduce γ_{a1}
$\varepsilon q_0\ \boxed{0}_1\ 1^+ q_1\ a\ 1^+ q_2$	$1^+\boxed{0}\gamma_{a1}$	*baab*	shift a, ε
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5$	$1^+\boxed{1}\beta_{a1}$ | $1^+1\boxed{0}\gamma_{a1}$	*baab*	reduce β_{a1}
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5\ b\ 2^+ q_8$	$1^+\boxed{1}\beta_{a1}$ | $1^+1\boxed{0}\gamma_{a1}$	*aab*	shift $b, 2^+$
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5\ \boxed{2}_1\ \varepsilon\ q_9$	$1^+\boxed{1}\beta_{a1}\quad 2^+\boxed{2}\gamma_{b1}$ | $1^+1\boxed{0}\gamma_{a1}$	*aab*	reduce γ_{b1}
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5\ \boxed{2}_1\ \varepsilon\ q_9\ a\ 1^+ q_{12}$	$1^+\boxed{1}\beta_{a1}\quad 2^+\boxed{2}\gamma_{b1}$ | $1^+1\boxed{0}\gamma_{a1}$	*ab*	shift $a, 1^+$
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5\ \boxed{2}_1\ \varepsilon\ q_9\ \boxed{0}_2\ 1^+ q_{10}$	$1^+\boxed{1}\beta_{a1}\quad 2^+\boxed{2}\gamma_{b1}$ | $1^+1\boxed{0}\gamma_{a2}$	*ab*	reduce γ_{a2}
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5\ \boxed{2}_1\ \varepsilon\ q_9\ \boxed{0}_2\ 1^+ q_{10}\ a\ 1^+ q_{11}$	$1^+\boxed{1}\beta_{a1}\quad 2^+\boxed{2}\gamma_{b1}$ | $1^+1\boxed{0}\gamma_{a2}$	*b*	shift a, ε
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5\ \boxed{2}_1\ \varepsilon\ q_9\ \boxed{1}_2\ \varepsilon\ q_{13}$	$1^+\boxed{1}\beta_{a2}\quad 2^+\boxed{2}\gamma_{b1}$	*b*	reduce β_{a2}
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5\ \boxed{2}_1\ \varepsilon\ q_9\ \boxed{1}_2\ \varepsilon\ q_{13}\ b\ 2^+ q_{16}$	$1^+\boxed{1}\beta_{a2}\quad 2^+\boxed{2}\gamma_{b1}$	ε	shift $b, 2^+$
$\varepsilon q_0\ \boxed{1}_1\ \varepsilon\ q_5\ \boxed{2}_1\ \varepsilon\ q_9\ \boxed{1}_2\ \varepsilon\ q_{13}\ \boxed{2}_2\ \varepsilon\ q_{17}$	$1^+\boxed{1}\beta_{a2}\quad 2^+\boxed{2}\gamma_{b2}$	ε	reduce γ_{b2}
$\varepsilon q_0\ \boxed{3}_1\ \varepsilon\ q_4$	$\varepsilon\boxed{3}\alpha_1$	ε	reduce α_1

Figure 19. Sample LR parser run with $w = aabaab$ (only successful configurations).

As mentioned, this grammar does not allow for a deterministic parsing. With one additional lookahead, the grammar would become deterministic (the ε-possibility is to be taken when being at the end of one of the a^n or b^m substrings of the input).

4.5. Deterministic Parsing

As mentioned, with one lookahead, LR parsing of the LCFRS in the previous example would become deterministic.

In general, when doing natural language processing, the grammars will always be ambiguous and a deterministic parsing is not possible. Instead, some way to share structure in the style of *generalized LR* [29] parsing has to be found. In other applications, however, we might have LCFRSs that allow for a deterministic parsing, similar to LR(k) parsing as in [27].

Non-determinism with LCFRS can arise not only from shift-reduce or reduce-reduce conflicts as in CFGs but also from a conflict between different *goto* transitions that have the same component non-terminal but different node addresses (see the multiple entries in the *goto* table of our example). We therefore need lookaheads not only for reduce (and, if $k > 1$, shift) but also for the *goto* entries.

We now define, similar to the CFG case [27], sets of *First* symbols for dotted productions, which contain all terminals that are reachable as first element of the yield, starting from the dotted production. For every $\gamma_{i,j}$, let us define $First(\gamma, i, j)$ as follows. (Remember that $lhs(\gamma, i, j)$ gives the jth symbol of the ith component in the lefthand side of γ, i.e., the element following the dot.)

1. $\varepsilon \in First(\gamma, i, j)$ if $lhs(\gamma, i, j)$ is not defined (i.e, the dot is at the end of the ith component);
2. $a \in First(\gamma, i, j)$ if $lhs(\gamma, i, j) = a$ (i.e., the dot precedes a);
3. $a \in First(\gamma, i, j)$ if $lhs(\gamma, i, j) = x \in V$, x is the lth component in a righthand side element of γ with non-terminal B, and there is a $\gamma_B \in P$ with $lhs(\gamma_B) = B$ such that $a \in First(\gamma_B, l, 0)$.
4. Nothing else is in $First(\gamma, i, j)$.

Besides this, we also need the notion of *Follow*. *Follow* sets are defined for all component nonterminals A_i ($A \in N, 1 \leq i \leq dim(A)$) of our LCFRS. They contain the terminals that can immediately follow the yield of such a component nonterminal.

1. $\$ \in Follow(S_1)$ where S is the start symbol and $\$ \notin T$ is a new symbol marking the end of the input.
2. For every production γ with $x \in V$ being the jth element of the ith lefthand side component (indices start with $i = 0$ and $j = 0$) and x being the lth element of the yield of a nonterminal B in the righthand side (also starting with $l = 0$), we have $First(\gamma_{i,j+1}) \subseteq Follow(B_{l+1})$, provided $First(\gamma_{i,j+1})$ is defined.
3. For every production γ with $lhs(\gamma) = A$ and with $x \in V$ being the last element of the ith lefthand side component (indices start with $i = 0$) and the lth element of the yield of a nonterminal B in the righthand side (also starting with $l = 0$), we have $Follow(A_{i+1}) \subseteq Follow(B_{l+1})$.
4. Nothing else is in the sets $Follow(A_i)$ of our component nonterminals.

For $LR(1)$ parsing, i.e., LR parsing with 1 lookahead, we replace the items in our states with triples consisting of the adress (regular expression), the dotted production $\gamma_{i,j}$ and the possible next terminal symbols. To this end, we define sets $Next(\gamma, i, j)$ that are the *First* sets for dots that are not at the end of components and otherwise the *Follow*-sets of the component non-terminal:

$Next(\gamma, i, j) = First(\gamma, i, j)$ if $\gamma_{i,j+1}$ exists and $Next(\gamma, i, j) = Follow(lhs(\gamma)_{i+1})$ otherwise.

A simple way to extend our automaton to $LR(1)$ is by extending the items in the states as follows: Every item with dotted production $\gamma_{i,j}$ is equipped with the lookahead set $Next(\gamma, i, j)$. Furthermore, transitions are also labeled with triples where the new third element that is added is the terminal in case of a shift transition and, in case of a transition that moves the dot over a non-terminal, the *Next* set of the result of moving the dot according to the transition.

