BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

SEQUENTIAL AND PARALLEL GRAMMARS: PROPERTIES AND APPLICATIONS

SEKVENČNÍ A PARALELNÍ GRAMATIKY: VLASTNOSTI A APLIKACE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. DOMINIKA KLOBUČNÍKOVÁ**
AUTOR PRÁCE

SUPERVISOR **Prof. RNDr. ALEXANDER MEDUNA, CSc.**
VEDOUCÍ PRÁCE

BRNO 2019

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

21128

Studentka:	**Klobučníková Dominika, Bc.**
Program:	Informační technologie Obor: Informační systémy
Název:	**Sekvenční a paralelní gramatiky: vlastnosti a aplikace**
	Sequential and Parallel Grammars: Properties and Applications
Kategorie:	Teoretická informatika

Zadání:

1. Seznamte se podrobně se sekvenčními a paralelními gramatikami dle pokynů vedoucího. Zaměřte se na normální formy.
2. Demonstrujte nové normální formy dle pokynů vedoucího.
3. Dle pokynů vedoucího uvažujte řadu syntaktických struktur, včetně struktur, které nejsou bezkontextové. Popište jejich syntaktickou analýzu založenou na výše modifikovaných gramatikách v normálních formách.
4. Aplikujte a implementujte syntaktickou analýzu navrženou v předchozím bodě. Testujte ji.
5. Zhodnoťte dosažené výsledky. Diskutujte další vývoj projektu.

Literatura:

- Meduna, A.: Automata and Languages, Springer, London, 2000, ISBN 978-1-85233-074-3
- Rozenberg, G. and 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

Při obhajobě semestrální části projektu je požadováno:

- Splnění prvních dvou bodů zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce:	**Meduna Alexander, prof. RNDr., CSc.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1. listopadu 2018
Datum odevzdání:	22. května 2019
Datum schválení:	23. října 2018

Abstract

This thesis deals with the topic of sequential and parallel grammars. Both of these groups cover a large number of grammar families, most of which, however, are not widely used because of the difficulties related to their processing. The thesis examines some of these grammar types, such as scattered-context grammars, multigenerative systems, and interactive L-systems, with focus on their normal forms. Subsequently, it introduces a set of algorithms utilising properties of the discussed grammar types as well as their normal forms. These algorithms are based on the Cocke-Younger-Kasami algorithm for context-free grammars, and are capable of parsing any grammar in the corresponding normal form. Finally, a program implementing the proposed algorithms is presented.

Abstrakt

Táto práca sa zaoberá problematikou sekvenčných a paralelných gramatík. Obe skupiny zastrešujú veľké množstvo gramatických tried, väčšina ktorých však nemá veľké uplatnenie kvôli komplikáciám spojeným s ich spracovaním. Práca skúma niektoré takéto gramatiky, ako napríklad gramatiky s rozptýleným kontextom, multigeneratívne gramatické systémy a interaktívne L-systémy s dôrazom na ich normálne formy. Práca následne predstavuje niekoľko algoritmov využívajúcich vlastnosti týchto gramatík, ako aj ich normálnych foriem. Tieto algoritmy sú založené na Cocke-Younger-Kasami algoritme pre bezkontextové gramatiky a dokážu spracovať ľubovoľnú gramatiku v príslušnej normálnej forme. Posledná časť práce predstavuje program implementujúci navrhnuté algoritmy.

Keywords

syntax analysis, normal form, Kuroda, Penttonen, Chomsky, 2-limited, context-sensitive grammar, scattered context grammar, multigenerative grammar system, L-system, Cocke-Younger-Kasami, Cocke-Kasami-Younger, CYK, CKY

Klíčová slova

syntatická analýza, normálna forma, Kuroda, Penttonen, Chomsky, 2-obmedzená (2-limited), kontextová gramatika, gramatika s rozptýleným kontextom, multigeneratívny gramatický systém, L-systém, Cocke-Younger-Kasami, Cocke-Kasami-Younger, CYK, CKY

Reference

KLOBUČNÍKOVÁ, Dominika. *Sequential and Parallel Grammars: Properties and Applications.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. RNDr. Alexander Meduna, CSc.

Rozšířený abstrakt

Táto práca sa zaoberá problematikou sekvenčných a paralelných gramatík, ako aj ich aplikácií. Jej cieľom je navrhnúť skupinu algoritmov schopných syntaktickej analýzy gramatík patriacich do týchto skupín, s dôrazom na nie-bezkontextové gramatiky. Predstavené algoritmy sú založené na algoritme Cocke-Younger-Kasami pre bezkontextové gramatiky v Chomského normálnej forme a dokážu spracovať ľubovoľnú gramatiku v korektnej normálnej forme.

Pomocou Chomského hierarchie ako základu pre porovnanie sekvenčných gramatík táto práca skúma typy gramatík, ktoré sú jej súčasťou, ale ležia aj mimo jej hraníc. Práca sa zameriava na kontextové gramatiky, multigeneratívne neterminálovo-synchronizované gramatické systémy a gramatiky s rozptýleným kontextom. Následne sú skúmané L-systémy – je popísaný ich význam a vlastnosti, ako napríklad paralelizmus a vetvenie. Preskúmané sú hlavné triedy L hierarchie, najmä rozšírené a interaktívne systémy. Pre tieto typy gramatík sú následne predstavené príslušné normálne formy.

Predstavené algoritmy pracujú už so spomínanými typmi gramatík, pričom využívajú štrukturálnu podobnosť medzi Chomského normálnou formou a vhodnými normálnymi formami príslušných typov gramatík. Každý z týchto typov však disponuje množinou špecifických vlastností, ktoré vyžadujú aplikáciu dodatočných mechanizmov pre zaistenie spoľahlivosti výsledkov algoritmov. Táto práca celkovo predstavuje päť algoritmov syntaktickej analýzy.

Rozšírenie pre kontextové gramatiky pracuje s gramatikami v Kurodovej normálnej forme. Keďže táto normálna forma predstavuje priame rozšírenie Chomského normálnej formy, jadro algoritmu bolo rozšírené o manipuláciu s kontextovými pravidlami. V prípade, ak by bol vo finálnom derivačnom strome použitý len jeden neterminál kontextového páru, výsledky algoritmu by mohli byť nesprávne. Tomuto problému je predídené zavedením množiny verzií spracúvanej matice. Vždy, keď sa nejaká množina kontextových pravidiel aplikuje na dvojicu prvkov, vytvoria sa dve verzie matice – jedna, v ktorej je umelo zabránené aplikovaniu pravidla, a druhá, ktorá obsahuje len zredukované neterminály, bez prítomnosti ich predkov; týmto vzniká potreba použiť oba neterminály pre prijatie vstupného reťazca. Tento algoritmus dosahuje časovú zložitosť $O(|G| \cdot n^3)$.

Rozšírenie pre kanonické multigeneratívne neterminálovo-synchronizované systémy pracuje so systémami pozostávajúcimi zo sekvencie bezkontextových gramatík v Chomského normálnej forme. Keďže tieto systémy využívajú najľavejšiu deriváciu pre kontrolu prepisovania, toto rozšírenie využíva dodatočný sled spracúvaných matíc pre jej zaznamenie – jednu pre každú gramatiku, teda aj každú normálnu maticu. Proces syntaktickej analýzy je obdobný algoritmu Cocke-Younger-Kasami s rozdielom, že redukované neterminály sú ukladané do spomínaných dodatočných matíc. Tieto matice sú potom filtrované, aby obsahovali len neterminály vzniknuté počas spätnej simulácie najľavejšej derivácie, ktoré sú následne porovnané s kontrolnými sekvenciami, a opätovne filtrované. Nakoniec sa matice synchronizujú s ohľadom na počet redukovaných prvkov, teda podreťazcov, a ich obsah je uložený do zodpovedajúcich hlavných matíc. Kvôli nutnosti použitia sekvencie matíc, časová zložitosť tohoto algoritmu dosahuje $O(N \cdot n^3)$.

Posledné sekvenčné rozšírenie spracúva gramatiky s rozptýleným kontextom v 2-obmedzenej (2-limited) normálnej forme. Kvôli jej štruktúre toto rozšírenie simuluje aplikáciu dvoch bezkontextových pravidiel v rôznych častiach spracovávanej matice. Avšak podobne ako pri rozšírení pre kontextové gramatiky môže byť konzistencia matice narušená, ak simulácia výsledného derivačného stromu obsahuje len jeden neterminál simultánne redukovaného páru. Narozdiel od predošlého rozšírenia sú všetky redukcie vykonávané v rámci

jednej matice, nakoľko predošlý systém by viedol k neprimeranému zvýšeniu časovej zložitosti algoritmu. Namiesto toho na konci spracovania algoritmus skúma pôvod všetkých zredukovaných počiatočných symbolov, a odstráni tie, ktorých rodokmeň buď neobsahuje všetkých očakávaných členov, alebo obsahuje duplicitné symboly. Kvôli semi-paralelnému charakteru gramatík s rozptýleným kontextom je celková časová náročnosť pôvodného algoritmu umocnená na druhú, z čoho vyplýva relatívne vysoká zložitosť $O(|G| \cdot n^6)$.

Hlavný problém prispôsobenia algoritmu Cocke-Younger-Kasami pre paralelné gramatiky predstavoval už spomínaný fakt, že paralelizmus prítomný v tejto skupine gramatík nie je možné simulovať sekvenčnými gramatikami. Podobne ako pri niektorých predošlých rozšíreniach používa algoritmus pre rozšírené L-systémy dvojicu matíc – jednu, v ktorej uchováva predošlú vetnú formu, a druhú, do ktorej ukladá vetnú formu, ktorá je práve generovaná. Tieto matice sú periodicky obmieňané, vďaka čomu sú nadbytočné symboly z predošlých vetných foriem odstránené. Taktiež, aby sa v matici predišlo prekrytiu redukovaných vetných foriem, algoritmus pokračuje v prehľadávaní vpravo od predtým redukovaných podreťazcov. Celková časová zložitosť tohoto rozšírenia dosahuje $O(|G| \cdot n^3)$.

Rozšírenie pre interaktívne systémy zdieľa algoritmické jadro s predchádzajúcim algoritmom. Oproti nemu však toto rozšírenie pridáva niekoľko kontrol orientovaných na prostredie. Narozdiel od pojatia kontextu v sekvenčných gramatikách interaktívne L-systémy žiadnym spôsobom nezasahujú do svojho prostredia; kvôli sekvenčnému charakteru algoritmu Cocke-Younger-Kasami však prostredie v čase redukcie ešte nemusí existovať. Toto je riešené zavedením dvoch vĺn redukčných kontrol – najskôr je overované, či súčasná matica obsahuje potomkov všetkých reťazcov prostredia svojho rodiča; následne je overované, či v tejto matici existuje prostredie nutné pre vznik všetkých nových neterminálov. Predstavenie týchto kontrolných mechanizmov zvyšuje časovú náročnosť rozšírenia až na celkovú hodnotu $O(|G|^2 \cdot n^3)$.

Posledná časť tejto práce je venovaná funkčnému prototypu, ktorý implementuje navrhnuté algoritmy. Je to program v jazyku C++ konfigurovateľný cez argumenty príkazového riadku. Program využíva princípy objektovo-orientovaného programovania, a nasadzuje množinu polymorfných tried zodpovedajúcich jednotlivým algoritmom a im príslušným typom gramatík. Je preberaná celková štruktúra hierarchie tried, ako aj detailný popis práce jednotlivých modulov, ich cieľ a odvodené triedy. Nakoniec sú porovnávané experimentálne hodnoty časovej a priestorovej zložitosti s ich teoretickými variantami.

Sequential and Parallel Grammars: Properties and Applications

Declaration

I hereby declare that this master's thesis was prepared as an original author's work under the supervision of Prof. RNDr. Meduna, CSc. All the relevant information sources, which were used during preparation of this project, are properly cited and included in the list of references.

. . . .
Dominika Klobučníková
May 21, 2019

Acknowledgements

I would like to express my sincere gratitude to my advisor Prof. RNDr. Alexander Meduna, CSc. for the continuous support and guidance during my work on this master's thesis.

Contents

Chapter 1

Introduction

In the formal language theory, the concept of formal grammars has been established as a way of describing various formal models. Based on their properties, there exists a wide variety of grammar types; however, the most notable criterion is the form of production rules the individual grammars allow. Generally, the fewer restrictions are placed on the rule form, the more languages can be described by the given grammar family, and the greater the generative power of the family is. Another way to categorise formal grammars is the order in which their sentential forms are rewritten – in case a single symbol is rewritten at a time, the grammar in question is sequential; if the entire sentential form is replaced, the grammar is parallel – also called an L-system. However, the less restricted a grammar type is, the more complex the process of its parsing is, resulting in a lack of dedicated parsing methods.

Syntax analysis, or parsing, can be utilised in many disciplines focused on string evaluation; these include program compilation, speech analysis, or simulation of biological processes. However, in practice it is usually limited by a number of factors – most algorithms are designed for application on sequential, at most context-free, grammars. Even inside this family, application of individual parsing algorithms is conditioned by the absence of various grammar properties, such as existence of left recursion, grammar ambiguity, symbol erasing, or repeated occurrences of the right-hand side prefix in rules. Naturally, it is not always possible to construct an equivalent grammar that meets these conditions.

However, there exists a specific group of parsing methods, called the general methods, that are able to bypass these restrictions, and therefore process any grammar of the family they were designed for. Parsing methods based on normal forms present a special subgroup of the general parsing methods. These utilise the fact that for any grammar, an equivalent grammar in a normal form can be constructed, and base their behaviour around the specific production rule forms allowed by the individual normal forms. A well-known representative of this group of parsing algorithms is the Cocke-Younger-Kasami parsing algorithm for context-free grammars in the Chomsky normal form. To determine a string's correspondence to the specified grammar, the algorithm builds alternative parse trees for the input string in a bottom-up way.

Because of the lack of dedicated parsing algorithms for non-context-free grammars, there have been notions of adapting existing parsing algorithms to suitable grammar types with an emphasis on the structural similarities between the original and adapted grammar family. This notion can be represented by the extension of the Cocke-Younger-Kasami algorithm for Watson-Crick grammars [18]. Naturally, the addition of the concept of context to parsing results in increased temporal requirements in the algorithms; in this case, the original time complexity of the Cocke-Younger-Kasami algorithm is multiplied by a power of two. However, there exist grammar types whose structure requires changes result in lower final time complexity.

The aim of this thesis is to design a set of algorithms capable of making a decision of whether a string belongs into the language generated by the specified non-context-free grammar. These algorithms are all based on the Cocke-Younger-Kasami algorithm, and work with grammars in normal forms most reminiscent of the Chomsky normal form that the original algorithm was designed for.

The extensions of the Cocke-Younger-Kasami algorithm presented in this thesis work with both sequential and parallel grammars. These sequential algorithms are designed for context-sensitive grammars, scattered context grammars and multigenerative grammar systems, whereas the parallel algorithms work with both non-interactive L-systems, and L-systems with unlimited environment size. Apart from the core provided by the CYK algorithm, each grammar type presents a set of complications that need to be prevented for the algorithm to yield reliable results. However, despite these complications, the universality offered by the algorithm remains intact.

This thesis is composed of several chapters. The first part focuses on the individual theoretical aspects of its background, followed by a theoretical introduction to the proposed algorithms, and their implementation.

Chapter 2 presents a theoretical introduction to sequential grammars. First, it gives an overview of formal terms used in the context of sequential grammars. Then it presents the Chomsky hierarchy as a means of categorising sequential grammars based on their generative power. Finally, it introduces some of the grammar types examined in this thesis in greater detail, along with the corresponding normal forms.

Similarly, Chapter 3 presents an introduction to parallel grammars, their purpose, and differences with sequential grammars. Then it presents the system of grammar descriptors used to modify the properties of individual families, and by extension their generative power. Finally, it examines some of the major grammar families; for these grammars, a set of normal forms is introduced.

The theoretical background of this thesis is concluded by Chapter 4, which serves as an introduction to parsing techniques. It examines the role of the parser in a compiler, and its role in general. Then it introduces basic terms used to define properties of parsing techniques, such as parse tree or grammar ambiguity. Finally, it focuses on each of the main types of parsing methods: the top-down methods, the bottom-up methods, and lastly the general methods, which are represented by the Cocke-Younger-Kasami algorithm. For this algorithm, graphical representation of the parsing process is supplied.

Chapter 5 presents the proposed extensions of the Cocke-Younger-Kasami algorithm. For each of these extensions, an informal description is given; all major parts of the algorithms are further represented by pseudocode descriptions. Finally, graphical representation of the extensions' progression, similar to that of the original algorithm, is offered. For each of the algorithms, worst-case time and space complexity is derived.

Finally, Chapter 6 introduces the working C++ prototype implementing these algorithms. The chapter first presents the overall architecture of the program with a focus on the polymorphic classes. Then, it examines the individual modules of the program responsible for input parsing, grammar and matrix representation, and implementation of the parsing algorithms. The last part of the chapter then compares the time and space complexity of the algorithm implementation to their theoretical counterpart.

Chapter 2

Sequential Grammars

This chapter serves as an introduction to sequential grammars. The first part of the chapter presents a survey of basic terms used in the formal language theory, followed by an introduction to sequential grammars as perceived in this thesis. The later part of the chapter presents the Chomsky hierarchy as a means of categorising grammar families based on their generative power, followed by several sections focusing on significant families that are either included in the hierarchy, or do not abide by the restrictions it places. The final part of the chapter focuses on normal forms of the aforementioned grammar types.

2.1 Alphabet, Word, and Language

In the formal language theory, languages are defined using alphabets and words. An alphabet, Σ, is a finite non-empty set consisting of symbols, also called letters. A string, or a word, is a finite sequence of symbols contained in the alphabet Σ [17]. This thesis uses the terms interchangeably.

Definition 2.1.1. A word consisting of no symbols is called the empty word, denoted as ε. It is defined as follows:

- ε is a word over the alphabet, Σ,
- if w is a string over the alphabet Σ, and $x \in \Sigma$, wx is a string over Σ [12].

Definition 2.1.2. Let w be a word over the alphabet, Σ. The length of the word denotes the number of the symbols the word consists of, and is defined as follows:

- if $w = \varepsilon$, then $|w| = 0$,
- if $w = a_1a_2 \dots a_n$, where $n \geq 1$ and $a_i \in \Sigma$ for some $1 \leq i \leq n$, then $|w| = n$ [12].

Let Σ be an alphabet, where $\Sigma = \{0, 1\}$. Then, the string ε, 0, 1, 001 are strings over Σ.

Definition 2.1.3. The set of all string over an alphabet, Σ, is denoted by Σ^*; this set includes the empty string, ε. The set of all non-empty words in the alphabet, Σ, is denoted as Σ^+. It is defined as:

$$\Sigma^+ = \Sigma^* \setminus \{\varepsilon\} \text{ [17].} \tag{2.1}$$

Operations over Words

This subsection focuses on operations used to manipulate words. These operations present the basis for formal language generation, and are used by the grammar types discussed in the following sections.

Definition 2.1.4. Let w, x be words over an alphabet, Σ. Then the concatenation of these words is defined as follows:

- concatenation of w and x is equal to wx,
- if $x = \varepsilon$, then $wx = xw = w$ [12].

Definition 2.1.5. Let w be a word over the alphabet, Σ. Then, the power of w is recursively defined for non-negative integers as:

- $w^0 = \varepsilon$,
- $w^i = ww^{i-1}$ for some $i \geq 1$ [12].

Definition 2.1.6. Let w, x be words over an alphabet, Σ. The word x is a prefix of w if there exists a word, y, such that $xy = w$. If $y \notin \{\varepsilon, w\}$, the word x is the proper prefix of w.

For a word, w, the function $prefix(w)$ denotes the set of all viable prefixes. It is defined as follows:

$$prefix(w) = \{x : \ x \text{ is a prefix of } w\} \text{ [12]}. \tag{2.2}$$

Definition 2.1.7. Let w, x be words over an alphabet, Σ. The word x is a suffix of w if there exists a word, y, such that $yx = w$. If $y \notin \{\varepsilon, w\}$, the word x is the proper suffix of w.

For a word, w, the function $suffix(w)$ denotes the set of all viable prefixes. It is defined as follows:

$$suffix(w) = \{x : \ x \text{ is a suffix of } w\} \text{ [12]}. \tag{2.3}$$

Definition 2.1.8. Let w, x be words over an alphabet, Σ. The word x is a subword of w if there exist two words, y and y', such that $yxy' = w$. If $x \notin \{\varepsilon, w\}$, the word x is a proper subword of w.

For a word, w, the function $subword(w)$ denotes the set of all its subwords. It is defined as follows:

$$subword(w) = \{x : \ x \text{ is a subword of } w\} \text{ [12]}. \tag{2.4}$$

For every word, w, in an alphabet, Σ, the following properties always hold:

- $prefix(w) \subseteq subword(w)$,
- $suffix(w) \subseteq subword(w)$,
- $\{\varepsilon, w\} \subseteq prefix(w) \cap suffix(w) \cap subword(w)$ [12].

Languages, and Operations over Languages

This section introduces formal languages as a set of strings over an alphabet, Σ, followed by a list of important language operations.

Definition 2.1.9. Let Σ be an alphabet, and let $L \subseteq \Sigma^*$. Then, L is a language over Σ [12].

By this definition, sets $\varnothing$ and $\{\varepsilon\}$ both present languages. However, these languages are not equal – the former contains no strings, whereas the latter contains a single empty string. For any alphabet, Σ, the set Σ^* presents a language of all its strings; such language is called the universal language over Σ [12].

Definition 2.1.10. Let L be a language. The language is finite if $card(L) = n$, otherwise it is infinite [12].

The operations utilised when dealing with formal languages can be also used to generate new languages. Based on the fact that languages are actually sets of words, these operations are inherited from Boolean algebra, such as union, intersection, or complement. For formal languages, these are defined as follows:

$$L_1 \cup L_2 = \{w : \; w \in L_1 \text{ or } w \in L_2\}, \tag{2.5}$$
$$L_1 \cap L_2 = \{w : \; w \in L_1 \text{ and } w \in L_2\}, \tag{2.6}$$
$$L_1 \setminus L_2 = \{w : \; w \in L_1 \text{ and } w \notin L_2\} \text{ [12]}. \tag{2.7}$$

The operations of power of language and language concatenation are defined analogously to their word counterparts.

Definition 2.1.11. The power of the language, L, is defined for non-negative integers as:

- $L^0 = \{\varepsilon\}$,
- $L^i = LL^{i-1}$ for some $i \geq 1$ [12].

Definition 2.1.12. Concatenation of two languages, L_1 and L_2, is defined as follows:

$$L_1L_2 = \{w_1w_2 | w_1 \in L_1, w_2 \in L_2\} \text{ [12]}. \tag{2.8}$$

Definition 2.1.13. The closure and the positive closure of a language, L, are respectively defined as:

$$L^* = \bigcup_{i=0}^{\infty} L^i, \tag{2.9}$$
$$L^+ = \bigcup_{I=1}^{\infty} L^i \text{ [12]}. \tag{2.10}$$

The rest of this thesis focuses on languages generated by formal grammars. These will be discussed further in the following sections.

2.2 Grammar, and Derivation Relation

The following sections focus on formal grammars as a construct serving to transform strings forming a language, as well as the automaton types capable of recognising them. These are based on the definition of phase-structure grammars.

Phase-structure grammar is the most general type of sequential grammar, as well as the main type of the Chomsky hierarchy. Its subfamilies are created by gradual restriction of its properties.

