



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

REGULATED GRAMMAR SYSTEMS

SYSTÉMY REGULOVANÝCH GRAMATIK

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

MARTIN TOMKO

SUPERVISOR

VEDOUCÍ PRÁCE

Prof. RNDr. ALEXANDER MEDUNA, CSc.

BRNO 2018

Brno University of Technology - Faculty of Information Technology

Department of Information Systems

Academic year 2017/2018

Master's Thesis Specification

For: **Tomko Martin, Bc.**

Branch of study: Mathematical Methods in Information Technology

Title: **Regulated Grammar Systems**

Category: Theoretical Computer Science

Instructions for project work:

1. Study grammar systems and regulated grammars based upon the supervisor's instructions.
2. Introduce regulated grammar systems by analogy with classical grammar systems.
3. Study the properties of regulated grammar systems based upon the supervisor's instructions.
4. Discuss the applications of regulated grammar systems and formalize them by using regulated grammar systems.
5. Implement regulated grammar systems formalized in item 4. Focus on visualization aspects of the implementation.
6. Evaluate the achieved results. Discuss further development of the project.

Basic references:

- Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1-3, Springer, 1997, ISBN 3-540-60649-1
- Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition), Pearson Education, 2006, ISBN 0-321-48681-1

Requirements for the semestral defense:

Items 1 and 2.

Detailed formal specifications can be found at <http://www.fit.vutbr.cz/info/szz/>

The Master's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

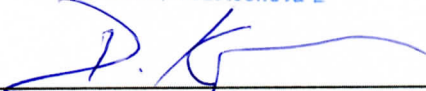
Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Meduna Alexander, prof. RNDr., CSc., DIFS FIT BUT**

Beginning of work: November 1, 2017

Date of delivery: May 23, 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
602 00 Brno, Ústřetěchova 2



Dušan Kolář

Associate Professor and Head of Department

Abstract

This thesis recaps a basic theory of formal languages, regulated grammars, and the parsing of LL(1) languages. An algorithm for parsing programmed grammars inspired by LL(1) parsing is suggested and analyzed. The class of languages accepted by this algorithm is shown to be a strict superclass of LL(1) languages, containing some non-context-free languages. However, this class appears to be incomparable with the class of context-free languages.

Abstrakt

Práce poskytuje přehled základů teorie formálních jazyků, regulovaných gramatik a analýzy LL(1) jazyků. Je zde navržen a analyzován algoritmus pro analýzu programovaných gramatik, inspirován LL(1) analyzátozem. Třída jazyků přijímaná tímto algoritmem je striktní nadtřídou LL(1) jazyků, obsahující některé jazyky, které nejsou bezkontextové. Tato třída se však jeví být neporovnatelná s třídou bezkontextových jazyků.

Keywords

regulated grammars, parsing, LL parsing, programmed grammars

Klíčová slova

regulované gramatiky, syntaktická analýza, LL analýza, programované gramatiky

Reference

TOMKO, Martin. *Regulated Grammar Systems*. Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. RNDr. Alexander Meduna, CSc.

Rozšířený abstrakt

Táto práca sa zaoberá návrhom a analýzou algoritmu pre syntaktickú analýzu programovaných gramatík inšpirovanú LL analýzou bezkontextových gramatík.

V prvých kapitolách práca poskytuje prehľad potrebnej teórie. Začína základmi teórie formálnych jazykov a gramatík, a ďalej uvádza gramatiky regulované na základe obmedzovania možných postupností pravidiel, špeciálne programované gramatiky. Pokračuje zhrnutím základných prístupov k syntaktickej analýze bezkontextových gramatík – analýzy zhora nadol a analýzy zdola nahor, pričom špeciálnu pozornosť venuje LL analýze, ktorá je ďalej využitá ako základ pre navrhovaný algoritmus.

Myšlienkou za týmto algoritmom, nazývaným *Table-resort algorithm*, je prispôbenie LL(1) analýzy mechanizmom programovaných gramatík – riadime sa predovšetkým posledným použitým pravidlom a jemu náležiacou množinou pravidiel, z ktorých môžeme vybrať ďalšie použité pravidlo. Ak táto voľba nie je jednoznačná, algoritmus sa rozhoduje podľa tzv. TR tabuľky (*Table Resort*), trojrozmiernej analógie k LL tabuľke, ktorú indexujeme okrem vrcholu zásobníka a symbolu na vstupe aj spomínanou množinou pravidiel. Toto rozhodovanie podľa tabuľky inšpirovalo názov algoritmu – algoritmus sa v prípade nejasností *uchýľuje* k tabuľke.

Popísaný algoritmus má nežiaducu vlastnosť, že jazyk ním prijímaný nemusí prijímať ten istý jazyk, ako generuje gramatika, podľa ktorej je jeho TR tabuľka vypočítaná. Tejto vlastnosti sa však pri algoritme založenom na LL analýze všeobecne nedá vyhnúť – ide mimo iné o priamy dôsledok toho, že algoritmus prepisuje vždy najľavejší výskyt daného neterminálu. Ďalším zdrojom týchto odchýlok je nedokonalosť TR tabuľky, na ktorej výpočet sa využívajú iba bezkontextové formy zadanej gramatiky, pričom mechanizmy programovanej gramatiky, t. j. množiny pravidiel nasledujúcich dané pravidlo, sú pre jednoduchosť ignorované. Tu však práca taktiež poukazuje na hranice, ktoré sa pri predpovedaní derivácií programovaných gramatík nedajú prekročiť.

Algoritmus ďalej môže pre určité vstupy nekonečne cykliť – tento problém je v práci analyzovaný a pre programované gramatiky bez ε -pravidiel a kontroly výskytu je navrhnuté riešenie tohoto problému, ktoré je aplikované vo vylepšenej verzii algoritmu. Riešenie spočíva v priebežnej kontrole počtu vygenerovaných terminálov, pričom bez prekročenia veľkosti vstupu môžu nekonečné cykly pozostávať iba z jednoduchých pravidiel (pravidiel tvaru $A \rightarrow B$). Pre tieto je vďaka vlastnostiam algoritmu možné predvídať, či ich aplikácia za určitých okolností povedie k nekonečnému cyklu, čo je možné zhrnúť v tzv. CL tabuľke (*Candidate Loop*), pomocou ktorej algoritmus priebežne kontroluje svoj aktuálny stav.

Trieda jazykov prijímaná týmto algoritmom je ostrou nadtriedou triedy **LL(1)**, javí sa však byť neporovnateľná s triedou bezkontextových jazykov – na jednej strane algoritmus dokáže prijímať jazyky ako $\{a^n b^n c^n : n \geq 0\}$, ktoré nie sú bezkontextové, pravdepodobne však nevie prijať napr. jazyk $\{ww^R : w \in \{a, b\}^*\}$, ktorý je bezkontextový, a potenciálne dokonca ani deterministický bezkontextový jazyk $\{a^i b^j : i \geq j \geq 0\}$.

Bližšie určenie triedy jazykov prijímaných algoritmom zostáva otvorenou otázkou, podobne ako otázka, pre ktoré gramatiky sa jazyk generovaný gramatikou odlišuje od jazyka prijímaného algoritmom za využitia tabuľky založenej na tejto gramatike. Ďalšie smery, ktorými sa dá od tejto práce uberať, sú analýza zacyklenia algoritmu pre všeobecnejšie typy programovaných gramatík a prípadné vylepšenia základných mechanizmov algoritmu, predovšetkým konštrukcie TR tabuľky.

Regulated Grammar Systems

Declaration

I declare that this thesis was composed by myself under the supervision of prof. RNDr. Alexander Meduna, CSc., and I have documented all the sources and materials used during the preparation of this thesis.

.....
Martin Tomko
May 23, 2018

Acknowledgements

I would like to thank my supervisor, prof. Meduna, for his guidance, advice and motivation throughout the writing of this thesis.

Contents

1	Introduction	2
2	Basic Concepts	3
2.1	Basic Definitions	3
2.1.1	Strings	3
2.1.2	Languages	4
2.1.3	Grammars	5
2.1.4	The Chomsky Hierarchy	6
2.2	Regulated Grammars	8
2.2.1	Regular-Controlled Grammars	8
2.2.2	Matrix Grammars	11
2.2.3	Programmed Grammars	12
2.2.4	Generative power	13
3	Parsing of Context-Free Grammars	14
3.1	Leftmost Derivations	14
3.2	Pushdown Automata	14
3.3	Top-Down Parsing	17
3.4	Bottom-Up Parsing	18
3.5	LL Parsing	19
4	Deterministic Parsing of Programmed Grammars	23
4.1	Basic Ideas	23
4.2	The TR Table	24
4.3	A Naive Table-resort Algorithm	27
4.3.1	Language Discrepancy	29
4.3.2	Accepting Power	30
4.4	Looping	32
4.4.1	Restricting Sentential Form Size	33
4.4.2	Finding Candidate Loops	34
4.5	An Improved Table-resort Algorithm	36
5	Conclusion	38
	Bibliography	40
A	Contents of the Attached CD	41

Chapter 1

Introduction

Regulated grammars provide an interesting extension to context-free grammars; however, parsing them is in general not as well-explored and well-handled. This thesis introduces and explores an extension of the LL parser designed to parse a subclass of programmed grammars.

In chapters 2 and 3, the necessary theory is reviewed – chapter 2 introduces the basics of formal languages and regulated grammars, and chapter 3 talks about approaches to the parsing of context-free grammars, notably LL(1) parsing.

In chapter 4, an algorithm to parse programmed grammars called the *table-resort* algorithm is suggested, which is based on LL(1) parsing. The first, naive version turns out to have the disadvantage of possibly not halting for certain inputs. This is examined in the rest of the chapter, and at the end, an improved version is proposed, which is just as powerful, but also guaranteed to halt for propagating grammars without appearance checking.

An implementation of the presented algorithms can be found on the accompanying CD, described in appendix A.

Chapter 2

Basic Concepts

In this chapter, we will briefly discuss the terms necessary for studying formal languages and grammars, and some parsing techniques applied to particular classes of grammars. We will also introduce several kinds of regulated grammars.

This chapter assumes a basic knowledge of set theory (sets, set operations, binary relations, ...). The knowledge of formal language theory is also useful, but these terms are reviewed in the first sections.

2.1 Basic Definitions

For completeness, we will begin by defining the basic terms related to formal languages.

For a more complete or alternative description of these concepts, many publications can be consulted. One such publication is [6], which also serves as the basis for the study of regulated grammars in this thesis.

2.1.1 Strings

An *alphabet* is a finite, nonempty set of *symbols*. We do not define symbols more closely, other than as being distinct from one another and as basic elements of strings. Some simple examples of alphabets include $\{0, 1\}$, $\{a, b, c\}$ or $\{\heartsuit, \diamondsuit, \clubsuit, \spadesuit\}$, but we can also think of a set of words from a particular language as an alphabet.

Let Σ be an alphabet. A sequence $w = a_1 \dots a_n$ of (not necessarily distinct) symbols from Σ is called a *string over Σ* ($a_i \in \Sigma$ for $1 \leq i \leq n$). The number n of symbols from which the string is composed is called its *length*, denoted as $|w|$. For example, $aacb$, $bcbcbc$ and b are examples of strings over $\{a, b, c\}$ (of lengths 4, 6 and 1, respectively). Sentences of some natural language (e.g. English) can be thought of as strings over the set of its words, and programs written in a particular programming language are strings over the set of its lexemes. We will usually simply refer to strings over a particular alphabet simply as *strings*, omitting the alphabet, unless the distinction is desirable.

A special case of a string is the *empty string*, denoted as ε ¹, which is a sequence of 0 symbols. Note that $|\varepsilon| = 0$, and for any alphabet Σ , ε is a string over Σ .

We can also consider the number of occurrences of a particular symbol $a \in \Sigma$ in a string $w \in \Sigma^*$ – we will denote it as $|w|_a$. For example, $|abbcbbaab|_b = 4$, $|0111|_0 = 1$ and $|aba|_c = 0$.

¹In literature, the symbol λ is often used instead.

For a set of symbols $\Gamma \subseteq \Sigma$, we can denote by $|w|_\Gamma$ the number of occurrences of any symbol from Γ in w – for example, $|\text{occurrence}|_{\{c,e\}} = 5$. Note that $|w|_\Gamma = \sum_{a \in \Gamma} |w|_a$.

The *concatenation* of two strings, $u = a_1 \dots a_m$ and $w = b_1 \dots b_n$ is the string $uw = a_1 \dots a_m b_1 \dots b_n$. Note that $\varepsilon w = w\varepsilon = w$ for any string w , and that concatenation is not commutative, so in general, $uw \neq wu$. Concatenation is, however, associative, so $u(vw) = (uv)w$ for any strings u, v, w .

The *power* of a string w , w^n , is the string w repeated n times. More precisely, we can define it recursively as follows:

$$w^0 = \varepsilon,$$

$$w^n = ww^{n-1}, \text{ for } n \geq 1.$$

The *reversal* of a string $w = a_1 \dots a_n$ is the string $w^R = a_n \dots a_1$, that is, the symbols of w in reverse order. For example, *koob* is the reversal of the string *book*.

Any contiguous subsequence of $w = a_1 \dots a_n$ is called a *substring* of w . More precisely, u is a substring of w if and only if $w = xuy$ for some strings x, y . Furthermore, u is a *proper substring* if $u \neq \varepsilon$ and $u \neq w$. We call u a *prefix* of w if $w = uy$ for some y and we call u a *suffix* of w if $w = xu$ for some x . For example, the string *abba* has the substrings $\varepsilon, a, b, ab, ba, bb, abb, bba, abba$, with $\varepsilon, a, ab, abb, abba$ being prefixes and $\varepsilon, a, ba, bba, abba$ being suffixes.

2.1.2 Languages

We denote by Σ^* the set of all strings over Σ , and any of its subsets $L \subseteq \Sigma^*$ is a *language* over Σ . That is, any language over Σ is simply a set of some strings over Σ . For example, $\{0, 01, 10, 1010\}$, $\{0000\}$, $\{0^n 1^n : n \in \mathbb{N}\}$, $\{\varepsilon\}$, \emptyset and $\{0, 1\}^*$ are all languages over $\{0, 1\}$. As with strings, we will simply talk about *languages*, not mentioning the specific alphabet unless we need to.