Consider the sample automaton in Figure 17. q_0 has two outgoing A_1 edges with different addresses. One (derivation tree node address ε) arises from moving the dot over the variable x in $S(\bullet xyzu) \rightarrow A(x,z)B(y,u)$. We have $Next(S(x \bullet yzu) \rightarrow A(x,z)B(y,u)) = First(B(\bullet b, b) \rightarrow \varepsilon) \cup First(B(\bullet xb, yb) \rightarrow B(x,y)) = \{b\}$, therefore this transition would be relabeled $\varepsilon : A_1 : \{b\}$. The other transition (address 1^+) leads to $A(x \bullet a, ya) \rightarrow A(x,y)$, the *First* set is $\{a\}$ and therefore this transition would be relabeled $1^+ : A_1 : \{a\}$.

Consequently, the *goto* part of the parse table, that would now have entries for every combination of component non-terminal and lookahead symbol, would have at most one entry per field. Hence, LR(1) parsing of our sample grammar is deterministic.

Concerning the reduce steps in the parse table, we would also have a lookahead. Here we can use the elements from the *Follow* set of the reduced components as lookaheads. The $r(\beta_a, 1)$ reduce entry for state q_2 for instance would have the lookaheads from $Follow(A_1) = \{a, b\}$.

Instead of this simple LR technique one can also compute the lookahead sets while constructing the automaton, in the style of canonical LR parsing for CFGs. This would allow for more precise predictions of next moves.

There are LCFRSs that do not allow for a deterministic *LR* parsing, no matter how long the length k of the lookahead strings (the non-deterministic CFGs, which are 1-LCFRSs are such grammars). But a precise characterization of the class of deterministic LCFRSs and an investigation of whether, for every $LR(k)$ LCFRS, we can find a weakly equivalent $LR(0)$ LCFRS (as is the case for CFG), is left for further research.

4.6. Soundness and Completeness of the LR Algorithm

In order to show soundness and completeness of the LR algorithm, we will relate it to the LCFRS-TA for the same grammar. We will slightly modify the deduction rules for TA by assuming that (1) call and predict steps are combined and the initial configuration is omitted, i.e., we start with the result of the first predict steps; (2) we omit the publish steps; and (3) we don't delete threads of finished rules. The modified rules are given Figure 20. Concerning the LR automaton, we replace the regular expressions in our parsing configurations with concrete addresses. In principal we can then have infinitely many configurations in cases of left recursions (just like in the TA case), i.e., in cases where a shift or goto transition comes with a regular expression denoting an infinite language. Furthermore, we do not delete daughters of completed rules in the constraint set either, i.e., we assume that the rule 2(b) in Definition 11 concerning the application of $f_{\lhd}$ to the constraint set does not apply. Clearly, this does not change the languages recognized by these automata, it just increases the set of possible configurations. Since we use no longer regular expressions but single addresses, the deduction rules of the LR parser get simpler. Immediate dominance constraints are no longer needed since they follow from the addresses. The relevant deduction rules are given in Figure 21.

Initial configurations: $\dfrac{}{\langle 0, \varepsilon, \{\varepsilon : \alpha_{0,0}\}\rangle}$ $lhs(\alpha) = S$

Predict: $\dfrac{\langle l, p, \mathcal{S} \cup \{p : \gamma_{k,i}\}\rangle}{\langle l, pj, \mathcal{S} \cup \{p : \gamma_{k,i}\} \cup \{pj : \gamma'_{0,0}\}\rangle}$ $\gamma_{k,i} \rightarrow [\gamma_{k,i}]A \in \Theta, A \in N, \delta(\gamma_{k,i}) = j+1, A \rightarrow \gamma'_{1,0} \in \Theta$

Scan: $\dfrac{\langle j, p, \mathcal{S} \cup \{p : \gamma_{k,i}\}\rangle}{\langle j+1, p, \mathcal{S} \cup \{p : \gamma_{k,i+1}\}\rangle}$ $\gamma_{k,i} \overset{w_{j+1}}{\rightarrow} \gamma_{k,i+1} \in \Theta$

Suspend: $\dfrac{\langle i, pj, \mathcal{S} \cup \{p : \gamma_{k,i}, pj : \beta_{l,m}\}\rangle}{\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,i+1}, pj : \beta_{l,m}\}\rangle}$ $[\gamma_{k,i}]\beta_{l,m} \rightarrow \gamma_{k,i+1}[\beta_{l,m}] \in \Theta$ or $\beta_{l,m} \rightarrow ret, [\gamma_{k,i}]ret \rightarrow \gamma_{k,i+1} \in \Theta$

Resume: $\dfrac{\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,i}, p\delta(\gamma_{k,i}) : \beta_{l,j}\}\rangle}{\langle i, p\delta(\gamma_{k,i}), \mathcal{S} \cup \{p : \gamma_{k,i}, p\delta(\gamma_{k,i}) : \beta_{l+1,0}\}\rangle}$ $\gamma_{k,i}[\beta_{l,j}] \rightarrow [\gamma_{k,i}]\beta_{l+1,0} \in \Theta$

Goal item: $\langle |w|, \varepsilon, \{\varepsilon : \alpha_{0,|lhs(\alpha,1)|}\rangle$ with $lhs(\alpha) = S$.

Figure 20. Revised deduction rules for TA configurations.

Initial configuration: $\dfrac{}{\langle \varepsilon q_0, \emptyset, 0\rangle}$ Shift: $\dfrac{\langle \Gamma pq, C, j\rangle}{\langle \Gamma pq w_{j+1} pp'q', C, j+1\rangle}$ $\begin{array}{l} action(q, w_{j+1}) \\ = s(r, q'), p' \in L(r) \end{array}$

Reduce 1: $\dfrac{\langle \Gamma p_k q_k X_k \dots p_1 q_1 X_1 pq, C, j\rangle}{\langle \Gamma p_k q_k \boxed{l}_1 p_k p'q', C \cup \{p\boxed{l}\gamma_1\}, j\rangle}$ $\begin{array}{l} r(\gamma,1) \in action(q,-), k = |lhs(\gamma,0)|, \\ goto(q_k, lhs(\gamma)_1) = \langle r, q'\rangle, p' \in L(r), \boxed{l} \notin \mathcal{N}_C \end{array}$

Reduce 2: $\dfrac{\langle \Gamma p_k q_k X_k \dots p_1 q_1 X_1 pq, C \cup \{p\boxed{l}\gamma_{i-1}\}, j\rangle}{\langle \Gamma p_k q_k \boxed{l}_i p_k p'q', C \cup \{p\boxed{l}\gamma_i\}, j\rangle}$ $\begin{array}{l} r(\gamma,i) \in action(q,-), \\ 1 < i, k = |lhs(\gamma, i-1)|, \\ goto(q_k, lhs(\gamma)_i) = \langle r, q'\rangle, p' \in L(r) \end{array}$

where for both reduce steps, any component pointer $\boxed{i}_m$ among the $X_1, \dots, X_k$ has an address pj for some $j \in \{1, \dots, |rhs(\gamma)|\}$.

Goal items: $\langle \varepsilon q_0 \, \boxed{i}_1 \, \varepsilon q_{acc}, \{\varepsilon\boxed{i}\alpha_1\}, |w|\rangle$ for some $\boxed{i} \in \mathcal{N}$ and $\alpha \in P$ with $lhs(\alpha) = S$.

Figure 21. LR deduction rules without regular expressions.