Definition 2.2.1. A phase-structure or type 0 grammar is a quadruple $G = (N, T, P, S)$, where:

- N is a non-empty set of nonterminal symbols,
- T is a set of terminal symbols, where $N \cap T = \varnothing$,
- P is a set of production rules in the form of $v \longrightarrow w$, where $v \in (N \cup T)^* N (N \cup T)^*$, $w \in (N \cup T)^*$,
- S is the start symbol [17].

Definition 2.2.2. For some $v, w \in (N \cup T)^*$, $\alpha \in (N \cup T)^* v (N \cup T)^*$, $\beta \in (N \cup T)^* w (N \cup T)^*$, $v \longrightarrow w \in P$, one can say that α directly derives into β, written as $\alpha \Longrightarrow_G \beta$. If G is implicit, the index can be omitted. The reflexive and transitive closure of the derivation relation, $\Longrightarrow$, can be written as $\Longrightarrow^*$ [12].

Definition 2.2.3. For some $p = v \longrightarrow w \in P$, v is called the left-hand side of production, $lhs(p) = v$. Analogously, w is called the right-hand side, $rhs(p) = w$ [12].

Definition 2.2.4. For some $p \in P$, p represents a left-recursive production if

$$rhs(p) \in \{lhs(p)\}(N \cup T)^* \text{ [12].} \tag{2.11}$$

Definition 2.2.5. For some $p \in P$, $x \in (N \cup T)^*$ and $y \in (N \cup T)^*$, $x\mathrm{lhs}(p)y$ directly derives $x\mathrm{rhs}(p)y$ according to p in G, as denoted by

$$x\mathrm{lhs}(p)y \Longrightarrow x\mathrm{rhs}(p)y \text{ [12].} \tag{2.12}$$

Definition 2.2.6. For some $p \in P$, $x \in T^*$ and $y \in (N \cup T)^*$, $x\mathrm{lhs}(p)y$ directly derives $x\mathrm{rhs}(p)y$ according to p in G in the leftmost way, as denoted by

$$x\mathrm{lhs}(p)y \Longrightarrow_{lm} x\mathrm{rhs}(p)y. \tag{2.13}$$

Analogously, $y\mathrm{lhs}(p)z$ directly derives $y\mathrm{rhs}(p)x$ according to p in G in the rightmost way, as denoted by

$$y\mathrm{lhs}(p)x \Longrightarrow_{rm} y\mathrm{rhs}(p)x \text{ [12].} \tag{2.14}$$

Definition 2.2.7. A string, $w \in (N \cup T)^*$, $S \Longrightarrow^* w$ is called a sentential form. If $w \in T^*$, it is also a sentence [12].

The language generated by a grammar, G, is a set of all sentences generated by G. It is defined as:

$$L(G) = \{S \Longrightarrow^* w \mid w \in T^*\}. \tag{2.15}$$

Two grammars, G_1 and G_2, are equivalent if $L(G_1) = L(G_2)$ [17].

2.3 Chomsky Hierarchy

The Chomsky hierarchy orders grammar types based on their generality, and consequently, their generative power. Overall, the grammar type number increases as its power decreases. It is divided into four sets:

- *type 0*, unrestricted or recursively-enumerable grammars,
- *type 1*, context-sensitive grammars,

- *type 2*, context-free grammars,
- *type 3*, regular grammars [12].

For each of these types, there exists a type of automaton of an equivalent generative power – it is capable of recognising languages generated by the family, but not one of a less restricted types. The automaton types are as follows:

- *type 0*, Turing machine,
- *type 1*, linear-bounded automaton,
- *type 2*, push-down automaton,
- *type 3*, finite state automaton [12].

As such, the hierarchy still retains its importance despite it no longer being the only means of grammar classification. Another example of a hierarchy of similar importance is the L family system discussed in the following chapter. Properties of individual grammar types present in the hierarchy can be found in Table 2.1.

	UR	CS	CF	REG
Union	Y	Y	Y	Y
Intersection	Y	Y	N	Y
Complement	N	Y	N	Y
Concatenation	Y	Y	Y	Y
Kleene *	Y	Y	Y	Y

(a) Closure properties.

	UR	CS	CF	REG
Equivalence	N	N	N	D
Inclusion	N	N	N	D
Membership	N	D	D	D
Emptiness	N	N	D	D
Finiteness	N	N	D	D

(b) Decidability properties.

Table 2.1: Properties of grammar families of the Chomsky hierarchy [17].

Naturally, not all types of sequential grammars adhere to the Chomsky hierarchy. An example of this is the family of multigenerative grammar systems, or the family of scattered context grammars. These will be further discussed in sections 2.6 and 2.7 respectively.

2.4 Unrestricted and Context-Sensitive Grammars

This section focuses on the family of unrestricted grammars, and the family of context-sensitive grammars as their proper subset, focusing on the differences between these sets.

As mentioned at the beginning of this chapter, there are no restrictions on type 0 production rules other than that they need to contain at least one nonterminal on the left-hand side of the rule. As such, they are identical with phase-structure grammars introduced in the definition 2.2.1.

Definition 2.4.1. Context-sensitive grammar is an unrestricted grammar, $G = (N, T, P, S)$, such that every production rule, $p \in P$ is in the form $\alpha A\beta \longrightarrow \alpha w\beta$, where $A \in N$, $\alpha, \beta, w \in (N \cup T)^*$, $w \neq \varepsilon$. In addition, P may contain a rule in the form $S \longrightarrow \varepsilon$, in which case S does not appear on the right-hand side of any production [17].

Language generated by an unrestricted grammar is defined as follows:

$$UR = \{L \mid \exists \text{ an unrestricted grammar, } G\text{, such that } L = L(G)\}. \tag{2.16}$$

Analogously, language generated by a context-sensitive grammar is defined as follows:

$$CS = \{L \mid \exists \text{ a context-sensitive grammar, } G\text{, such that } L = L(G)\} \text{ [17]}. \tag{2.17}$$

Alternatively, type 1 grammars can be defined by placing conditions on the relative length of the left-hand side and the right-hand side of their rules.

Definition 2.4.2. A length increasing (monotonous) grammar is a type 0 grammar, $G = (N, T, P, S)$, such that for each production $p = v \longrightarrow w \in P$, $|v| \leq |w|$. In addition, P may contain a rule in the form $S \longrightarrow \varepsilon$, in which case S does not appear on the right-hand side of any production [17].

The generative power of the family of context-sensitive grammars is equal to that of monotonous grammars, thus they can be used interchangeably.

Turing Machines, and Linear-Bounded Automata

Turing machines, and their partially restricted version, the linear-bounded automata, present the automata types whose generative power is equal to that of unrestricted and context-sensitive grammars respectively.

Definition 2.4.3. A Turing Machine is an ordered system, $M = (Q, \Sigma, \Gamma, \delta, q_0, \Delta, F)$, where Q is a finite set of states, Σ is the input alphabet, Γ is the tape alphabet, $\Gamma \cap Q = \varnothing$ and $\Sigma \subset \Gamma$, $q_o \in Q$ is the initial state, $\Delta \in \Gamma \setminus \Sigma$ is the blank symbol, $F \subseteq Q$ is the set of final states and δ is the transition function,

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}) \text{ [17]}. \tag{2.18}$$

The linear-bounded automata present a special case of Turing machines, analogously to the relationship between unrestricted and context-sensitive grammars. The tape of these machines contains read-only markers signalising the ends of the tape, marked as # and \$ respectively. The differences in their behaviour can be seen in Figure 2.1.

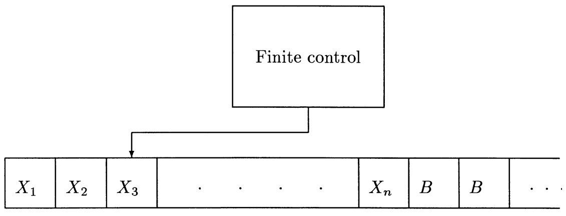
The equivalence of Turing machines and unrestricted grammars is a logical consequence of the Church-Turing thesis – any unrestricted grammar presents a procedure [17]. However, the thesis is not formalised and therefore cannot serve as a proof of equivalence. Instead, it is based on the possibility of conversion between the Turing machines and unrestricted grammars.

Let $G = (N, T, P, S)$ be an unrestricted grammar, and M be a three-tape Turing machine, such that $L(G) = L(M)$. For an input string, $w \in T^*$, M nondeterministically chooses a production $p = u \longrightarrow v \in P$ and a position in the string w. If the right-hand side of p, which is equal to v, is found at the randomly selected position of w, the substring is rewritten. If, after a finite number of operations, the tape contains the start symbol, S, the string is accepted.

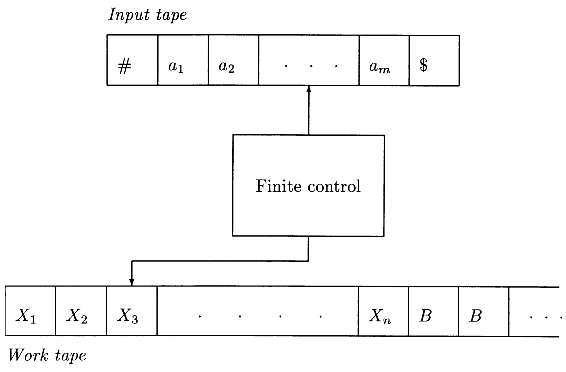
Conversely, a Turing machine can be converted into a grammar, $G = (N, T, P, S)$, where

$$N = ((\Sigma \cup \{\varepsilon\}) \times \Gamma) \cup Q \cup \{S_0, S_1, S_2\}. \tag{2.19}$$

and the production set, P, is as follows:



(a) Turing machine.



(b) Linear-bounded automaton.

Figure 2.1: Behaviour differences between a Turing machine, and its special case, the linear-bounded automaton [17]. B represents the blank symbol, Δ.

1. $S_0 \longrightarrow q_0 S_1$,
2. $S_1 \longrightarrow (a,a)S_1$ for all $a \in \Sigma$,
3. $S_1 \longrightarrow S_2$,
4. $S_2 \longrightarrow (\varepsilon, \Delta)S_2$,
5. $S_2 \longrightarrow \varepsilon$,
6. $q(a,X) \longrightarrow (a,Y)p \iff (p,Y,R) \in \delta(q,X)$, such that $a \in \Sigma \cup \{\varepsilon\}$, $q \in Q$, $X, Y \in \Gamma$,
7. $(b,Z)q(a,X) \longrightarrow p(b,Z)(a,Y) \iff (p,Y,L) \in \delta(q,X)$, such that $a, b \in \Sigma \cup \{\varepsilon\}$, $p, q \in Q$, $X, Y, Z \in \Gamma$,
8. $(a,X)q \longrightarrow qaq$, $q(a,X) \longrightarrow qaq$, $q \longrightarrow \varepsilon$, such that $q \in F$, $a \in \Sigma \cup \{\varepsilon\}$, $X \in \Gamma$.

If the rewriting proceeds in three steps, first when rules $1-5$ are used, second when the rules $6-7$ are used, and the final, which uses the rule 8, the process simulates the rewriting of the Turing machine [17, pg. 178-179]. This way, the equality of the languages is proven.

Analogously, since the linear-bound automaton cannot exceed the number of tape cells provided at the beginning of the rewriting, the linear-bounded automaton simulates the monotonousness of the context-sensitive grammars.

2.5 Matrix Grammars

A matrix grammar, G, is a context-free grammar containing an extra component, M. This component represents a final set of production sequences of P_G. During each derivation step in this grammar, a sequence $m = p_1 p_2 \dots p_n \in M$ is chosen, and each of the contained productions must be applied in the predefined order [11].

Definition 2.5.1. Let $H = (G, M)$ be a matrix grammar, and let G allow a sequence of context-free derivation steps as follows:

$$u_0 \Longrightarrow u_1[p_1] \Longrightarrow u_2[p_2] \Longrightarrow \cdots \Longrightarrow u_n[p_n] \tag{2.20}$$

and let $m = p_1p_2 \dots p_n \in M$ be a sequence. Then u_0 directly derives into u_n in the matrix grammar H according to the matrix m, denoted as $u_0 \Longrightarrow u_n[m]$, also written as $u_0 \Longrightarrow u_n$ [11].

Definition 2.5.2. Let $H = (G, M)$ be a matrix grammar.

- Let $u \in (N \cup T)^*$. Then u derives in zero steps, written as $u \Longrightarrow^0 u[\varepsilon]$.
- Let $u_0, u_1, \dots u_n \in (N \cup T)^*$ and $u_{i-1} \Longrightarrow u_i[p_i]$ for all $1 \leq i \leq n$. Then u_0 derives into u_n in n steps, written as $u_0 \Longrightarrow^n u_n[p_1p_2 \dots p_n]$ [11].

Definition 2.5.3. Let $H = (G, M)$ be a matrix grammar, where $G = (N, T, P, S)$. The language generated by H, denoted as $L(H)$, is defined as:

$$L(H) = \{w : w \in T^* \wedge S \Longrightarrow^* w \text{ in the matrix grammar } H\} \text{ [11]}. \tag{2.21}$$

For every matrix grammar, H, there exists an equivalent unrestricted grammar, H', and therefore their generative powers are equal [11]. This property can be used to derive the generative power of multigenerative grammar systems discussed in the following section.

2.6 Multigenerative Grammar Systems

Multigenerative grammar systems are similar to the matrix grammars in the sense that they take advantage of symbol sequences to direct rewriting. Because of this, an equivalent matrix grammar can be constructed for any multigenerative grammar system – their generative power is equal to that of the unrestricted grammars [11].

An n-generative grammar system is composed of n context-free grammars rewritten in parallel. During each derivation step, a production rule is applied on all current sentential forms. These sentential forms are then compared to a control sequence of nonterminal or production rules. Finally, once all sentences have been finalised, a language operation is applied on the n sentences to produce the overall result.

Generally, three main operations can be applied on the languages – *union*, *concatenation*, and *first*. Based on the control mechanism, there exist two types of multigenerative systems of equal power – nonterminal-synchronised and rule-synchronised systems [11]. All of these types are capable of generating non-context-free grammars.

The main type of multigenerative grammar systems are the canonical systems. In these systems, the rewriting is restricted to the use of the left derivation.

Definition 2.6.1. Canonical n-generative nonterminal-synchronised grammar system (n-NSG) is an $(n + 1)$-tuple defined as:

$$\Gamma = (G_1, G_2, \dots G_n, Q) \tag{2.22}$$

where

- $G_i = (N_i, T_i, P_i, S_i)$ is a context-free grammar for all $1 \leq i \leq n$,
- Q is a finite set of nonterminal control sequences in the form of $(A_1, A_2, \dots A_n$, where $A_i \in N_i$ for all $1 \leq i \leq n$ [11].

Definition 2.6.2. Multiform for a multigenerative grammar system is defined as $\chi = (x_1, x_2, \dots, x_n)$, where $x_i \in (N_i \cup T_i)^*$ for all $1 \leq i \leq n$.

Definition 2.6.3. Let $\chi = (u_1A_1v_1, u_2A_2v_2 \dots u_nA_nv_n)$ and $\bar{\chi} = (u_1x_1v_1, u_2x_2v_2 \dots u_nx_nv_n)$ be multiforms of an n-NSG, $A_i \in N_i$, $u_i \in T_i^*$, $v_i, x_i \in (N_i \cup T_i)^*$, $A_i \longrightarrow x_i \in P_i$ for all $1 \leq i \leq n$, and $(A_1, A_2, \dots, A_n) \in Q$. Then χ directly derives into $\bar{\chi}$, denoted as $\chi \Longrightarrow \bar{\chi}$ [11].

Definition 2.6.4. An n-language generated by Γ is defined as:

$$n - L(\Gamma) = \{(w_1, \dots w_n) : (S_1, \dots S_n) \Longrightarrow^* (w_1, \dots w_n), w_i \in T_i^* \text{ for all } 1 \leq i \leq n\} \text{ [11]}. \tag{2.23}$$

As mentioned at the beginning of this section, the final language generated by a multi-generative grammar system is created by applying an operation on the sentences generated by the individual context-free grammars. Based on the applied operation mode, the generated language is defined as follows.

Definition 2.6.5. Let $\Gamma = (G_1, G_2, \dots, G_n, Q)$ be a canonical n-generative nonterminal-synchronised system. The language generated by Γ in the union mode is defined as:

$$L_{union}(\Gamma) = \bigcup_{i=1}^{n} \{w_i : (w_1, w_2, \dots, w_n) \in n - L(\Gamma)\} \text{ [11]}. \tag{2.24}$$

Definition 2.6.6. Let $\Gamma = (G_1, G_2, \dots, G_n, Q)$ be a canonical n-generative nonterminal-synchronised system. The language generated by Γ in the concatenation mode is defined as:

$$L_{conc}(\Gamma) = \{w_1w_2 \dots w_n : (w_1, w_2, \dots, w_n) \in n - L(\Gamma)\} \text{ [11]}. \tag{2.25}$$

Definition 2.6.7. Let $\Gamma = (G_1, G_2, \dots, G_n, Q)$ be a canonical n-generative nonterminal-synchronised system. The language generated by Γ in the first mode is defined as:

$$L_{first}(\Gamma) = \{w_1 : (w_1, w_2, \dots, w_n) \in n - L(\Gamma)\} \text{ [11]}. \tag{2.26}$$

As opposed to the nonterminal-synchronised grammar systems, the production-synchronised systems work by picking a production from the set of control sequences, Q, and applying it on a nonterminal in the processed sentential form.

Definition 2.6.8. Canonical n-generative production-synchronised grammar system (n-PSG) is an $(n+1)$-tuple defined as:

$$\Gamma = (G_1, G_2, \dots G_n, Q) \tag{2.27}$$

where

- $G_i = (N_i, T_i, P_i, S_i)$ is a context-free grammar for all $1 \leq i \leq n$,
- Q is a finite set of production control sequences in the form of $(p_1, p_2, \dots p_n$, where $p_i \in P_i$ for all $1 \leq i \leq n$ [11].

Definition 2.6.9. Let $\chi = (u_1A_1v_1, u_2A_2v_2 \dots u_nA_nv_n)$ and $\bar{\chi} = (u_1x_1v_1, u_2x_2v_2 \dots u_nx_nv_n)$ be multiforms of an n-PSG, $A_i \in N_i$, $u_i \in T_i^*$, $v_i, x_i \in (N_i \cup T_i)^*$ for all $1 \leq i \leq n$. Followingly, let $p_i = A_i \longrightarrow x_i \in P_i$ for all $1 \leq i \leq n$ and $(p1, p_2, \dots, p_n) \in Q$. Then χ directly derives into $\bar{\chi}$, denoted as $\chi \Longrightarrow \bar{\chi}$ [11].

So far, this section has dealt exclusively with canonical grammar systems, which always apply the leftmost nonterminal in each of their grammars. The remaining system types include the general grammar systems, and the hybrid systems.

As opposed to the canonical systems, the general systems are not limited to rewriting the leftmost nonterminal, and may therefore rewrite their sentential forms in any order.

The final type of multigenerative grammar systems are the hybrid systems. These incorporate both of the previously mentioned approaches, and allow a combination of either canonical or general rewriting in their grammars.

These systems are defined analogously to the canonical systems. However, this thesis focuses on the canonical systems, and therefore, the other types will be omitted.

2.7 Scattered Context Grammars

The scattered context grammars present a form of semi-parallel grammars – they allow application of several context-free rules at different positions of a sentential form at the same time. Similarly to multigenerative grammar systems in the concatenation and union modes, they are capable of generating non-context-free languages [14]. The generative power of grammars of this family depends on several factors, such as the degree of context-sensitivity and the presence of symbol erasure, both of which will be discussed in this section.

Definition 2.7.1. A scattered context grammar is a quadruple $G = (V, T, P, S)$ where:

- V is the total alphabet,
- $T \subset V$ is the set of terminals,
- P is a finite set of productions in the form

  $$(A_,1A_2, \dots A_n) \longrightarrow (x_1, x_2, \dots x_n),$$

  where $n \geq 1$, $A_i \in V \setminus T$, $x_i \in T^*$ for all $1 \leq i \leq n$,
- $S \in V \setminus T$ is the start symbol [14].

Definition 2.7.2. If $u = u_1A_1 \dots u_nA_nu_{n+1}$, $v = u_1x_1 \dots u_nx_nu_{n+1}$, $p = (A_1, \dots A_n) \longrightarrow (x_1, \dots x_n) \in P$, where $u_i \in V^*$ for all $1 \leq i \leq n+1$, then G makes a derivation step from u to v according to p, written as $u \Longrightarrow_G v[p]$ or simply $u \Longrightarrow_G v$ [14].

Definition 2.7.3. Let $p = (A_1, A_2, \dots, A_n) \longrightarrow (x_1, x_2, \dots, x_n) \in P$ for some $n \geq 1$ be a scattered context production. The length of the rule, $len(p)$ is equal to n [14].

Definition 2.7.4. Let G be a scattered context grammar. Then, the degree of context-sensitivity of G is equal to the number of productions, $p \in P$, such that $len(p) > 1$ [4].

Let $p \in P$ be a scattered context production. If $len(p) = 1$, the production is context-free. Consequently, if the degree of context-sensitivity of a scattered context grammar, G, is equal to zero, the grammar is context-free.

Definition 2.7.5. A scattered context grammar, G, is propagating, if each production $p = (A_1, \dots A_n) \longrightarrow (x_1, \dots x_n) \in P$ satisfies $x_i \in V^+$ for all $1 \leq i \leq n$ [14].

Definition 2.7.6. Language generated by a scattered context-grammar, G, is defined as follows:

$$L(G) = \{x : x \in T^*, S \Longrightarrow_G x\} \text{ [14]}. \tag{2.28}$$

As mentioned previously, context-free grammars are a special case of scattered context grammars. Consequently, by increasing the degree of context-sensitivity of a scattered context grammar, its generative power increases. Thus, the family of context-free grammars is properly contained in the family of propagating scattered context grammars, whose power is equal to that of the context-sensitive grammars. Formally,

$$\mathcal{L}_{CF} \subset \mathcal{L}_{PSCG} \subseteq \mathcal{L}_{CS} \text{ [14]}. \tag{2.29}$$

Finally, for any unrestricted grammar, G, an equivalent scattered context grammar, G', containing at most six context-sensitive productions and six nonterminals, or seven context-sensitive productions and five nonterminals, can be constructed [14]. This thesis will focus on the family of propagating scattered context grammars, $\mathcal{L}_{PSCG}$.

2.8 Normal Forms of Sequential Grammars

This section focuses on normal forms of the discussed sequential grammar types. Normal forms of grammars present a uniform format of grammar representation, which allows easier processing while maintaining the generative power of the grammars [12].

The following subsections discuss significant normal forms of the previously mentioned grammar types in the same order as they were introduced. These normal forms share the characteristic of being structurally similar to the Chomsky hierarchy discussed in the first subsection. The similarities will be discussed in their respective subsections.

Context-Free Grammars and Multigenerative Systems

This subsection focuses on normal form of context-free grammars. First, the Chomsky normal form, which is arguably the most significant normal form of context-free grammars, is presented, followed by the Greibach normal form.

Definition 2.8.1. Let $G = (N, T, P, S)$ be a context-free grammar. G is in the weak Chomsky normal form if each nonterminal rule has a right member in N^* and each terminal rule has a right member in $T \cup \{\varepsilon\}$ [17].