As languages are sets, we can combine them using the usual set operations such as union, intersection, and complement (over an alphabet). For any L_1, L_2 :

$$L_1 \cup L_2 = \{w : w \in L_1 \vee w \in L_2\} - \text{the union of } L_1, L_2,$$

$$L_1 \cap L_2 = \{w : w \in L_1 \wedge w \in L_2\} - \text{the intersection of } L_1, L_2,$$

$$L_1 \setminus L_2 = \{w : w \in L_1 \wedge w \notin L_2\} - \text{the difference of } L_1, L_2,$$

$$\bar{L}_1 = \Sigma^* \setminus L_1 = \{w \in \Sigma^* : w \notin L_1\} - \text{the complement of } L_1 \text{ over } \Sigma.$$

Note that the value of the complement depends on the alphabet chosen. We can also introduce new operations, such as concatenation and iteration.

The *concatenation* of the languages L_1, L_2 is the language $L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$. In other words, it is the set of all concatenations xy of any string x from the first language with any string y from the second language. For example, for $L_1 = \{a, ab\}$ and $L_2 = \{bc, c\}$, $L_1 L_2 = \{abc, abbc, ac\}$. Note that for any language L , $L\emptyset = \emptyset L = \emptyset$ and $L\{\varepsilon\} = \{\varepsilon\}L = L$. Just like with strings, concatenation of languages is non-commutative and associative.

We can now define the *power* L^n of a language L analogously as with strings:

$$L^0 = \{\varepsilon\},$$

$$L^n = LL^{n-1}, \text{ for } n \geq 1.$$

We can also define the *iteration* L^* and *positive iteration* L^+ of a language L as follows:

$$L^* = \bigcup_{i \geq 0} L^i = L^0 \cup L^1 \cup L^2 \cup \dots,$$

$$L^+ = \bigcup_{i \geq 1} L^i = L^1 \cup L^2 \cup \dots$$

Note that $L^+ = L^*L = LL^*$, whereas $L^* = L^+ \cup \{\varepsilon\}$. Also note that we can think of Σ^* as the iteration of the set of all strings over Σ of length 1 – we will end up with exactly the set of all strings over Σ .

Relatively small, finite languages can be described simply by enumerating all their strings. For more complex languages, possible approaches include the set builder notation, as with $\{w \in \{a, b\}^* : |w| \text{ is a prime}\}$, or applying known operations to simpler languages, as with $\{a\}\{a, b, c\}^*$, which is the set of all strings over $\{a, b, c\}$ that start with a . In fact, using only finite languages and finitely many applications of union, concatenation and iteration, we can describe exactly the class **REG** of regular languages (described later in section 2.1.4, see [5] for proof of equivalence).

If we want to discuss more complex languages, however, these approaches might be impractical or even insufficient. There are two kinds of mechanisms that are commonly used to describe languages – grammars and automata, the former of which will be introduced in the following section, and the latter of which will be briefly discussed in section 3.2.

2.1.3 Grammars

This section will introduce a practical mechanism for describing more complex languages – grammars.

The idea behind grammars is to start with a string consisting only of a single symbol, called the *start symbol*, and keep rewriting parts of this string according to production rules of the form $\alpha \rightarrow \beta$ (rewrite α to β), where α and β are strings. Strings that can be produced this way are called *sentential forms*, and those consisting solely of terminal symbols are said to be the *sentences* generated by the grammar. The language generated by a grammar consists exactly of the sentences it generates.

These ideas will be illustrated by an example later in the section, and are made more precise in the following definitions:

Definition 2.1.1. A (*phrase structure*) grammar is a 4-tuple $G = (N, \Sigma, P, S)$, where:

- N is the set of nonterminal symbols (*nonterminals*);
- Σ is the set of terminal symbols (*terminals*);
- For convenience, let V denote $N \cup \Sigma$;
- $P \subseteq (V^*NV^*) \times V^*$ is the set of production rules;
- $S \in N$ is the start symbol.

A production rule $r = (\alpha, \beta) \in P$ will be denoted as $\alpha \rightarrow \beta$. α is called the *left-hand side* of the rule, denoted by $\text{lhs}(r)$, whereas β is called the *right-hand side* of the rule, denoted by $\text{rhs}(r)$. We can also add labels to the rules, in the form of a set Ψ of labels and a bijection $\psi : \Psi \leftrightarrow P$, which will assign a unique label to each rule. We can then mark

rules as $r : \alpha \rightarrow \beta \in P$ (which means that $\psi(r) = \alpha \rightarrow \beta \in P$) and denote the grammar as a 5-tuple $G = (N, \Sigma, \Psi, P, S)$. When using this notation, we will talk about rules and their labels interchangeably where appropriate. We will use this notation in section 2.2, when talking about regulated grammars.

Definition 2.1.2. Let $G = (N, \Sigma, P, S)$ be a grammar. The string μ derives λ in one step, denoted by $\mu \Rightarrow_G \lambda$, where $\mu, \lambda \in V^*$, if and only if there exists a rule $(\alpha \rightarrow \beta) \in P$ and strings $\gamma, \delta \in V^*$, such that $\mu = \gamma\alpha\delta$ and $\lambda = \gamma\beta\delta$.

In other words, $\mu \Rightarrow_G \lambda$ means that a substring α of μ can be replaced by β according to the rule $(\alpha \rightarrow \beta) \in P$, thus creating λ . We will omit the G if it is clear what grammar we are talking about, preferring to simply write $\mu \Rightarrow \lambda$.

For grammars with rule labels, we can write the label of the rule used for rewriting next to the derivation: $\mu \Rightarrow_G \lambda [r]$ means that μ was rewritten to λ using the rule $r : \alpha \rightarrow \beta$ (the only rule with the label r).

Definition 2.1.3. Let G be a grammar and \Rightarrow_G the relation on V^* as defined above. Then we can define the relations \Rightarrow_G^k for $k \in \mathbb{N}$, \Rightarrow_G^* and \Rightarrow_G^+ as follows:

- $\mu \Rightarrow_G^0 \lambda$ if and only if $\mu = \lambda$;
- $\mu \Rightarrow_G^k \lambda$ for $k \geq 1$ if and only if there exists a string $\kappa \in V^*$, such that $\mu \Rightarrow_G \kappa$ and $\kappa \Rightarrow_G^{k-1} \lambda$ (\Rightarrow_G^k is the k -th power of \Rightarrow_G);
- $\mu \Rightarrow_G^* \lambda$ if and only if $\mu \Rightarrow_G^k \lambda$ for some $k \geq 0$ (\Rightarrow_G^* is the transitive and reflexive closure of \Rightarrow_G);
- $\mu \Rightarrow_G^+ \lambda$ if and only if $\mu \Rightarrow_G^k \lambda$ for some $k \geq 1$ (\Rightarrow_G^+ is the transitive closure of \Rightarrow_G).

Again, we prefer to write \Rightarrow^k , \Rightarrow^* and \Rightarrow^+ , omitting the G subscript wherever possible. Strings over V that can be derived from the start symbol, that is, strings $\alpha \in V^*$, such that $S \Rightarrow_G^* \alpha$, are called *sentential forms*.

For grammars with rule labels, we can write the sequence of rules used for a particular derivation next to it: $\mu \Rightarrow_G^k \lambda [r_1 \dots r_k]$.

Finally, we can define the language generated by a grammar:

Definition 2.1.4. Let $G = (N, \Sigma, P, S)$ be a grammar. The language generated by G is the language $L(G) = \{w \in \Sigma^* : S \Rightarrow_G^* w\}$.

An example of a grammar is $G = (\{S\}, \{a, b\}, \{S \rightarrow aSb, S \rightarrow \varepsilon\}, S)$. It can be shown that the language of this grammar is $L(G) = \{a^n b^n : n \geq 0\}$. An example of a derivation using this grammar is $S \Rightarrow_G aSb \Rightarrow_G aaSbb \Rightarrow_G aabb$. The existence of such a derivation means that $aabb \in L(G)$.

2.1.4 The Chomsky Hierarchy

By placing varying limitations on the form of production rules of grammars, we can restrict the generative power of grammars, meaning that certain languages cannot be generated by grammars with certain limitations on their production rules. One important hierarchy arising from such limitations is the Chomsky Hierarchy, named after Noam Chomsky, which will be used throughout this thesis as a reference scale for the power of various types of grammars.

The Chomsky Hierarchy consists of four types of grammars, all complying with definition 2.1.1, but introducing more limitations on the production rules with each level, which also leads to the corresponding classes of languages being strictly smaller than their superclasses:

- *Type 0 grammars*, also called *phrase structure grammars* or *unrestricted grammars*, have production rules of the form $\alpha \rightarrow \beta$, where $\alpha \in V^*NV^*$, $\beta \in V^*$. That is, they correspond exactly to definition 2.1.1 with no additional restrictions placed on the form of the production rules.

The languages generated by type 0 grammars are referred to as *type 0 languages* or as *recursively enumerable languages*. The class of these languages is denoted as \mathcal{L}_0 or **RE**.

- *Type 1 grammars*, also called *context-sensitive grammars*, have production rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $\alpha, \beta \in V^*$, $A \in N$, $\gamma \in V^+$. That is, they only rewrite one nonterminal with a non-empty string, with α and β serving as a context – different rewritings of the same nonterminal might be possible in different contexts.

One exception for the non-emptiness of the right-hand side must be made possible to allow generating the empty string – for this reason, a context-sensitive grammar may have a single rule of the form $S \rightarrow \varepsilon$, where S is the start symbol, in which case S cannot appear on the right-hand side of any rule.

The languages generated by type 1 grammars are referred to as *type 1 languages* or as *context-sensitive languages*. The class of these languages is denoted as \mathcal{L}_1 or **CS**.

We also add without proof that the very same class of languages is generated by *essentially noncontracting grammars*, which have rules of the unrestricted form $\alpha \rightarrow \beta$, but with the additional requirement that $|\alpha| \leq |\beta|$, again with the same single exception for $S \rightarrow \varepsilon$, in which case S may not appear on the right-hand side of any rule.

- *Type 2 grammars*, also called *context-free grammars*, have rules of the form $A \rightarrow \gamma$, where $A \in N$, $\gamma \in V^*$. That is, compared to unrestricted grammars, only a single nonterminal is on the left-hand side of any rule, and unlike in context-sensitive grammars, no context is considered. Also note that the right-hand side of any rule can be empty.

The languages generated by type 2 grammars are referred to as *type 2 languages* or as *context-free languages*. The class of these languages is denoted as \mathcal{L}_2 or **CF**.

- *Type 3 grammars*, also called *right-linear grammars*, have rules of the form $A \rightarrow xB$ or $A \rightarrow x$, where $A, B \in N$, $x \in \Sigma^*$. That is, just like with context-free grammars, only a single nonterminal is on the left-hand side of any rule, and in addition, there is at most one nonterminal on the right-hand side, always in the rightmost position.

The languages generated by type 3 grammars are referred to as *type 3 languages* or as *regular languages*. The class of these languages is denoted as \mathcal{L}_3 or **REG**.

We add without proof that the same class of languages is generated by *left-linear grammars* – grammars with rules of the form $A \rightarrow Bx$ or $A \rightarrow x$, so the same as right-linear, except the nonterminal on the right-hand side is always in the leftmost position. Note that combining both types of rules (with a nonterminal on either side) would result in a greater expressive power.

Another common variant of type 3 grammars are *right-regular* (and *left-regular*) grammars – these have production rules of the form $A \rightarrow aB$ ($A \rightarrow Ba$) or $A \rightarrow a$, where $a \in \Sigma$ is a single terminal, with a possible exception for $S \rightarrow \varepsilon$ as with type 1 grammars.

Each of the aforementioned classes is a proper subclass of all the classes with smaller type numbers. This can be summarized as follows:

$$\mathbf{REG} \subset \mathbf{CF} \subset \mathbf{CS} \subset \mathbf{RE}$$

It should be noted that not all languages can be described by phrase structure grammars – there is an even greater class of all languages, which **RE** is a proper subclass of. In fact, for any nonempty finite alphabet Σ , there are countably many recursively enumerable languages, but there are uncountably many languages.

2.2 Regulated Grammars

Grammars, as described in the previous sections, can be enhanced with various regulation mechanisms, enabling them to place further restrictions on when various rules can be used, possibly increasing their generative power. In this section, we will discuss three such enhancements – regular-controlled grammars, matrix grammars, and programmed grammars, which all use variants of so-called *rule-based regulation*, and which all turn out to generate the same classes of languages. For a more complete treatment of these and more types of regulated grammars, see [6], which this section is based on.

Before we introduce the first type of regulated grammars, recall that a grammar can be defined as a 5-tuple $G = (N, \Sigma, \Psi, P, S)$, where Ψ is a set of rule labels, from which every production rule is assigned a unique rule label, and these rule labels can be written next to derivations to specify which production rules they are based on.

2.2.1 Regular-Controlled Grammars

Definition 2.2.1. A regular-controlled (context-free) grammar is a pair $H = (G, \Xi)$, where

- $G = (N, \Sigma, \Psi, P, S)$ is a context-free grammar called the *core grammar* with the set Ψ of rule labels,
- $\Xi \subseteq \Psi^*$ is a regular language over the set Ψ of rule labels called the *control language*.

The idea behind regular-controlled grammars is that a derivation is only considered valid if the sequence of rules used to produce it is a string of the control language. This idea is formalized in the following definition:

Definition 2.2.2. The language $L(H)$ generated by a regular-controlled grammar $H = (G, \Xi)$ is defined as $L(H) = \{w \in \Sigma^* : S \Rightarrow_G^* w [\alpha], \alpha \in \Xi\}$.

Note that for $\Xi = \Psi^*$, it is the case that $L(H) = L(G)$ – the language generated by H is the same as the language generated by its core grammar G . So context-free grammars can be thought of as special cases of regular controlled grammars (with $\Xi = \Psi^*$). Also note that the derivation relation \Rightarrow_G is defined the same as for ordinary grammars – the only difference is the introduction of the control language.