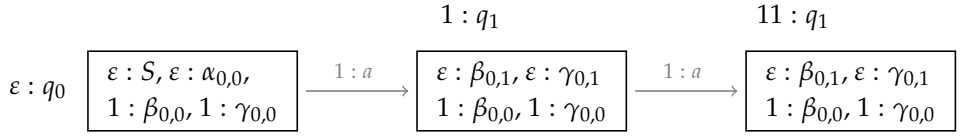
Figure 22 shows the correspondence between TA configurations and LR configurations for our sample derivation of *aaba*.

TA Configurations	**LR Configurations**
$\langle 0, \varepsilon, \{\varepsilon : \alpha_{0,0}\}\rangle$ $\langle 0, 1, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,0}\}\rangle$ $\langle 0, 1, \{\varepsilon : \alpha_{0,0}, 1 : \gamma_{0,0}\}\rangle$	$\langle \varepsilon q_0, \emptyset, 0\rangle$
$\langle 1, 1, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}\}\rangle$ $\langle 1, 1, \{\varepsilon : \alpha_{0,0}, 1 : \gamma_{0,1}\}\rangle$ $\langle 1, 11, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}, 11 : \beta_{0,0}\}\rangle$ $\langle 1, 11, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}, 11 : \gamma_{0,0}\}\rangle$	$\langle \varepsilon q_0 \; a \; 1q_1, \emptyset, 1\rangle$
$\langle 2, 11, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}, 11 : \beta_{0,1}\}\rangle$ $\langle 2, 11, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}, 11 : \gamma_{0,1}\}\rangle$ $\langle 2, 111, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}, 11 : \beta_{0,1}, 111 : \beta_{0,0}\}\rangle$ $\langle 2, 111, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}, 11 : \beta_{0,1}, 111 : \gamma_{0,0}\}\rangle$	$\langle \varepsilon q_0 \; a \; 1q_1 \; a \; 11q_1, \emptyset, 2\rangle$
$\langle 2, 1, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,2}, 11 : \gamma_{0,1}\}\rangle$	$\langle \varepsilon q_0 \; a \; 1q_1 \; \boxed{0}_1 \; 1q_3, \{11\boxed{0}\gamma_1\}, 2\rangle$
$\langle 2, \varepsilon, \{\varepsilon : \alpha_{0,1}, 1 : \beta_{0,2}, 11 : \gamma_{0,1}\}\rangle$ $\langle 2, 1, \{\varepsilon : \alpha_{0,1}, 1 : \beta_{1,0}, 11 : \gamma_{0,1}\}\rangle$ $\langle 2, 11, \{\varepsilon : \alpha_{0,1}, 1 : \beta_{1,0}, 11 : \gamma_{1,0}\}\rangle$	$\langle \varepsilon q_0 \; \boxed{1}_1 \; \varepsilon q_2, \{11\boxed{0}\gamma_1, 1\boxed{1}\beta_1\}, 2\rangle$
$\langle 3, 11, \{\varepsilon : \alpha_{0,1}, 1 : \beta_{1,0}, 11 : \gamma_{1,1}\}\rangle$	$\langle \varepsilon q_0 \; \boxed{1}_1 \; \varepsilon q_2 \; b \; 11q_6, \{11\boxed{0}\gamma_1, 1\boxed{1}\beta_1\}, 3\rangle$
$\langle 3, 1, \{\varepsilon : \alpha_{0,1}, 1 : \beta_{1,1}, 11 : \gamma_{1,1}\}\rangle$	$\langle \varepsilon q_0 \; \boxed{1}_1 \; \varepsilon q_2 \; \boxed{0}_2 \; 1q_4, \{11\boxed{0}\gamma_2, 1\boxed{1}\beta_1\}, 3\rangle$
$\langle 4, 1, \{\varepsilon : \alpha_{0,1}, 1 : \beta_{1,2}, 11 : \gamma_{1,1}\}\rangle$	$\langle \varepsilon q_0 \; \boxed{1}_1 \; \varepsilon q_2 \; \boxed{0}_2 \; 1q_4 \; a \; 1q_5, \{11\boxed{0}\gamma_2, 1\boxed{1}\beta_1\}, 4\rangle$
$\langle 4, \varepsilon, \{\varepsilon : \alpha_{0,2}, 1 : \beta_{1,2}, 11 : \gamma_{1,1}\}\rangle$	$\langle \varepsilon q_0 \; \boxed{1}_1 \; \varepsilon q_2 \; \boxed{1}_2 \; \varepsilon q_7, \{11\boxed{0}\gamma_2, 1\boxed{1}\beta_2\}, 4\rangle$
	$\langle \varepsilon q_0 \; \boxed{2}_1 \; \varepsilon q_8, \{11\boxed{0}\gamma_2, 1\boxed{1}\beta_2, \varepsilon\boxed{2}\alpha_1\}, 4\rangle$

Figure 22. TA configurations and corresponding LR configurations for our sample parse of $w = aaba$.

We now define a function f_{TA} that gives for every LR configuration the set of corresponding TA configurations. The idea is the following: An LR configuration describes a path in our LR automaton. This path yields several possible thread contents.

Before giving the formal definition, let us consider two examples from Figure 22 for illustration. Take as a first example the LR configuration $\langle \varepsilon q_0 \; a \; 1q_1 \; a \; 11q_1, \emptyset, 2\rangle$. The corresponding path in our automaton is



Each thread store in the corresponding TA configuration corresponds to a possible path through these states.

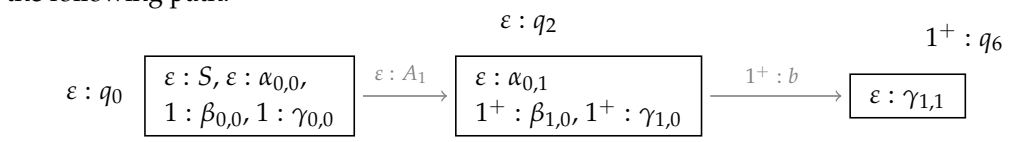
- For every ε-element in the last state of this path (here $\varepsilon : \beta_{0,1}$ or $\varepsilon : \gamma_{0,1}$), we compute the TA configurations $\langle 2, 11, \mathcal{S}_\beta \rangle$ and $\langle 2, 11, \mathcal{S}_\gamma \rangle$ respectively for these elements:

  Let us consider first the case of $\mathcal{S}_\beta$. We first define a set $\mathcal{S}'_\beta$, starting with $\mathcal{S}'_\beta = \{11 : \beta_{0,1}\}$:

  - Since $11 : \beta_{0,1}$ was obtained via the a-transition, from $1 : \beta_{0,0}$ in $1 : q_1$, we add $11 : \beta_{0,0}$ to $\mathcal{S}'_\beta$.
  - $11 : \beta_{0,0}$ in turn is in the second state on the path since we obtained it via prediction from $1 : \beta_{0,1}$, which is added to $\mathcal{S}'_\beta$ as well.
  - Again, $1 : \beta_{0,1}$ arose out of a shift of a from $1 : \beta_{0,0}$, which is added as well and then, because of predict in q_0, we add $\epsilon : \alpha_{0,0}$. This gives $\mathcal{S}'_\beta = \{11 : \beta_{0,1}, 11 : \beta_{0,0}, 1 : \beta_{0,1}, 1 : \beta_{0,0}, \epsilon : \alpha_{0,0}\}$.
  - We then remove all items where there is an item with the same address and rule in the set but with a lefthand side index that is more advanced. This yields $\mathcal{S}'_\beta = \{11 : \beta_{0,1}, 1 : \beta_{0,1}, \epsilon : \alpha_{0,0}\}$, which is the thread set $\mathcal{S}_\beta$ we were looking for.
  - In a similar way, for the γ case, we obtain $\mathcal{S}_\gamma = \{11 : \gamma_{0,1}, 1 : \beta_{0,1}, \epsilon : \alpha_{0,0}\}$