Definition 2.8.2. Let $G = (N, T, P, S)$ be a context-free grammar. G is in the Chomsky normal form if every rule, $p \in P$ satisfies $rhs(p) \in (T \cup N^2)$ [12]. Based on this, grammar in the Chomsky normal form has productions in one of the following forms:

- $A \longrightarrow BC$, where $A, B, C \in N$,
- $A \longrightarrow a$, where $A \in N$, $A \in T$ [12].

Definition 2.8.3. Let $G = (N, T, P, S)$ be a context-free grammar. G is in the Greibach normal form if every production $p \in P$ satisfies

$$rhs(p) \in TN^* \text{ [12]}. \tag{2.30}$$

Originally, these normal forms were defined for the context-free grammars only. However, they can be applied to other grammar types, such as the multigenerative grammar systems and matrix grammars, both of which are based on context-free grammars.

Unrestricted and Context-Sensitive Grammars

The lack of structural limitations in unrestricted and context-sensitive grammars allows them to contain more than one nonterminal on the left-hand side of their productions. Naturally, this phenomenon is reflected in the normal forms used for these grammar families. This subsection presents two such normal forms – the Kuroda normal form, and its special case, the Penttonen normal form.

Definition 2.8.4. A grammar, G, is in the Kuroda normal form if its rules are in one of the following forms:

- $AB \longrightarrow CD$, where $A, B, C, D \in N$,
- $A \longrightarrow BC$, where $A, B, C \in N$,
- $A \longrightarrow a$, where $A \in N$, $A \in T$,
- $A \longrightarrow \varepsilon$, where $A \in N$ [12].

In the special case of the first rule form where $A = C$, the grammar is also in the Penttonen normal form [12].

To adapt the Kuroda normal form to context-sensitive grammars, the last rule form introduced in the definition 2.8.4 is removed. This results in the grammar becoming non-erasing, which satisfies the properties of context-sensitive grammars.

Scattered Context Grammars

The normal form used for scattered context grammars reflects the semi-parallelism of the grammar type. The 2-limited normal form simulates application of two context-free productions at different positions in the sentential form. However, as opposed to the Chomsky normal form introduced in the definition 2.8.2, the normal form does not require the right-hand side of productions to consist of a single symbol type.

Definition 2.8.5. The 2-limited normal defines rule forms as follows:

- $(A_1, \ldots A_n) \longrightarrow (x_1, \ldots x_n)$ implies than $n \leq 2$, and for every $1 \leq i \leq n$, $1 \leq |w_i| \leq 2$ and $w_i \in (V \setminus \{S\})^*$,
- $A \longrightarrow w$ implies $A = S$ [14].

The final difference, when compared to the previously mentioned normal forms, lies in the fact that the normal form requires all productions to be context-sensitive. Context-free productions are allowed only at the beginning of the rewriting process, and are used to generate the first pair of nonterminals.

Chapter 3

Parallel Grammars

L-systems, or parallel grammars, were created as a result of the attempt to translate the phenomenons of developmental biology to formal language theory. As such, L-systems allow formal respresentation and mirroring of multicellular processes [6]. Based on defining properties of the processes, several language families were introduced. Notably, these include the families of D0L, 0L, DT0L, E0L and EPIL languages, some of which will be discussed in this chapter. Nowadays, the fundamental L families constitute a testing ground of significance comparable to that of the Chomsky hierarchy [17].

As opposed to sequential grammars discussed in the previous chapter, L-systems require every symbol to be rewritten during a derivation step. This property could be partially simulated by a sequence of sequential steps; however, some of the mechanisms, such as the rewriting synchronisation, are not established in sequential operations. Therefore, the parallelism cannot be simulated to its full extent [17].

An example of this behaviour can be demonstrated on the string a. Applying only the rule $a \longrightarrow aa$, sequential rewriting would lead to the language $\{a^i \mid i \geq 1\}$. However, if the same rule was to be applied in parallel, it would lead to the final language $\{a^{2^i} \mid i \geq 0\}$ [17]. Thus, the generated languages differ.

Biological processes do not always process in a linear way – cell division and death often takes effect. To reflect the division, L-systems offer evolutionary branching mechanisms. In textual representation, bounds of the new branch are marked with $[,]$ symbols; their visual representation can be seen in Figure 3.1. However, this mechanism can only be used for the non-interactive systems without further adjustments. The biological background of interactive L-systems complicates their application, and significantly restricts it [6].

As mentioned at the beginning of this chapter, many new families of L-systems have been introduced since their initial establishment, creating a hierarchy of importance comparable to the Chomsky hierarchy, which was discussed in the previous chapter. The hierarchy will be further discussed in Section 3.2. The following sections will focus on some of the core families, such as the 0L and table systems, and other important families, such as the extended and interactive systems, discussed in sections 3.3 and 3.5 respectively. Finally, the last part of the chapter will present normal forms of these types of grammars.

3.1 Substitutions and 0L Systems

As opposed to sequential grammars, L-systems in their purest form do not differentiate between types of symbols – the basic rewriting mechanism is the morphism. This section introduces the term, and based on it, defines the simplest L-system family – the 0L systems.

Definition 3.1.1. A finite substitution σ over an alphabet Σ is a mapping of Σ^* into the set of all finite nonempty languages defined as follows. For each letter $a \in \Sigma$, $\sigma(a)$ is a finite nonempty language, $\sigma(\varepsilon) = \varepsilon$, and for all words $w_1, w_2 \in \Sigma^*$, $\sigma(w_1)\sigma(w_2) = \sigma(w_1 w_2)$ [17].

Definition 3.1.2. If none of the languages $\sigma(a)$, $a \in \Sigma$, contains the empty word, the substitution σ is referred to as ε-free or non-erasing. If each $\sigma(a)$ consists of a single word, σ is called a morphism [17].

Customarily, letter-to-letter morphisms found in L-systems are called coding; morphisms mapping each letter to a letter or to the empty word, ε, are called the weak codings. Subsequently, if $\sigma(a)$ presents a morphism, the resulting rewriting process is deterministic [16].

The morphisms presented in this section are generally parallel – no part of the processed string can remain unchanged. In L-systems this notion can be used to define generated languages.

Definition 3.1.3. For all languages, $L \subseteq \Sigma^*$, the following stands:

$$\sigma(L) = \{u \mid u \in \sigma(w), \text{ for some } w \in L\} \text{ [17].} \tag{3.1}$$

Definition 3.1.4. An 0L system is a triple $G = (\Sigma, \sigma, w_o)$, where:

- Σ is an alphabet,
- σ is a finite substitution on Σ,
- w_0, referred to as the axiom, is a word over Σ.

Definition 3.1.5. The language generated by the system, G, is defined as follows:

$$L(G) = \{w_0\} \cup \sigma(w_0) \cup \sigma(\sigma(w_0)) \cup \cdots = \bigcup_{i \geq 0} \sigma^i(w_0) \text{ [17].} \tag{3.2}$$

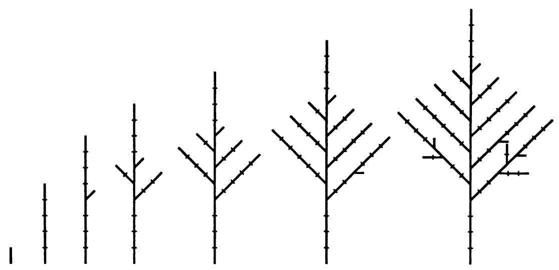


Figure 3.1: L-system branching [17]. Angled lines represent new branches of the organism development.

D0L systems were one of the first established families. Despite their simplicity, they offer valuable tools in modelling both biological processes and sequences.

Definition 3.1.6. A 0L system, (Σ, σ, w_0), is deterministic, or a D0L system, if σ is a morphism [17].

Definition 3.1.7. Let (Σ, h, w_0) be a D0L system. The system, G generates its language $L(G)$ in a specific order, as a sequence:

$$w_0, w_1 = h(w_0), w_2 = h(w_1), \ldots. \tag{3.3}$$

The sequence is denoted by $S(G)$ [17].

Definition 3.1.8. Let G_1 and G_2 be D0L systems. G_1 and G_2 are sequence-equivalent if $S(G_1) = S(G_2)$. They are language-equivalent if $L(G_1) = L(G_2)$ [17].

The determinism of D0L systems allows formation of periodicity in the generated language. For example, let a word, w_i, appear twice in the same sequence, $w_i = w_{i+k.j}$ for some $i, k \geq 0$, $j > 0$. Then, after the period j, the sequence starts repeating, which makes the generated language finite. Consequently, the appearance of repetition in $S(G)$ presents the necessary condition for finiteness of the generated language, $L(G)$ [17].

Definition 3.1.9. An infinite sequence of words, w_i, $i \geq 0$, is locally catenative if, for some positive integers $k, i_1, \ldots i_k$ and $q \geq max(i_1, \ldots i_k)$, $w_n = w_{n-i_1} \ldots w_{n-i_k}$ whenever $n \geq q$. A D0L system, G, is locally catenative if the sequence $S(G)$ is locally catenative [17].

This section serves as an introduction to the basic families of L-systems. However, the hierarchy offers a wide variety of L-system property modifiers, which will be discussed in the following section.

3.2 System Modifiers

0L and D0L systems, which were presented in the previous section, offer no means of regulating the generated language. Considering the L-systems were created to mirror biological processes, which do not differentiate between right or wrong states of the generated string, this behaviour is to be expected. However, in the formal language theory, filtering mechanisms similar to those present in sequential grammars are expected.

For this purpose, a number of filtering mechanisms were designed, such as the extended L-systems and table L-systems, discussed in Sections 3.3 and 3.4 respectively. However, these are not the only reasons for language property extension. The following list aims to illustrate some of the commonly used modifiers:

- *A.* adult – adult word, adult language,
- *C.* coding – letter-to-letter morphism,
- *D.* deterministic,
- *E.* extended – discussed in Section 3.3,
- *F.* finite – a finite set of axioms,
- *I.* interactive – discussed in Section 3.5,

- O. non-interactive,
- P. propagating – non-erasing, no cell death,
- T. table – discussed in Section 3.4,
- U. unary – alphabet consisting of one letter [17].

These modifiers can be combined at will – except for the obvious contradiction of interactive and non-interactive modifiers – to create a system of desired properties. However, as opposed to the Chomsky hierarchy, the generative power of the individual families does not increase in a linear matter. A comparison of their power can be seen in Figure 3.2.

3.3 Extended L-Systems

Extended L-systems introduce filtering mechanisms similar to those present in sequential grammars – they divide the overall alphabet into two disjoint sets, terminals and nonterminals. The use of nonterminals serves as a way to signalise that the examined string is not yet complete [16]. From the biological point of view, the existence of extended L-systems does not make sense, however, this problem is countered by the increase in generative power the systems offer [6].

Definition 3.3.1. An E0L system is a quadruple $G = (V, \Sigma, P, S)$, where:

- V is the total alphabet,
- Σ is the finite set of terminals – the target alphabet,
- P is the set of rules in the form $A \longrightarrow w$, where $A \in V$, $w \in V^*$,

$$\begin{tikzcd}
 & & \begin{array}{c} HD0L = WD0L = \\ = HPD0L = WPD0L = \\ = HDF0L = WDF0L = \\ = HPDF0L = WPDF0L \\ = NDF0L = CDF0L \\ = NPDF0L = CPDF0L \end{array} & \\
 & ND0L = NPD0L = CD0L \arrow[ur] & & EDF0L \arrow[ul] \\
CPD0L \arrow[ur] & ED0L \arrow[u] \arrow[urr] & DF0L \arrow[ur] & EPDF0L \arrow[u] \\
 & EPD0L \arrow[ul] \arrow[u] & D0L \arrow[ul] \arrow[u] & PDF0L \arrow[ul] \arrow[u] \\
 & & PD0L \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Figure 3.2: Hierarchy of generative power of L-system families. Power of classes on the same level cannot be directly compared [17].

- S is the axiom [17].

A derivation step in the extended L-systems presents a middle ground between the parallel morphisms as per Definition 3.1.1 and the sequential derivation step as introduced in Definition 2.2.5 – the rewriting happens for all symbols in parallel, however, only nonterminals may be rewritten into other symbols. Once the symbol is rewritten into a terminal, the value is always replaced by itself [6].

In terms of generative power, E0L grammars are more powerful than 0L grammars. Within the L hierarchy, their power is equal to that of the following families:

$$\begin{aligned} E0L &= C0L = N0L = W0L = H0L = NP0L = EP0L = WP0L \\ &= HP0L = EF0L = CF0L = NF0L = WF0L = HF0L \\ &= EPF0L = NPF0L = WPF0L = HPF0L \text{ [17].} \end{aligned} \tag{3.4}$$

When compared to the Chomsky hierarchy, they are more powerful than the family of context-free grammars, but less powerful than the family of context-sensitive languages,

$$\mathcal{L}(CF) \subset \mathcal{L}(E0L) \subset \mathcal{L}(CS) \text{ [6].} \tag{3.5}$$

3.4 Table L-systems

In parallel rewriting, tables may be used to regulate the development. In this context, a table is a set of production rules – as such, this notion is of no significance in sequential grammars. A system has a finite number of tables, and during each step, a single table must be used [17].

This mechanism mirrors the properties of real organisms, as they often require different conditions during their development, or may simply behave in a different way [6].

Definition 3.4.1. T0L system is a triple $G = (\Sigma, S, w_o)$, where S is a finite set of finite substitutions such that, for each $\sigma \in S$, the triple (Σ, S, w_0) is a 0L system. The language of the T0L system, $L(G)$, consists of w_0 and of all words in all languages $\sigma_1 \ldots \sigma_k(w_0)$, where $k \geq 1$ and each σ_o belongs to S – some of the σ_is may also coincide. If all substitutions in S are, in fact, morphisms then G is deterministic or a DT0L system [17].

The definition above presents the notion of DT0L system, as opposed to DT0L language. This is caused by the fact that the system uses a set of tables, whose order of use is not defined. Therefore, the system does not generate a DT0L language in a sequence. To achieve a deterministic language, the order of use must be defined in a special way [6].

Definition 3.4.2. For an infinite sequence w_i, where $i \geq 0$ of words, the function $f(n) = |w_n|$ – mapping the set of non-negative integers into itself – is termed the growth function of the sequence [17].

The families of table L-systems present one of the most widely studied groups of families in the L-hierarchy [17]. The hierarchy of the basic table families can be seen in Figure 3.3.

3.5 Interactive L-Systems

The class of interactive L-systems is unique in the notion that the individual cells, or symbols, interact with environment – other cells on their sides. Overall, number of interacting

$$\begin{tikzcd}
 & \mathrm{DTF0L} & \\
\mathrm{DT0L} \arrow[ur] & & \mathrm{PDTF0L} \arrow[ul] \\
 & \mathrm{PDT0L} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 3.3: Hierarchy of the generative power of table L-systems without further filtering mechanisms [17].

cells on either side does not matter when determining power of the system; only the total environment size is important. Subsequently, the environment size increases the generative power of the system. All of the following families are equal:

$$(4,1)L = (3,2)L = (2,3)L = (1,4)L \text{ [17]}. \tag{3.6}$$

However, the generative power of the $(5,0)L$ and $(0,5)L$ families, and by extension the other $(0,x)$ and $(x,0)$ systems, is not comparable, as discussed in Section 3.6.

Definition 3.5.1. Let k and l be non-negative integers. A (k, l) system is a quadruple $G = (\Sigma, P, g, w_0)$, where:

- Σ is the alphabet,
- P is the set of productions, $P \subset (\Sigma \cup \{g\})^k \times \Sigma \times (\Sigma \cup \{g\})^l \times \Sigma^*$,
- g is the environment marker, $g \notin \Sigma$,
- w_0 is the axiom,

such that for some $p = (w_1, a, w_3, w_4) \in P$, p satisfies the following conditions:

- if $w_1 = \bar{w_1} g \bar{\bar{w_1}}$ for some $\bar{w_1}, \bar{\bar{w_1}} \in (\Sigma \cup \{g\})^*$, then $w_1 \in \{g\}^*$,
- if $w_3 = \bar{w_3} g \bar{\bar{w_3}}$ for some $\bar{w_3}, \bar{\bar{w_3}} \in (\Sigma \cup \{g\})^*$, then $\bar{\bar{w_3}} \in \{g\}^*$ [6].

To maintain the consistence of rule representation, this thesis will use the notation $p = w_1 < \ a \ > w_3 \longrightarrow w_4$.

The family of interactive L-systems is specific in one more way – as mentioned at the beginning of this chapter, the family does not allow development branching during the rewriting process, as this concept disrupts the biological background of the family [6].

Extended Interactive L-Systems

The addition of nonterminals further increases the power of the system. Furthermore, this class of L-systems can serve as a simple interlink to sequential grammars. Union of the system types presented in the previous sections yields the following definition.

Definition 3.5.2. Let k and l be non-negative integers. An extended (k, l) system is the quintuple $G = (V, \Sigma, P, g, S)$, where:

- V is the total alphabet,
- Σ is the set of terminals,
- P is the set of productions in the form $w_1 < \ a \ > w_3 \longrightarrow w_4$,
- g is the environment marker, $g \notin V$,
- S is the axiom.

It has been established that the EIL family generates all recursively enumerable languages. Furthermore, the family of EPIL – propagating EIL systems – produces the family of context-sensitive languages [17].

3.6 Normal Forms of Parallel Grammars

This section focuses on normal forms of L-system families that were discussed earlier in the chapter. The first part focuses on a normal form for E0L that has been introduced in several publications [17, 16], along with its equivalent used in EIL systems; the second part focuses on normalisation of environment size in IL systems.

Extended L-Systems

This section focuses on normal forms of extended L-systems. Considering normal forms are a concept usually applied in sequential grammars, the normal form is very similar to the Chomsky normal form.

Definition 3.6.1. An E0L system contains rules in one of the following forms:

- $A \longrightarrow BC$, where $A, B, C \in V \setminus \Sigma$,
- $A \longrightarrow B$, where $A, B \in V \setminus \Sigma$,
- $A \longrightarrow a$, where $A \in V \setminus \Sigma$, $a \in \Sigma$,
- $a \longrightarrow A$ where $a \in \Sigma$, $A \in V \setminus \Sigma$,
- $A \longrightarrow \varepsilon$ where $A \in V \setminus \Sigma$ [17, 16].

Extended Interactive L-Systems

The normal form presented in the previous section can be further adapted to IL systems. However, no steps are taken to regulate the context in any way.

Definition 3.6.2. The normal form for EPIL systems allows rules in one of the following forms:

- $w_1 < A > w_3 \longrightarrow BC$, where $A, B, C \in V \setminus \Sigma$, $w_1, w_3 \in V^*$,
- $w_1 < A > w_3 \longrightarrow B$, where $A, B \in V \setminus \Sigma$, $w_1, w_3 \in V^*$,
- $w_1 < A > w_3 \longrightarrow a$, where $A \in V \setminus \Sigma$, $a \in \Sigma$, $w_1, w_3 \in V^*$,
- $w_1 < a > w_3 \longrightarrow A$ where $a \in \Sigma$, $A \in V \setminus$, $w_1, w_3 \in V^*$,
- $w_1 < A > w_3 \longrightarrow \varepsilon$ where $A \in V \setminus \Sigma$, $w_1, w_3 \in V^*$.

With the removal of the last rule form, this normal form can be used for EPIL systems.

Interactive L-Systems

Unlike the previously mentioned normal forms, the normal form for IL systems does not focus on the form of the rules themselves, but on the environment surrounding them.

For every IL system, G, it is possible to find a nonnegative integer l and a $(1, l)$L system, $\bar{G}$, such that G and $\bar{G}$ are equal. This is done by creating a system that moves the bounds of the environment until the desired stare is achieved [6].

This, however, only applies to systems that interact with environment on both sides – it is not applicable to $(n, 0)L$ or $(0, n)L$ systems.

Chapter 4

Syntax Analysis

This chapter focuses on established methods of syntax analysis. First, it introduces the basic terms and concepts concerning parsing methods, followed by an overview of some of the significant methods based on the direction they work in. Finally, it introduces general parsing methods, such as the Earley algorithm, and the Cocke-Younger-Kasami algorithm, which is discussed in detail.

Generally speaking, syntax analysis is one of the central phases of compilation. Code compilation is the process of translating a source program into a target program while maintaining its semantics.

First, the lexer separates the source programs into lexemes, which are then reclassified into tokens and gradually sent to the parser – the compiler component responsible for syntax analysis. This module verifies whether the received sequence of tokens could present a valid sentence of the compiler's grammar, as well as detect any syntactical ambiguity of the input. Based on the general approach of the compiler, this module either generates the parse tree, which is then passed on for semantic analysis and code generation, or directs the processes itself. In the latter case, the process is called syntax-directed translation [1].

Traditionally, compilation makes use of parsing methods created for context-free grammars. However, application of syntax analysis is not limited to code compilation. Based on the grammar type used, it can be utilised for a wide variety of processes focused on data analysis; these will be further explored in the following chapter. The remaining part of this chapter deals with parsing methods focused on context-free grammars.

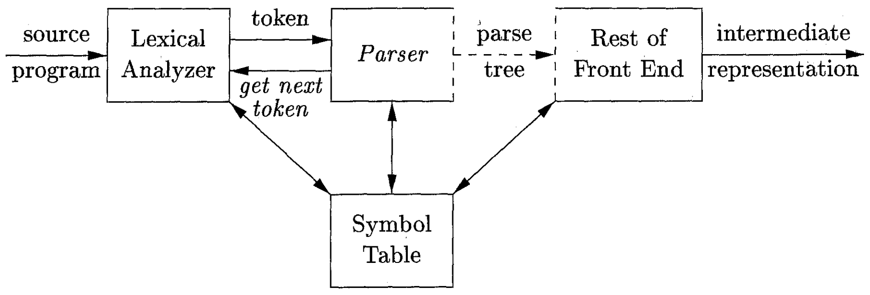
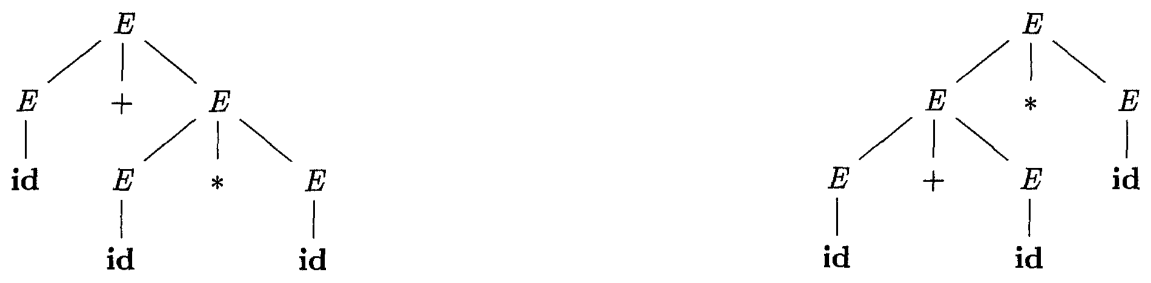


Figure 4.1: The role of the parser in the compiler [1].



(a) Leftmost derivation. (b) Rightmost derivation.

Figure 4.2: Alternate parse trees for the sentence $id + id * id$ [1].

4.1 Derivation, and Parse Tree

A parse tree is a graphical representation of the order in which the sentential form, as per Definition 2.2.7, is rewritten.

As mentioned at the beginning of this chapter, the parse tree is constructed mainly for context-free grammars. As such, the individual nodes of the tree represent the nonterminal of the left-hand side of the applied rule, and the child nodes represent the individual symbols on the right-hand side of the rule, as per Definition 2.2.3.