As an example of a regular controlled grammar, consider $H_{abc} = (G_{abc}, \Xi)$, where $G_{abc} = (\{S, A, C\}, \{a, b, c\}, \Psi, P, S)$, with P containing the following rules with the following labels from Ψ :

- $0 : S \rightarrow AC,$
- $1 : A \rightarrow aAb,$
- $2 : C \rightarrow cC,$
- $3 : A \rightarrow \varepsilon,$
- $4 : C \rightarrow \varepsilon,$

and $\Xi = \{0\}\{12\}^*\{34\}$. It can be shown that $L(H_{abc}) = \{a^n b^n c^n : n \geq 0\}$, which is a non-context-free language, immediately showing that regular-controlled grammars are strictly stronger than ordinary context-free grammars. For example, the following derivation can be considered:

$$S \Rightarrow AC \Rightarrow aAbC \Rightarrow aAbcC \Rightarrow aaAbbcC \Rightarrow aaAbbccC \Rightarrow aabbccC \Rightarrow aabbcc [0121234]$$

Seeing as $0121234 \in \Xi$, it is also the case that $aabbcc \in L(H_{abc})$. After any application of rule 1 (generation of a and b), an application of rule 2 (generation of c) must follow. Eventually, an application of rule 3 must occur (erasing A), which will necessarily be followed by an application of rule 4 (erasing C). This will keep the numbers of a , b and c in balance, restricting the language of the core grammar, which is $L(G_{abc}) = \{a^m b^m c^m : m, n \geq 0\}$.

We can further introduce an additional mechanism – *appearance checking*. The idea is that we will allow certain rules to be „skipped“ if their left-hand side does not appear in the sentential form. The labels of all such rules comprise the *appearance checking set*. Let us state this notion formally:

Definition 2.2.3. A regular-controlled (context-free) grammar with appearance checking is a triple $H = (G, \Xi, W)$, where

- G, Ξ have the same meaning as in definition 2.2.1;
- $W \subseteq \Psi$ is the appearance checking set.

We must also define a special kind of derivation:

Definition 2.2.4. Let $H = (G, \Xi, W)$ be a regular-controlled grammar with appearance checking, $G = (N, \Sigma, \Psi, P, S)$, $V = N \cup \Sigma$. For any $\mu \in V^+, \lambda \in V^*, r : \alpha \rightarrow \beta \in P$, we write

$$\mu \Rightarrow_{(G, W)} \lambda [r]$$

if and only if one of the following is true:

- a) $\mu = \gamma\alpha\delta$ and $\lambda = \gamma\beta\delta$ for some $\gamma, \delta \in V^*$ (in other words, $\mu \Rightarrow_G \lambda [r]$),
- b) $r \in W$, μ does not contain α as a substring, and $\lambda = \mu$.

Note that this relation is independent of the control language Ξ – it only depends on the core grammar G and the appearance checking set W . We can also define $\Rightarrow_{(G, W)}^k, \Rightarrow_{(G, W)}^*$, and $\Rightarrow_{(G, W)}^+$ as before.

Definition 2.2.5. *The language generated by a regular-controlled grammar with appearance checking $H = (G, \Xi, W)$ is defined as $L(H) = \{w \in \Sigma^* : S \Rightarrow_{(G,W)}^* w[\alpha], \alpha \in \Xi^*\}$.*

Note that the ordinary regular-controlled grammar, specified in definition 2.2.1 earlier in this section is just a special case of a regular-controlled grammar with appearance checking, where the appearance checking set is empty ($W = \emptyset$).

Generative Power

A regular-controlled grammar (with appearance checking) is called *propagating*, if it contains no derivation rules of the form $A \rightarrow \varepsilon$, with a possible exception for $S \rightarrow \varepsilon$, as described in section 2.1.4 (S may not appear on the right-hand side of any rule in that case).

We denote the class of languages generated by regular-controlled grammars as \mathbf{rC} , and we add a subscript $_{ac}$ or a superscript $^{-\varepsilon}$ to specify grammars with appearance checking or propagating grammars, respectively. To sum this up explicitly, we use the following notation:

- \mathbf{rC} – the class of languages generated by *regular-controlled grammars*,
- $\mathbf{rC}^{-\varepsilon}$ – the class of languages generated by *propagating regular-controlled grammars*,
- \mathbf{rC}_{ac} – the class of languages generated by *regular-controlled grammars with appearance checking*,
- $\mathbf{rC}_{ac}^{-\varepsilon}$ – the class of languages generated by *propagating regular-controlled grammars with appearance checking*.

It is clear from the definitions given in this section that $\mathbf{CF} \subseteq \mathbf{rC}^{-\varepsilon} \subseteq \mathbf{rC} \subseteq \mathbf{rC}_{ac}$, and that $\mathbf{rC}^{-\varepsilon} \subseteq \mathbf{rC}_{ac}^{-\varepsilon} \subseteq \mathbf{rC}_{ac}$. In fact, almost all of these inclusions are known to be proper – the only exception is $\mathbf{rC}^{-\varepsilon} \subseteq \mathbf{rC}$, where it is an open problem whether the inclusion is proper or whether the classes are equal. Furthermore, it turns out that $\mathbf{rC}_{ac}^{-\varepsilon} \subset \mathbf{CS}$ and $\mathbf{rC}_{ac} = \mathbf{RE}$. The proofs of these claims may be found in [1].

These relations are summed up in the Hasse diagram on figure 2.1, where a dashed line is used to represent improper inclusion, and all other inclusions are proper.

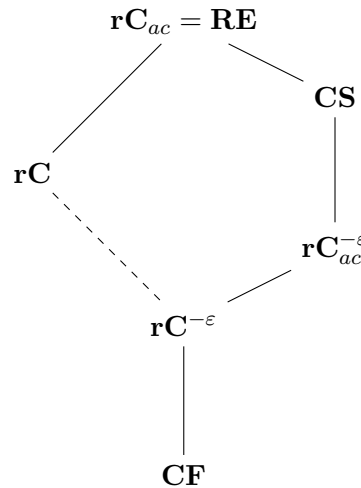


Figure 2.1: Generative power of regular-controlled grammars

2.2.2 Matrix Grammars

Matrix grammars use a slightly different approach to rule-based regulation than regular-controlled grammars – rather than specifying a language of allowed derivations, they specify sequences of rules, called *matrices*, and require that any derivation in the grammar be composed of these matrices. Let us state this formally, introducing appearance checking right away this time, as the version without appearance checking will just be a special case again:

Definition 2.2.6. *A Matrix grammar with appearance checking is a triple $H = (G, M, W)$, where*

- $G = (N, \Sigma, \Psi, P, S)$ is a context-free grammar called the core grammar,
- $M \subseteq \Psi^+$ is a finite language, the elements of which are called matrices,
- $W \subseteq \Psi$ is the appearance checking set.

We can now define a derivation in such a grammar and thereafter its language.

Definition 2.2.7. *Let $H = (G, M, W)$ be a matrix grammar with appearance checking. For any $x, y \in V^*$, we say that x derives y in one step in H , denoted by $x \Rightarrow_H y$, if and only if there exists a matrix $m = r_1 \dots r_n \in M$ such that $x \Rightarrow_{(G, W)}^n y [r_1 \dots r_n]$ (using the relation $\Rightarrow_{(G, W)}$ as defined in definition 2.2.4).*

In other words, to perform a derivation in a matrix grammar (with appearance checking), you cannot in general apply just a single rule, you must apply a whole matrix – an entire sequence of rules from a pre-defined finite set. It is, however, possible for matrices to be of length 1, in which case they correspond to single rules. We can define \Rightarrow_H^k , \Rightarrow_H^* and \Rightarrow_H^+ as usual.

Definition 2.2.8. *Let $H = (G, M, W)$ be a matrix grammar with appearance checking. The language generated by H is defined as $L(H) = \{w \in \Sigma^* : S \Rightarrow_H^* w\}$.*

If $W = \emptyset$, the grammar is simply called a *matrix grammar* (without appearance checking) and can be thought of as just a pair $H = (G, M)$. Note that a context-free grammar is a special case of a matrix grammar, where $M = \Psi$.

Generative Power

Again, a *propagating* matrix grammar (with appearance checking) is a grammar with no rules of the form $A \rightarrow \varepsilon$ (with the usual exception for $S \rightarrow \varepsilon$, as described in 2.1.4). To describe the classes of languages generated by matrix grammars, we will use the symbol \mathbf{M} , the subscript $_{ac}$ and the superscript $^{-\varepsilon}$, as before:

- \mathbf{M} – the class of languages generated by *matrix grammars*,
- $\mathbf{M}^{-\varepsilon}$ – the class of languages generated by *propagating matrix grammars*,
- \mathbf{M}_{ac} – the class of languages generated by *matrix grammars with appearance checking*,
- $\mathbf{M}_{ac}^{-\varepsilon}$ – the class of languages generated by *propagating matrix grammars with appearance checking*.

It turns out that these classes are equal to the classes generated by regular-controlled grammars – that is, the following equalities hold: $\mathbf{M} = \mathbf{rC}$, $\mathbf{M}^{-\varepsilon} = \mathbf{rC}^{-\varepsilon}$, $\mathbf{M}_{ac} = \mathbf{rC}_{ac}$, and $\mathbf{M}_{ac}^{-\varepsilon} = \mathbf{rC}_{ac}^{-\varepsilon}$. The proofs of these equalities can be found in [1].

2.2.3 Programmed Grammars

Let us now take a slightly different approach – instead of adding an extra control mechanism on top of the unchanged structure of ordinary context-free grammars, we will modify the production rules themselves. Each rule r will now contain two extra sets, subsets of Ψ , which will be the sets of rules that can follow rule r in a derivation. The first set, σ_r , called the *success field* of r , is the set of rules that can be used after using r to rewrite a part of the sentential form (so after *successfully* applying rule r), whereas the second set, φ_r , called the *failure field* of r , is the set of rules that can be used after r if r could not be applied due to its left-hand side not appearing in the sentential form (so after *failing* to apply rule r). The latter set serves as an analogue to appearance checking.

Definition 2.2.9. A programmed grammar with appearance checking is a quintuple $G = (N, \Sigma, \Psi, P, S)$, where

- N, Σ, Ψ and S are defined as usual (see definition 2.1.1),
- $P \subseteq \Psi \times N \times V^* \times 2^\Psi \times 2^\Psi$ ² is the set of production rules, with the requirement that if $(r, A, x, \sigma_r, \varphi_r), (s, A, x, \sigma_s, \varphi_s) \in P$, then $(r, A, x, \sigma_r, \varphi_r) = (s, A, x, \sigma_s, \varphi_s)$.

We denote a rule $(r, A, x, \sigma_r, \varphi_r) \in P$ as $(r : A \rightarrow x, \sigma_r, \varphi_r)$. As mentioned before, the sets $\sigma_r, \varphi_r \subseteq \Psi$ are called the *success field* and the *failure field* of r , respectively.

We will define derivations in these grammars slightly differently than usual: \Rightarrow_G will be a binary relation over $V^* \times \Psi$ rather than just V^* . It should be noted that the very same symbol (\Rightarrow_G) is used for this relation and the relation defined for phrase-structure grammars (in definition 2.1.2). The distinction between these two relations depends on the type of the grammar G – whether it's a programmed grammar (with appearance checking) or an ordinary phrase-structure grammar. In this section, we will always assume the former case.

Definition 2.2.10. Let $G = (N, \Sigma, \Psi, P, S)$ be a programmed grammar with appearance checking. For any $\mu, \lambda \in V^*$, $(r : A \rightarrow x, \sigma_r, \varphi_r) \in P$, and $s \in \Psi$, it holds that $(\mu, r) \Rightarrow (\lambda, s)$ if and only if one of the following statements holds:

- $\mu = yAz$ and $\lambda = yxz$ for some $y, z \in V^*$, and $s \in \sigma_r$;
- $\mu = \lambda$, μ does not contain A and $s \in \varphi_r$.

We can define $\Rightarrow_G^k, \Rightarrow_G^*, \Rightarrow_G^+$ as usual.

Definition 2.2.11. Let $G = (N, \Sigma, \Psi, P, S)$ be a programmed grammar with appearance checking. The language generated by G is defined as $L(G) = \{w \in \Sigma^* : (S, r) \Rightarrow_G^* (w, s) \text{ for some } r, s \in \Psi\}$.

Furthermore, we define an ordinary *programmed grammar* (without appearance checking) to be a programmed grammar with appearance checking $G = (N, \Sigma, \Psi, P, S)$ such that for all $r \in \Psi$, $\varphi_r = \emptyset$. That is, the failure field is empty for all production rules of the grammar.

Based on these definitions, we can remark that a context-free grammar is a special case of a programmed grammar – one in which for each rule, the success field contains all rules, and the failure field is empty. That is, $\sigma_r = \Psi$ and $\varphi_r = \emptyset$ for all $r \in \Psi$.

²The notation 2^Ψ is used to denote the *power set* of Ψ , that is, the set of all subsets of the set Ψ .

2.2.4 Generative power

Just like before, we will define a *propagating* programmed grammar (with appearance checking) as a programmed grammar (with appearance checking) with no rules of the form $(r : A \rightarrow \varepsilon, \sigma_r, \varphi_r)$, with the usual possible exception for a rule $(s : S \rightarrow \varepsilon, \sigma_s, \varphi_s)$, in which case S may not appear on the right-hand side of any rule.

We will use the symbol \mathbf{P} , the subscript $_{ac}$ and the superscript $^{-\varepsilon}$, just like before, to describe the classes of languages generated by the various types of programmed grammars:

- \mathbf{P} – the class of languages generated by *programmed grammars*,
- $\mathbf{P}^{-\varepsilon}$ – the class of languages generated by *propagating programmed grammars*,
- \mathbf{P}_{ac} – the class of languages generated by *programmed grammars with appearance checking*,
- $\mathbf{P}_{ac}^{-\varepsilon}$ – the class of languages generated by *propagating programmed grammars with appearance checking*.

Just like with matrix grammars, it can be shown that these classes are equal to the classes generated by regular-controlled grammars – that is, the following equalities hold: $\mathbf{P} = \mathbf{rC}$, $\mathbf{P}^{-\varepsilon} = \mathbf{rC}^{-\varepsilon}$, $\mathbf{P}_{ac} = \mathbf{rC}_{ac}$, and $\mathbf{P}_{ac}^{-\varepsilon} = \mathbf{rC}_{ac}^{-\varepsilon}$. For the proof of these equalities see [1].