- In addition, we add all configurations that we can reach from the ones constructed so far via predict or resume. These are reflected in corresponding items in the last state of our path. In our sample case, $1 : \beta_{0,0}$ and $1 : \gamma_{0,0}$ are both elements of the last state and arose directly out of $\epsilon : \beta_{0,1}$ via predict. Therefore, we add these to the thread store with $11 : \beta_{0,1}$ being the active thread, which leads to $\langle 2, 111, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}, 11 : \beta_{0,1}, 111 : \beta_{0,0}\}\rangle$ and $\langle 2, 111, \{\varepsilon : \alpha_{0,0}, 1 : \beta_{0,1}, 11 : \beta_{0,1}, 111 : \gamma_{0,0}\}\rangle$.
- These are all configurations in $f_{TA}(\langle \varepsilon q_0\ a\ 1q_1\ a\ 11q_1, \emptyset, 2\rangle)$.

Now consider as a second example $\langle \varepsilon : q_0\ \boxed{1}_1\ \varepsilon : q_2\ b\ 11 : q_6, \{11\boxed{0}\gamma_1, 1\boxed{1}\beta_1\}, 3\rangle$, corresponding to the following path:



Again, we have to go back from the ε-element in the last state in order to construct the thread store. The last element gives us $11 : \gamma_{1,1}$.

Consequently, the configuration is $\langle 3, 11, \mathcal{S}\rangle$ and we construct $\mathcal{S}$ starting with $\mathcal{S}' = \{11 : \gamma_{1,1}\}$. Following the shift transition back adds $11 : \gamma_{1,0}$ to $\mathcal{S}'$. $11 : \gamma_{1,0}$, in turn, arose from a resume applied to $1 : \beta_{1,0}$, which is therefore added as well. $1 : \beta_{1,0}$ was predicted from $\varepsilon : \alpha_{0,1}$, which is added as well. Now we follow the A_1-transition backwards and therefore add $\varepsilon : \alpha_{0,0}$. Finally, since for a specific address and rule, we keep only the most advanced position, we obtain $\langle 3, 11, \{\varepsilon : \alpha_{0,1}, 1 : \beta_{1,0}, 11 : \gamma_{1,1}\}\rangle$.

In order to define the corresponding TA-sets, we annotate the LR automaton with pointers that show for each element c in any of the states, which other element c' has lead to c, either because of moving the dot over a terminal or a component variable or because of a predict/resume closure operation. In the latter case, the pointer is labeled with the daughter index of the predict/resume operation. For this annotation, we assume that instead of pairs $r : c$ of regular expression and comutation point, we use onyl pairs $p : c$ of address and computation point. This means of course, that we have potentially infinite states and potentially infinite predecessor paths through a state and from one state to another.

Definition 12 (Predecessor Annotation). *Let M be an LR-automaton with state set Q constructed from an LCFRS G. $\mathcal{Q} = \{\langle x, q\rangle \mid q \in Q, x \in q\}$.*

We define the predecessor annotation *of M as a labeled relation Predec $\subset \mathcal{Q} \times \mathcal{Q} \times (\{1, \ldots, rank(G)\}^* \cup \{-\})$ such that for every $q \in Q$ and every $p : c \in q$:*

- *If there is a* $p' : c' \in q$ *such that* $p = p'i$ *and* $pi : c$ *can be deduced from* $p : c'$ *by the predict/resume rule from Definition 7,* $\langle\langle p : c, q\rangle, \langle p' : c', q\rangle, i\rangle \in Predec$.
- *If* $p = \varepsilon$ *and there is a* $q' \in Q$ *and a* $p' : c' \in q'$ *such that* $\varepsilon : c \in read(\{p' : c'\}, x, p')$ *for some* $x \in T \cup \{A_i \mid A \in N, 1 \leq i \leq dim(A)\}$, *then* $\langle\langle p : c, q\rangle, \langle p' : c', q'\rangle, -\rangle \in Predec$.

Nothing else is in Predec.

Now we can define the set of possible TA stores for a given LR-configuration.

For every set of constraints C, we define $Nodes(C)$ as the set of pointers in C. Furthermore, for every pointer $\boxed{i} \in Nodes(C)$, we define $addr(\boxed{i}, C)$ as the address associated with $\boxed{i}$, and $label(\boxed{i}, C)$ as its label β_j.

For a given LR configuration, we compute the corresponding TA configurations via deduction rules. The items in these rules have the form $[\mathcal{S}, \langle\Gamma, C\rangle, i, p, p_2 : c]$ with $\mathcal{S}$ a thread store, $\langle\Gamma, C\rangle$ a LR configuration, i the index in the input, p an address and $p_2 : c$ an element of the topmost state in Γ. We start with an empty thread store and our LR configuration. The deduction rules then either move backwards inside the topmost Γ state or remove elements from Γ while adding corresponding threads to the thread store. In the end, we will have an empty stack Γ and then, the thread store $\mathcal{S}$ together with the index i and the address p of the current thread gives us one of the TA configurations corresponding to the LR configuration we started with. p stays constant throughout the entire computation. p_1 represents the address of the daughter that has lead to moving into the current state.

Definition 13. *Let* G *be an LCFRS,* M_G *its LR automaton. For every input* $w \in T^*$ *and every configuration* $\langle\Gamma, C, i\rangle$ *that can be deduced via* M_G *for the input* w, *we define* $f_{TA}(\langle\Gamma, C, i\rangle) = \{\langle i, p, \mathcal{S}\rangle \mid [\mathcal{S}, \langle\varepsilon, \emptyset\rangle, i, p, -, -]$ *can be deduced from* $[\emptyset, \Gamma, C, i, -, -, -]$ *via the following deduction rules*$\}$.

- *Current thread:* $\dfrac{[\emptyset, \langle\Gamma p_1 q, C\rangle, i, -, -]}{[\{p_1 p_2 : c\}, \langle\Gamma p_1 q, C\rangle, i, p_1 p_2, p_2 : c]}\ p_2 : c \in q$
- *Predict/resume backwards:* $\dfrac{[\mathcal{S}, \langle\Gamma p_1 q, C\rangle, i, p, p_2 j : c]}{[\mathcal{S} \cup \{p_1 p_2 : c'\}, \langle\Gamma p_1 q, C\rangle, i, p, p_2 : c']}$ $\langle\langle p_2 j : c, q\rangle, \langle p_2 : c', q\rangle, j\rangle \in Predec$
- *Shift backwards:* $\dfrac{[\mathcal{S}, \langle\Gamma p_1 q' a\, p_1 p_2 q, C\rangle, i, p, \varepsilon : c]}{[\mathcal{S}, \langle\Gamma p_1 q', C\rangle, i, p, p_2 : c']}$ $a \in T$ and $\langle\langle \varepsilon : c, q\rangle, \langle p_2 : c', q'\rangle, -\rangle \in Predec$
- *Shift non-terminal backwards:* $\dfrac{[\mathcal{S}, \langle\Gamma p_1 q' \boxed{k}_j\, p_1 p_2 q, C\rangle, i, p, \varepsilon : c]}{[\mathcal{S}, \langle\Gamma p_1 q', C\rangle, i, p, p_2 : c']}$ $\exists n$ such that $p_1 p_2 n = addr(\boxed{k}, C)$ and $\langle\langle \varepsilon : c, q\rangle, \langle p_2 : c', q'\rangle, -\rangle \in Predec$
- *Empty stack:* $\dfrac{[\mathcal{S}, \langle\varepsilon q_0, C\rangle, i, p, \varepsilon : c]}{[\mathcal{S}, \langle\varepsilon, C\rangle, i, p, -]}$
- *Add completed components:* $\dfrac{[\mathcal{S}, \langle\varepsilon, C\rangle, i, p, -]}{[\mathcal{S} \cup \{p' : \beta_{i-1,|lhs(\beta,i)|}\}, \langle\varepsilon, C\rangle, i, p, -]}$ $p'\boxed{k}\beta_i \in C$, p' does not occur in $\mathcal{S}$