In top-down parsing methods, which will be discussed in Section 4.2, the tree is constructed starting from the grammar's start symbol, which poses as the root of the tree. Gradually, new nodes are added to the tree until either all leaves are represented by terminals, in which case the sentence is syntactically correct, or there are no viable rules to apply to the tree. In bottom-up parsing methods, discussed in Section 4.3, tree constructions starts with a sequence of terminals – or tokens – and the goal is to reach the root of the tree [1].

Sentence Ambiguity

The final structure of the parse tree is determined by the order in which the production rules are applied. During the rewriting process, the tree is usually constructed by the continuous application of the leftmost derivation, in which case, it is constructed based on the left parse [1]. The sequence of rightmost derivations is defined analogously. In deterministic grammars, there exists a single parse tree for every sentence in the generated language. Otherwise, the grammar is ambiguous.

Let $G = (\{E\}, \{id, +, *\}, P, E)$ be a context-free grammar, such that

$$\begin{aligned} P = \{\ & E \longrightarrow E + E, \\ & E \longrightarrow E * E, \\ & E \longrightarrow id\ \}. \end{aligned} \tag{4.1}$$

For the input sentence $id + id * id$, there exist two distinct parse trees, which can be seen in Figure 4.2. Most parsing methods used during the parsing process are not capable of handling such ambiguity. However, some exceptions, such as the GLR method, exist [1].

4.2 Top-Down Parsing Methods

Top-down parsing methods handle the creation of the parse tree by starting at the root node, and gradually adding nodes, until all tokens of the input string have been handled. In concrete terms, they find the leftmost derivation for the input sentence – in each step of the process, the parser attempts to match the currently examined nonterminal to a production rule, and subsequently match its right-hand side. These methods are not capable of handling left recursion, as per Definition 2.2.4, because they are unable to determine the required level of recursion, and may fall into an infinite loop [1].

The recursive-descent parsing presents one of the general forms of top-down parsing. A recursive-descent program contains functions for every production rule in its grammar, and uses recursion to simulate nonterminal rewriting. However, this method may require backtracking to choose the correct production rule to expand, and thus they may require several scans of the input string.

To avoid backtracking, predictive parsing methods use a fixed number of look-ahead tokens to deterministically choose the production rule to apply. These are usually constructed for grammars of the $LL(k)$ class, most notably the $LL(1)$ class.

$LL(k)$ Parsing Method

The $LL(k)$ class parsers scan the input from left to right and construct the leftmost derivation. Intuitively, k stands for the number of look-ahead tokens.

The construction of $LL(1)$ parsers requires the construction of FIRST and FOLLOW sets. However, these sets can be utilised during the creation of bottom-up parsers as well; these will be discussed in Section 4.3. During top-down parsing, these sets are used to determine which production rule to apply. The FOLLOW set can be also used during error recovery to estimate the position to continue the analysis [1].

The $FIRST(\alpha)$ set, where α is any sequence of symbols, is defined as the set of the string that can be derived from α. If the sequence can be derived into ε, $\alpha \Longrightarrow^* \varepsilon$, it is included in the set. This set is used to determine which production rule should be applied.

The $FOLLOW(A)$ set, where A is a nonterminal, is defined as the set of terminals that can appear to the immediate right of A in any sentential form; if any of the symbols can be erased, the symbols following them are included as well. Moreover, if the nonterminal, A, may pose as the rightmost symbol of any sentential form, the $FOLLOW(A)$ set includes the end marker, \$. Usually, this set is used to determine whether a part of the sentential form has been erased and whether the production has been fully recognised.

A grammar, G, is $LL(1)$ if the following holds for any two productions, $A \longrightarrow \alpha$ and $A \longrightarrow \beta$:

- there exists no terminal, a, such that both α and β start with a,
- at most one of the sequences α and β can derive into ε,
- if $\beta \Longrightarrow^* \varepsilon$, α must most derive into a string starting with a terminal, $a \in FOLLOW(A)$; if $\alpha \Longrightarrow^* \varepsilon$, β must not derive into a string starting with a terminal, $b \in FOLLOW(A)$ [1].

The $LL(1)$ class of parsers covers a major part of programming languages. However, considering the grammar must not contain left recursion or ambiguity, it is not suitable for all grammars.

4.3 Bottom-Up Parsing Methods

Bottom-up parsing corresponds to the process of constructing a parse tree for the input string starting with leaves and gradually working up to the root of the tree. Usually this process is not done explicitly but as a part of syntax analysis.

This type of parsing methods is represented by the shift-reduce parsing approach, whose major representative is the class of $LR(k)$ parsers. Generally speaking, shift-reduce parsing is the process of reducing the input string into the start symbol; during each step of the process, the right-hand side of a production rule is matched and reduced. The core of this approach lies in the decision of when to reduce the substring and what production to apply [1].

During a left-to-right scan of the input string, the shift-reduce parser creates the rightmost derivation in reverse – a right parse. This is done by matching the right-hand side of a rule, called a handle, and reducing it into its left-hand side. A right parse can be obtained by handle pruning, that is by gradual reduction into the start symbol. For a sentence w and a right parse of the length n, have the following right-parse:

$$S = \gamma_0 \Longrightarrow_{rm} \gamma_1 \Longrightarrow_{rm} \cdots \Longrightarrow_{rm} \gamma_n = w. \tag{4.2}$$

To reconstruct the derivation in reverse, the handle β_i is found in γ_i and replaced by the complete production. If, after a finite number of iterations, the sentential form contains only the start symbol, the string is accepted [1]. This mechanism presents the core of shift-reduce parsing.

Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing that uses a stack to hold a temporary sequence of grammar symbols and a an input buffer holding the unprocessed part of the input string. During the scan of the string, the parser shifts tokens onto the stack until they can be reduced into a nonterminal. This process is repeated until either the string is accepted, or no operations are available. The primary operations of a shift-reduce parser include the following:

- *shift* – shift the next token on the stack,
- *reduce* – reduce a handle; the right end of the handle must be at the top of the stack,
- *accept* – accept the input string,
- *error* – reject the input string because of a syntax error [1].

Not all context-free grammars are suitable for use with shift-reduce parsers, as they may cause operation conflicts. There exist two main types of such conflicts: the shift/reduce conflicts, when the parser cannot determine whether to shift another token or to reduce a handle, and reduce/reduce conflicts, when two productions share the same handle [1].

$LR(k)$ Parsing Method

The main representative of shift-reduce parsing is the $LR(k)$ class. The parsers of this class are table-driven and use a number of look-ahead tokens; however, because of the complexity of these tables, usually no more than a single look-ahead token is used. Despite these limitations, the $LR(1)$ family of grammars presents a proper superset of the LL grammars presented in Section 4.2.

An LR parser makes shift-reduce decisions by keeping state information on its position in the parse. The states consist of a list of items – productions with a dot mark representing the position in the parse. For example, for the rule $A \longrightarrow BC$, there exist the following items:

$$A \longrightarrow \cdot BC,\ A \longrightarrow B \cdot C,\ A \longrightarrow BC \cdot . \tag{4.3}$$

These states are then used to construct an LR automaton. To define the transitions between the states, the functions CLOSURE and GOTO are defined. The CLOSURE function is used to determine the set of items the parser can move to when shifting a particular token, whereas the GOTO function serves as a new entry point into the automaton after a handle is reduced [1].

Based on the specifics of their work, there exist several types of LR parsers – $LR(1)$, SLR, $LALR$ and GLR. The $LR(1)$ parsers, while offering the greatest generative power, are not widely used because the creation of their states is parameterised by the look-ahead symbol, which subsequently results in a largely repetitive set of states. Some of the other methods, such as the $LALR$ method, solve this problem to a degree by using the state set as generated by the $LR(0)$ method, while sacrificing only a small portion of their generative power. Because of its efficiency, the $LALR$ method is used by the parser generator software, Bison [5]. Finally, the GLR method is capable of handling grammar ambiguity [1].

4.4 General Parsing Methods

The parsing methods presented in the previous sections put a number of conditions on the grammars they can be applied to, such as the absence of left recursion or ambiguity. There exists a group of parsing algorithms capable of surpassing these limitations, and therefore of parsing any grammar – the general parsing methods. The representatives of this group of algorithms include the Earley algorithm [8], which works in a top-down way, and the Cocke-Younger-Kasami algorithm, which works in a bottom-up way. In their base form, both of these methods present recognisers – methods only capable of recognising the correspondence of the sentence to the specified grammar. The latter will be discussed in detail.

Cocke-Younger-Kasami Algorithm

The Cocke-Younger-Kasami (CYK) algorithm is a representative of algorithms based on normal forms [15, 1]. The algorithm works in a bottom-up way with grammars in the Chomsky normal form, as per Definition 2.8.2.

Given a grammar, $G = (N, T, P, S)$ in the Chomsky normal form and an input string, $w = a_1 a_2 \dots a_n$, where $a_i \in T$ for some $1 \leq i \leq n$, the algorithm makes a decision of whether the input string, w, can be generated by the grammar, G. It works by constructing a matrix of sets $CYK[i, j]$ for some $1 \leq i \leq j \leq n$, where each set holds reduced nonterminals.

To populate these sets, the algorithm first reduces the individual terminals in the input string, a_i and adds them to their respective $CYK[i, i]$ sets. These nonterminals serve as the base for further reductions – whenever $B \in CYK[i, j]$ and $C \in CYK[j + 1, k]$, such that there exists a production $A \longrightarrow BC \in P$, the nonterminal A is added to $CYK[i, k]$. Considering that $B \Longrightarrow^* a_i \dots a_j$ and $C \Longrightarrow^* a_{j+1} \dots a_k$, the process results in gradual

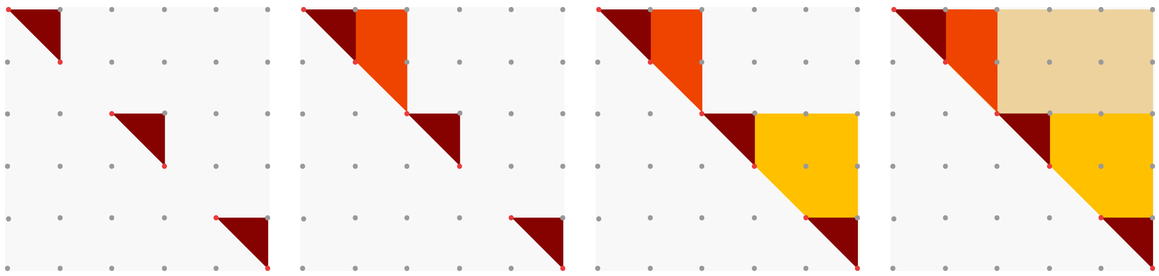


Figure 4.3: Application of the CYK algorithm. The final reduction is made by combining the a_1a_2 and $a_3 \dots a_6$ substrings, the input string was accepted.

reduction of longer substrings, in this case $A \Longrightarrow^* a_i \dots a_k$. Finally, once no more nonterminals can be reduced, the $CYK[1, n]$, which represents the root of the parse tree, is tested for presence of the start symbol, S. If it is found, a tree exists for the input string, and it is accepted. Otherwise, the string is rejected [13].

Algorithm 1 Cocke-Younger-Kasami Algorithm [13]

```
Input  a context-free grammar, $G = (N, T, P, S)$ in the Chomsky normal form
       $w = a_1a_2 \dots a_n$ with $a_i \in T$, $1 \leq i \leq n$, for some $n \geq 1$
Output ACCEPT if $w \in L(G)$
       REJECT if $w \notin L(G)$
 1: introduce sets $CYK[i, j] = \varnothing$ for $1 \leq i \leq j \leq n$
 2: function CYK($w$)
 3:     for $i = 1$ to $n$ do
 4:         if $A \longrightarrow a_i \in P$ then add $A$ to $CYK[i, i]$
 5:     repeat
 6:         if $B \in CYK[i, j], C \in CYK[j+1, k], A \longrightarrow BC \in P$ for some $A, B, C \in N$ then
 7:             add $A$ to $CYK[i, k]$
 8:     until no change
 9:     if $S \in CYK[1, n]$ then
10:         ACCEPT
11:     else REJECT
```

The work of this algorithm can be seen in Figure 4.3. The triangles represent the individual reductions. Not all reduced nonterminals are used in the final simulation of the parse tree – in this case the reduction of the substring $a_1a_2a_3$, represented by the bright orange triangle, was omitted.

Chapter 5

Extensions of the Cocke-Younger-Kasami Algorithm

This chapter introduces the proposed extensions of the Cocke-Younger-Kasami algorithm, which was first introduced in Section 4.4. The presented extensions work with both sequential and parallel grammars, and make use of the similarities between the normal forms of the respective grammar types and the Chomsky normal form the Cocke-Younger-Kasami algorithm was designed for.

The first part of this chapter presents the extensions for sequential grammar families – context-sensitive grammars, multigenerative systems and scattered context grammars. The later part introduces extensions for extended and interactive L-systems.

As opposed to the original algorithm, most of these grammar types incorporate the notion of context. Consequently, the order in which the CYK matrix is scanned may alter the results and is therefore strictly defined in all extensions. Moreover, every extension deals with complications caused by the nature of the examined grammar type. These complications, as well as their solutions, are discussed in their respective sections.

5.1 Extension for Context-Sensitive Grammars

The extension for context-sensitive grammars works with grammars in the Kuroda normal form, as per Definition 2.8.4. This normal form presents a direct extension of the Chomsky normal form, and adds the $AB \longrightarrow CD$ rule form.

Intuitively, a major part of the algorithm's behaviour remains unchanged. Application of context-sensitive productions respects the purpose of the individual matrix elements – these represent the nonterminals reduced using a substring of the input string whose bounds are symbolised by the position of the matrix element. Otherwise, the parsing proceeds analogously, as seen in Algorithm 2. However, this behaviour requires the addition of control mechanisms tracking the origin of the nonterminals, as described later in the chapter.

Informal Description

Given a context-sensitive grammars, $G = (N, T, P, S)$ in the Kuroda normal form, the algorithm makes a decision of whether an input string, $w = a_1a_2 \dots a_n$ where $a_i \in T$ for some $1 \leq i \leq n$, is generated by the language $L(G)$ in a bottom-up way. It uses a matrix of nonterminal sets, $CYK[i, j]$ for some $1 \leq i \leq j \leq n$. For each reduced nonterminal,

Algorithm 2 CYK Algorithm Adapted for Context-Sensitive Grammars

```
Input  a context-sensitive grammar, $G = (N, T, P, S)$, in the Kuroda normal form
       $w = a_1a_2 \dots a_n$ with $a_i \in T$, $1 \le i \le n$, for some $n \ge 1$
Output ACCEPT if $w \in L(G)$
       REJECT if $w \notin L(G)$
introduce sets $CYK[i, j] = \varnothing$ for some $1 \le i \le j \le n$      ▷ global working matrix
introduce sets $CYK'[i, j] = \varnothing$ for some $1 \le i \le j \le n$     ▷ global alternative matrix
introduce set $V = \varnothing$                                             ▷ global set of matrix versions
function CYK$_{\mathrm{CS}}(w)$
    for $i = 1$ to $n$ do
        if $A \longrightarrow a_i \in P$ for some $A \in N$ then
            add $A$ to $CYK[i, i]$                                          ▷ matrix initialisation
            introduce set $R_A = \varnothing$                               ▷ set of nonterminals reduced from $A$
    add $CYK$ to $V$
    repeat
        flag a member of the $V$ set as the working $CYK$ matrix
        for $level = 1$ to $n - 1$ do
            for $i = 1$ to $n - level$ do
                let $k = i + level$
                for $offset = 0$ to $level - 1$ do
                    let $j = i + offset$
                    Reduce$(i, j, k)$
                    $CYK'[i, j] = \varnothing$ for some $1 \le i \le j \le n$ ▷ reset the alternative matrix
        if $S \in CYK[1, n]$ then
            ACCEPT
        remove the working $CYK$ matrix from $V$
    until $V = \varnothing$
    REJECT
```

A, there exists an ancestor set, R_A, holding the nonterminals A has been reduced from; the contents of this set can be used to recursively track the nonterminal's ancestry tree.

The algorithm starts by scanning the input terminals, a_i for some $1 \le i \le n$, and adding a nonterminal, A to a set on the main matrix diagonal, $CYK[i, i]$ for every $A \longrightarrow a_i \in P$. Afterwards, the algorithm iterates through the matrix to try to reduce the rest of the sets. For every combination of nonterminals representing neighbouring substrings, $C \in CYK[i, j]$ and $D \in CYK[j + 1, k]$, it is checked whether the nonterminals can be reduced using a context-free rule. If so, the resulting nonterminal, A, is added to the $CYK[i, k]$ set for any $A \longrightarrow CD \in P$. For every such nonterminal, A, a reduced-from set, $R_A = \{C, D\} \cup R_C \cup R_D$, is constructed. Subsequently, the same test is executed for context-sensitive productions, and the new nonterminals are added to the respective parent sets – $CYK[i, j]$ and $CYK[j + 1, k]$. In this case, the pair symbol, B, is added to the R_A set for any $AB \longrightarrow CD \in P$ as well.

However, a problem arises if only one nonterminal of a context-sensitive pair is used in the final simulation of the parse tree; such state would never come into existence normally, and continuing the parsing process based on it may lead to false acceptance of the input string. To prevent this from happening, a version control system is introduced.

Each time a context-sensitive rule is reduced, its left-hand side nonterminals are saved to an alternate matrix, CYK', holding a copy of the current parse. In this matrix, all of the symbols' ancestors are deleted using the R_A sets, and as a result, both symbols of the pair have to be used to finish the parsing successfully; this behaviour is demonstrated in Algorithm 3. After all context-sensitive nonterminals have been reduced for a pair of elements, the CYK' matrix is added to the set holding alternate matrices, V.

Algorithm 3 Nonterminal Reduction Used in the $\mathbf{CYK_{CS}}$ function

Input matrix coordinates of the examined sets, i, j, k

```
1: procedure REDUCE($i$, $j$, $k$)
2:     if $C \in CYK[i,j]$, $D \in CYK[j+1,k]$, $rhs(p) = CD$ for some $C, D \in N$, $p \in P$ then
3:         if $A \longrightarrow CD \in P$ for some $A \in N$ then
4:             add $A$ to $CYK[i,k]$
5:             introduce set $R_A = \{C, D\} \cup R_C \cup R_D$
6:         if $AB \longrightarrow CD \in P$ for some $A, B \in N$ then
7:             if $\exists X \in R_C \cup R_D$ and $X \in CYK'[i',j']$ such that $X$ is context-sensitive then
8:                 continue
9:             if $CYK'[i',j'] = \varnothing$ for all $1 \leq i' \leq j' \leq n$ then
10:                let $CYK'[i',j'] = CYK[i',j']$
11:            for $A$ in $R_C \cup R_D$ do
12:                remove $A$ from the corresponding CYK' set
13:            add $A$ to $CYK'[i,j]$
14:            add $B$ to $CYK'[j+1,k]$
15:            introduce set $R_A = R_C \cup R_D \cup B$
16:            introduce set $R_B = R_C \cup R_D \cup A$
17:        if $CYK'[i',j'] \neq \varnothing$ for some $1 \leq i' \leq j' \leq n$ then
18:            unset $CYK'[i',j']$ as context-sensitive
19:            add $CYK'$ to $V$                          ▷ context-sensitive production used
```

As opposed to the original algorithm, parsing does not end once a matrix fails – it is repeated as long as the version set, V, contains any viable matrices. However, once the set is empty, the input string is rejected.

The behaviour of the extension is demonstrated in Figure 5.1. Pictures 1–3 show application of a context-sensitive rule on the substrings $CYK[2,3]$ and $CYK[4,6]$, which subsequently leads to the acceptance of the string. The last picture shows an example of false acceptance caused by the use of an incomplete context-sensitive pair; this behaviour is prevented by the introduced version control system.

Time and Space Complexity

The initialisation phase of the algorithm iterates through the width of the matrix once, taking $O(n)$ time. In the worst case scenario, the algorithm generates a matrix for every production rule, $p \in P$, and subsequently scans it. During each of these scans, the three nested cycles are executing, bringing the time complexity of the block to $O(|P| \cdot n^3)$. Thus, the total time complexity is equal to:

$$O(t) = n + |P| \cdot n^3, \tag{5.1}$$

$$O(t) = |P| \cdot n^3. \tag{5.2}$$

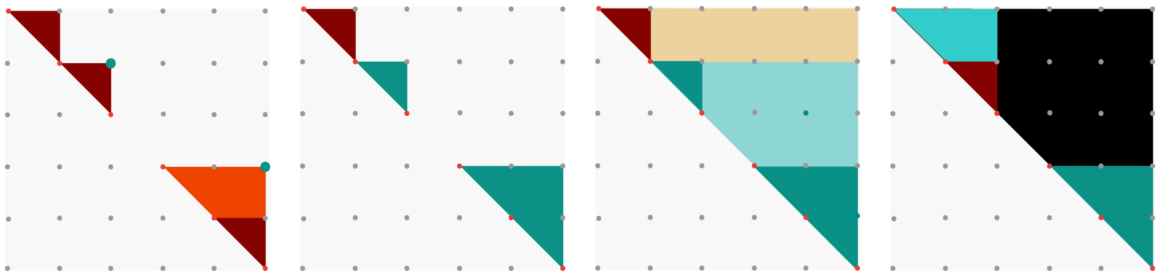


Figure 5.1: Extension of the CYK algorithm for context-sensitive grammars. The first three pictures show the progression of the algorithm with use of a context-sensitive production, whereas the last picture demonstrates the need for control mechanisms.

In the instance described in previous paragraph, the algorithm generates $|P|$ distinct matrices. In case the input grammar is written in a way that allows all nonterminals to be reduced in every matrix element, the space complexity of a single matrix reaches $O(|N| \cdot n^3)$. Overall, the space complexity of the algorithm reaches:

$$O(s) = |P| \cdot |N| \cdot n^2. \tag{5.3}$$

5.2 Extension for Multigenerative Grammar Systems

A multigenerative system consists of a sequence of context-free grammars that are rewritten in parallel. Because of this, the extension works with grammars in the Chomsky normal form. The extension works with canonical nonterminal-synchronised systems, as per Definition 2.6.1. The generative power of all mentioned grammar systems is equal [11], and thus, this type was chosen because of the convenience the canonical systems offer. Considering a multigenerative system consists of a sequence of context-free grammars, the parsing core remains unchanged. However, synchronisation between individual grammars requires implementation of auxiliary mechanisms, as discussed later in the section.

Informal Description

Given an n-generative nonterminal synchronised system, $\Gamma = (G_1, G_2, \ldots, G_n, Q)$ with G_m in the Chomsky normal form for some $1 \leq m \leq n$, the algorithm makes a decision of whether a sequence of input strings, $w_1, w_2, \ldots w_n$ are generated by $L(\Gamma)$, such that $w_m = a_{1m}a_{2m} \ldots a_{|w_m|m}$ and $L(\Gamma)$ is not in an editing mode. The algorithm uses a sequence of matrices, CYK_m and CYK'_m, each of which is composed of nonterminal sets, $CYK_m[i, j]$ and $CYK'[i, j]$ respectively, for some $1 \leq i \leq j \leq |w_m|$.

The CYK_m matrices are used to store permanent nonterminals – because of the semi-parallel nature of the CYK algorithm, reduction are not made in a strict order. However, canonical systems make use of comparison between leftmost nonterminals and control sequences. To accommodate this, a sequence of future matrices, CYK'_m is introduced. These matrices are used to store the reduced nonterminals, which are filtered and moved back to CYK_m after each iteration.