Chapter 3

Parsing of Context-Free Grammars

Let us discuss some properties of context-free grammars and then introduce some models of their parsing, which we will later use as inspiration for designing parsers for regulated grammars. The only kind of grammars we will discuss in the entire chapter are context-free grammars.

3.1 Leftmost Derivations

The derivation relation defined in definition 2.1.2 allows the rewriting of any nonterminal of the sentential form. We can, however, restrict this to a specific nonterminal, such as the leftmost one. It turns out that such a restriction does not affect the language of a context-free grammar.

Definition 3.1.1. Let $G = (N, \Sigma, P, S)$ be a context-free grammar. We define the binary relation \Rightarrow_{lm} on V^* as follows: For any $\mu, \lambda \in V^*$, $\mu \Rightarrow_{lm} \lambda$ if and only if there exists a rule $(A \rightarrow \alpha) \in P$ and strings $\beta \in \Sigma^*$ and $\gamma \in V^*$, such that $\mu = \beta A \gamma$ and $\lambda = \beta \alpha \gamma$. Note that $\beta \in \Sigma^*$, so no nonterminals appear to the left of the one rewritten in the derivation – the leftmost nonterminal is rewritten.

We call this relation the *leftmost derivation*. We can now define the language of a context-free grammar G derived by leftmost derivations as $L_{lm}(G) = \{w \in \Sigma^* : S \Rightarrow_{lm}^* w\}$. It can be shown that for any context-free grammar G , $L_{lm}(G) = L(G)$ (see [5] for proof). This makes sense, as in a context-free grammar, once you generate a nonterminal A , you can derive the same strings from A , regardless of what context it appears in, as context-free rules only have a single nonterminal on their left-hand side. Therefore, whenever you have multiple nonterminals in a sentential form, it doesn't matter which one you rewrite first – it won't affect what you can do with the others.

As another example, we could define a *rightmost derivation*, \Rightarrow_{rm} , similarly to definition 3.1.1, except that we would require that $\beta \in V^*, \gamma \in \Sigma^*$ – this time, the rightmost nonterminal would be rewritten.

3.2 Pushdown Automata

We will now introduce the (*extended*) *pushdown automaton* (the PDA, or EPDA), another model for describing context-free grammars. While grammars start with a start symbol and try to generate a string using various rewritings, automata take the reverse approach – they

start with a string on their input, perform some computation over it, and eventually either accept or reject it (or, in some special cases, loop indefinitely). Automata are discussed only very briefly in this thesis – a more detailed introduction can be found in [5].

A simpler variant, the *finite automaton* (FA), is a finite state machine – it has a finite set of states and can be in exactly one of them at any given time. Always starting in a specified *initial state*, it goes through an input string from left to right, making a transition from one state to the next with each input symbol processed. The next state is defined as a function of the current state and the currently read symbol. Some of the states can be denoted as *final states*. The input string is accepted or rejected based on whether the automaton ends up in a final or a non-final state after reading the whole input. Finite automata accept exactly the class of *regular languages*, **REG** (see section 2.1.4 for definition and see [5] for proof).

A pushdown automaton is essentially a finite automaton extended with a *pushdown* – a last-in, first-out memory. In every computational step, it reads a symbol from the top of the pushdown (or a string, in the case of the *extended pushdown automaton*), and based on this data, the current state, and the current input symbol, it decides what state to move to and what string to write to the top of the pushdown. Just like finite automata, pushdown automata accept or reject a string based on whether they end up in a final or a non-final state after processing the whole input. However, we can define a variant of the PDA, which accepts the string if and only if its pushdown is empty after processing the input. Yet another variant requires both a final state and an empty pushdown.

All the aforementioned variants of the pushdown automaton (EPDA/PDA, three acceptance conditions variants) have the same accepting power – they accept exactly the class of *context-free languages*, **CF** (see section 2.1.4 for definition and see [5] for proof).

We will first define the EPDA, as the PDA can be considered a special case thereof.

Definition 3.2.1. An extended pushdown automaton (EPDA) is a septuple

$$P = (Q, \Sigma, \Gamma, \delta, z_0, q_0, F),$$

where:

- Q is a finite set of states,
- Σ is the input alphabet,
- Γ is the pushdown alphabet,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^* \rightarrow 2^{Q \times \Gamma^*}$ is the transition function,
- $z_0 \in \Gamma$ is the start pushdown symbol,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of final states.

The transition function gives a set of possible pairs of state and pushdown string that can be chosen for the current computational step, based on the current state, the current input symbol and the top of the pushdown. From a selected pair, the state becomes the current state of the EPDA, and the string gets pushed to the top of the pushdown. This will be defined more rigorously in definition 3.2.2.

The definition of a PDA is the same with a small difference in the definition of the transition function – on its input, it only has Γ rather than Γ^* :

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$$

That is, an ordinary PDA only reads a single symbol from the pushdown when deciding on the next step.

To get the full picture of the purpose of each component, we need a couple more definitions – the *configuration*, which contains not only the current state of the EPDA, but also its unprocessed input and pushdown contents; the *transition relation*, analogous to the derivation relation of grammars, which describes what configurations can be reached from another configuration in a single computational step; and the *language* of an EPDA:

Definition 3.2.2. Let $P = (Q, \Sigma, \Gamma, \delta, z_0, q_0, F)$ be an EPDA. A configuration of P is any triple $\xi \in Q \times \Sigma^* \times \Gamma^*$. A configuration of the form (q_0, w, z_0) for some $w \in \Sigma^*$ is called an initial configuration, while a configuration of the form (f, ε, γ) for some $f \in F, \gamma \in \Gamma^*$ is called a final configuraion when considering acceptance by a final state. We can also consider acceptance by empty pushdown, in which case a final configuration is of the form $(q, \varepsilon, \varepsilon)$ for any $q \in Q$, or acceptance by a final state and empty pushdown, in which case a final configuration is of the form $(f, \varepsilon, \varepsilon)$ for any $f \in F$. We define the transition relation \vdash_P on the set of configurations as follows:

$$(q, w, \beta) \vdash_P (q', w', \beta') \Leftrightarrow w = aw', \beta = \alpha\eta, \beta' = \gamma\eta, (q', \gamma) \in \delta(q, a, \alpha), \eta \in \Gamma^*.$$

We can also define \vdash^k, \vdash^* and \vdash^+ as before.

We will define the language of P in three ways, depending on whether we accept strings by a final state, by empty pushdown, or by both – these three languages will in general be different from each other for a particular P , and will respectively be denoted as $L_f(P), L_\varepsilon(P)$, and $L_{f\varepsilon}(P)$. In each case, it will be the set of strings that allow the EPDA to move from the initial configuration to a final configuration of the corresponding type:

- $L_f(P) = \{w \in \Sigma^* : (q_0, w, z_0) \vdash^* (f, \varepsilon, \gamma), f \in F, \gamma \in \Gamma^*\};$
- $L_\varepsilon(P) = \{w \in \Sigma^* : (q_0, w, z_0) \vdash^* (q, \varepsilon, \varepsilon), q \in Q\};$
- $L_{f\varepsilon}(P) = \{w \in \Sigma^* : (q_0, w, z_0) \vdash^* (f, \varepsilon, \varepsilon), f \in F\};$

Although in general, the three languages defined by a particular EPDA P (that is, $L_f(P), L_\varepsilon(P)$, and $L_{f\varepsilon}(P)$) will not be the same, it can be shown that for any P and any two selected acceptance conditions $a, b \in \{f, \varepsilon, f\varepsilon\}$, there exists an EPDA P' such that $L_a(P) = L_b(P')$, so no matter which acceptance condition we choose, we get the same class of languages. Furthermore, it can be shown that for any EPDA P , there is an equivalent PDA P' , regardless of which specific acceptance conditions we choose, and conversely, PDAs are a special case of EPDAs, so PDAs are as computationally strong as EPDAs. The proof of these claims can again be found in [5].

Note that EPDAs (and PDAs) are nondeterministic in general – for any particular configuration (q, w, α) , where w starts with a , we can make a move according to $\delta(q, a, \beta)$ or $\delta(q, \varepsilon, \beta)$ for any β which is a prefix of α – this can be many sets in general, each of which can contain multiple pairs (q', γ) . Even if we restrict ourselves to PDAs, that is we decide on a move based on a symbol $z \in \Gamma$, which is just the first symbol of α , we still get

two sets $(\delta(q, a, z))$ and $(\delta(q, \varepsilon, z))$ to choose from, each of which may contain multiple pairs (q', γ) . If we add the requirement that there can be at most one such pair in the union of all applicable rules, we get *deterministic EPDAs*, or *deterministic PDAs*, shortened to *DEPDAs* and *DPDAs*.

DEPDAs and DPDAs are strictly weaker than EPDAs and PDAs – the class of languages accepted by them (by a final state), called the *deterministic context-free languages*, denoted by **DCF**, is a strict subclass of context-free languages: **REG** \subset **DCF** \subset **CF**. Furthermore, the different acceptance conditions no longer lead to the same language classes – for example, no DEPDA accepting with empty pushdown can accept the language $\{a^i b^j : j = i \vee j = 0\}$ (see [5] for proof).

In the two following sections, we will describe two approaches to transforming context-free grammars into equivalent pushdown automata, which are used as a basis for practical parsing algorithms.

3.3 Top-Down Parsing

In top-down parsing, we start with the start symbol S on the pushdown and try to derive the input string on the pushdown according to the production rules of the grammar, while also checking that the input corresponds to what was generated on the pushdown.

The rules of the pushdown automaton we will create can be divided into two types:

- *Expand* – when a nonterminal A is on top of the pushdown, expand it according to a production rule $A \rightarrow \alpha$;
- *Pop* – when a terminal a is on top of the pushdown, remove it from the pushdown while also reading the same terminal from the input.

Note that for this to work, we need to push the right-hand sides of the production rules to the pushdown in such a way, that the leftmost symbol will be nearest to the top of the pushdown – this happens to coincide with how we defined a computational step of the EPDA (which the PDA is a special case of), so we don't need to do any explicit string reversal.

Formally, this approach to parsing can be specified as follows. Let $G = (N, \Sigma, P, S)$ be a context-free grammar. We will define a PDA M accepting with empty pushdown, such that $L(G) = L_\varepsilon(M)$, as $M = (\{q\}, \Sigma, N \cup \Sigma, \delta, S, q, \emptyset)$, with δ defined as follows:

- $(q, \alpha) \in \delta(q, \varepsilon, A)$ for any rule $(A \rightarrow \alpha) \in P$ – the *expand* moves;
- $\delta(q, a, a) = \{(q, \varepsilon)\}$ for all $a \in \Sigma$ – the *pop* moves.

It can be proven by mathematical induction, that for any $A \in N, w \in \Sigma^*$, the following holds:

$$A \Rightarrow^* w \Leftrightarrow (q, w, A) \vdash^* (q, \varepsilon, \varepsilon)$$

Plugging in $A = S$, we get $L(G) = L_\varepsilon(M)$.

Note that with this approach, you are always simulating a leftmost derivation of the input string on the pushdown – you are always rewriting the nonterminal on top of the pushdown, which corresponds to the left of the unprocessed sentential form.

This approach can serve as a basis for parsing context-free languages, but we must come up with a resolution to the possible nondeterminism: with a nonterminal $A \in N$ on top of the pushdown, we can expand it according to any rule of the form $A \rightarrow \alpha \in P$, and we need to know which rule to use. One possible approach to this, called LL parsing, is described in section 3.5.

3.4 Bottom-Up Parsing

In this section we will introduce a converse approach – we will push symbols of the input string onto the pushdown, and once there is a string on top of the pushdown that corresponds to the right-hand side of a production rule, we can replace it with the left-hand side of said rule. If we manage to reduce the input string all the way to S , we can move to a final state.

Note that with this approach, we will move input symbols to the pushdown one by one, resulting in the leftmost symbols being nearer the bottom. However, we have defined the top of the stack as the left side of the pushdown string in a configuration (see definition 3.2.2). Therefore, when reducing substrings on the stack according to production rules, we must consider the reversal of the right-hand sides. (We would not need to do this reversal if we defined the top of the pushdown to be on the right in the transition relation.)

Let us state these notions formally. Let $G = (N, \Sigma, P, S)$ be a context-free grammar. We will define an EPDA M accepting with final state, such that $L(G) = L_f(M)$, as $M = (\{q, r\}, \Sigma, N \cup \Sigma \cup \{\#\}, \delta, \#, q, \{r\})$, with δ defined as follows:

- $(q, A) \in \delta(q, \varepsilon, \alpha^R)$ for any rule $A \rightarrow \alpha \in P$ – the *reduce* moves;
- $\delta(q, a, \varepsilon) = \{(q, a)\}$ for all $a \in \Sigma$ – the *shift* moves;
- $\delta(q, \varepsilon, S\#) = \{(r, \varepsilon)\}$ – the single end rule.

Again, it can be shown by induction that $L(G) = L_f(M)$.

Note that this method simulates the reverse of a rightmost derivation of the input string on the pushdown – at the end, you reduce the sentential form on the pushdown to S , and for any nonterminals appearing there, the ones nearest to the top are the ones that were reduced to the latest, which would mean they appear the earliest in the actual derivation. Seeing as we’re working with the reversal of the actual sentential form (due to how we define a computational step of an EPDA), the top of the pushdown corresponds to the right side of the sentential form, so the rightmost nonterminals are derived from first.

This method has a similar problem as top-down parsing from the perspective of determinism – the right-hand sides of various rules may be prefixes of one another, which means that from a particular configuration, you might be able to perform several different reductions, or you may be able to wait until more symbols are shifted, after which you would be able to perform different reductions. To base a parsing algorithm on this method, we need a mechanism to determine which rules should be chosen for reductions. One such mechanism is offered by so-called LR parsers, which are not discussed in this thesis, but more information on them can be found in [4].

3.5 LL Parsing

We can now define a variant of top-down parsing that can be used to parse a subset of context-free languages efficiently (in linear time) – *LL parsing*, with the LL denoting the fact that this algorithm reads the input from *Left* to right, and constructs a *Leftmost* derivation – in every step, the leftmost nonterminal of the current sentential form is rewritten. More specifically, we will talk about *LL(1) parsing*, meaning that 1 symbol of the unprocessed input is used to help decide which rule to use. It is possible to define $LL(k)$ parsers analogously, and it turns out that they get more powerful with increasing k , but we will not deal with that in this thesis.