Current thread picks one element $p_2 : c$ from the topmost state q in Γ as the starting element. The address of the corresponding thread is a concatenation of the address p_1 preceding q on the stack and the address p_2 of the chosen state element. **Predict/resume backwards** applies whenever the address of the current state element is not ε, we then follow the *Predec* link and add a corresponding new thread to our thread store. **Shift backwards** follows a *Predec* link that corresonds to a reversed edge in the automaton labeled with a terminal. In this case, we do not change the thread store since we already have a thread for the rule application in question with a more advanced index. We only remove the shifted terminal and state from the LR stack and we split the address of the state and the element within it according to the *Predec* link. **Shift non-terminal backwards** is a similar move following a *Predec* link that corresonds to a reversed edge in the automaton labeled with a component non-terminal. These four rules serve to reduce the LR stack while constructing threads out of it. This process terminates with a single application of **Empty stack**. Once the stack is empty, we have to take care of the components described in C. For all of these, if no corresponding thread exists, wee add it by applying **Add completed component**.

Lemma 14. *Let $G = (N, T, V, P, S)$ be a LCFRS of rank m. Let $\langle N', T, S, ret, \delta, \Theta\rangle$ be the LCFRS-TA of G, and let M be the LR-automaton constructed for G with $\langle action, goto\rangle$ being the LR parse table one can read off M.*

For every input word $w \in T^*$, with $\mathcal{C}_{TA}$ being the set of TA configurations, $\mathcal{C}_{LR}$ the set of LR configurations arising from parsing w with the TA or the LR parser:

$$\mathcal{C}_{TA} = \bigcup_{c_{LR} \in \mathcal{C}_{LR}} f_{TA}(c_{LR})$$

Proof. Proof by induction over the deduction rules of the TA automaton and the LR parser:

- **Initial configurations**: $c_0 = \langle \varepsilon q_0, \emptyset, 0\rangle \in \mathcal{C}_{LR}$ can be added as initial configuration in the LR parser iff $\langle 0, \varepsilon, \{\varepsilon : \alpha_{0,0}\}\rangle$ for every $\alpha \in P$ with $lhs(\alpha) = S$ are initial configurations in the TA and, furthermore, for any configuration one can obtain from one of these initial TA configurations via applications of *Predict* it holds that there is a corresponding element in q_0. Conversely, except for $\langle \varepsilon : S\rangle \in q_0$, for every element in q_0, there is a corresponding configuration in the TA containing this element and all its predecessors as threads.

  Predict in the TA starting from any of the $\langle 0, \varepsilon, \{\varepsilon : \alpha_{0,0}\}\rangle$:

  $$\frac{\langle 0, p, \mathcal{S} \cup \{p : \gamma_{0,0}\}\rangle}{\langle 0, pj, \mathcal{S} \cup \{p : \gamma_{0,0}\} \cup \{pj : \gamma'_{0,0}\}\rangle} \quad \begin{array}{l}\gamma_{0,0} \to [\gamma_{0,0}]A \in \Theta, A \in N, \delta(\gamma_{0,0}) = j+1,\\ A \to \gamma'_{1,0} \in \Theta, \text{i.e., } A = lhs(\gamma')\end{array}$$

  Elements in q_0 are obtained by

  - Initial predict: $\dfrac{\varepsilon : S}{\varepsilon : \alpha_{0,0}}$ $lhs(\alpha) = S$
  - Predict closure on q_0 in the LR automaton: $\dfrac{p : \gamma_{0,0}}{pj : \gamma'_{0,0}}$ $\begin{array}{l}lhs(\gamma, 0, 0) = rhs(\gamma, j-1, 0),\\ rhs(\gamma, j) = lhs(\gamma')\end{array}$

    with a *Predec* link labeled j from the consequent item to the antecedent item.

  Consequently, we obtain that the set of TA configurations we get by using only the initial rule and *Predict* is exactly $f_{TA}(c_0)$, i.e., the union of all $f_{TA}(c)$ for LR configurations we can obtain by the initial rule.
- **Scan** and **Shift**: We now assume that we have sets $\mathcal{C}_{TA}$ of TA configurations and $\mathcal{C}_{LR}$ of LR configurations such that $\mathcal{C}_{TA} = \bigcup_{c_{LR} \in \mathcal{C}_{LR}} f_{TA}(c_{LR})$.

  To show: For any pair $c_{LR}, f_{TA}(c_{LR})$ contained in the two sets and for any terminal a and any address p: $c_{TA,1}, \ldots, c_{TA,k}$ are all the configurations in $f_{TA}(c_{LR})$ with current thread address p that allow for an application of *Scan* with a, the new configurations being $c'_{TA,1}, \ldots, c'_{TA,k}$ iff c_{LR} allows to derive a new configuration c'_{LR} via an application of *Shift* with the terminal a and address p, such that the following holds: $f_{TA}(c'_{LR}) = \bigcup_{i=1}^{k} \overline{c'_{TA,i}}$ where $\overline{c'_{TA}}$ is the closure of $\{c'_{TA}\}$ under applications of *Predict* and *Resume*.

  Let us repeat the operations *Scan* and *Shift*:

  Scan in the TA: $\dfrac{\langle j, p, \mathcal{S} \cup \{p : \gamma_{k,i}\}\rangle}{\langle j+1, p, \mathcal{S} \cup \{p : \gamma_{k,i+1}\}\rangle}$ $\gamma_{k,i} \overset{w_{j+1}}{\to} \gamma_{k,i+1} \in \Theta$

  Shift in the LR automaton: $\dfrac{\langle \Gamma p_1 q, C, j\rangle}{\langle \Gamma p_1 q w_{j+1} p_1 p_2 q', C, j+1\rangle}$ $\begin{array}{l}action(q, w_{j+1})\\ = s(r, q'), p_2 \in L(r)\end{array}$

  Let $c_{TA,i} = \langle j, p, \mathcal{S} \cup \{p : \gamma^i_{m_i,n_i}\}\rangle$ for $1 \le i \le k$. According to our induction assumption, c_{LR} has a form $\langle \Gamma p_1 q, C, j\rangle$ such that there exist a $p_2 : \gamma^i_{m_i,n_i} \in q$ with $p = p_1 p_2$ for all $1 \le i \le k$. Furthermore, the dot in $\gamma^i_{m_i,n_i}$ precedes a terminal $a = w_{j+1}$, we consequently can perform scans on $c_{TA,1}, \ldots, c_{TA,k}$ and there is a shift transition $action(q, a) = s(r, q'), p_2 \in L(r)$. Consequently, we can obtain new TA configurations $c'_{TA,i} = \langle j+1, p, \mathcal{S} \cup \{p : \gamma^i_{m_i,n_i+1}\}\rangle$ and a new LR configuration $\langle \Gamma p_1 q w_{j+1} p_1 p_2 q', C, j+1\rangle$ in the LR automaton.