The algorithm starts its work by scanning the input strings, w_m for some $1 \leq m \leq n$, and adding a nonterminal, A, to its respective $CYK'_m[i, i]$ set for any $A \longrightarrow a_{mi} \in P_m$. The matrices are then filtered to contain only left-most nonterminals, and their correspondence to control sequences is checked; remaining nonterminals are then moved to their respective $CYK_m[i, i]$ elements. In this step, it is not needed to synchronise the number of reduced elements individually, as at most one nonterminal is reduced during each step.

Afterwards, the individual matrices are scanned and reduced in parallel. For every combination of neighbouring nonterminals, $B \in CYK_m[i, j]$ and $C \in CYK_m[j+1, k]$, a nonterminal A is added to $CYK_m[i, k]$ for any $A \longrightarrow BC \in P_m$.

Once no changes can be made in any of the matrices, the leftmost coordinate of the reduced substrings is found for every matrix; subsequently, nonterminals representing substrings located further to the right are erased from their respective CYK'_m matrices. Afterwards, the remaining nonterminals are compared to the nonterminal control sequences, $q \in Q$, and any nonterminal that is not a part of a completed control sequence, q, is erased. Finally, to maintain the synchronicity of the grammar system, the same number of nonterminals needs to be reduced in every matrix. The individual sets, $CYK'_m[i, j]$, represent reduction options for their respective substrings; as such, only one member of each set may be used in the final simulation of the parse trees. Because of this, the algorithm obtains a global minimum of non-empty elements of CYK'_m – as opposed to the number of nonterminals – and the elements exceeding this number are reset. Afterwards, the remaining nonterminals are moved to their respective CYK_m matrices, and the parsing process continues. The series of inspections can be seen in Algorithm 4.

The parsing ends once no matrix can be edited, and by extension, all CYK'_m matrices are empty. In case a matrix gets completed earlier than the rest of the system, no control sequences are matched, and all matrices are erased, resulting in a total halt. Finally, the sequence of input string is accepted if $S_m \in CYK_m[1, |w_m|]$, otherwise it is rejected. The graphical representation of this extension can be seen in Figure 5.2.

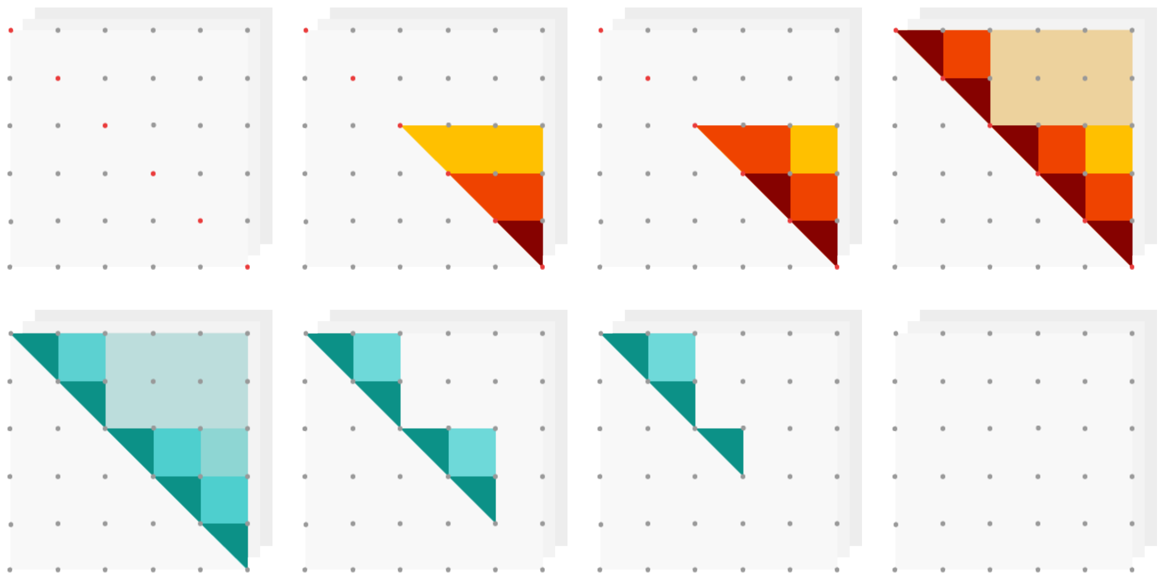


Figure 5.2: Extension of the CYK algorithm for multigenerative grammar systems. The extension uses an extra set of matrices to keep track of new nonterminals; the nonterminals are gradually moved to the main matrix, where the sequence of strings is accepted.

Algorithm 4 CYK Algorithm Adapted for n-Generative Nonterminal-Synchronised Grammar Systems

```
Input an $n$-generative nonterminal-synchronised system, $\Gamma = (G_1, G_2, \dots, G_n, Q)$, with
        $G_m = (N_m, T_m, P_m, S_m)$ for some $1 \leq m \leq n$ in the Chomsky normal form,
        $w_m = a_{m1}a_{m2}\dots a_{mx}$, with $a_{mi} \in T_i$, $1 \leq i_m \leq x_m$, for some $1 \leq m \leq n$, $x \geq 1$
Output ACCEPT if $(w_1, w_2, \dots w_m) \in L(\Gamma)$
       REJECT if $(w_1, w_2, \dots w_m) \notin L(\Gamma)$
 1: introduce sets $CYK_m[i, j] = \varnothing$ for some $1 \leq i_m \leq j_m \leq x_m$
 2: introduce sets $CYK'_m[i, j] = \varnothing$ for some $1 \leq i_m \leq j_m \leq x_m$
 3: function $\mathrm{CYK_{NGS}}(w_1, w_2, \dots, w_m)$
 4:     repeat
 5:         for $m = 1$ to $n$ do in parallel
 6:             for $i = 1$ to $x_m$ do
 7:                 if $A \longrightarrow a_{mi} \in P_m$ for some $A \in N_m$ then
 8:                     add $A$ to $CYK_m[i, i]$                    ▷ matrix initialisation
 9:             if $\exists i'' : i'' < i' \wedge CYK'[i'', i''] \neq \varnothing$ for some $1 \leq i'' \leq x_m$ then
10:                 let $CYK'_m[i', i'] = \varnothing$
11:             if $\nexists q = (A_1, A_2, \dots A_m) \in Q$ such that $A_i \in CYK'_m[i', i']$ then
12:                 remove $A_i$ from $CYK'_m[i', j']$
13:     until no change
14:     repeat
15:         for $m = 1$ to $n$ do in parallel
16:             for $level = 1$ to $x_m - 1$ do
17:                 for $i = 1$ to $x_m - level$ do
18:                     for $j = i$ to $i + level$ do
19:                         let $k = i + level$
20:                         if $B \in CYK_m[i, j]$, $C \in CYK_m[j + 1, k]$, $A \longrightarrow BC \in P_m$ then
21:                             add $A$ to $CYK'_m[i, k]$
22:         for $m = 1$ to $n$ do in parallel
23:             if $\exists i'' : i'' < i' \wedge CYK'_m[i'', j''] \neq \varnothing$ for some $1 \leq i'' \leq j'' \leq x_m$ then
24:                 let $CYK'_m[i', j'] = \varnothing$ for all $i' \leq j' \leq x_m$
25:             if $\nexists q = (A_1, A_2, \dots A_m) \in Q$ such that $A_i \in CYK'_m[i', j']$ for some $1 \leq i'$
                  $\leq j' \leq x_m$ then remove $A_i$ from $CYK'_m[i', j']$
26:             let $min = min_{global}(\{x : CYK_m[i'', j''] \neq \varnothing$ for some $1 \leq i'' \leq j'' \leq x_m\}_{\#})$
27:             let $min_{local} = \{x : CYK_m[i'', j''] \neq \varnothing$ for some $1 \leq i'' \leq j'' \leq x_m\}_{\#}$
28:             if $min_{local} > min$ then
29:                 for $iter = 1$ to $min_{local} - min$ do
30:                     let $CYK'[i', j'] = \varnothing$ for some $1 \leq i' \leq j' : \nexists j'' > j' : CYK'[i', j''] \neq \varnothing$
31:             let $CYK_m[i, j] = CYK'_m[i, j]$ for some $1 \leq i \leq j \leq x_m$
32:             let $CYK'_m[i, j] = \varnothing$ for some $1 \leq i \leq j \leq x_m$
33:     until no change
34:     if $S_m \in CYK_m[1, x_m]$ for any $1 \leq m \leq n$ then
35:         ACCEPT
36:     REJECT
```

Time and Space Complexity

An n-generative system consists of n distinct context-free grammars used to rewrite n input strings, x_i for some $1 \leq i \leq n$. For each of these grammars, a matrix is initiated, and the three nested cycles are executed. Subsequently, each of the matrices is scanned again to erase non-leftmost symbols, nonterminals not conforming to a control sequence, and excess elements, bringing the time complexity of the block to $O(3|x_i|)$. Overall, the time complexity of the algorithm equals:

$$\begin{aligned} O(t) &= n \cdot (i + i^3 + 3i) && (5.4)\\ &= n \cdot (4i + i^3), && (5.5)\\ O(t) &= n \cdot i^3. && (5.6) \end{aligned}$$

In the worst case scenario, the algorithm reduces every possible nonterminal during each iteration. For an n-generative system, $\Gamma = (G_1, G_2, \dots G_n)$ such that $G_i = (N_i, T_i, P_i, S_i)$ for some $1 \leq i \leq n$, this means the creation of approximately $i^2 \cdot N_i$ new symbols. The total space complexity of the algorithm therefore reaches:

$$O(s) = n \cdot i^2 \cdot |N_i|. \tag{5.7}$$

5.3 Extension for Scattered Context Grammars

The extension for scattered context grammars works with grammars in the 2-limited normal form, as per Definition 2.8.5. Compared to the Chomsky normal form, the 2-limited form allows application of up to two context-free productions in the string. Moreover, the right-hand sides of these productions do not have to be composed of a single symbol type.

The main challenge of adapting the Cocke-Younger-Kasami algorithm to scattered context grammars lies in the fact that the members of the scattered pair do not need to be located next to each other. This introduces two problems – first, the respective right-hand side strings are reduced individually, and second, most reductions are context-sensitive, making the previous strategies not viable. The core of the extension can be seen in Algorithm 5.

Informal Description

Given a context-sensitive grammar, $G = (V, \Sigma, S, P)$ in the 2-limited normal form, the algorithm makes a decision of whether an input string, $w = a_1 a_2 \dots a_n$ where $a_i \in \Sigma$ for some $1 \leq i \leq n$, belongs to the language generated by G, $L(G)$. The algorithm uses a matrix of nonterminal sets, $CYK[i, j]$ for some $1 \leq i \leq j \leq n$, as well as sets of partial productions, P_L and P_R. These sets are populated before the parsing begins; for every $(A, B) \longrightarrow (w_1, w_2) \in P$, it adds the production $A \longrightarrow w_1$ to P_L and $B \longrightarrow w_2$ to P_R. Moreover, for every reduced nonterminal, A, there exists an ancestor set, R_A, holding the nonterminals A was reduced from. If the nonterminal, A, was reduced using a scattered production, a pair nonterminal, S_A is defined as well.

At the beginning of the parsing process, the input tokens, a_i, for some $1 \leq i < j \leq n$, are scanned. For any $A \longrightarrow a_i \in P_L$, the right partial scan is triggered. Whenever an a_j is found such that $B \longrightarrow a_j \in P_R$ and $(A, B) \longrightarrow (a_i, a_j) \in P$, the nonterminals A such that $S_A = B$ and B such that $S_B = A$ are added to sets $CYK[i, i]$ and $CYK[j, j]$ respectively.

Afterwards, the matrix is scanned in a linear way. For every symbol, $B \in CYK[i,j]$ such that $A \longrightarrow B \in P_L$ or a pair of neighbouring symbols $B \in CYK[i,j]$ and $C \in CYK[j+1,k]$ such that $A \longrightarrow B \in P$ or $A \longrightarrow BC \in P_L$, an analogous search for right partial derivation is triggered, as seen in Algorithm 6. For every matched partial rule, $B \longrightarrow w_B \in P_R$ such that $(A,B) \longrightarrow (B,w) \in P$ or $(A,B) \longrightarrow (BC,w) \in P$, these partial context free productions are applied at their respective positions in the matrix, e.g. A is added to $CYK[i,j]$ or $CYK[i,k]$, depending on the length of the handle; the coordinates of B are

Algorithm 5 CYK Algorithm Adapted for Scattered Context Grammars

```
Input a 2-limited propagating scattered context grammar, $G = (V, \Sigma, S, P)$
        $w = a_1 a_2 \dots a_n$ with $a_i \in T$, $1 \le i \le n$, for some $n \ge 1$
Output  ACCEPT if $w \in L(G)$
        REJECT if $w \notin L(G)$
 1: introduce sets $P_L = \varnothing$, $P_R = \varnothing$        ▷ global set of partial productions
 2: introduce sets $CYK[i,j] = \varnothing$ for some $1 \le i \le j \le n$        ▷ global working matrix
 3: function CYK$_{SC}(w)$
 4:     if $(A,B) \longrightarrow (x_A, x_B)$ for some $A, B \in V \setminus \Sigma$, and $x_A, x_B \in V^*$ then
 5:         add $A \longrightarrow x_A$ to $P_L$
 6:         add $B \longrightarrow x_B$ to $P_R$
 7:     for $i = 1$ to $n-1$ do        ▷ matrix initialization
 8:         if $A \longrightarrow x_i \in P_L$ for some $A \in V \setminus \Sigma$ then
 9:             for $j = i+1$ to $n$ do
10:                 if $B \longrightarrow x_j \in P_R$ and $(A,B) \longrightarrow (x_i, x_j) \in P$ for some $B \in V \setminus \Sigma$ then
11:                     add $A$ to $CYK[i,i]$
12:                     add $B$ to $CYK[j,j]$
13:                     introduce sets $R_A = \varnothing$, $R_B = \varnothing$
14:                     let $S_A = B$, $S_B = A$
15:     repeat
16:         for $level = 1$ to $n-1$ do
17:             for $i = 1$ to $n - level$ do
18:                 let $k = i + level$
19:                 for $offset = 0$ to $level - 1$ do
20:                     let $j = i + offset$
21:                     if $B \in CYK[i,j], A \longrightarrow B \in P_L$ then
22:                         GetScatteredPair($i$, $j$, $j$)        ▷ call for the nonterminal
23:                     if $j < n - level$ then        ▷ enough remaining tokens
24:                         if $B \in CYK[i,j]$, $C \in CYK[j+1,k]$, $A \longrightarrow BC \in P_L$ then
25:                             GetScatteredPair($i$, $j$, $k$)
26:                         if $B \in CYK[1,j]$, $C \in CYK[j+1,n]$, $S \longrightarrow BC \in P$ then
27:                             add $S$ to $CYK[1,n]$
28:                             introduce set $R_S = R_B \cup R_C \cup \{B, C\}$
29:     until no change
30:     CleanAncestry( )
31:     if $S \in CYK[1,n]$ then
32:         ACCEPT
33:     REJECT
```

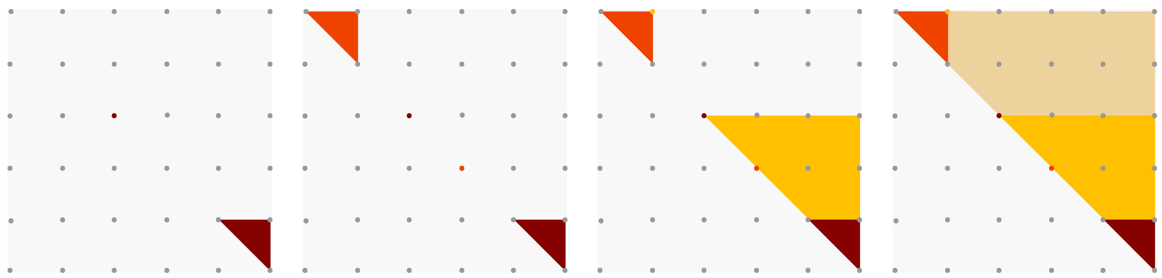


Figure 5.3: Extension of the CYK algorithm for scattered context grammars. The nonterminal pairs reduced at the same time are colour-matched. Subsequently, they are used to reduce the start symbol located in the upper right-hand corner.

decided in the same way. For each of these nonterminals ancestor sets are created to allow ancestry backtracking during the last phase of the parsing. The algorithm does no checks to assure the matrix consistency at the time of the reductions – most of the productions used are context-sensitive, and therefore, a control mechanism executed during reduction would increase the time complexity exponentially.

Once no more changes can be made to the matrix, the ancestry of all start symbols, $S \in CYK[1, n]$ is validated, as seen in Algorithm 8. During this scan, the R_S sets are recursively searched to reconstruct the parse tree. If a paired symbol, S_A, is not found for some $A \in R_S$, or there exists a substring of the input string, w, that was not reduced during the process, the symbol is erased. For this purpose, symbols with different predecessor sets, R_S, are considered distinct. Finally, if the $CYK[1, n]$ set still contains an instance of the start symbol, S, the input string, w, is accepted. Otherwise, it is rejected.

The work of the algorithm, including the application of a scattered-context rule, can be seen in Figure 5.3. In this example, the substrings a_3 and a_5a_6 are reduced first, followed by the reduction of substrings a_1a_2 and a_4. Finally, a context-free rule is applied to complete the reduction.

Algorithm 6 Coordinates of the Scattered Set Lookup

Input matrix coordinates of the left reduction sets, i, j, k

```
1:  procedure GetScatteredPair($i$, $j$, $k$)
2:      for $level = 1$ to $n - 1$ do
3:          for $i' = i + 1$ to $n - level$ do
4:              let $k' = i' + level$
5:              for $offset = 0$ to $level - 1$ do
6:                  let $j' = i' + offset$
7:                  if $B \in CYK[i', j'], A \longrightarrow B \in P_R$ then
8:                      Reduce($split$, $i$, $j$, $k$, $i'$, $j'$, $j'$)        ▷ call for the nonterminal
9:                  if $j \leq n - level$ then                  ▷ not the last set on the diagonal
10:                     if $B \in CYK[i', j'], C \in CYK[j' + 1, k'], A \longrightarrow BC \in P_R$ then
11:                         Reduce($split$, $i$, $j$, $k$, $i'$, $j'$, $k'$)
```

Algorithm 7 Nonterminal Reduction Used in the $\mathbf{CYK_{SC}}$ procedure

```
Input coordinates of the left reduction sets, $i, j, k$
      coordinates of the right reduction sets, $i', j', k'$
1: procedure REDUCE(, $i$, $j$, $k$, $i'$, $j'$, $k'$)
2:     introduce sets $PR_L = \varnothing$, $PR_R = \varnothing$          ▷ applicable partial reductions
3:     if $B \in CYK[i, j]$, $(j = k,\ p = A \longrightarrow B)$ or $(C \in CYK[j+1, k],\ p = A \longrightarrow BC)$
        for some $p \in P_L$ then add $p$ to $PR_L$
4:     if $B \in CYK[i', j']$, $(j' = k',\ p = A \longrightarrow B)$ or $(C \in CYK[j'+1, k'],\ p = A \longrightarrow BC)$
        for some $p \in P_R$ then add $p$ to $PR_R$
5:     if $A = lhs(p_L)$, $B = lhs(p_R)$, $lhs(p) = (A, B)$ for some $p_L \in PR_L$, $p_R \in PR_R$,
        $p \in P$ then
6:         add $A$ to $CYK[i, k]$
7:         add $B$ to $CYK[i', k']$
8:         introduce set $R_A = rhs(p_L) \cup rhs(p_R)$
9:         introduce set $R_B = rhs(p_L) \cup rhs(p_R)$
10:        let $S_A = B$, $S_B = A$
```

Algorithm 8 Ancestry Cleanup Used in the $\mathbf{CYK_{SC}}$ procedure

```
1: procedure CLEANANCESTRY( )
2:     for $A$ in $CYK[1, n]$ do
3:         if $A \neq S$ then continue
4:         for $A \in R_S$ do
5:             if $\nexists B \in R_S$ such that $S_A = B$ then          ▷ not all pairs covered
6:                 remove $S$ from $CYK[1, n]$
7:         if $\nexists A \in S_S$ such that $A \longrightarrow a_i$ for any $1 \leq i \leq n$ then  ▷ not all tokens covered
8:             remove $S$ from $CYK[1, n]$
```

Time and Space Complexity

The algorithm uses three nested cycles to search for left partial reductions. After a reduction has been detected, the same sequence of cycles is run to find the right partial reduction, bringing the time complexity of the step to $O(n^6)$. Once the reduction of the matrix is complete, the ancestry tree needs to be scanned. In case it contains duplicities and a prevalence of unary partial derivations, the scan may cover the entire matrix, making the time complexity $O(n^3)$. Overall, the time complexity of the algorithm reaches:

$$O(t) = |P| \cdot (n^6 + n^3), \tag{5.8}$$

$$O(t) = |P| \cdot n^6. \tag{5.9}$$

In the worst case scenario, every element of the matrix is populated by all possible nonterminals, as well as the ancestry tree. In this case, the space complexity of each part is equal to $O(n^2 \cdot |N|)$, bringing the total space complexity to:

$$O(s) = 2n^2 \cdot |N|, \tag{5.10}$$

$$O(s) = n^2 \cdot |N|. \tag{5.11}$$

5.4 Extension for EP0L Systems

Systems of the EP0L family use a normal form similar to the Chomsky normal form, as per Definition 3.6.1. Therefore, the only change that needs to be done to the base algorithm is the addition of the unary rule form, $A \longrightarrow B$. Other than this property, it is necessary to simulate the base properties of the L-systems – parallelism and synchronised rewriting.

As mentioned in Chapter 3, parallelism cannot be fully simulated by context-free grammars alone, which is solved by the addition of a future matrix, $FCYK$, holding the currently generated sentential form. Another problem arises from the fact that the Cocke-Younger-Kasami algorithm generates alternate nonterminals for all matched handles, regardless of their order or overlapping, which proved problematic when dealing with multiple reductions per sentential form.

Informal Description

Given an EP0L system, $G = (V, \Sigma, P, S)$ in normal form per Definition 3.6.1, the algorithm makes a decision of whether an input string, $w = a_1a_2 \dots a_n$ where $a_i \in \Sigma$ for some $1 \le i \le n$, belong to the language generated by G, $L(G)$. For its work, the algorithm uses a pair matrix of nonterminal matrices, $CYK[i, j]$ and $FCYK[i, j]$ for some $1 \le i \le j \le n$, as well as a set of past matrix states, V.

The algorithm starts its work by scanning the input string, and for every a_i for some $1 \le i \le n$ such that $A \longrightarrow a_i \in P$, adding the nonterminal, A, to $CYK[i, i]$. From this point on, the matrix is only used for reading until the iteration.

Subsequently, the rest of the matrix is scanned. For every $B \in CYK[i, j]$ such that $A \longrightarrow B \in P$ for some $1 \le i \le j \le n$, the nonterminal B is added to $FCYK[i, j]$; for every $B \in CYK[i, j]$ and $C \in CYK[j+1, k]$ such that $A \longrightarrow BC \in P$, the nonterminal A is added to $FCYK[i, k]$. Whenever all substring combinations have been inspected for a combination of substring bounds i and k, the value of the left bound, i, is adjusted to be located to the immediate right value of the shortest reduced substring during the step, as seen in Algorithm 9. Considering the $FCYK$ matrix contains an entire sentential form, this mechanism is required to avoid reduction overlapping, and by an extension, false acceptance of the input string, w.