The idea is that we will enhance the top-down parsing algorithm with a table which will help decide which rule to use whenever multiple rules are applicable. The table will decide based on the nonterminal A on top of the pushdown, and the terminal a in the front of the unprocessed input. We will also use the symbol $\$$ to denote the end of the input. The table is based on a set *Predict* of terminals that can start a string derivable from A and the rest of the sentential form following it, if A is rewritten using a particular production rule. We will describe this set in the following section.

The *Predict* set

Definition 3.5.1. Let $G = (N, \Sigma, \Psi, P, S)$ be a context-free grammar and $r \in \Psi$ the label of a production rule $A \rightarrow \alpha \in P$. We define the set $Predict(r)$ (alternatively denoted as $Predict(A \rightarrow \alpha)$) as follows:

$$\begin{aligned} Predict(r) = \{ & a \in \Sigma : S \Rightarrow^* \beta A \beta' \wedge \beta \alpha \beta' \Rightarrow^* \beta a \gamma, \text{ where } \beta, \beta', \gamma \in V^* \} \\ & \cup \{ \$: S \Rightarrow^* \beta A \beta' \wedge \beta \alpha \beta' \Rightarrow^* \beta, \text{ where } \beta, \beta', \gamma \in V^* \} \end{aligned}$$

The idea is that if we have a nonterminal A on top of the pushdown and a terminal a at the start of the unprocessed input, we must apply a rule $r : A \rightarrow \alpha$ such that $a \in Predict(r)$. Otherwise, there is no way we can continue the derivation and end up with a as the starting symbol of the remainder.

Ignoring ε -rules, $Predict(A \rightarrow \alpha)$ is simply the set of possible leftmost terminals of strings derivable from A starting with $A \rightarrow \alpha$. However, we must consider whatever may follow A , if it can eventually be rewritten to ε starting with $A \rightarrow \alpha$. We also include $\$$ if A can occur at the end of a sentential form and can also be rewritten to ε after using this rule.

We can put *Predict* in more easily readable terms and also describe how to compute it at the same time, if we define a few other sets first:

Definition 3.5.2. Let $G = (N, \Sigma, \Psi, P, S)$ be a context-free grammar. For any string $\mu \in V^*$, we define the following two sets:

- $Empty(\mu) = \{\varepsilon : \mu \Rightarrow^* \varepsilon\},$
- $First(\mu) = \{a \in \Sigma : \mu \Rightarrow^* a\gamma, \gamma \in V^*\}.$

Furthermore, for any nonterminal $A \in N$, we may define the following set:

- $Follow(A) = \{a \in \Sigma : S \Rightarrow^* \beta A a \gamma, \text{ where } \beta, \gamma \in V^*\} \cup \{ \$: S \Rightarrow^* \beta A, \beta \in V^* \}.$

From the definition, $Empty(\mu)$ will be $\{\varepsilon\}$ if the empty string can be derived from μ , and it will be empty otherwise. $First(\mu)$ is the set of terminals that can begin a string derived from μ . Both of these can be precomputed for sentential forms of length 1 (that is, members of V) and then used to compute the value for any sentential form.

The set $Follow(A)$ is the set of terminals that can immediately follow A in a sentential form derivable in the grammar. If A can occur at the end of such a sentential form, $Follow(A)$ also includes $\$$. This set can be computed based on the production rules of the grammar, using any rules where A appears on the right-hand side and the strings following it.

We will now provide an alternative definition of $Predict$ which can be shown to coincide with the previous one. In a context-free grammar $G = (N, \Sigma, \Psi, P, S)$ we define $Predict(A \rightarrow \alpha)$ for any $A \rightarrow \alpha \in P$ as follows:

- If $Empty(\alpha) = \emptyset$, then $Predict(A \rightarrow \alpha) = First(\alpha)$;
- If $Empty(\alpha) = \{\varepsilon\}$, then $Predict(A \rightarrow \alpha) = First(\alpha) \cup Follow(A)$.

Having defined this set, we can define an LL(1) grammar:

Definition 3.5.3. *A context-free grammar $G = (N, \Sigma, \Psi, P, S)$ is an LL(1) grammar if and only if for every $a \in \Sigma$ and every $A \in N$, there is at most one rule $r \in \Psi$, such that A is the left-hand side of r and $a \in Predict(r)$.*

We may sometimes refer to these simply as *LL grammars*, as the general case of $LL(k)$ grammars is not covered in this thesis.

So if we encounter a pair A, a such that there are multiple A -rules in P with a in their $Predict$ sets, the grammar is not an LL(1) grammar and the LL(1) parsing algorithm cannot be used to parse it. It is possible to transform some non-LL(1) grammars into equivalent LL(1) grammars, but this is not possible for all context-free grammars. In fact, if we denote by **LL(1)** the class of languages generated by LL(1) grammars, it can be shown that it is a strict subclass of the context-free languages: **REG** \subset **LL(1)** \subset **CF**.

The LL Table

Based on the $Predict$ set, we can now define the LL table for a particular grammar G . We will denote the entry of the table corresponding to a nonterminal $A \in N$ and a terminal $a \in \Sigma \cup \{\$\}$ as $T[A, a]$.

Definition 3.5.4. *Let $G = (N, \Sigma, \Psi, P, S)$ be an LL(1) grammar. For any $a \in \Sigma \cup \{\$\}$, $A \in N$, we define $T[A, a]$ as the single rule $r : A \rightarrow \alpha \in P$ such that $a \in Predict(r)$, or as empty if no such rule exists. There cannot be more than one such rule by the definition of an LL(1) grammar (3.5.3).*

We can therefore construct the LL table by first constructing the $Predict$ sets for all rules (which would be itself preceded by computing the $First$, $Empty$ and $Follow$ sets) and then simply going through all terminals and nonterminals and asking how many applicable rules there are. During this process, we can also detect if a particular grammar is not LL – if we end up with any pair A, a with multiple applicable rules.

Having built the LL table, we can now go through with describing the LL parsing algorithm itself.

The LL Parsing Algorithm

The description of the algorithm itself can be seen labeled as algorithm 1. In the description of this algorithm, we will denote the bottom of the stack by $\#$ and the end of the input by $\$$. We will also use a to denote the current input symbol, that is, the first symbol of the unprocessed input (or $\$$ if the whole string has already been processed), and X to denote the symbol on top of the pushdown (which can be a nonterminal, a terminal, or $\#$).

Algorithm 1: LL table parsing algorithm

Input : The LL table T for a grammar $G = (N, \Sigma, \Psi, P, S)$, a string $w \in \Sigma^*$
Output: A sequence of rules to produce a leftmost derivation of w in G , if $w \in L(G)$, or an error otherwise.

```

Pushdown.Push( $\#$ );
Pushdown.Push( $S$ );
while  $\neg$ Pushdown.Empty() do
     $X \leftarrow$  Pushdown.Top();
     $a \leftarrow$  Input.Current();
    switch  $X$  do
        case  $X = \#$  do
            if  $a = \$$  then
                | SUCCESS – the input string was successfully parsed;
            else
                | ERROR – unprocessed input remaining after the derivation is
                | finished;
        case  $X \in \Sigma$  do
            if  $X = a$  then
                | Pushdown.Pop();
                | Input.Move();
            else
                | ERROR – string generated on pushdown does not agree with string
                | on input;
        case  $X \in N$  do
            if  $r : X \rightarrow \alpha \in T[X, a]$  then
                | Pushdown.Pop();
                | Pushdown.Push( $\alpha$ );
                | Output( $r$ );
            else
                | ERROR – no  $A$ -rule  $r$  with  $a \in \text{Predict}(r)$ ;

```

We use the following operations in the algorithm:

- $\text{Pushdown.Push}(\gamma)$ – pushes the string γ to the pushdown, starting from the right;
- $\text{Pushdown.Top}()$ – returns the symbol currently on top of the pushdown;
- $\text{Pushdown.Pop}()$ – removes the top of the pushdown from the pushdown;
- $\text{Pushdown.Empty}()$ – returns *True* if the pushdown is empty, *False* otherwise;
- $\text{Input.Current}()$ – returns the first symbol of the unprocessed input string;

- *Input.Move()* – marks the first symbol of the unprocessed input string as processed, removing it from the unprocessed portion;
- *Output(x)* – prints x to the output (x will always be a rule label in this case).

We also use the following two keywords to mark the end of the algorithm:

- *SUCCESS* – the algorithm has successfully parsed the input string and the rule labels printed previously can be used to reconstruct the leftmost derivation of the input string;
- *ERROR* – the algorithm has ruled that the input is not in the language generated by the supplied grammar.

Chapter 4

Deterministic Parsing of Programmed Grammars

We would now like to design an efficient algorithm for parsing programmed grammars. In fact, we will settle for a parser that only works with some, not all programmed grammars – we know that $\mathbf{P}_{ac}^\varepsilon = \mathbf{RE}$, so parsing all of them would be way too ambitious. We would like to base this algorithm on the ideas of LL parsing (as described in section 3.5), but in such a way that will allow us to parse a greater class of languages.

We will first summarize the basic ideas behind a proposed extension in section 4.1, then formalize these ideas by defining the table used by the parsing algorithm in section 4.2, and describing a naive version of the algorithm in section 4.3, which also includes a quick analysis of the algorithm. This naive version will turn out to have a serious flaw – it may not halt for certain inputs. This problem is further analysed in section 4.4, at the end of which an improved version of the algorithm is proposed for propagating programmed grammars without appearance checking.

An implementation of all the algorithms presented in this chapter can be found on the accompanying CD, as described in appendix A.

4.1 Basic Ideas

One straightforward way of extending the LL parsing algorithm to programmed grammars is to consider the success and failure fields of the last used rule r , σ_r and φ_r , before consulting a table. If the corresponding set contains at most one rule, it's clear what rule needs to be applied next. Otherwise, we can consult a table, but this time, instead of just deciding based on the topmost nonterminal A on the pushdown and the leftmost unprocessed terminal a on the input, we can also consider the set ρ (corresponding to σ_r or φ_r of the previous rule r) of rule labels which we are restricting ourselves to. The table therefore needs to be three-dimensional, with entries of the form $T[A, a, \rho]$. This will allow us to make choices we wouldn't be able to do with LL parsing, because while there might be multiple A -rules r' with $a \in \text{Predict}(r')$, it might happen that only one of them is in ρ . We will usually use the symbol ρ to denote the set of rule labels being selected from, regardless of whether it is the success field or the failure field of the previous rule.

We will call the resulting algorithm the *Table-resort* algorithm, or the *TR* algorithm for short, as the algorithm first checks whether there is at most one rule in the set ρ of

currently permitted rules, and only *resorts* to the table otherwise. This three-dimensional table, described precisely in section 4.2, will be called a *TR Table*.

For the purposes of determinism, we will always need to rewrite the leftmost nonterminal, or the leftmost occurrence of the left-hand side of the selected rule. However, unlike with context-free grammars, rewriting the nonterminals in different orders can lead to different rules being applicable after individual derivation steps, so this focus of leftmost occurrences of nonterminals can lead to the language of the grammar changing. Therefore, the language of strings accepted by the algorithm using a table based on a particular grammar G , which we will denote by $L_{TR}(G)$, may be different from the actual language $L(G)$ of all strings generated by the grammar. It will always be the case that $L_{TR}(G) \subseteq L(G)$, as the algorithm will only ever accept strings that are generated by the grammar, but this inclusion will sometimes be proper.

4.2 The TR Table

We will now define $T[A, a, \rho]$ for all relevant A, a, ρ , where A is the topmost nonterminal on the pushdown and a is the leftmost unprocessed terminal on the input, just like in the LL table, while ρ is the set of rules we are allowed to select from. The table will be defined very similarly to an LL table, just restricting the considered rules to ones in ρ – if we used T' to denote a variant of the LL table of the underlying context-free grammar, in which $T'[A, a]$ can contain the set of all the A -rules r' such that $a \in \text{Predict}(r')$ (not just the one, but multiple ones, allowing us to construct an „LL table“ for non-LL context-free grammars), we could simply use a definition akin to $T[A, a, \rho] = T'[A, a] \cap \rho$. In other words, an entry for $T[A, a, \rho]$ will contain rules all rules $r \in \rho$ of the form $A \rightarrow \alpha$ such that $a \in \text{Predict}(r)$.

For practical reasons, we also need to require that $|T[A, a, \rho]| \leq 1$, as nondeterminism could arise in the table-resort algorithm otherwise. If any such entries appear in a grammar, we must reject the grammar (although this does not necessarily mean that the language of the grammar cannot be generated by another grammar, which can be processed by the table-resort algorithm).

We also need to consider the selection of the first rule – programmed grammars add no extra restrictions on this compared to ordinary context-free grammars, so we will have to require that just like with LL grammars, there is at most one S -rule r_a for each $a \in \Sigma \cup \{\$\}$, such that $a \in \text{Predict}(r_a)$. However, this is not guaranteed by what was discussed in the previous paragraphs.

Let $\Psi_S := \{r \in \Psi : \text{lhs}(r) = S\}$ be the set of possible starting rules. This is the set we must select the first rule from, but we have no assurance that $\Psi_S = \sigma_r$ or $\Psi_S = \varphi_r$ for some $r \in \Psi$, so the TR table as described so far may not contain the information necessary for this selection, and the uniqueness of S -rules r_a such that $a \in \text{Predict}(r_a)$ is not guaranteed. We could resolve this by adding entries of the form $T[S, a, \Psi_S]$ to the table, but we will take a different approach – we will add an *implicit starting rule* $(r_0 : S' \rightarrow S, \Psi_S, \emptyset)$ ¹ to the grammar, where S' will be the new start symbol of the grammar.

This can be done for any programmed grammar and it has no effect on the grammar's generated language. There are grammars which would be TR under the original, more

¹ We cannot set $\sigma_{r_0} = \Psi$, as we would then include Ψ among the sets for which TR table entries are computed, suddenly requiring that for each $A \in N, a \in \Sigma \cup \{\$\}$, there is at most one A -rule $r_{A,a}$ among *all* rules, such that $a \in \text{Predict}(r_{A,a})$, effectively asking that the underlying grammar be LL. It is for this reason that in general, if Ψ appears as σ_r or φ_r for any rule $r \in \Psi$, the grammar is TR only if the underlying context-free grammar is LL.

relaxed requirements, but will no longer be TR after this rule is added. However, this will happen exactly when it might not be clear what rule to start the derivation of some particular input with, so this is exactly the sort of nondeterminism we are trying to combat. We will therefore require that each TR grammar has an implicit starting rule like the one described above, although we will often not mention explicitly it in example grammars, simply relying on the fact that it can be done.