According to the LR automaton construction, $\overline{\{\varepsilon : \gamma^1_{m_1,n_1+1}, \ldots, \varepsilon : \gamma^k_{m_k,n_k+1}\}} = q'$ with $\varepsilon : \gamma^i_{m_i,n_i+1}$ having a predecessor link pointing to $p_2 : \gamma^i_{m_i,n_i}$ in q. Further elements in q' are obtained by predict and resume and are linked to the elements with address ε by a predecessor chain. The same predict and resume operations can be applied to the corresponding TA configurations $c'_{TA,1}, \ldots, c'_{TA,k}$ as well. Therefore our induction assumption holds.

- **Suspend** and **Reduce**: We assume again that we have sets $\mathcal{C}_{TA}$ of TA configurations and $\mathcal{C}_{LR}$ of LR configurations such that $\mathcal{C}_{TA} = \bigcup_{c_{LR} \in \mathcal{C}_{LR}} f_{TA}(c_{LR})$.

  To show: For any pair $c_{LR}, f_{TA}(c_{LR})$ contained in the two sets: there is a $c_{TA} \in f_{TA}(c_{LR})$ that allows to derive a new configuration c'_{TA} via an application of *suspend* iff c_{LR} allows to derive a new configuration c'_{LR} via an application of *reduce 1/2* such that $f_{TA}(c'_{LR}) = \overline{c'_{TA}}$.

  Let $c_{TA} = \langle i, pj, \mathcal{S} \cup \{p : \gamma_{k,n}, pj : \beta_{l,m}\}\rangle$ such that a suspend can be applied resulting in $\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,n+1}, pj : \beta_{l,m}\}\rangle$. According to our induction assumption, c_{LR} has a form $\langle \Gamma pjq, C, i\rangle$ such that there exists a $\varepsilon : \beta_{l,m} \in q$ (without loss of generality, we assume our grammars to be ε-free. Consequently, computation points with the dot at the end of a component cannot arise from predict or resume operations and therefore always have address ε). Furthermore, since in $\beta_{l,m}$, the dot is at the end of the $(l+1)$th component, we have $r(\beta, l+1) \in action(q, -)$.

  Let us remind the suspend and reduce operations:

  TA Suspend: $\dfrac{\langle i, pj, \mathcal{S} \cup \{p : \gamma_{k,n}, pj : \beta_{l,m}\}\rangle}{\langle i, p, \mathcal{S} \cup \{p : \gamma_{k,n+1}, pj : \beta_{l,m}\}\rangle}$ $\quad [\gamma_{k,i}]\beta_{l,m} \to \gamma_{k,n+1}[\beta_{l,m}] \in \Theta$ or $\beta_{l,m} \to ret, [\gamma_{k,n}]ret \to \gamma_{k,n+1} \in \Theta$

  LR Reduce 1: $\dfrac{\langle \Gamma p_k q_k X_k \ldots p_1 q_1 X_1 pjq, C, i\rangle}{\langle \Gamma p_k q_k \boxed{l}_1 p_k p' q', C \cup \{pj\boxed{l}\beta_1\}, i\rangle}$ $\quad r(\beta,1) \in action(q,-), k = |lhs(\gamma,0)|,$ $goto(q_k, lhs(\gamma)_1) = \langle r, q'\rangle, p' \in L(r), \boxed{l} \notin \mathcal{N}_C$

  LR Reduce 2: $\dfrac{\langle \Gamma p_k q_k X_k \ldots p_1 q_1 X_1 pjq, C \cup \{pj\boxed{l}\beta_{l-1}\}, i\rangle}{\langle \Gamma p_k q_k \boxed{l}_l p_k p' q', C \cup \{pj\boxed{l}\beta_l\}, i\rangle}$ $\quad r(\beta, l) \in action(q,-),$ $1 < l, k = |lhs(\beta, l-1)|,$ $goto(q_k, lhs(\beta)_l) = \langle r, q'\rangle, p' \in L(r)$

  where for both reduce steps, any component pointer $\boxed{i}_m$ among the $X_1, \ldots, X_k$ has an address pjm for some $m \in \{1, \ldots, |rhs(\beta)|\}$.

  The only difference between the two reduce steps is that in the first, a first component is found and therefore a new pointer added to C, while in the second, only the component index of an already present pointer in C is incremented.

  In our case, there must be elements $p' : \gamma_{k,n}, p'j : \beta_{l,0} \in q_k$ such that $p_k p' = p$ and $\varepsilon : \gamma_{k,n+1} \in q'$. This means that for the resulting LR configuration, we obtain exactly the TA thread store plus eventually additional store entries obtained from predict or resume operations.

□

4.7. Comparison to TAG Parsing

As mentioned in the beginning, LR parsing has also been proposed for TAG [31,41]. The LR states in this case contain dotted productions that correspond to small subtrees (a node and its daughters) in the TAG elementary trees. They are closed under moving up and down in these trees, predicting adjunctions and predicting subtrees below foot nodes. There are two different transitions labeled with non-terminal symbols in the LR automata constructed for TAG: Transitions that correspond to having finished a subtree and moving the dot over the corresponding non-terminal node and transitions that correspond to having finished the part below a foot node and moving back into the adjoined auxiliary tree. The latter transitions are labeled with the adjunction site. In the parser, when following such a transition, the adjunction site is stored on the stack of the parser. Adjunctions can be nested, consequently, we can have a list (actually a stack) of such adjunction sites that we have to go back to later. This way to keep track of where to go back to once the adjoined tree is entirely processed

corresponds to the derivation node addresses we have in the thread automata and the LCFRS LR parser since both mechanisms capture the derivation structure. The difference is that with TAG, we have a simple nested structure without any interference of nested adjunctions in different places. And the contribution of auxiliary trees in the string contains only a single gap, consequently only a single element in the daughter list of a node contains such information about where to go back to when having finished the auxiliary tree. Therefore each of these chains of adjunctions can be kept track of locally on the parser stack.