At the end of each iteration, the future matrix, $FCYK$, is tested for presence of the start symbol, S. If it is found, the string is accepted. Otherwise, the contents of the CYK matrix are replaced by the $FCYK$ matrix, which is in turn erased. Because of this mechanism, if a part of the sentential form is not rewritten, a blank space in the matrix is created, which ultimately leads to rejection of the input string. Before the swap is executed, however, the algorithm checks whether the previous version set, V, contains a matrix identical to $FCYK$. If so, the grammar contains a loop, and the string will never be reduced successfully. In this case, as well as in the case the $FCYK$ matrix is empty, the input string, w, is rejected. Otherwise, the $FCYK$ matrix is added to V, and the parsing continues until one of the previous conditions is satisfied.

Progression of the algorithm can be seen in Figure 5.4. The CYK matrix uses shades of orange, the $FCYK$ matrix is blue.

Time and Space Complexity

The potentially most expensive part of this algorithm is the matrix switching – during each iteration, one of the matrices is reset, whereas the other scanned again. In case only one

Algorithm 9 CYK Algorithm Adapted for EP0L systems

```
Input an EP0L system, $G = (V, \Sigma, P, S)$, in the normal form per Definition 3.6.1
      $w = a_1 a_2 \ldots a_n$ with $a_i \in \Sigma$, $1 \leq i \leq n$, for some $n \geq 1$
Output ACCEPT if $w \in L(G)$
       REJECT if $w \notin L(G)$
 1: introduce sets $CYK[i, j] = \varnothing$ for some $1 \leq i \leq j \leq n$
 2: introduce sets $FCYK[i, j] = \varnothing$ for some $1 \leq i \leq j \leq n$
 3: introduce set $V = \varnothing$                                         ▷ past matrices
 4: function CYK$_{\text{EP0L}}(w)$
 5:     for $i = 1$ to $n$ do
 6:         if $A \longrightarrow a_i \in P$ for some $A \in V \setminus \Sigma$ then
 7:             add $A$ to $CYK[i, i]$                                 ▷ matrix initialisation
 8:     repeat
 9:         for $level = 0$ to $n - 1$ do                              ▷ unary productions
10:             for $i = 1$ to $n - level$ do
11:                 for $j = i$ to $n - level$ do
12:                     if $B \in CYK[i, j]$, $A \longrightarrow B \in P$ for some $A, B \in V \setminus \Sigma$ then
13:                         add $A$ to $FCYK[i, j]$
14:         for $level = 1$ to $n - 1$ do                    ▷ distance from the main diagonal
15:             let $i_0 = 1$
16:             for $i = i_0$ to $n - level$ do           ▷ starting position of the first substring
17:                 let $i_0 = \infty$                                     ▷ reset the value
18:                 for $j = i$ to $i + level$ do
19:                     let $k = i + level$
20:                     let $unary = (FCYK[i, j] \neq \varnothing)$          ▷ substring offset indicator
21:                     if $B \in CYK[i, j]$, $C \in CYK[j+1, k]$, $A \longrightarrow BC \in P$ for some $A, B, C$
                        $\in V \setminus \Sigma$ then
22:                         add $A$ to $FCYK[i, k]$
23:                     if $unary$ then $i_0 = min(i_0, j + 1)$      ▷ skip only the first substring
24:                     else $i_0 = min(i_0, k + 1)$             ▷ skip the whole reduced substring
25:         if $S \in FCYK[1, n]$ then                             ▷ axiom reduction test
26:             ACCEPT
27:         if $FCYK \in V$ then
28:             REJECT
29:         add $FCYK$ to $V$
30:         let $CYK[i, j] = FCYK[i, j]$ for some $1 \leq i \leq j \leq n$          ▷ matrix swap
31:         let $FCYK[i, j] = \varnothing$ for some $1 \leq i \leq j \leq n$
32:     until $CYK[i, j] = \varnothing$ for all $1 \leq i \leq j \leq n$
33:     REJECT
```

production rule is applied during a scan, the matrices can be switched $|P|$ times in total. Overall, this brings the algorithm time complexity to:

$$O(t) = |P| \cdot n^3. \tag{5.12}$$

At any given time, the algorithm uses two active matrices – one for reading, and one for writing; each element in these matrices may contain a total of $|N|$ nonterminals. More-

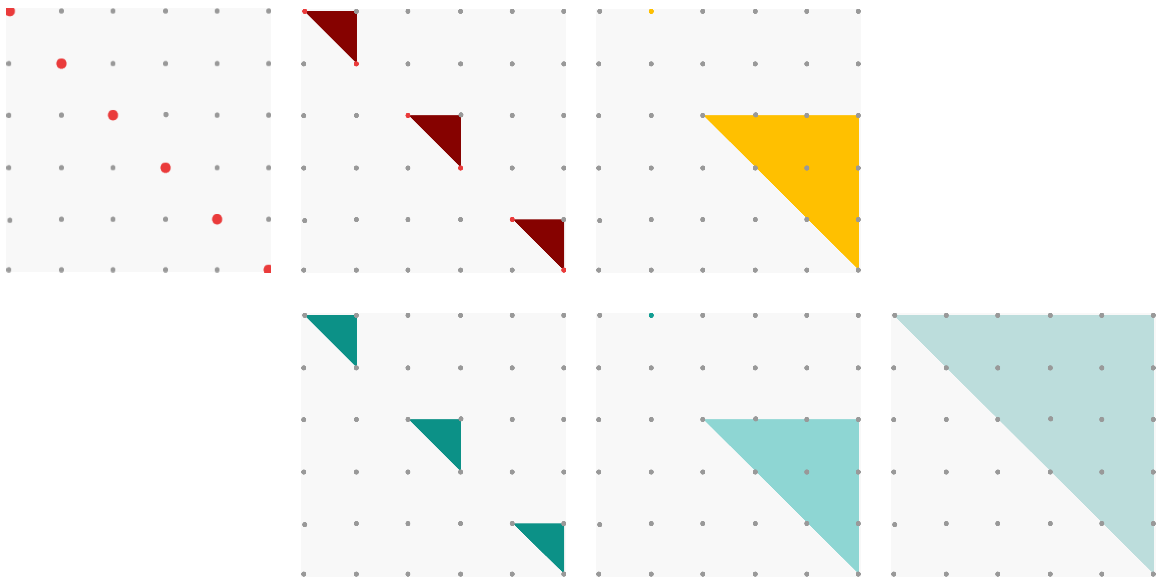


Figure 5.4: Extension of the CYK algorithm for EP0L systems. Orange hues represent the CYK matrix, whereas blue ones represent the $FCYK$ matrix. The input string is accepted once the start symbol is found in $FCYK[1, n]$.

over, the matrix history set, V, may hold a total of $|P|$ fully used matrices. In total, this brings the space complexity of the algorithm to:

$$\begin{aligned} O(s) &= 2n \cdot |N| + n^2 \cdot |P| \cdot |N| && (5.13)\\ &= n \cdot |N| + n^2 \cdot |N| \cdot |P| && (5.14)\\ &= (n \cdot |N|) \cdot (1 + n \cdot |P|), && (5.15)\\ O(s) &= n^2 \cdot |N| \cdot |P|. && (5.16) \end{aligned}$$

5.5 Extension for EPIL Systems

EPIL systems use an extended version of the normal form presented in the previous section extended by environment conditions, as per Definition 3.6.2. The extension is based on the same principle as the one for EPOL systems – it saves the newly reduced nonterminals to the future matrix, $FCYK$, and moves them back to the main matrix, CYK, at the end of the iteration. However, EPIL systems require a certain degree of context-sensitivity unseen in EP0L systems. This is achieved by using ancestor sets, R_A for some nonterminal A, as previously introduced in section 5.1.

Informal Description

Given an EPIL system, $G = (V, \Sigma, P, g, S)$ in a normal form per Definition 3.6.2, the algorithm makes a decision of whether an input string, $w = a_1a_2 \ldots a_n$ where $a_i \in \Sigma$ for some $1 \leq i \leq n$ is generated by $L(G)$. For its work, it uses a pair of matrices, $CYK[i, j]$ and $FCYK[i, j]$ for some $1 \leq i \leq j \leq n$, used to store present and future variations of the generated sentential forms respectively, as well as a set of past matrix versions, V.

At the start of the parsing process, the CYK matrix is initialised by scanning the input string, and adding a nonterminal, A, to the respective $CYK[i, i]$ set for any $A \longrightarrow a_i \in$

P. As there exists no previous sentential form at this point, the reductions are saved to the CYK matrix immediately. Once the initialisation phase is complete, the CYK matrix is used only to detect production handles until the end of each iteration.

Afterwards, the CYK matrix is scanned, and for every nonterminal, $B \in CYK[i, j]$ for some $1 \leq i \leq j \leq n$ such that $w_1 < A > w_3 \longrightarrow B \in P$, the nonterminal A is added to $FCYK[i, j]$, and set $R_A = \{B\} \cup R_B$ is introduced. Similarly, for every pair of neighbouring nonterminals, $B \in CYK[i, j]$ and $C \in CYK[j + 1, k]$ for some $1 \leq i \leq j \leq k \leq n$ such that $w_1 < A > w_3 \longrightarrow BC \in P$, the nonterminal A is added to $FCYK[i, k]$, and set $R_A = \{B, C\} \cup R_B \cup R_C$ is introduced. Once all reductions have been made for a pair of substring bounds, i and k, the bounds of the scanned substring are moved to the immediate right of the shortest reduced substring. No environment checks are executed at this time, as the environment needed to reduce individual nonterminals may not exist yet because of the linear character of the matrix scan.

Algorithm 10 Matrix Filtering for EPIL systems

```
1:  procedure MatrixCleanup( )
2:      if $\exists A \in FCYK[i, k]$ for some $A \in V \setminus \Sigma,\ 1 \leq i \leq k \leq n$ such that $R_A \neq \varnothing$ then
3:          for $R$ in $R_A$ do
4:              let $w_1 = w_1$ such that $R$ was reduced using $w_1 < R > w_3 \longrightarrow w_4 \in P$
5:              let $w_3 = w_3$ such that $R$ was reduced using $w_1 < R > w_3 \longrightarrow w_4 \in P$
6:              if !IsEnvFound($w_1, i, true, true$) or !IsEnvFound($w_3, k, false, true$) then
7:                  remove $A$ from $FCYK[i, k]$
8:      if $\exists A \in FCYK[i, k]$ for some $A \in V \setminus \Sigma, 1 \leq i \leq k \leq n$ such that $R_A \neq \varnothing$ then
9:          let $w_1 = w_1$ such that $A$ was reduced using $w_1 < A > w_3 \longrightarrow w_4 \in P$
10:         let $w_3 = w_3$ such that $A$ was reduced using $w_1 < A > w_3 \longrightarrow w_4 \in P$
11:         if !IsEnvFound($w_1, i, true, false$) or !IsEnvFound($w_3, k, false, false$) then
12:             remove $A$ from $FCYK[i, k]$
```

Once no more changes can be made to the $FCYK$ matrix, the algorithm validates whether all nonterminals respect the constraints presented by the production rule that was used to reduce them. The validation consists of two waves, both of which can be seen in Algorithm 10. First, the algorithm checks that for every reduced nonterminal, the matrix contains children of it's ancestors' environment. This assures that environment remains consistent thorough the iterations. Second, the nonterminals' own environment is checked to assure the environment constraints, w_1 and w_3 for some $w_1 < A > w_3 \longrightarrow w_4 \in P$, are satisfied. The matched environment symbols must be located next to each other, as seen in Algorithm 11.

Finally, the $FCYK$ matrix is tested for the presence of the start symbol, S. If the nonterminal is found, the string is accepted. Otherwise, it is tested whether the $FCYK$ matrix is a member of V. If so, the grammar contains a loop, and therefore, the string is rejected. If not, it is added to the set, and its contents are used to replace the CYK matrix. In this way, the algorithm assures that whenever a part of the processed sentential form is not reduced, a blank space is created in the matrix; subsequently, this leads to rejections of the final string. Once the CYK matrix is updated, the parsing continues analogously until a decision is made.

The progression of the algorithm can be seen in Figure 5.5. It uses the same base grammar as Figure 5.4, this time, however, the environment checks are employed, and ultimately lead to the rejection of the input string.

Algorithm 11 Environment Test Used for EPIL Systems

Input w – scanned environment string
pos_0 – starting position to scan
$direction$, $ancestry$ – scan modificators

```
1:  function isEnvFound($w$, $pos_0$, $direction$, $ancestry$)
2:      if $|w| = 0$ then return true
3:      let $head = a_1$ such that $w = a_1 a_2 \dots a_m$
4:      let $body = a_2 \dots a_m$ such that $w = a_1 a_2 \dots a_m$
5:      if $head = g$ then return true                ▷ no more environment requirements
6:      if $direction$ then
7:          for $j = pos_0 - 1$ downto $0$ do
8:              if $j = 0$ then return false
9:              for $i = j$ downto $1$ do
10:                 if $ancestry$ and $\exists A \in FCYK[i,j]$ such that $head \in R_A$ and isEnvFound($body$, $i$, $direction$) then return true
11:                 else if $!ancestry$ and $head \in FCYK[i,j]$ and isEnvFound($body$, $i$, $direction$) then return true
12:     else
13:         for $i = pos_0 + 1$ to $n + 1$ do
14:             if $i > n$ then return false
15:             for $j = 1$ to $n$ do
16:                 if $ancestry$ and $\exists A \in FCYK[i,j]$ such that $head \in R_A$ and isEnvFound($body$, $i$, $direction$) then return true
17:                 else if $!ancestry$ and $head \in FCYK[i,j]$ and isEnvFound($body$, $i$, $direction$) then return true
```

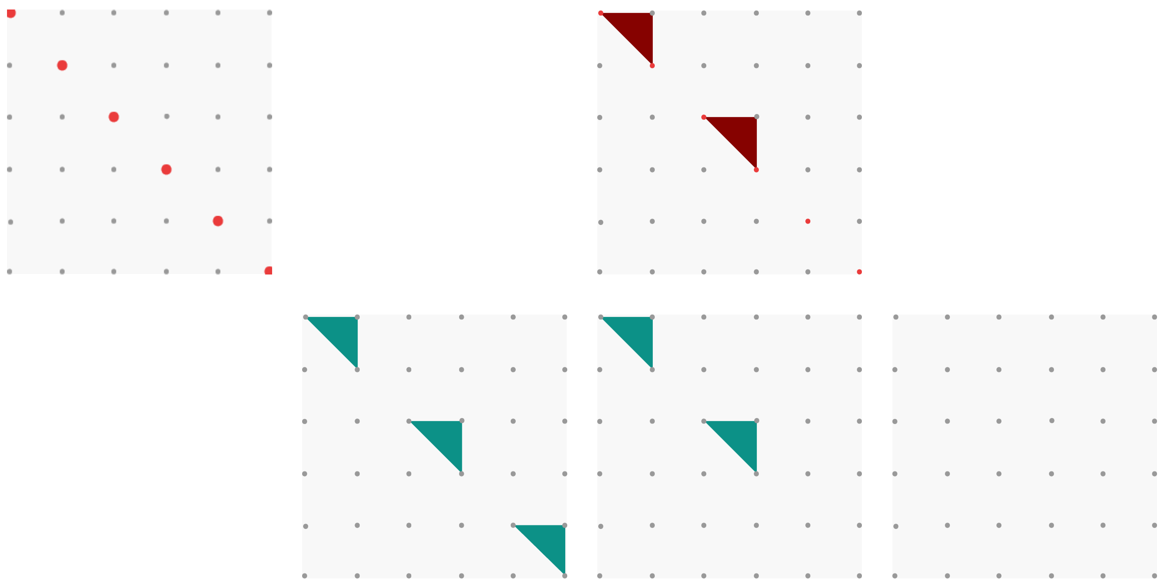


Figure 5.5: Extension of the CYK algorithm for EPIL systems. Orange hues represent the CYK matrix, whereas blue ones represent the $FCYK$ matrix. In the example, an environment check fails, causing nonterminals deletion and subsequent string rejection.

Time and Space Complexity

The core of this algorithm proceeds identically to the algorithm introduced in Section 5.4. However, after every iteration, the symbols' environment and their ancestors' environment is checked. In worst scenario, that means checking $2|N|$ symbols $|P|$ times. Overall, the time complexity of the algorithm reaches:

$$\begin{aligned} O(t) &= |P| \cdot n^3 \cdot |P| \cdot 2|N| && (5.17)\\ &= |P|^2 \cdot |N| \cdot n^3, && (5.18)\\ O(t) &= |N| \cdot |P|^2 \cdot n^3. && (5.19) \end{aligned}$$

In terms of spatial requirements, the algorithm needs to save a total of $2n$ symbols, as the environment is never bigger than the matrix itself. As a result, the time complexity of the algorithm is equal to:

$$\begin{aligned} O(s) &= n^2 \cdot |N| \cdot |P| \cdot 2n && (5.20)\\ &= 2n^3 \cdot |N| \cdot |P|, && (5.21)\\ O(s) &= n^3 \cdot |N| \cdot |P|. && (5.22) \end{aligned}$$

Chapter 6

Implementation of Working Prototype

This chapter focuses on the working prototype implementing the algorithm extensions presented in the previous chapter. The prototype was implemented using the C++ programming language [7] as a console application configurable through command line arguments. The configuration files are specified using the YAML mark-up language [3].

The first part of this chapter introduces the overall architecture of the application, with a focus on the roles of the individual modules. These are then discussed in further detail, and the differences between implementation of support systems required by the individual grammar and parser types are compared.

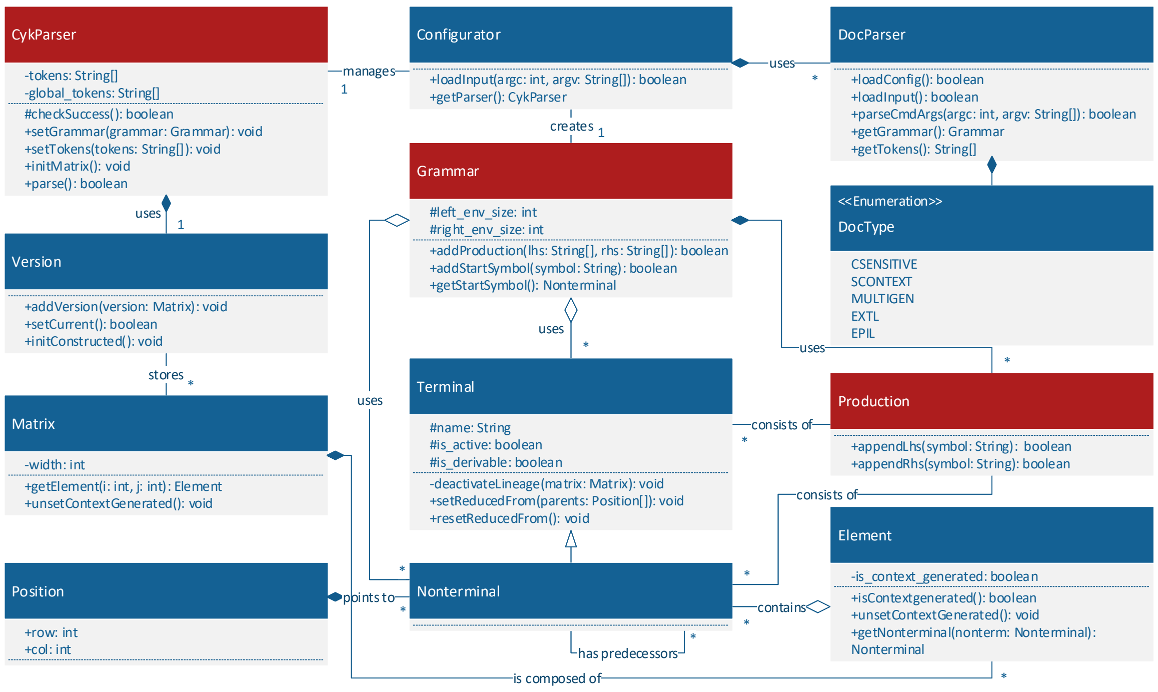
Finally, experimental data representing the time and space complexity of the individual algorithms is presented. The data is then compared to the theoretical values presented in Chapter 5.

6.1 Program Architecture

The program consists of several major modules responsible for different parts of its work, ranging from configuration validation, through its parsing and internal representation, to application of parsing algorithms, as seen in Figure 6.1. The entry point of the program is presented by the `Configurator` class, which is responsible for management of the configuration file parsing class, `DocParser`, and subsequent initialisation of the parsing module based on the user-defined parameters. The module consists of several polymorphic classes, such as the `Grammar` and `Production` classes, managed by an instance of the `CykParser` child class. The details of their implementation will be discussed later in the chapter.

6.2 Configuration Parser

As mentioned before, this module represents the entry point to the program's work, and is used to initialise the modules used for parsing. It is responsible for validation of configuration files supplied by the user as command line arguments, and their subsequent transformation into the internal data model. The module is managed by the `Configurator` class, which server as an interlink between the `DocParser` class used to parse the configuration files and the rest of the program.



CykParser
-tokens: String[]
-global_tokens: String[]
#checkSuccess(): boolean
+setGrammar(grammar: Grammar): void
+setTokens(tokens: String[]): void
+initMatrix(): void
+parse(): boolean
Configurator
+loadInput(argc: int, argv: String[]): boolean
+getParser(): CykParser
DocParser
+loadConfig(): boolean
+loadInput(): boolean
+parseCmdArgs(argc: int, argv: String[]): boolean
+getGrammar(): Grammar
+getTokens(): String[]
manages
1
uses
*
creates
1
Grammar
#left_env_size: int
#right_env_size: int
+addProduction(lhs: String[], rhs: String[]): boolean
+addStartSymbol(symbol: String): boolean
+getStartSymbol(): Nonterminal
<<Enumeration>>
DocType
CSENSITIVE
SCONTEXT
MULTIGEN
EXTL
EPIL
uses
1
Version
+addVersion(version: Matrix): void
+setCurrent(): boolean
+initConstructed(): void
stores
*
Matrix
-width: int
+getElement(i: int, j: int): Element
+unsetContextGenerated(): void
uses
*
uses
uses
*
Terminal
#name: String
#is_active: boolean
#is_derivable: boolean
-deactivateLineage(matrix: Matrix): void
+setReducedFrom(parents: Position[]): void
+resetReducedFrom(): void
Production
+appendLhs(symbol: String): boolean
+appendRhs(symbol: String): boolean
consists of
*
consists of
Element
-is_context_generated: boolean
+isContextgenerated(): boolean
+unsetContextGenerated(): void
+getNonterminal(nonterm: Nonterminal): Nonterminal
Position
+row: int
+col: int
points to
*
*
Nonterminal
*
contains
*
has predecessors
*
is composed of
*

Figure 6.1: General architecture of the prototype. Classes marked by red headers are polymorphic, and their implementation depends on the grammar type they are used with.

Once the `DocParser` has finished its work successfully, the `Configurator` instance uses a factory method to create a `CykParser` instance based on the determined grammar type. Finally, it passes the completed `Grammar` instance and the sequence of input tokens to the parser object, and returns it to the main program flow.

Grammar Configuration Parsing

After accessibility of the configuration files has been established, a `DocParser` object attempts to process their contents. The program uses two configuration files – a text file containing the input string, and a grammar specification file in the YAML 1.2 language [3]. The YAML language was chosen because of the high level of human readability and a high content-to-syntax ratio it offers. The format of the grammar configuration file can be found in Appendix C.