With these ideas in mind, let us now formally define *TR grammars* and the TR table:

Definition 4.2.1. Let $G = (N, \Sigma, \Psi, P, S)$ be a programmed grammar (with appearance checking) with an implicit starting rule and G' be its underlying context-free grammar (so G without the success and failure fields in the rules). Then G is a TR grammar if and only if for each $A \in N$, $a \in \Sigma \cup \{\$$ and $\rho \in \{\sigma_r, \varphi_r : r \in \Psi\}$, there is at most one rule $r \in \rho$, such that A is the left-hand side of r and $a \in \text{Predict}(r)$, where *Predict* is computed on the underlying context-free grammar G' .

Definition 4.2.2. Let $G = (N, \Sigma, \Psi, P, S)$ be a TR grammar. For any $a \in \Sigma \cup \{\$$, $A \in N$, $\rho \in \{\sigma_r, \varphi_r : r \in \Psi\}$, we define $T[A, a, \rho]$ as the single rule $r : A \rightarrow \alpha \in \rho$, such that $a \in \text{Predict}(r)$, or as empty if no such rule exists. There cannot be more than one such rule by the definition of a TR grammar (4.2.1).

Seeing as we are indexing the table by subsets of Ψ , it might appear that the table will grow exponentially with the number of production rules, but notice that we are only using sets that are either Ψ_S or correspond to σ_r or φ_r of some production rule r , meaning there will be at most $2|P| + 1$ such relevant subsets, so the size will only grow linearly with the number of rules in the grammar (as well as with the number of nonterminals and with the number of terminals).

Restricting the applicable rules to r such that $a \in \text{Predict}(r)$ is valid, as by definition of $\text{Predict}(r)$, no other rules can be used to derive a string beginning with a . However, this restriction may be too permissive for some grammars, as *Predict* is defined for context-free rules and the computation of the set does not take the success and failure fields into account. To make this clear, let us define a set $\text{Predict}_P(r)$ for each $r \in \Psi$, which will take these fields into account:

Definition 4.2.3. Let $G = (N, \Sigma, \Psi, P, S)$ be a programmed grammar (with appearance checking) and $r \in \Psi$ the label of a production rule $(r : A \rightarrow \alpha, \sigma_r, \varphi_r) \in P$. We define the set $\text{Predict}_P(r)$ as follows:

$$\begin{aligned} \text{Predict}_P(r) = & \{a \in \Sigma : (S, r_0) \Rightarrow^* (\beta A \beta', r) \wedge (\beta \alpha \beta', r_1) \Rightarrow^* (\beta a \gamma, r_2), \\ & \beta, \beta', \gamma \in V^*, r_1 \in \sigma_r, r_0, r_2 \in \Psi\} \\ & \cup \{\$: (S, r_0) \Rightarrow^* (\beta A \beta', r) \wedge (\beta \alpha \beta', r_1) \Rightarrow^* (\beta, r_2), \\ & \beta, \beta', \gamma \in V^*, r_1 \in \sigma_r, r_0, r_2 \in \Psi\} \end{aligned}$$

This is really just an analogy to the *Predict* set for ordinary context-free grammars – for an A -rule r , where $A \in N$, $\text{Predict}_P(r)$ is the set of terminals that can, in some derivation of the programmed grammar (not just the underlying context-free grammar), eventually stand where A currently stands in the sentential form, provided that r is the very next rule applied. A derivation in a programmed grammar can be thought of as a special case of a derivation in the underlying context-free grammar, so it is to be expected that $\text{Predict}_P(r) \subseteq \text{Predict}(r)$ for any given rule $r \in \Psi$. However, this inclusion is sometimes

strict: Consider a grammar $G = (\{S, A, B\}, \{a, b\}, \{0, 1, 2, 3, 4\}, P, S)$ with the following production rules in P :

- 0 : $S \rightarrow AB, \{1, 2\}, \emptyset$
- 1 : $A \rightarrow \varepsilon, \{3\}, \emptyset$
- 2 : $A \rightarrow a, \{4\}, \emptyset$
- 3 : $B \rightarrow b, \{0\}, \emptyset$
- 4 : $B \rightarrow a, \{0\}, \emptyset$

Considering just the underlying context-free grammar, clearly $a \in \text{Predict}(1)$, as B can be rewritten to a . However, in the actual programmed grammar, this will never happen – an application of rule 1 will be followed by an application of rule 3, and the only string that can be generated this way is b , so $a \notin \text{Predict}_P(1)$.

This demonstrates that $\text{Predict}(r)$ can contain terminals that can not actually be derived at the start of a string when deriving using r , so we will reject grammars that could be parsed by this algorithm perfectly well if we used better approximations of the Predict_P sets. However, as the following theorem shows, we cannot compute the Predict_P sets algorithmically in the general case:

Theorem 4.2.1. *Let $\text{Predict}_P = \{\langle G, a, r \rangle : a \in \text{Predict}_P(r)\}$ denote the decision problem of whether in a particular programmed grammar with appearance checking $G = (N, \Sigma, \Psi, P, S)$, for a particular terminal $a \in \Sigma$ and for a particular rule $r \in \Psi$ it is the case that $a \in \text{Predict}_P(r)$. Then Predict_P is undecidable.*

Proof. We will prove this by reduction from $NEP = \{\langle M \rangle : L(M) \neq \emptyset\}$, the nonemptiness problem for Turing Machines. For a given Turing Machine M , we must construct a triple G, a, r , where G is a programmed grammar with appearance checking and $a \in \Sigma$, $r \in \Psi$, such that $L(M) \neq \emptyset \Leftrightarrow a \in \text{Predict}_P(r)$.

From M , we can easily construct a Turing Machine M' such that $L(M') = \{a\}L(M)$. Clearly, $L(M) = \emptyset \Leftrightarrow L(M') = \emptyset$. We can also construct a programmed grammar with appearance checking G' equivalent to M' – that is, $L(G') = L(M')$ (see [3] for the proof that this is possible). Then we construct a grammar G from G' by the addition of a new start symbol, S , and a rule $(r : S \rightarrow S', \Psi, \emptyset)$, where S' is the start symbol of G' (assuming, without loss of generality, that S and r don't yet represent anything in the grammar G'). This addition will clearly not affect the language of G' , so $L(G) = L(G') = L(M') = \{a\}L(M)$. Clearly, any string generated by G will begin with a , and the derivation must start with the rule r . Therefore, $a \in \text{Predict}_P(r) \Leftrightarrow L(G) \neq \emptyset$, and furthermore, $L(G) \neq \emptyset \Leftrightarrow L(M) \neq \emptyset$. By the transitive property of equivalence, $a \in \text{Predict}_P(r) \Leftrightarrow L(M) \neq \emptyset$. \square

This means that there are limits to how well we can restrict the set of rules applicable to a nonterminal in a leftmost position based on the first unprocessed terminal. However, it does not mean that we cannot do better than $\text{Predict}(r)$ – this approach was chosen for its simplicity and is the only one used in this thesis, but other approaches can be explored.

4.3 A Naive Table-resort Algorithm

Employing the TR table defined in the previous section, we can now define a simple extension of the LL parsing algorithm with the ideas introduced in section 4.1. The algorithm itself is described as algorithm 2.

We will assume that the input grammar will have an implicit starting rule labelled r_0 , which can be the unique rule with the start symbol as its left-hand side, or can be added as suggested in section 4.2 if there is no such rule. This is mostly useful so that Ψ_S is included among the sets for which entries of the TR table are defined and it is therefore clear what rule to start with.

As mentioned before, the algorithm only resorts to the table when the set ρ of allowed rules contains more than one rule. When selecting a rule from the table, the algorithm decides based on the following information:

- The nonterminal on the top of the pushdown, A ,
- The latest input, a terminal a ,
- The set from which the next rule is to be selected, ρ (which corresponds to σ_r or φ_r of the previous rule r , depending on whether its application was successful, or to Ψ at the very beginning of the algorithm).

When indexed, $T[A, a, \rho]$ will contain at most a single rule label r , such that A is on the right-hand side of the corresponding rule, $r \in \rho$ and $a \in \text{Predict}(r)$, as it is required that the input grammar be a TR grammar. The last used rule is not used without being rewritten first, so it doesn't matter very much how we initialize it for now.

The algorithm utilizes a pushdown with the start symbol $\#$ and an input, the end of which will be denoted by $\$$. The following notation is also used:

- a – the latest input, a terminal;
- X – the symbol at the top of the pushdown (can be a terminal, a nonterminal, or $\#$ – the bottom of the pushdown);
- r – the rule applied in the previous step, or the rule being currently applied after selection;
 - The individual components of the rule to be applied are denoted as follows:
 $(r : A \rightarrow \alpha, \sigma_r, \varphi_r)$
- ρ – the set from which a next rule must be selected.

We also use the same operations and keywords as described in section 3.5 for algorithm 1, with the addition of the following operations:

- *Pushdown.Replace*(A, α) – replaces the topmost instance of the nonterminal A with the string α , with the leftmost symbol of α being placed nearest to the top.
- $\rho.\text{Get}()$ – returns any member of the set ρ – is only be used when $|\rho| = 1$ so that it's always clearly defined which member to select;

The following sections will provide some basic analysis of the algorithm, such as its accepting power. Note that as mentioned before, the algorithm may never halt for certain inputs and certain grammars, which is discussed in more depth in section 4.4.

Algorithm 2: Table-resort algorithm

Input : The TR table T for a TR $G = (N, \Sigma, \Psi, P, S)$ with an implicit starting rule labelled r_0 , a string $w \in \Sigma^*$

Output: A sequence of rules to produce a derivation of w in G , if $w \in L_{TR}(G)$, or an error otherwise.

```
 $r \leftarrow r_0;$ 
 $\rho \leftarrow \Psi_S;$ 
 $Pushdown.Push(\#);$ 
 $Pushdown.Push(S);$ 
while  $\neg Pushdown.Empty()$  do
   $X \leftarrow Pushdown.Top();$ 
   $a \leftarrow Input.Current();$ 
  switch  $X$  do
    case  $X = \#$  do
      if  $a = \$$  then
         $SUCCESS$  – the input string has been successfully processed, printed
        rule labels can be used to construct a derivation of  $w$  in  $G$ ;
      else
         $ERROR$  – unprocessed input remaining after the derivation is
        finished;
    case  $X \in \Sigma$  do
      if  $X = a$  then
         $Pushdown.Pop();$ 
         $Input.Move();$ 
      else
         $ERROR$  – string generated on pushdown does not agree with string
        on input;
    case  $X \in N$  do
      switch  $|\rho|$  do
        case  $0$  do
           $ERROR$  – no rule to apply;
        case  $1$  do
           $(r : A \rightarrow \alpha, \sigma_r, \varphi_r) \leftarrow \rho.Get();$ 
          if  $A$  not on pushdown then
             $\rho \leftarrow \varphi_r;$ 
          else
             $Pushdown.Replace(A, \alpha);$ 
             $Output(r);$ 
             $\rho \leftarrow \sigma_r;$ 
          otherwise do
            if  $(r : A \rightarrow \alpha, \sigma_r, \varphi_r) \in T[X, a, \rho]$  then
               $Pushdown.Replace(A, \alpha);$ 
               $Output(r);$ 
               $\rho \leftarrow \sigma_r;$ 
            else
               $ERROR$  – no rule to apply;
```

4.3.1 Language Discrepancy

As mentioned before, one problem of the algorithm is that for some grammars G , the language accepted by this algorithm, denoted by $L_{TR}(G)$, may differ from the language $L(G)$ of the grammar from which we build the TR table. Consider these simple examples: Let $G_1 = (\{S, A\}, \{a, b\}, \{0, 1, 2\}, P, S)$ and let P consist of the following rules:

$$\begin{aligned} 0 : S &\rightarrow AA, \{1\}, \emptyset \\ 1 : A &\rightarrow a, \{2\}, \emptyset \\ 2 : A &\rightarrow b, \{0\}, \emptyset \end{aligned}$$

Clearly, $L(G_1) = \{ab, ba\}$, however, $ba \notin L_{TR}(G_1)$ – after rewriting S to AA , rule 1 must be used, and the algorithm will insist on applying it to the leftmost nonterminal, so only ab can be accepted. Next, let $G_2 = (\{S, A, B\}, \{a, b\}, \{0, 1, 2, 3\}, P, S)$ and let P consist of the following rules:

$$\begin{aligned} 0 : S &\rightarrow AB, \{1, 2\}, \emptyset \\ 1 : A &\rightarrow a, \{3\}, \emptyset \\ 2 : B &\rightarrow a, \{1\}, \emptyset \\ 3 : B &\rightarrow b, \{1\}, \emptyset \end{aligned}$$

It can be shown that $L(G_2) = \{aa, ab\}$, but $aa \notin L_{TR}(G)$. This is because after rewriting S to AB , the algorithm will be selecting the next rule from the set $\{1, 2\}$, so it will only consider rules with the topmost nonterminal on the pushdown (that is, A) as their left-hand side, so rule 1 will always be chosen with a on the input. Finally, let $G_3 = (\{S, A, B\}, \{a, b, c\}, \{0, 1, 2, 3\}, P, S)$ and let P consist of the following rules:

$$\begin{aligned} 0 : S &\rightarrow AB, \{2, 3\}, \emptyset \\ 1 : A &\rightarrow a, \{0\}, \emptyset \\ 2 : B &\rightarrow b, \{1\}, \emptyset \\ 3 : B &\rightarrow c, \{1\}, \emptyset \end{aligned}$$

The language of the grammar is $L(G_3) = \{ab, ac\}$, but if we compute the TR table T for this grammar, it turns out that $T[A, a, \{2, 3\}]$ for this grammar would be empty. This means that $L_{TR}(G_3) = \emptyset$.

All of these grammars are artificial examples constructed to have properties incompatible with the algorithm and can quite easily be transformed into grammars which will work well with the algorithm (and don't even need such a complex algorithm, given that they all generate small finite languages), but they nevertheless demonstrate that such problems can come up in the general case.