Restricting our LR parser to well-nested 2-LCFRS amounts to restricting it to TAG. This restriction means that our LCFRS rules have the following form: Assume that $\gamma = A(\alpha_0, \ldots, \alpha_{dim(A)-1}) \rightarrow A_1(x_0^{(1)}, \ldots, x_{dim(A_1)-1}^{(1)}) \cdots A_m(x_0^{(m)}, \ldots, x_{dim(A_m)-1}^{(m)})$ is a rule in such an LCFRS. Then we can partition the righthand side into minimal subsequences $A_1 \ldots A_{i_1}, \ldots, A_{i_k} \ldots A_m$ such that a) the yields of these subsequences do not cross; b) every subsequence either (i) contains only a single non-terminal or (ii) has a first element of fan-out 2 that wraps around the yields of the other non-terminals. In this latter case, the yield of the second to last non-terminals can again be partitioned in the same way. Consider as an example a rule $A(x_1x_2x_3x_4x_5x_6x_7) \rightarrow B(x_1)C(x_2, x_6)D(x_3, x_5)E(x_4)F(x_7)$. Here, we can partition the righthand side into the daughter sequences B and C, D, E and F that can be parsed one after the other without having to jump back and forth between them. Concering C, D, E, the C wraps around D, E and, concerning this shorter sequence, D wraps around E. This means that during parsing, when processing the yield of one of these subsequences, we have to keep track of the last first component of some non-terminal that we have completed since this will be the first one for which we have to find a second component. We can push this information on a stack and have to access only the top-most element. There is no need to search through the entire partial derivation structure that has already been built in order to find a matching first component when reducing a second one. On the parse stack, it is enough to push the rule names for which we have completed a first component and still need to find a second. Pointers to the derivation structure are no longer needed. And node addresses are not needed either. Besides nestings between sisters as in this example, we can also have embedded nestings in the sense that a first comoponent is part of some other component. In this case, we have to finish the lower non-terminal before finishing the higher. In reductions of first components, we keep can keep track of embedded first components by removing them from the stack and popping them again after having added the new element. This makes sure that the second component of the most deeply embedded first component inside the rightmost first component of what we have processed so far must be completed next.

The LR automaton does not need node addresses any longer. Besides this, it looks exactly like the automata we have built so far. As an example consider the LR automaton in Figure 23.

Whenever we reduce a first component of a rule with fan-out 2, we push the rule name on the component stack and any stack symbols corresponding to daughters of this component (i.e., the first component non-terminals are part of the reduced yield but not the corresponding second one) are taken from this stack and then pushed on an extra stack attached to the new rule symbol, such that the rightmost daughter is the highest element on this embedded stack. Whenever we reduce a second component, we must have the corresponding rule name as the top stack element and then we can delete it. The top is defined as follows: If the top is an element without attached stack, it is itself the top of the entire stack. If not, the top of its attached stack is the top of the entire stack structure. An example is given in Figure 24. This example involves two reduce steps with first components, the first in state q_9 where the rule name γ is pushed on the stack. In the second reduce step, we reduce the first component of β. This contains a non-terminal A_1 for which we have not seen the corresponding A_2 yet. Therefore, one symbol is popped from the stack (for A_1), and the new β is pushed with an attached stack containing the removed γ. Whenever we reduce a second component of a rule, the top-most stack element must be the name of that rule.

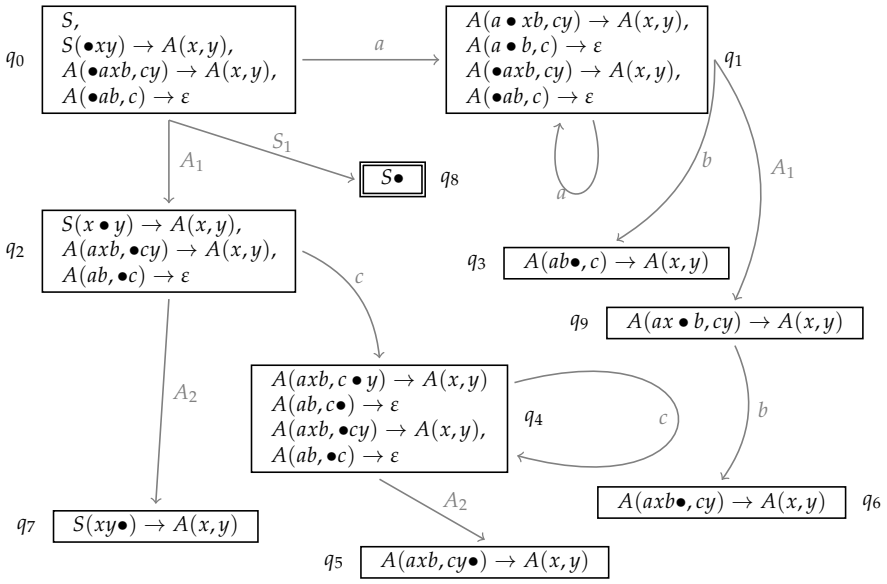


Figure 23. LR automaton for an LCFRS with rules $\alpha : S(xy) \to A(x,y)$, $\gamma : A(ab,c) \to \varepsilon$, $\beta : A(axb,cy) \to A(x,y)$.

parsing stack	rem. input	component stack	operation
q_0	aabbcc	–	
$q_0\ a\ q_1$	abbcc	–	shift
$q_0\ a\ q_1\ a\ q_1$	bbcc	–	shift
$q_0\ a\ q_1\ a\ q_1\ b\ q_3$	bcc	–	shift
$q_0\ a\ q_1\ A_1\ q_9$	bcc	γ	reduce γ_1
$q_0\ a\ q_1\ A_1\ q_9\ b\ q_6$	cc	γ	shift
$q_0\ A_1\ q_2$	cc	$\beta[\gamma]$	reduce β_1 (γ_1 is a daughter)
$q_0\ A_1\ q_2\ c\ q_4$	c	$\beta[\gamma]$	shift
$q_0\ A_1\ q_2\ c\ q_4\ c\ q_4$	–	$\beta[\gamma]$	shift
$q_0\ A_1\ q_2\ c\ q_4\ A_2\ q_5$	–	β	reduce γ_2
$q_0\ A_1\ q_2\ A_2\ q_7$	–	–	reduce β_2
$q_0\ S_1\ q_8$	–	–	reduce α_1

Figure 24. Parsing trace for *aabbcc* (only successful configurations), well-nested 2-LCFRS parsing.

5. Conclusions

This paper presents the first LR style algorithm for LCFRS parsing. It extends the well-known LR parsing techniques from CFG to the case of LCFRS where a single non-terminal can span several non-adjacent strings in the input. Even though the derivation structure needs to be kept track off during parsing in order to make sure that the correct components are related to each other, the proposed algorithm constructs only a finite automaton, by capturing derivation tree addresses in regular expressions whenever left recursions occur in some component.

The resulting parsing algorithm extends previous work on incremental Earley-style LCFRS parsing and on Thread Automata, an automaton model for LCFRS that proceeds in an incremental way but that does not precompile predict and resume steps the way it is proposed for the LR construction in this paper.

Acknowledgments: The research presented in this paper was partly funded by the German Science Foundation (DFG). We are grateful to anonymous reviewers of this paper and of a previous short version of this work that has been presented at NAACL-HLT 2015. Their comments and suggestions helped considerably to improve the paper.