First, the grammar file is processed using the `yaml-cpp` library [2]. It is parsed to determine the used grammar type, and an instance of the corresponding grammar class is created, as discussed in Section 6.3. Then, the instance proceeds to validate and save the grammar's terminals and nonterminals, as well as the start symbol. When scanning the two symbol sets, their disjunction is continuously checked; the parsing is executed in a way that allows easy validation of the grammar. Finally, the production rules are processed. Based on the saved symbol sets, their validation and handling is managed by the previously created `Grammar` instance.

In case the program is working with a multigenerative grammar system, this process is repeated for each of its specified grammars; subsequently, the set of control sequences is scanned. For each of these sequences, it is checked whether its length is equal to the size of the system, and whether all used nonterminals are defined in their respective grammars.

Input String Parsing

After the grammar configuration has been completed, the input string is processed. It is divided into input tokens based on the delimiter specified in the configuration file. If a multi-generative grammar system was used, the contents of the file are first separated into a sequence of input strings based on the user-defined delimiter; naturally, the global and the token delimiters must differ. This step concludes the work of the `DocParser` class. Once it is done, the results are acquired by the `Configurator` class.

6.3 Grammar Management Module

This module serves as the internal representation of a formal grammar. As such, it is responsible for storing all symbols, productions and other properties defined by the grammar, and posing as a mediator for their manipulation. The module works with several classes representing the individual components of a formal grammar – the `Terminal` class representing symbols, the `Production` class combining defined symbols into production rules, and finally, the `Grammar` class, which serves as an intermediary between these classes and the rest of the program.

Symbol Representation

The symbols used in formal grammars are represented by two classes – `Terminal` and `Nonterminal`, each of which represents the corresponding symbol type. To facilitate use of instances of these classes, the `Nonterminal` class is inherited from the `Terminal` class, and it works with the same set of member variables. Because of this, the two classes can be used interchangeably.

These classes manage all information related to the symbol their instances represent. Naturally, they store information about the symbol name and its "derivability" – ability to be located on the left-hand side of a production. Other than these, the class keeps track of the activity of a symbol, which represents its ability to be used to reduce any symbols. This property is used to simulate symbol deletion while preventing it from being reduced again. Finally, the classes store information about the symbol's ancestors and its paired symbol.

Throughout the program, symbols are usually stored in instances of `std::unordered_set`, which uses the `std::hash` function to determine the position of individual instances inside the container. Since the symbol classes are not a part of the `std` namespace, the `std::hash` function was extended to cover these classes; the function ignores the activity flag of the instances to simulate permanent deletion of a symbol. By using this it, the need for an explicit comparison function was prevented.

Production Representation

The production rules of a formal grammar are represented by the `Production` class and its derived classes. Based on the corresponding grammar type, there exist several subclasses with different compositions. Each of these classes stores pointers to the symbols on their left-hand side and right-hand sides, represented by the `lhs` and `rhs` member variables in their respective order, and offers its own implementation of the corresponding getter and setter methods. There exist the following production rule classes:

- `Production` – Context-free production rule usable in multigenerative grammar systems and EP0L systems. Its `lhs` member is a pointer to a single nonterminal, whereas its `rhs` member contains a `std::vector` of nonterminal pointers.
- `SensitiveProduction` – Context-sensitive production. It extends the `Production` class; its `lhs` member is shadowed to contain a `std::vector` of nonterminal pointers.
- `ScatteredProduction` – Production type used in scattered context grammars. It extends the `SensitiveProduction` class, and its `rhs` member is shadowed to contain a two-dimensional `std::vector` of symbol pointers.
- `InteractiveProduction` – Production type used in EPIL systems. It is based on the `Production` class, however, with the addition of left and right environment members and corresponding getter and setter methods.

Validation of production forms is handled by the `Grammar` subclasses, as the production often lacks necessary information to make the decision itself, such as the size of the environment, or symbol definition and type.

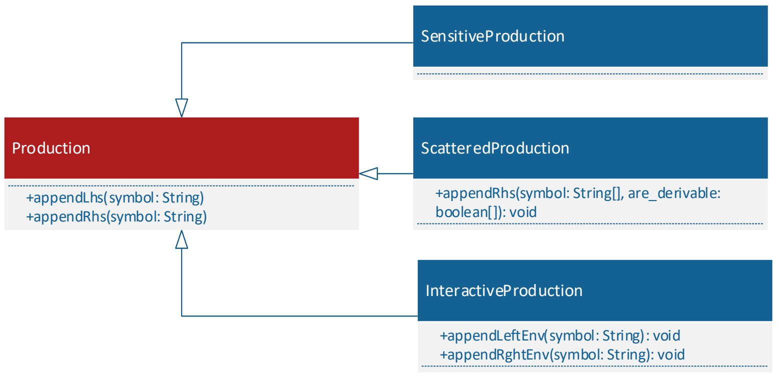

SensitiveProduction
Production
+appendLhs(symbol: String)
+appendRhs(symbol: String)
ScatteredProduction
+appendRhs(symbol: String[], are_derivable: boolean[]): void
InteractiveProduction
+appendLeftEnv(symbol: String): void
+appendRghtEnv(symbol: String): void

Figure 6.2: Inheritance hierarchy of the `Production` class responsible for production rule representation.

Grammar Representation

The `Grammar` class serves as a base class for the classes representing the individual grammar types the program works with. Because of this, definitions of most of its methods are empty, and are overridden in each of the derived classes to create a unified interface with the rest of the program.

Each class is responsible for determining the type of the added rule based on the phase of its application, as well as validating whether its form respects the normal form used with its respective grammar type. During the parsing, the class is used to find all possible reductions based on the contents of the provided nonterminal sets; for each valid combination of symbols, it creates a structure containing information about the reduced symbols and the symbols they were reduced from, `ReductionInfo`, which can be later used to detect their ancestry. The specifics of the derived classes are discussed below.

`ExtendedGrammar` and `InteractiveGrammar` Classes

The `ExtendedGrammar` class is based on EP0L systems as presented in Section 5.4. Considering it is almost identical with context-free grammars, the class is also used to represent the individual grammars of a multigenerative grammar system. It uses basic context-free production rules, represented by the `Production` class. Based on the time of their application, the rules are divided into three `std::vector` members:

- `terminal_productions` – productions applied during the initialisation phase of the matrix; the right-hand side is composed of a single terminal,
- `unary_productions` – applied during the first part of matrix reduction; the right-hand side is composed of a single nonterminal,
- `nonterminal_productions` – productions used during the final part of matrix reduction; the right-hand side is composed of two nonterminals.

For each of the presented production types, there exists a separate reduction method – `getTerminalLhs()`, `getUnaryLhs()`, and `getBasicLhs()` in their respective order.

The `InteractiveGrammar` class represents the EPIL systems as presented in Section 3.5. The behaviour of this class extends the `ExtendedSystem` class, and adapts it to work with `InteractiveProduction` class. Otherwise, most of their behaviour is shared, as the environment requirements in an EPIL system are ignored until the final part of a parsing iteration.

`ContextSensitiveGrammar` Class

This class represents context-sensitive grammars as presented in Section 2.4. Considering the members of an EP0L system are similar to context-free grammars, this class extends the behaviour offered by the `ExtendedGrammar` class.

Other than the production types used by the base class, the `ContextSensitiveGrammar` class contains the `binary_productions` member, which is composed of `SensitiveProduction` instances. These productions are used during the final part of reduction checks

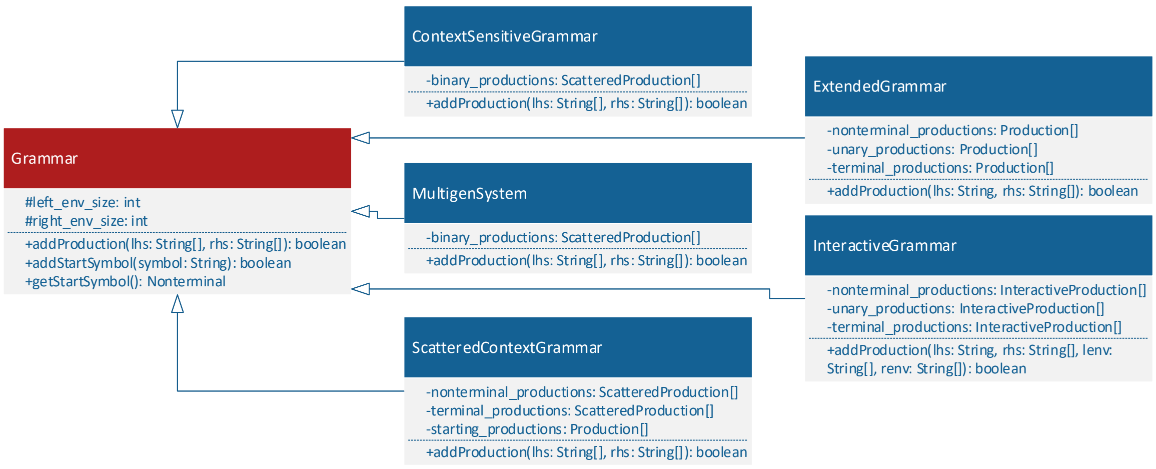

ContextSensitiveGrammar
-binary_productions: ScatteredProduction[]
+addProduction(lhs: String[], rhs: String[]): boolean
ExtendedGrammar
-nonterminal_productions: Production[]
-unary_productions: Production[]
-terminal_productions: Production[]
+addProduction(lhs: String, rhs: String[]): boolean
Grammar
#left_env_size: int
#right_env_size: int
+addProduction(lhs: String[], rhs: String[]): boolean
+addStartSymbol(symbol: String): boolean
+getStartSymbol(): Nonterminal
MultigenSystem
-binary_productions: ScatteredProduction[]
+addProduction(lhs: String[], rhs: String[]): boolean
InteractiveGrammar
-nonterminal_productions: InteractiveProduction[]
-unary_productions: InteractiveProduction[]
-terminal_productions: InteractiveProduction[]
+addProduction(lhs: String, rhs: String[], lenv: String[], renv: String[]): boolean
ScatteredContextGrammar
-nonterminal_productions: ScatteredProduction[]
-terminal_productions: ScatteredProduction[]
-starting_productions: Production[]
+addProduction(lhs: String[], rhs: String[]): boolean

Figure 6.3: Inheritance hierarchy of the `Grammar` class, which is responsible for internal representation of formal grammars and their management.

in an iteration by the `getBinaryLhs()` method. As opposed to the previous methods, this method generates two separate sets of reduced nonterminals – one for each source set.

`MultigenSystem` Class

This class represents multigenerative nonterminal-synchronised grammar systems as presented in Section 2.6. It is composed of a sequence of `ExtendedGrammar` instances representing the individual context-free grammars, and a two-dimensional `std::vector` of nonterminals representing the control sequences.

During the parser configuration process, the components of the individual grammars are created separately. To facilitate creation of complete instances stored by the `MultigenSystem` class, methods used to signalize their initialisation and completion have been introduced by the class – these are the `startGrammar()` and `addGrammar()` methods respectively. By using these methods, the need for explicit creation of an `ExtendedGrammar` instance inside the configuration parser is also prevented, leading to lower coupling between the classes.

`ScatteredContextGrammar` Class

This class represents scattered context grammars as presented in Section 2.7. As opposed to the previous types, scattered context grammars are semi-parallel, and apply multiple context-free productions during a single reduction. To simulate this behaviour, the class uses production form represented by the `ScatteredProduction` class. Based on the time of their application, the rules are divided into three `std::vector` members:

- `terminal_productions` – productions applied during the initialisation phase; the right-hand side strings are composed exclusively from terminals,
- `nonterminal_productions` – productions applied during the parsing process,
- `starting_productions` – context-free productions used to reduce the start symbol.

For both of the scattered context containers, there exist corresponding partial left-hand side getters similar to the ones mentioned in the previous classes. These are accompanied by the `existsTerminalRule()` and `existsNonterminalRule()` methods to check the existence of a complete production rule.

6.4 Matrix Manipulation Module

This module is responsible for representation of the matrices utilised by the Cocke-Younger-Kasami algorithm. This includes the individual elements of a matrix, the matrices as a whole, and management of matrix versions. These are represented by the `Element`, `Matrix` and `Version` classes in their respective order.

Parsing Matrix Representation

The matrices used by the Cocke-Younger-Kasami parsing algorithm are composed of nonterminal sets represented by the `Element` class. The class uses an `std::unordered_set` to simulate the corresponding element of the CYK matrix. It also offers allowing manipulation of the element's contents and checks required by the parsing algorithms.

The Matrix class represents a two-dimensional grid of Element instances simulated by a one-dimensional std::vector, whose size is equal to the second power of the input string length. This class is used as an intermediary between the parser classes and the non-terminal sets, and is used to handle individual elements as well as recursive operations, such as search and deactivation of the predecessor tree. During the parsing, the Matrix instance is used for acquisition of nonterminal sets at given two-dimensional coordinates, and subsequently adding the new nonterminals to the corresponding elements. The individual Element instances do not know their own coordinates; the Matrix class uses a coordinate mapping function for correct element acquisition.

Version Control System

A major part of the implemented algorithms utilises a matrix version set. This set is represented by the Version class, which is used to manage instances of the Matrix class. It works with three fields – the scanned matrix, current, the currently edited alternate matrix, constructed, and a list of alternate matrices waiting to be scanned. It also offers functionality for copying matrices, resetting their elements individually, and appointing a new working matrix.

Appointment of working matrices is achieved by using the setCurrent() method. The method attempts to appoint the last added alternate version as the new working matrix, and returns a bool based on its success. The Matrix instance is picked from the end of the std::vector to assure the lowest possible number of operations is needed to finish parsing the matrix.

In parsing methods that do not remove unsuccessful matrices, the class also offers functionality to save past matrix states; this can be utilised to determine whether the parsing process has entered an infinite loop and must therefore be halted. This part of its functionality is managed by the addMatrix() and containsVersion() methods respectively.

6.5 Grammar Parsing Core

This module is represented by the CykParser class, and is responsible for implementation of the algorithms introduced in Chapter 5. The CykParser class serves as an abstract base class for the classes implementing individual algorithms, and provides a unified interface between the main program flow, which does not need to know what type of grammar it is dealing with, and the implementation of the parsing methods. There exist five derived parser classes, one for each algorithm; each of these classes contains an instance of the corresponding grammar class, and instances of both the Matrix class, represented by the matrix field, and the Version class, represented by the versions field.

Generally, the parsing process is divided into three separate phases – matrix initialisation, nonterminal reduction, and success checking. These are carried out by the initMatrix(), parseMatrix(), and checkSuccess() methods in their respective order. During the first two phases, the matrix is scanned in the order defined by the individual algorithms, and reduced nonterminals are acquired using the corresponding left-hand side getter methods. These are then inserted into their destination Element instances, and if needed, their ancestry and paired symbol are recorded. Implementation of the last phase differs based on the grammar type.

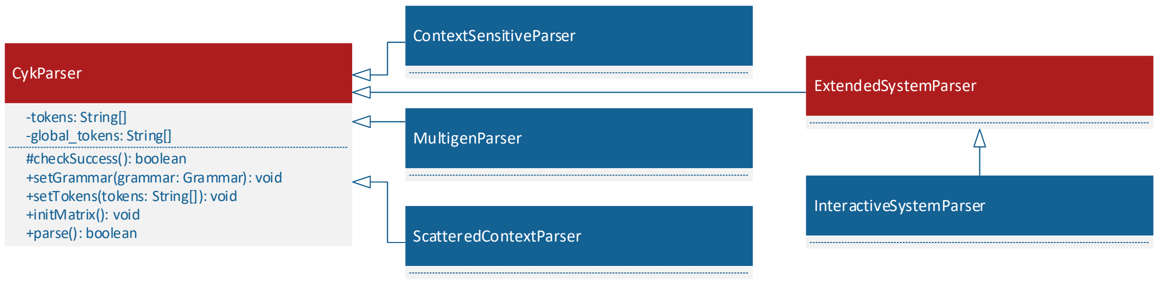


Figure 6.4: Inheritance hierarchy of the `CykParser` class, which is responsible for implementation of the individual parsing algorithms.

`ContextSensitiveParser` Class

This class implements the parsing algorithm for context-sensitive grammars in the Kuroda normal form introduced in Section 5.1. It works with an instance of the `ContextSensitive-Grammar` class, and uses the `Version` class for matrix handling.

During the initialisation phase, the program iterates through the `tokens` list, and attempts to reduce the individual tokens using terminal productions of the bundled grammar. Each of the resulting nonterminals is then inserted into the corresponding element of the `matrix` field.

The parsing phase is separated into two parts: first, context-free reductions are made in a way analogous to the previous phase. Then, the symbols reduced using context-sensitive rules are acquired. An inactive copy of both nonterminals is added to the scanned matrix, `matrix`, to prevent their repeated reduction. In the `constructed` matrix, the ancestry tree of both nonterminals is deactivated to assure their use, and active copies of both reduced nonterminals are saved.

Once the value of the `matrix` field can no longer be changed, it is checked whether the start symbol of the grammar can be found in the element representing the upper right-hand corner of the matrix. If so, the string is accepted. Otherwise, if the version list contains any `Matrix` instances, a new one is moved to the `matrix` field, and parsing continues. If no more matrices are left, the string is rejected.

`MultigenParser` Class

This class implements the parsing algorithm for multigenerative grammar systems in the Chomsky normal form introduced in Section 5.2. It works with an instance of the `Mul-tigenSystem` class, as well as a sequence of current matrices, `current_matrices`, and a sequence of working matrices, `candidate_matrices`. For parallel matrix reduction, it utilises the `std::thread` functionality.

During the initialisation phase, the individual tokens of all grammars are scanned and reduced, and the resulting nonterminals are inserted into the corresponding `Element` instances. This phase is executed sequentially, as the resource cost of the thread creation is quite high compared to the cost of the phase itself.

The parsing phase, however, is executed in parallel – for each of the candidate matrices, a separate `std::thread` worker is created. After the matrices have been reduced and filtered to contain only leftmost symbols, the `checkControlSequences()` method is called. This method creates a list of sought control nonterminals for each matrix based on their position in the multigenerative system, and launches a sequence of `std::async` worker threads whose purpose is to delete nonterminals not found in the control sequence. Once all threads have returned flags indicating local success of individual control sequence checks, these are combined into a global flag. Again, incomplete sequences are removed. Finally, a global minimum of non-empty elements is found, and excess elements are reset. Remaining nonterminals are then moved to their respective main matrices, and parsing continues.

Once no nonterminals are added at the end of the iteration, the parsing is complete. The program then checks if all grammars have been reduced into a start symbol located in the symbolic upper right-hand corner of the matrices. If it is found in all main matrices, the sequence of input strings is accepted.

`ScatteredContextParser` Class

This class implements the parsing algorithm for scattered context grammars in the 2-limited normal form introduced in Section 5.3. It works with an instance of the `ScatteredContextGrammar` class, and is the only algorithm to use a single parsing matrix.

During the initialisation phase, the input tokens are first scanned to find all reductions made using the left partial terminal rules. First, this is done for unary productions, containing a single terminal on their right-hand side, followed by a scan for binary productions composed of a pair of neighbouring symbols. For each matched partial rule, an analogous search for the right partial derivation is triggered using the rest of the input tokens. Once a right reduction has been found, it is checked whether a complete production composed of the two partial productions exists. If so, the nonterminals are saved; for each saved nonterminal, its ancestors and paired symbol are recorded as well. In addition to the reduced nonterminals, the token itself is added into each matrix element.

The parsing phase works analogously with the initialisation phase, with the addition of starting productions – for each pair of neighbouring elements, the algorithm attempts to reduce the start symbol; the symbol has only ancestors, but no paired symbol.

Once the reduction has been completed, the start symbols located in the upper right-hand corner of the matrix are scanned. For each of these nonterminals, the ancestry tree is scanned recursively, and the paired symbol of every right ancestor is recorded in a `std::unordered_set`. Subsequently, the same scan is executed for the left ancestors – if they are not found in the set, or if they are found more than once, the ancestry tree is invalid, and the symbol is erased. Finally, if the examined element still contains any instances of the start symbol, the input string is accepted.

`ExtendedSystemParser` Class

This class implements the parsing algorithm for EP0L systems introduced in Section 5.4. It works with an instance of the `ExtendedSystem` class, and adds an additional matrix – the `future_matrix`, holding the currently reduced sentential form; it also uses a list of previous matrices represented by the `Version` class.

During the initialisation phase, the input tokens are scanned and reduced; the resulting nonterminals are then inserted into their respective elements. The parsing phase is composed of two parts: first, unary reductions are found for a substrings; this phase is executed first because in case an unary reduction has been made, the following step of the parsing process must start to the immediate right of the reduced symbol to prevent creation of empty spaces in the sentential form. Then, the same is done for a pair of neighbouring substrings using productions whose right-hand side is composed of two nonterminals.

Finally, once the `future_matrix` has been completed, it is checked whether it contains the start symbol in its upper right-hand corner, and therefore, the input string can be accepted. Subsequently, it is checked whether an identical matrix can be found in the version list. If so, it means the grammar is repeating, and will never lead to an accepted string. In this case, the string is rejected. Otherwise, the matrix is added to the version list, the `matrix` field is replaced by its value, and the parsing continues.

`InteractiveSystemParser` Class

This class implements the parsing algorithm for EPIL systems introduced in Section 5.5. It extends the `ExtendedSystemParser` class, and therefore, most of the classes' behaviour is shared. The `InteractiveSystemParser` class presents two additions to the behaviour of the base class.

The first difference is applied during the reduction phase. When saving nonterminals, the `addEnvRequirements()` method is called. This method is not implemented in the parent class, however, the `InteractiveSystemParser` uses it to save the value and relative positions of the nonterminals checked at the end of the iteration. This data is acquired from the `ReductionInfo` structure obtained from the left-hand side getter method.

The other difference is present at the end of the iteration, when the `validateContext()` is called. Again, the method is implemented only in the derived class. First, the method iterates through the reduced nonterminals, and finds their parents in the `matrix` field representing the previous state of the sentential form. For each of the found parents, it checks whether there exist nonterminals in the `future_matrix` field reduced using their environment. If not, the nonterminals are removed. Then, a second scan through the reduced nonterminals is executed, this time confirming existence of the symbols' own environment in the `future_matrix` field.

6.6 Experimental Results

This section presents experimental values of time and space complexity of the implementations of the presented algorithms. For each of the algorithms, the tests were run for every combination of input strings and used grammars presented. To simulate the worst case scenario the complexities were derived for, the grammars used work with the same alphabet as the input string, however, the strings are never accepted.

The experimental values were acquired using the `perf` tool available on Linux systems; specifically, the `perf-stat` and `perf-mem` tools were used to measure time and space complexity respectively. The tests were performed on a system running the Ubuntu 19.04 operating system, using the kernel version `5.0.0-13-generic`. The values given in the tables represent the mean values of five independent runs.