The deviations from the grammar's language happen for two reasons, both related to the last switch in algorithm 2:

- If $|\rho| = 1$, it is clear what rule needs to be applied, but the algorithm will always try to apply it to the leftmost instance of the particular nonterminal in the current sentential form (that is, to the instance of the particular nonterminal nearest to the top of the pushdown). However, unlike for context-free grammar, the order in which

nonterminals are rewritten matters for programmed grammars. More precisely, a programmed grammar might generate some particular words only by derivations, which at some point rewrite a nonterminal which is not the leftmost of its kind in the current sentential form, as is the case with the string ba and the grammar G_1 in the example above. The algorithm cannot simulate these derivations, and therefore won't accept such words;

- If $|\rho| > 1$, the algorithm will insist on rewriting the leftmost nonterminal of the sentential form (so the topmost nonterminal on the pushdown). This means that not only will the algorithm forgo any other instances of that particular nonterminal, like in the previous case, but it will also ignore other nonterminals on the pushdown that can be rewritten with other rules from ρ (see grammar G_2 above for illustration). This is even the case if the corresponding TR table entry is empty – the algorithm will simply reject the input, even though it could conceivably explore rewriting some of the other nonterminals using the other rules from ρ (see grammar G_3). However, this would introduce new questions of nondeterminism, so we abandon these cases for simplicity.

Therefore, even though $L_{TR}(G) \subseteq L(G)$ always holds, there will be cases when $L_{TR}(G) \subset L(G)$ holds. Most of these problems are a consequence of the algorithm always looking for leftmost derivations, so we would have to move further away from LL analysis to resolve them. Furthermore, any attempts to consider other derivations would have to provide a way to deal with the ensuing nondeterminism.

4.3.2 Accepting Power

Let us now discuss the accepting power of the algorithm. We will denote the class of languages that can be accepted by the table-resort algorithm as described in this section by \mathcal{L}_{TR} . For now, we will not burden ourselves with the fact that the algorithm may not halt for some inputs – we will deal with it in a later section, and from a theoretical standpoint, we can consider these inputs as not accepted by the algorithm, therefore not a part of its accepted language for the particular grammar.

It's not straightforward to consider the relation between this class and the classes generated by various kinds of programmed grammars – even if we could delimit the class of languages generated by TR grammars, it wouldn't necessarily be the same as \mathcal{L}_{TR} , as the languages accepted by the algorithm will sometimes differ from the languages generated by the grammars. This difference might lead to unexpected results – the class of languages generated by programmed grammars restricted to leftmost derivations is simply the class of context-free languages, but for slightly different definitions of leftmost derivations, which also consider the current set ρ of applicable rules, we can even increase the power of certain types of programmed grammars by such a restriction – see [2] for details and proofs.

If we use any LL(1) grammar (with $\sigma_r = \Psi$ and $\varphi_r = \emptyset$ for all $r \in \Psi$) as the input grammar of the algorithm, it should accept exactly the language generated by said grammar, as the algorithm is an extension of LL(1) parsing – it will *always* simply resort to the table in this case, from which we can remove the third dimension, as the only success and failure fields of any rules are Ψ and \emptyset , and we don't have to worry about restricting Ψ to S -rules for the initialization of the algorithm, as the grammar is LL(1) anyway, so we can think of the table as simply the LL table as described in section 3.5. From this, we can infer that $\mathbf{LL}(1) \subseteq \mathcal{L}_{TR}$.

Furthermore, the algorithm can accept some non-LL(1) languages. Consider the grammar $G_4 = (\{S, X\}, \{a, b, c\}, \{0, 1, 2, 3\}, P, S)$, where P consists of the following rules:

- 0 : $S \rightarrow aSX, \{0, 1\}, \emptyset$
- 1 : $S \rightarrow \varepsilon, \{2, 3\}, \emptyset$
- 2 : $X \rightarrow b, \{2\}, \emptyset$
- 3 : $X \rightarrow c, \{3\}, \emptyset$

The language of this grammar is $L(G_4) = L_{TR}(G_4) = \{a^n b^n, a^n c^n : n \geq 0\}$, which is a typical example of a deterministic context-free language, which is not LL(k) for any $k \geq 1$. This is enough to prove that the inclusion shown above is proper – $\mathbf{LL}(1) \subset \mathcal{L}_{TR}$.

In fact, the algorithm can accept languages that are not even context-free. Consider $G_5 = (\{S, A, C\}, \{a, b, c\}, \{0, 1, 2, 3, 4\}, P, S)$, where P consists of the following rules:

- 0 : $S \rightarrow AC, \{1, 2\}, \emptyset$
- 1 : $A \rightarrow aAb, \{3\}, \emptyset$
- 2 : $A \rightarrow \varepsilon, \{4\}, \emptyset$
- 3 : $C \rightarrow cC, \{1, 2\}, \emptyset$
- 4 : $C \rightarrow \varepsilon, \{0\}, \emptyset$

It can be shown that $L(G_5) = L_{TR}(G_5) = \{a^n b^n c^n : n \geq 0\}$, which is a context-sensitive language that is not context-free. Therefore, $\mathcal{L}_{TR} \setminus \mathbf{CF} \neq \emptyset$. It is not clear whether there are also non-context-sensitive languages which can be parsed by the TR algorithm.

However, although no formal proof is provided in this thesis, it appears that conversely, there are context-free languages that cannot be parsed by the TR algorithm. Consider the language $L_{wwr} = \{ww^R \mid w \in \{a, b\}^*\}$. The nondeterminism in not being able to tell where the first half of the word ends and where the second half begins makes it problematic to construct a TR grammar G , such that $L_{TR}(G) = L_{wwr}$ – the algorithm would necessarily end up in the same configuration after processing the whole of the word ab , the first half of the word $abba$, as well as the first third of the word $abbbba$, and so on.

It appears that even the relatively simple language $L_{ij} = \{a^i b^j \mid i \geq j\}$, cannot be parsed by TR analysis. This would delimit \mathcal{L}_{TR} more precisely, as unlike L_{wwr} , it is a deterministic context-free language (which is not LL(k) for any $k \geq 0$). We can consider a grammar like $G_6 = (\{S, X\}, \{a, b\}, \{0, 1, 2, 3\}, P, S)$ with the rules in P as follows:

- 0 : $S \rightarrow aSX, \{0, 1\}, \emptyset$
- 1 : $S \rightarrow \varepsilon, \{2, 3\}, \emptyset$
- 2 : $X \rightarrow b, \{2, 3\}, \emptyset$
- 3 : $X \rightarrow \varepsilon, \{3\}, \emptyset$

It is the case that $L(G_6) = L_{ij}$, but G_6 is not a TR grammar, as $T[X, b, \{2, 3\}]$ would have to contain multiple rules – the algorithm has no way of knowing, which rule to apply when it comes to process the b 's on the input, even though it's clear to us that it should apply 2 until the b 's run out and then switch to 3. This does not mean that there cannot be a different grammar G , which is TR and $L_{TR}(G) = L_{ij}$, but it illustrates how it might be hard for this algorithm to parse this language.

To summarize, here's what we know (or assume) about \mathcal{L}_{TR} :

- $\mathbf{LL}(0) \subset \mathcal{L}_{TR}$,
- $\mathcal{L}_{TR} \setminus \mathbf{CF} \neq \emptyset$,
- Presumably, $\mathbf{CF} \setminus \mathcal{L}_{TR} \neq \emptyset$ and even $\mathbf{DCF} \setminus \mathcal{L}_{TR} \neq \emptyset$.

This is summed up graphically in figure 4.1, with \mathcal{L}_{TR} filled in.

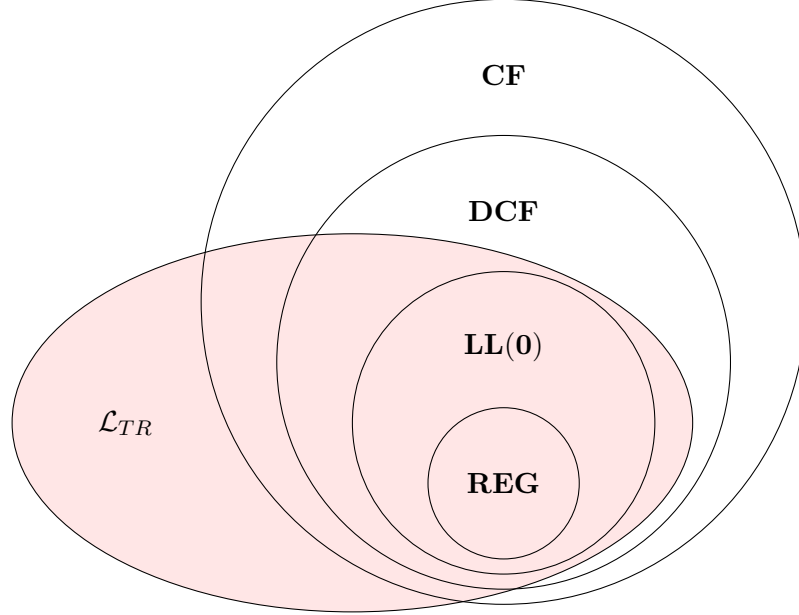


Figure 4.1: Accepting power of the TR algorithm

4.4 Looping

As noted in the previous section, the TR algorithm as described may not ever halt for certain inputs. We will refer to such occurrences as *looping*. For example, consider $G_0 = (\{S\}, \{a\}, \{0, 1, 2\}, P, S)$ where P contains the following rules:

- 0 : $S \rightarrow A, \{1\}, \emptyset$
- 1 : $A \rightarrow S, \{0\}, \emptyset$
- 2 : $A \rightarrow a, \{2\}, \emptyset^2$.

With e.g. the string a as the input, $\rho = \{0\}$ every step of the way, so it is always clear what rule to apply, and the algorithm will simply keep rewriting S to A and back to S , never halting. In this example, it is fairly obvious, but such control loops may be well-hidden in the structure of more complex grammars, and we must examine when the looping can happen and when it can be avoided.

In this section, we will first reduce this problem to the question of how many computation steps can occur without a terminal being generated (or without an even stricter condition occurring), and then try to resolve this new question. This seems to be fairly hard if we consider ε -rules, so we will restrict ourselves to propagating grammars in this section.

²Rule 2 is only included so that $a \in \text{Predict}(0)$, which is important for the initialization of the algorithm.

4.4.1 Restricting Sentential Form Size

One fairly simple mechanism we can add to the algorithm is to add the knowledge of the length of the input, $|w|$. This way we can know for sure when we've derived too many terminals. In fact, we can include nonterminals in this analysis, if we wish to do so: In a grammar $G = (N, \Sigma, \Psi, P, S)$, define for each $A \in N$ the *value* of A as the minimum length of a terminal string that can eventually be generated from A : $\|A\|_G := \min\{|w| : A \Rightarrow_G^* w\}$. For a programmed grammar, analysis on the underlying context-free grammar G' will suffice, providing $\|A\|_{G'}$ as a lower bound for $\|A\|_G$ ³. If we also define $\|a\|_G := 1$ for all $a \in \Sigma$, we can define the *value* of a string $\alpha = X_1 \dots X_n$ as $\|X_1 \dots X_n\|_G := \sum_{i=1}^n \|X_i\|_G$ for all $X_1 \dots X_n \in (N \cup \Sigma)^*$. We will omit the suffix in the rest of the chapter, as it does not matter in principle whether $\|\alpha\|_G$ or $\|\alpha\|_{G'}$ is used.

Note that other lower bounds of $\|A\|_G$ may suffice as $\|A\|$ for our purposes, although they may diminish the advantages of this approach – for the extreme case of $\|A\| = 0$ for all $A \in N$, the value of a string is simply equal to the number of terminals it contains. For propagating grammars, we can increase this global lower bound to 1, as every nonterminal will eventually generate at least one terminal (with the usual exception for $S \rightarrow \varepsilon$).

When analyzing an input string w , we can keep track of the value of the sentential form γ generated so far: We initialize it as $\|S\|$, and for each rule of the form $\alpha \rightarrow \beta$ applied, we can subtract $\|\alpha\|$ and add $\|\beta\|$. If we check in every step that the value of the sentential form has not superseded the length of the input, that is $\|\gamma\| \leq |w|$, we can detect all infinite loops which increase the value of the sentential form. Note that as $\|A\|$ for $A \in N$ is defined as the *minimum* length of a terminal string derivable from A (or a lower bound of said minimum), we can be sure that $\|\gamma\|$ will never decrease, assuming we use a reasonable lower bound, such as one of the ones mentioned above. Alternatively, we can count the number n_t of terminals generated so far and ensure that it doesn't go beyond $|w|$, which is what we will eventually use in the improved algorithm, although this approach is more permissive – we would detect fewer loops in the general case.

This means that the only loops that can occur undetected must not increase the value of the sentential form (or not generate terminals), so the question of whether the algorithm will loop for a particular input gets transformed into how many computation steps can the algorithm take without increasing the value of the sentential form (or generating terminals). In the upcoming sections, we will show how to answer this question for propagating grammars without appearance checking, which will also provide a starting point for the same analysis for propagating grammars with appearance checking. Propagating grammars have two great advantages that will make this analysis possible:

- In a propagating grammar, we can be sure that $\|A\| > 0$ for any $A \in N$, as no nonterminal can generate an empty string (with the usual exception for $S \rightarrow \varepsilon$). This means that if we use the lower bound $\|A\| = 1$ for the values of the individual nonterminals, the only way to not increase the value of the sentential form and at the same time not generate terminals is to use *simple rules* – rules of the form $A \rightarrow B$, where $A, B \in N$ (or fail to apply a rule, in the case of grammars with appearance checking), so we can restrict our analysis to these;
- In a propagating grammar, it's easier to keep track of the leftmost nonterminal of the sentential form – unless it gets rewritten to a terminal, it only changes when it is rewritten according to a rule, whereupon the leftmost nonterminal of the right-hand

³ Any derivation possible in G must also be possible in G' , thus clearly $\|A\|_{G'} \leq \|A\|_G$ for all A .

side of said rule becomes the leftmost nonterminal of the sentential form. This makes the analysis of possible loops easier thanks to specific properties of the TR algorithm.