Author Contributions: This work is an extension of previous research of the authors. The underlying ideas emerged from mutual discussion, to which both authors have contributed substantially. The present paper was mostly written by Laura Kallmeyer. Wolfgang Maier contributed the introduction and the first part of the second section.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Vijay-Shanker, K.; Weir, D.; Joshi, A.K. Characterising structural descriptions used by various formalisms. In Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics, Stanford, CA, USA , 6–9 July 1987; pp. 104–111.
2. Joshi, A. How much context-sensitivity is necessary for characterizing structural descriptions? In *Natural Language Processing: Theoretical, Computational and Psychological Perspectives*; Dowty, D., Karttunen, L., Zwicky, A., Eds.; Cambridge University Press: New York, NY, USA, 1985; pp. 206–250.
3. Brants, S.; Dipper, S.; Hansen, S.; Lezius, W.; Smith, G. The TIGER treebank. In Proceedings of the 1st Workshop on Treebanks and Linguistic Theories, Sofia, Bulgaria, 13–14 December 2002; pp. 24–42.
4. Maier, W.; Lichte, T. Characterizing discontinuity in constituent treebanks. In Proceedings of the 14th International Conference, Bordeaux, France, 25–26 July 2009; pp. 167–182.
5. Marcus, M.P.; Santorini, B.; Marcinkiewicz, M.A. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.* **1993**, *19*, 313–330.
6. Evang, K.; Kallmeyer, L. PLCFRS parsing of english discontinuous constituents. In Proceedings of the 12th International Conference on Parsing Technologies (IWPT 2011), Dublin, Ireland, 5–7 October 2011; pp. 104–116.
7. Haider, H.; Rosengren, I. *Scrambling*; Sprache und Pragmatik: Lund, Sweden, 1998.
8. Kübler, S.; McDonald, R.; Nivre, J. Dependency parsing. *Synth. Lect. Hum.Lang. Technol.* **2009**, *1*, 1–127.
9. Kuhlmann, M. Mildly non-projective dependency grammar. *Comput. Linguist.* **2013**, *39*, 355–387.
10. Skut, W.; Krenn, B.; Brants, T.; Uszkoreit, H. An annotation scheme for free word order languages. In Proceedings of the 5th Applied Natural Language Processing Conference, 31 March–3 April 1997; pp. 88–95.
11. Ranta, A. *Grammatical Framework: Programming with Multilingual Grammars*; CSLI Publications: Stanford, CA, USA, 2011.
12. Ljunglöf, P. Expressivity and Complexity of the Grammatical Framework. Ph.D. Thesis, Göteborg University, Gothenburg, Sweden, 2004.
13. Kallmeyer, L.; Maier, W.; Parmentier, Y.; Dellert, J. TuLiPA-Parsing extensions of TAG with range concatenation grammars. *Bull. Pol. Acad. Sci.* **2010**, *58*, 377–391.
14. Kallmeyer, L.; Parmentier, Y. On the relation between multicomponent tree adjoining grammars with tree tuples (TT-MCTAG) and range concatenation grammars (RCG). In Proceedings of the Second International Conference on Language and Automata Theory and Applications (LATA 2008), Tarragona, Spain, 13–19 March 2008; pp. 263–274.
15. Dada, A.; Ranta, A. Implementing an open source arabic resource grammar in GF. In *Perspectives on Arabic Linguistics: Papers from the Annual Symposium on Arabic Linguistics*; John Benjamins Publishing: Amsterdam, The Netherlands, 2007; pp. 209–231.
16. Botha, J.A.; Blunsom, P. Adaptor grammars for learning non-concatenative morphology. In Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, Washington, DC, USA, 18–21 October 2013; pp. 345–356.
17. Melamed, I.D.; Satta, G.; Wellington, B. Generalized multitext grammars. In Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Barcelona, Spain, 21–26 July 2004; pp. 661–668.
18. Kaeshammer, M. Hierarchical machine translation with discontinuous phrases. In Proceedings of the Tenth Workshop on Statistical Machine Translation, Lisbon, Portugal, 17–18 September 2015.
19. Kaeshammer, M. Synchronous linear context-free rewriting systems for machine translation. In Proceedings of the Seventh Workshop on Syntax, Semantics and Structure in Statistical Translation, Atlanta, GA, USA, 13 June 2013; pp. 68–77.
20. Seki, H.; Matsumura, T.; Fujii, M.; Kasami, T. On multiple context-free grammars. *Theor. Comput. Sci.* **1991**, *88*, 191–229.

21. Burden, H.; Ljunglöf, P. Parsing linear context-free rewriting systems. In Proceedings of the Ninth International Workshop on Parsing Technology, Vancouver, BC, Canada, 9–10 October 2005; pp. 11–17.
22. Kallmeyer, L.; Maier, W. An incremental earley parser for simple range concatenation grammar. In Proceedings of the 11th International Conference on Parsing Technologies (IWPT'09), Paris, France, 7–9 October 2009; pp. 61–64.
23. Villemonte de la Clergerie, E. Parsing mildly context-sensitive languages with thread automata. In Proceedings of the COLING 2002: The 19th International Conference on Computational Linguistics, Taipei, Taiwan, 26–30 August 2002.
24. Kallmeyer, L.; Maier, W. Data-driven parsing using probabilistic linear context-free rewriting systems. *Comput. Linguist.* **2013**, *39*, 87–119.
25. Van Cranenburgh, A. Efficient parsing with linear context-free rewriting systems. In Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, Avignon, France, 23–27 April 2012; pp. 460–470.
26. Angelov, K.; Ljunglöf, P. Fast statistical parsing with parallel multiple context-free grammars. In Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, Gothenburg, Sweden, 26–30 April 2014; pp. 368–376.
27. Knuth, D.E. On the translation of languages from left to right. *Inf. Control* **1965**, *8*, 607–639.
28. Tomita, M. LR parsers for natural languages. In Proceedings of the COLING 1984: The 10th International Conference on Computational Linguistics, Stanford, CA, USA, 2–6 July1984; pp. 354–357.
29. Tomita, M. An efficient augmented context-free parsing algorithm. *Comput. Linguist.* **1987**, *13*, 31–46.
30. Nederhof, M.J. An alternative LR algorithm for TAGs. In Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Montreal, QC, Canada, 10–14 August 1998; pp. 946–952.
31. Prolo, C.A. LR Parsing for Tree Adjoining Grammars and Its Application to Corpus-Based Natural Language Parsing. Ph.D. Thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA, 2003.
32. Aizikowitz, T.; Kaminski, M. LR(0) conjunctive grammars and deterministic synchronized alternating pushdown automata. In Proceedings of the 6th International Computer Science Symposium on Computer Science, Theory and Applications, St. Petersburg, Russia, 14–18 June 2011; pp. 345–358.
33. Okhotin, A. Generalized LR parsing algorithm for boolean grammars. *Int. J. Found. Comput. Sci.* **2006**, *17*, 629–664.
34. Barash, M.; Okhotin, A. Generalized LR parsing algorithm for grammars with one-sided contexts. *Theory Comput. Syst.* **2016**, doi:10.1007/s00224-016-9683-3.
35. Kallmeyer, L.; Maier, W. LR parsing for LCFRS. In Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, CO, USA, 31 May–5 June 2015; pp. 1250–1255.
36. Boullier, P. *Proposal for a Natural Language Processing Syntactic Backbone*; Research Report 3342; INRIA-Rocquencourt: Rocquencourt, France, 1998.
37. Weir, D. Characterizing Mildly Context-Sensitive Grammar Formalisms. Ph.D. Thesis, University of Pennsylvania, Philadelphia, PA, USA, 1988.
38. Kracht, M. *The Mathematics of Language*; Mouton de Gruyter: Berlin, Germany, 2003.
39. Kallmeyer, L. *Parsing beyond Context-Free Grammar*; Springer: Heidelberg, Germany, 2010.
40. Grune, D.; Jacobs, C. *Parsing Techniques. A Practical Guide*, 2nd ed.; Monographs in Computer Science; Springer: New York, NY, USA, 2008.
41. Nederhof, M.J. Solving the correct-prefix property for TAGs. In Proceedings of the Fifth Meeting on Mathematics of Language, Schloss Dagstuhl, Germany, 25–28 August 1997; pp. 124–130.