Conversely, in a grammar with ε -rules, the leftmost nonterminal can be erased, whereupon the formerly second leftmost nonterminal becomes the leftmost nonterminal, which can be erased again, and so on. Therefore we would need to keep track of the entire sentential form to be sure we don't lose the leftmost nonterminal, making complete analysis significantly harder.

In the rest of this section, unless specified otherwise, we will restrict ourselves to propagating grammars without appearance checking.

4.4.2 Finding Candidate Loops

We are now looking for *candidate loops* – possibly infinite branches of computation that the algorithm can fall into, which don't generate terminals or even increase the value of the sentential form. We will presume we are analyzing a grammar $G = (N, \Sigma, \Psi, P, S)$.

For analyzing loops that can happen without generating terminals, we can obviously restrict the space of rules we explore to rules without terminals on their right-hand sides. In fact, as we are only discussing propagating grammars, we can restrict ourselves to simple rules, that is, rules of the form $A \rightarrow B$, $A, B \in N$, as every nonterminal will generate at least one terminal, so there can never be more than $|w|$ nonterminals in the sentential form, so no non-simple rules may appear infinitely many times in infinite loops that don't increase the value of the sentential form. Let us therefore denote P_s as the set of simple rules in P , where P is the set of production rules of the grammar being analysed. Unless there is a candidate loop consisting of loops in P_s , there is no candidate loop in P .

In each step, the TR algorithm decides what to do based on three variables: The topmost symbol X on the pushdown, The leftmost unprocessed symbol a on the input, and the set of rules ρ where the next rule gets selected from. We can ignore the cases of X being the bottom of the pushdown $\#$ or a terminal, as the former can only occur at the end of the algorithm, and the latter can only keep occurring in a loop which generates terminals. When analysing the time between generating terminals, we can therefore assume that $X = A \in N$ is a nonterminal, and furthermore, a does not change.

We can use the values of these three variables, A , a and ρ , to fully determine the rule s that gets applied to the sentential form. Assuming grammars without appearance checking, ρ is also fully determined by the previous rule r , as $\rho = \sigma_r$. (the only other option is r not being applicable, in which case we don't care, because that means an error, and errors cannot happen in infinite loops). We can therefore represent the values needed to determine the next steps by the triple (A, a, r) .

A useful property of the TR algorithm is that if we restrict ourselves to propagating grammars without appearance checking, the knowledge of this rule allows us to uniquely determine the only possible next triple (A', a', r') , or whether a terminal was generated, in which case our analysis doesn't care about this branch of the computation, as it cannot be a part of an infinite loop:

- The next rule r' is fully determined by (A, a, σ_r) , either by σ_r only containing one rule, or through the TR table. We might also learn that there is no next rule, so the computation simply fails and we are therefore not in a loop;
- Assuming we haven't left the space of simple rules, it will be the case that $a' = a$;

- The next leftmost nonterminal A' can be inferred from the rule r' being applied in this computation step:
 - a) If r' is not a simple rule, we have left P_s , so we are not in a loop and we don't care about A' ;
 - b) If $r' : A \rightarrow B$, the current leftmost nonterminal gets rewritten, so $A' = B$;
 - c) if $r' : C \rightarrow D$ with $C \neq A$, some other nonterminal gets rewritten, so $A' = A$.

We can look for candidate loops simply by exploring the space $T := N \times (\Sigma \cup \{\$, \perp\}) \times P_s$ of possible triples, using the special symbol \perp to denote that we have left any possibility of being in a loop – either by leaving the space of simple rules, or by reaching an error and the algorithm failing. Having established this notation, we can sum up the previous paragraphs formally by defining a function $\text{next} : T \cup \{\perp\} \rightarrow T \cup \{\perp\}$ for any applicable A, a, r as follows:

- $\text{next}(A, a, r) = (A', a', r')$ as described above, if r' is defined and simple;
- $\text{next}(A, a, r) = \perp$ otherwise, that is, if the algorithm cannot select a next rule, or if the next rule is not simple;
- $\text{next}(\perp) = \perp$.

We can now construct a directed graph $SR = (T \cup \{\perp\}, E)$, where the aforementioned triples and \perp are the vertices, and the set of edges is $E = \{(u, v) : \text{next}(u) = v\}$. Any candidate loop in the grammar will manifest as a cycle in this graph – if there is a candidate loop which can get executed for some input, it will necessarily only consist of simple rules, and the algorithm will be deciding based on some triple $(A, a, r) \in T$ at some point in its execution. This however fully defines the remaining trajectory of the algorithm in T , which will either be by a cycle, or lead to \perp , in which case the computation we considered wasn't actually a loop. We can also leave out \perp altogether, as its inclusion will have no effect on the results of the analysis.

Looking for cycles in this graph is a question of looking for strongly connected components (SCCs) – as each node has only one outgoing edge, all nontrivial SCCs must be cycles. All SCCs with no outgoing edges will be either $\{\perp\}$, or correspond to a candidate loop.

We can construct a table, called the *candidate loop table*, or *CL table* for short, denoted by CL , indexed by entries from T marking whether a particular triple (A, a, r) is part of a cycle corresponding to a candidate loop, and in each step of the algorithm after applying a simple rule, we can check the relevant entry $CL[A, a, r]$ – we can be sure that the triples corresponding to candidate loops won't lead to anything meaningful, as they mean we are stuck in simple rule space. However, they don't necessarily mean the algorithm would actually loop – for the candidate loop to correspond to an actual loop, two conditions must be met:

- The loop must be *self-sustainable* – it must produce at least as many of each kind of nonterminal as it consumes. Otherwise, the sentential form will eventually „run out“ of nonterminals necessary for the candidate loop's execution;
- The loop must be *executable* – the current sentential form must contain at least the minimum number of occurrences of each nonterminal required for executing the loop the first time.

Furthermore, even a self-sustainable candidate loop may not be *reachable* – there might not exist any $w \in \Sigma^*$ such that the algorithm will eventually reach a configuration where it will decide based on a triple contained in a cycle corresponding to said candidate loop.

For convenience, we can also define $CL[A, a, r] := \text{false}$ for all $r \in P \setminus P_s$ and all $A \in N, a \in \Sigma$, which will allow us to describe the final algorithm more compactly. Also note that we can define for each $a \in \Sigma \cup \{\$\}$ a subspace $T_a := N \times \{a\} \times P_s$ of T and a separate graph SR_a as SR restricted to nodes from T_a and analyze these graphs separately if we prefer – we will not lose any edges this way, as the terminal on the input cannot change without leaving the simple rule space.

To sum up, the information we need for detecting loops at runtime is stored in the table CL , which we can construct as follows:

1. Prepare the set P_s of simple rules and compute $\text{next}(A, a, r)$ for all relevant triples in T ;
2. Construct the graph SR and compute its SCCs (strongly connected components);
3. Mark all the entries $CL[A, a, r]$ corresponding to triples which occur in SCCs with no outgoing edges as *true*, meaning the triple corresponds to a candidate loop. Mark all the other entries as *false*.

4.5 An Improved Table-resort Algorithm

We can now add the improvements discussed in the previous sections – counting the number of terminals generated in a variable n_t , and checking the CL table for whether we have reached a candidate loop. The algorithm is described as algorithm 3. This algorithm will accept the same input strings as algorithm 2 described in section 4.3, but in addition, it will always halt for propagating grammars without appearance checking.

The only changes from algorithm 2 are the periodic checking of the CL table and the number of terminals generated. Unlike with algorithm 2, we must make sure that the last used rule r is defined in the beginning, as it is used to index the CL table.

To use the algorithm with non-propagating grammars, or grammars with appearance checking, we will either have to provide a dummy CL table or modify the algorithm to skip the CL table checking. However, doing so will open us up to the algorithm possibly not halting for some inputs.

Note, however, that even if we restrict ourselves to propagating programmed grammars without appearance checking, we can still parse all the languages mentioned explicitly in section 4.3.2 – in particular, all the **LL**(1) languages, and the languages $\{a^i b^i, a^i c^i : i \geq 0\}$ and $\{a^n b^n c^n : n \geq 0\}$, as evidenced by their grammars being TR as well as propagating and without appearance checking.

Algorithm 3: Table-resort algorithm with loop detection

Input : The TR table T for a TR $G = (N, \Sigma, \Psi, P, S)$ with an implicit starting rule r_0 , a CL table CL for G , a string $w \in \Sigma^*$ of known length $|w|$

Output: A sequence of rules to produce a derivation of w in G , if $w \in L_{TR}(G)$, or an error otherwise.

$n_t \leftarrow 0, r \leftarrow r_0, \rho \leftarrow \Psi S;$

$Pushdown.Push(\#);$

$Pushdown.Push(S);$

while $\neg Pushdown.Empty()$ **do**

$X \leftarrow Pushdown.Top();$

$a \leftarrow Input.Current();$

switch X **do**

case $X = \#$ **do**

if $a = \$$ **then**

SUCCESS – the input string has been successfully processed, printed rule labels can be used to construct a derivation of w in G ;

else

ERROR – unprocessed input remaining after the derivation is finished;

case $X \in \Sigma$ **do**

if $X = a$ **then**

$Pushdown.Pop();$

$Input.Move();$

else

ERROR – string generated on pushdown does not agree with string on input;

case $X \in N$ **do**

if $CL[X, a, r]$ *is true* **then**

ERROR – we have reached a candidate loop, the parsing will not be successful;

switch $|\rho|$ **do**

case 0 **do**

ERROR – no rule to apply;

case 1 **do**

$(r : A \rightarrow \alpha, \sigma_r, \varphi_r) \leftarrow \rho.Get();$

if A *not on pushdown* **then**

$\rho \leftarrow \varphi_r;$

else

$Pushdown.Replace(A, \alpha);$

$Output(r), \rho \leftarrow \sigma_r, n_t \leftarrow n_t + |\alpha|_\Sigma;$

otherwise do

if $(r : A \rightarrow \alpha, \sigma_r, \varphi_r) \in T[X, a, \rho]$ **then**

$Pushdown.Replace(A, \alpha);$

$Output(r), \rho \leftarrow \sigma_r, n_t \leftarrow n_t + |\alpha|_\Sigma;$

else

ERROR – no rule to apply;

if $n_t > |w|$ **then**

ERROR – too many terminals were generated;

Chapter 5

Conclusion

This thesis introduced an extension of LL parsing, called the *table-resort* algorithm, which can be used to parse certain programmed grammars. The class of languages accepted by this algorithm turns out to be a strict superclass of LL(1) languages, and includes certain non-context-free languages, but is incomparable with context-free languages.

One disadvantage of the algorithm is that it will not always accept the same language as the input grammar generates. However, this cannot be helped in general if we want to restrict ourselves to rewriting the leftmost nonterminal.

The improved version of the algorithm, presented at the end of chapter 4, is guaranteed to halt for propagating programmed grammars without appearance checking, but the analysis of whether the algorithm will halt seems to be more complicated for more general versions of programmed grammars.

The TR table which both versions of the algorithm are based on is computed only using the information about the underlying context-free grammar, ignoring the success and failure fields of the rules. This information could conceivably be used to improve the accuracy of the approximations of the $Predict_P$ sets (and thus hopefully also the accepting power of the algorithm), but as is proven in section 4.2, the approximation can never be perfect in general.

Some questions related to the algorithm remain open:

- What exactly is the class of languages accepted by this algorithm? Is it really impossible to use it to parse some context-free languages, such as $\{ww^R : w \in \{a,b\}^*\}$ or $\{a^ib^j : i \geq j \geq 0\}$?
- When is the language $L(G)$ of the underlying grammar G different from the language $L_{TR}(G)$ accepted by the algorithm? Can we transform such a grammar into a grammar G' , such that $L(G) = L_{TR}(G')$? What grammars is this possible for?
- Can we detect candidate loops in more general grammars, especially propagating grammars with appearance checking? Is the problem significantly harder for grammars with appearance checking, given the very different structure of the graph SR that would arise, owing to each point of the space being explored having up to two different successors, and with the requirement to take the presence of various nonterminals in the sentential form into account?
- Can we provide a better foundation for constructing the TR table than the $Predict$ set computed on the underlying context-free grammar, such as a better approximation of the $Predict_P$ set?

- Are there other changes that can be made to the core concepts behind the algorithm in order to increase its accepting power?

Exploring these and other questions can not only lead to a better understanding of the algorithm and possible increases in its accepting power, but also to more interesting questions about the properties of programmed grammars and their parsing in general.

Bibliography

- [1] C. Martin-Vide, G. P., V. Mitrana: *Formal Languages and Applications*. Springer, Berlin. 2004.
- [2] Fernau, H.: Regulated Grammars with Leftmost Derivation. In *SOFSEM '98: Theory and Practice of Informatics*. 1998. ISBN 978-3-540-65260-1.
- [3] J. Dassow, G. P.: *Regulated Rewriting in Formal Language Theory*. Springer-Verlag. 1989. ISBN 978-3-642-74934-6.
- [4] Keith Cooper, L. T.: *Engineering a Compiler*. Morgan Kaufmann. second edition. 2012. ISBN 978-0-12-088478-0.
- [5] Meduna, A.: *Automata and Languages: Theory and Applications [Springer, 2000]*. Springer Verlag. 2005. ISBN 1-85233-074-0. 892 pp.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=6177
- [6] Meduna, A.; Zemek, P.: *Regulated Grammars and Automata*. Springer US. 2014. ISBN 978-1-4939-0368-9. 694 pp.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10498

Appendix A

Contents of the Attached CD

On the accompanying CD, a simple implementation of the algorithms presented in this thesis can be found. It is intended for demonstration, as neither efficiency nor generality were big goals when implementing it, but it can in principle be used for possible applications. To run it, you will need Python 3 with the *networkx*¹ library installed.

The following files can be found on the CD:

- `README.txt` – A short summary of information necessary for using the application;
- `grammars` – A directory containing several example grammars, including most of the ones used in this thesis;
- `Grammar.py` – Source file defining the structure used for programmed grammars, along with some basic analysis, including the computation of *Predict* sets;
- `TRParser.py` – Source file implementing the actual algorithms introduced in this thesis, that is:
 - TR table construction,
 - Naive TR parsing,
 - CL table construction,
 - Improved TR parsing;
- `Main.py` – A simple main function calling the functionality implemented in the other files.

For a quick example run of the application, try executing something like the following:

```
$ python3 Main.py grammars/g_abc.txt
```

¹<https://networkx.github.io/>