Experimental results for context-sensitive grammars can be seen in Table 6.1a; the comparison of these values to their theoretical counterparts presented in Chapter 5 be subsequently seen in Figure 6.5. Analogous tables for the remaining sequential grammar types, along with their visual representation, can be found in Appendix A. As visible from the figure, the increase in the consumed resources follows the same trend in both versions of the complexities. Apparently, the use of recursive ancestry checks in the context-sensitive and scattered context algorithms causes a drastic increase of the consumed resources for longer input strings. However, due to the nature of ancestry checks in the final phase of the scattered context parsing algorithm, the current implementation presents

$\lvert G\rvert$	$\lvert w\rvert$	Time [Instr]	Space [MB]
0	0	5,164,957	0.023
0	5	5,388,150	0.024
0	10	6,446,883	0.026
0	15	34,487,940	0.031
5	0	221,149,957	0.049
5	5	6,164,597	0.025
5	10	12,109,167	0.026
5	15	612,760,524	0.078
10	0	10,017,377,826	1.136
10	5	6,909,997	0.025
10	10	13,507,768	0.027
10	15	647,651,809	0.128
15	0	7,435,454,589	0.662
15	5	7,653,079	0.026
15	10	14,309,873	0.071
15	10	1,058,657,848,216	696.038

(a) Values for the context-sensitive parsing algorithm.

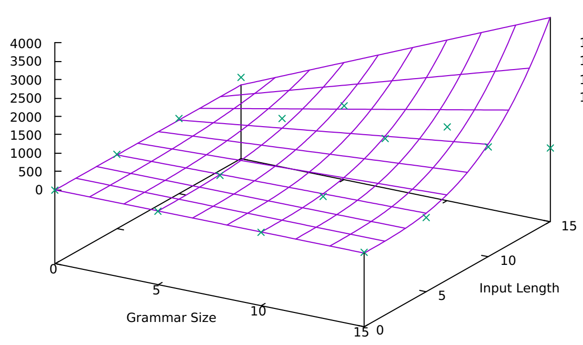
$\lvert G\rvert$	$\lvert w\rvert$	Time [Instr]	Space [MB]
0	0	5,315,901	0.024
0	5	5,509,703	0.024
0	10	5,982,849	0.024
0	15	6,759,919	0.024
5	0	6,014,984	0.024
5	5	7,330,312	0.024
5	10	9,480,175	0.025
5	15	12,763,288	0.026
10	0	6,753,253	0.024
10	5	8,269,055	0.025
10	10	10,828,568	0.026
10	15	14,857,878	0.026
15	0	7,294,814	0.024
15	5	8,900,354	0.024
15	10	11,520,127	0.025
15	15	16,047,356	0.026

(b) Values for the EP0L system parsing algorithm.

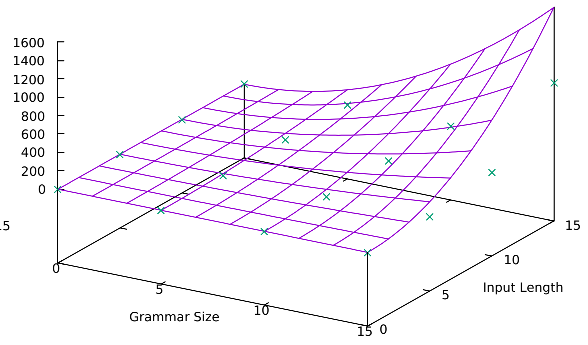
Table 6.1: Experimental complexities of the implemented algorithms. The values are given based on the size of the grammar, G, and length of the input string, w.

a compromise between the amount of saved data and length of the run time for all implemented algorithms.

Similarly, the values for the extension for EP0L systems can be seen in Table 6.1b; the visualisation of the presented values, along with the comparison to the theoretical complexities and the results for EPIL systems can be found in Appendix B.



(a) Time complexity.



(b) Space complexity.

Figure 6.5: Experimental results of time and space complexity of the context-sensitive parsing algorithm (green) compared to the respective theoretical values (purple).

Chapter 7

Conclusion

This thesis examines the properties of sequential and parallel grammars, and the applications of these grammars. Its aim is to design and present a set of algorithms capable of parsing grammar types belonging to these groups, with a focus on grammars that are not context-free. These algorithms are based on the Cocke-Younger-Kasami algorithm for context-free grammars in the Chomsky normal form and are capable of parsing any grammar in the corresponding normal form.

Using the Chomsky hierarchy as a base for comparison of sequential grammars, it discusses grammar types both presented as a part of the hierarchy, and not fully respecting the bounds placed by it. The thesis focuses on context-sensitive grammars, multigenerative nonterminal-synchronised grammar systems, and scattered context grammars. For each of the grammars outside the hierarchy, their generative power is compared to that of the family of unrestricted grammars, which represents its the major grammar type.

Then, normal forms of the presented sequential grammar types are inspected. First, the normal forms for context-free grammars are discussed, as the algorithms presented later in the thesis utilise the similarities between the normal forms of their respective grammar types and the normal forms for context-free grammars. Specifically, the Chomsky normal form and the Greibach normal form are presented; because of the structural similarities between the grammar types, these can be also applied to multigenerative grammar systems. Subsequently, the Kuroda normal form, and its special case – the Penttonen normal form – used for unrestricted and context-sensitive grammars are presented; these present a direct extension of the Chomsky normal form, and add a single binary rule form. Finally, the 2-limited normal form for scattered context grammars, using rule form similar to a sequence of two context-free productions in the Chomsky normal form, is presented.

Subsequently, parallel grammars – also called the L-systems – are examined. Their purpose and properties, such as parallelism and system branching, are discussed; languages generated by L-systems are compared to their sequential equivalent, and the inability of sequential grammars to simulate parallel rewriting because of a lack of synchronising mechanism is stated.

As opposed to sequential grammars, which form a sequence of proper subsets in the order strictly defined by the Chomsky hierarchy, a list of grammar modifiers, forming the L hierarchy, is given along with a comparison of generative powers of significant grammar types created by their combination. Contradictory to the Chomsky hierarchy, the generative power of some families proves to be incomparable. Finally, three major families are discussed – these are the extended systems, which act as a parallel counterpart to context-free grammars; table systems, which are unique to the L-systems; and finally, interactive systems, comparable to context-sensitive and unrestricted grammars of the Chomsky hierarchy.

A normal form for extended L-system similar to the Chomsky normal form is presented; it is subsequently adapted to work with extended interactive L-systems, allowing the extension of the Cocke-Younger-Kasami algorithm to these grammar types. To outline the significance of the environment in interactive systems, a final normal form focused on its size is presented.

As the final part of the theoretical background of this thesis, various types of parsing methods are discussed. After introducing the basic terms used in the context of parsing techniques, the chapter introduces top-down parsing methods, gives a brief overview of their work, and finally discusses the restrictions placed by some of their representatives. Subsequently, bottom-up parsing methods, working from input string up to the reduction of the start symbol, are introduced in a similar way. Finally, the group of parsing methods capable of parsing any grammar of the corresponding family – the general methods – are introduced. Their representative, the Cocke-Younger-Kasami algorithm, which serves as the core for parsing methods presented in this thesis, is introduced. It is accompanied by an informal description of its work, the corresponding pseudocode representation, and a graphical demonstration of its progression. These serve as a base of comparison to the presented parsing methods.

The following part of the thesis presents the proposed extensions of the Cocke-Younger-Kasami algorithm. These extensions work with grammar types introduced previously, and utilise the similarities between the Chomsky normal form and the respective normal forms of the individual grammar types. However, each of these types presents a set of unique properties that require the addition of mechanisms assuring the reliability of results offered by the algorithms. Overall, the thesis presents five distinct parsing algorithms.

The extension for context-sensitive grammars works with grammars in the Kuroda normal form. As this normal form presents a direct extension of the Chomsky normal form, the parsing core of the algorithm only needs to be extended to handle the binary production form. However, in case a single nonterminal of a context-sensitive pair is used in the final parse tree, the reliability of the algorithm's result is compromised. To counter this problem, the algorithm employs a set of matrix versions; every time a set of context-sensitive productions is applied on a pair of elements, two versions of the matrix are created – the former, which is artificially prevented from the applying the productions, and the latter, which contains only the reduced nonterminals, but none of their ancestors – resulting in the need to use both of these nonterminals to finish the parsing successfully. This algorithm reaches the time complexity of $O(|G| \cdot n^3)$.

The extension for canonical multigenerative nonterminal-synchronised systems works with systems composed of context-free grammars in the Chomsky normal form. Considering these systems compare the results of leftmost derivation to the attached control sequences, the extension uses an additional sequence of parsing matrices – one for each grammar, and therefore for each normal parsing matrix. The parsing process is analogous to that of the original algorithm, with the difference that reduced nonterminals are saved into the additional matrices. These are then filtered to contain only nonterminals resulting from the reverse simulation of leftmost derivation, which are then compared to control sequences and filtered further. Finally, the matrices are synchronised to contain the same number of reduced elements – and therefore substrings – and their contents are moved to the corresponding main matrices. Because of the need for a sequence of matrices, the time complexity of this algorithm for a system of size N reaches $O(N \cdot n^3)$.

The final sequential extension works with scattered context grammars in the 2-limited normal form. Because of its structure, the extension simulates the application of two context-free rules at different positions of the parsing matrix. However, similarly to the extension for context-sensitive rules, the consistency of the matrix is disrupted if the simulation of the resulting parse tree contains only one nonterminal of the simultaneously reduced pair. As opposed to the previous extension, all reductions are made inside a single matrix, as the previous system would lead to a vast increase of the time complexity of the algorithm. Instead, at the end of the parsing, the algorithm scans the ancestry of all reduced start symbols, and removes the ones whose ancestry tree is either lacking expected members, or contains duplicate symbols. Because of the semi-parallel nature of scattered context grammars, the time complexity of the original algorithm is squared, resulting in the relatively high complexity of $O(|G| \cdot n^6)$.

The main challenge of adapting the Cocke-Younger-Kasami algorithm to parallel grammars proved to be the fact that, as described earlier in the thesis, the parallelism of this group of grammars cannot be simulated by context-free grammars. Similarly to some of the previous extensions, the algorithm for extended L-systems uses a pair of matrices – one to hold the previous sentential form, and one to hold the sentential form that is currently being reduced. These matrices are periodically switched, and therefore, the leftover symbols from previous sentential forms are removed. Moreover, to prevent reduction of overlapping sentential forms in a matrix, the algorithm always scans the substring located to the right of the previously reduced symbols. Overall, the time complexity of this extension reaches $O(|G| \cdot n^3)$.

The extension for EPIL systems shares the core behaviour of the previous algorithm. Compared to it, this extension adds several environment-related checks. As opposed to the notion of context in sequential grammars, interactive L-systems do not alter their environment in any way; moreover, because of the sequential nature of the parsing method, the environment may not exist at the time of the reduction. This is countered by two waves of reduction checks – first, it is checked whether the current matrix contains descendants of all environment strings; then, it is checked whether the environment of all new nonterminals exists. The addition of these control mechanisms increases the time complexity of the extension, bringing it to the overall value of $O(|G|^2 \cdot n^3)$.

The final part of the thesis is dedicated to the working prototype implementing these algorithms. It is a C++ program configurable through command line arguments. The program utilises the principles of object-oriented programming, and employs a set of polymorphic classes mirroring the individual algorithms and their corresponding grammar types. First, the overall structure of the class hierarchy is discussed, followed by a detailed descriptions of the work of individual modules, their purpose, and derived classes. Finally, the experimental values of time and space complexity of the algorithms are compared to their theoretical counterparts.

This thesis deals with a complex topic, and while it yields significant results, it does not exhaust the topic completely. As such, it may be used as a basis for further research; the following topics may prove interesting during related future research:

- extension of the Cocke-Younger-Kasami algorithm to other grammar types, such as matrix grammars, or unrestricted grammar with restricted erasing,
- extension to other normal forms, such as the Greibach normal form, which offers an easily processable rule form while covering relatively long substrings,
- extension to different parsing algorithms, i.e. the algorithms of the LR family.

Bibliography

[1] Aho, A. V.; Lam, M. S.; Sethi, R.; et al.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley. 2006. ISBN 0-321-48681-1.

[2] Beder, J.: *yaml-cpp: A YAML parser and emitter in C++*. Accessed: March 12, 2019.
Retrieved from: https://github.com/jbeder/yaml-cpp

[3] Ben-Kiki, O.; Evans, C.; döt Net; I.: *YAML: YAML Ain't Markup Language*. Accessed: May 09, 2019.
Retrieved from: https://yaml.org/

[4] Fernau, H.; Meduna, A.: *A simultaneous reduction of several measures of descriptional complexity in scattered context grammars. Information Processing Letters*. vol. 86, no. 5. 2003: pp. 235–240.

[5] Free Software Foundation: *Bison - GNU Project*. Accessed: May 02, 2019.
Retrieved from: https://www.gnu.org/software/bison/

[6] Herman, G. T.; Rozenberg, G.: *Developmental Systems and Languages*. North-Holland. 1975. ISBN 0-7204-2806-8.

[7] ISO/IEC: *ISO/IEC 14882:2011 — Programming language C++*. International Organization for Standardization. 2011.
Retrieved from: https://www.iso.org/standard/5037.html

[8] Jurafsky, D.; Martin, J. H.: *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall. 2009. ISBN 978-0-13-187321-6.

[9] Klobučníková, D.: *General Grammars: Normal Forms with Applications*. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. 2017.
Retrieved from: http://www.fit.vutbr.cz/study/DP/BP.php?id=18090

[10] Klobučníková, D.: *Sequential and Parallel Grammars: Properties and Applications*. Term project. Brno University of Technology, Faculty of Information Technology. 2019.

[11] Lukáš, R.: *Multigenerative Grammar Systems*. PhD. Thesis. Brno University of Technology, Faculty of Information Technology. 2006.

[12] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer. 2000. ISBN 978-1-85233-074-3.

[13] Meduna, A.: *Elements of Compiler Design*. Auerbach Publications. 2008. ISBN 978-1-4200-6325-7.

[14] Meduna, A.; Techet, J.: *Scattered Context Grammars and Their Applications*. WIT Press. 2010. ISBN 978-1-84564-426-0.

[15] Rekers, J.; Schuerr, A.: *A graph grammar approach to graphical parsing*. In *Proceedings of Symposium on Visual Languages*. IEEE. 1995. ISBN 0818670452. pp. 195–202.

[16] Rozenberg, G.; Salomaa, A.: *The Mathematical Theory of L Systems*. Academic Press. 1980. ISBN 0-12-597140-0.

[17] Rozenberg, G.; Salomaa, A.: *Handbook of Formal Languages*. vol. 1. Springer. 1997. ISBN 978-3-642-63863-3.

[18] Zulkufli, M.; Liyana, N.; Sherzod, T.; et al.: *Watson–Crick Context-Free Grammars: Grammar Simplifications and a Parsing Algorithm. The Computer Journal*. vol. 61, no. 9. 01 2018: pp. 1361–1373. ISSN 0010-4620.

Appendix A

Experimental Results for Sequential Grammars

Experimental Results

This appendix presents the experimental values of time and space complexity of implementation of the corresponding sequential parsing algorithms. Fist, the data is presented in the tabular form, followed by its graphical representation and comparison with the theoretical values. Data for the context-sensitive algorithm can be found in Table 6.1a and Figure 6.5.

$\|G\|$	$\|w\|$	Time [Instr]	Space [MB]
0	0	7,772,303	0.40
0	5	8,417,602	0.037
0	10	10,452,359	0.038
0	15	14,654,074	0.039
5	0	10,191,619	0.036
5	5	13,480,018	0.046
5	10	20,039,445	0.053
5	15	37,968,424	0.066
10	0	12,346,996	0.035
10	5	31,906,507	0.099
10	10	106,031,248	0.167
10	15	326,667,795	0.249
15	0	6,882,497	0.026
15	5	26,322,047	0.071
15	10	120,340,804	0.165
15	15	356,510,584	0.262

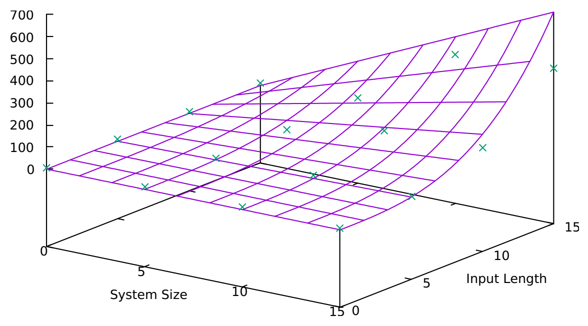
(a) Values for multigenerative grammar systems.

$\|G\|$	$\|w\|$	Time [Instr]	Space [MB]
0	0	5,900,367	0.024
0	5	5,981,543	0.024
0	10	10,042,903	0.026
0	15	19,896,480	0.028
5	0	6,476,144	0.023
5	5	8,169,710	0.024
5	10	19,498,876	0.027
5	15	49,471,827	0.032
10	0	7,622,626	0.024
10	5	30,754,922	0.029
10	10	176,586,524,785 k	19633.453
10	15	980,512,327,594 k	173636.37
15	0	8,927,581	0.024
15	5	17,410,782	0.026
15	10	190,252,526 M	19207.563
15	15	2,379,851,377 M	217045.462

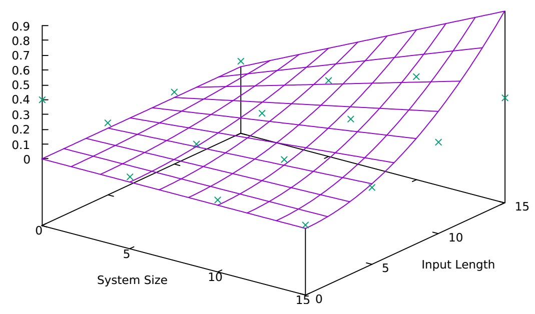
(b) Values for scattered context grammars.

Table A.1: Experimental complexities of sequential algorithms. The values are given based on the size of the grammar, G, and length of the input string, w. Multigenerative grammar systems work present a global sum of found values.

Visualisation of the Acquired Results

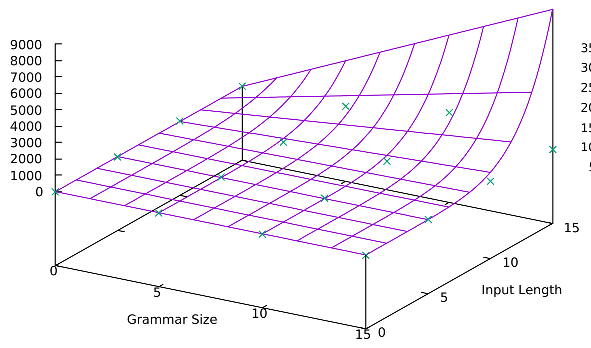


(a) Time complexity.

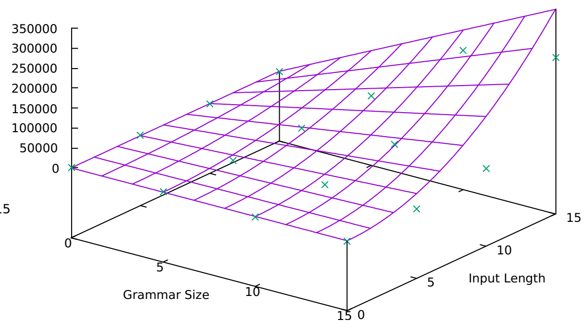


(b) Space complexity.

Figure A.1: Experimental values of time and space complexity of the parsing algorithm for multigenerative grammar systems (green) compared to their theoretical counterpart (purple). The visual representation does not consider the number of nonterminals in a system.



(a) Time complexity.



(b) Space complexity.

Figure A.2: Experimental values of time and space complexity of the parsing algorithm for scattered context grammars compared to their theoretical counterpart.

Appendix B

Experimental Results for Parallel Grammars

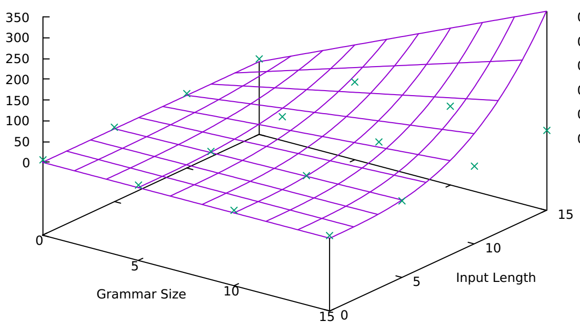
This appendix presents the experimental values of time and space complexity of implementation of the corresponding parallel parsing algorithms. Fist, the data for EPIL systems is presented in the tabular form, followed by the graphical representation of both EP0L and EPIL algorithm results and their comparison with the theoretical values. Data for the context-sensitive algorithm can be found in Table 6.1b.

Experimental Results

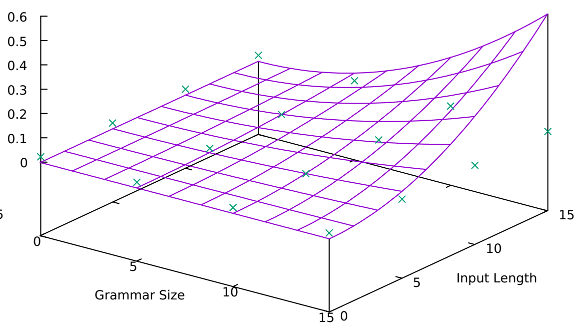
$\lvert G\rvert$	$\lvert w\rvert$	Time [Instr]	Space [MB]
0	0	5,445,486	0.024
0	5	5,590,554	0.024
0	10	6,004,689	0.024
0	15	6,666,704	0.025
5	0	6,348,203	0.024
5	5	6,655,790	0.025
5	10	7,141,648	0.024
5	15	8,030,734	0.025
10	0	7,171,710	0.024
10	5	7,877,296	0.025
10	10	9,630,884	0.025
10	15	11,733,910	0.026
15	0	8,044,261	0.025
15	5	8,717,743	0.025
15	10	11,622,802	0.026
15	15	14,354,274	0.028

Table B.1: Experimental complexities of the parsing algorithm for EPIL systems. The values are given based on the size of the grammar, G, and length of the input string, w.

Visualisation of the Acquired Results

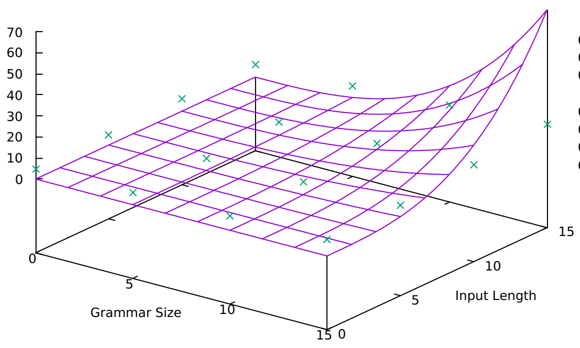


(a) Time complexity.

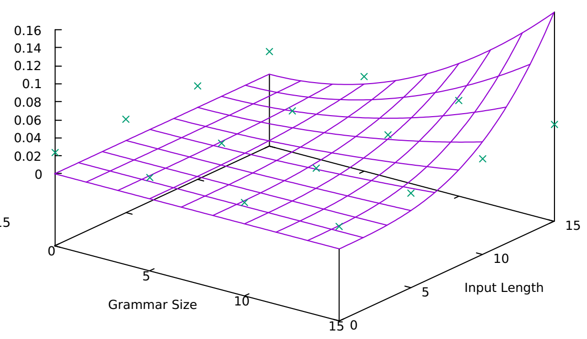


(b) Space complexity.

Figure B.1: Experimental values of time and space complexity of the parsing algorithm for EP0L systems (green) compared to their theoretical counterpart (purple).



(a) Time complexity.



(b) Space complexity.

Figure B.2: Experimental values of time and space complexity of the parsing algorithm for EPIL systems compared to their theoretical counterpart.

Appendix C

Examples of Grammar Configuration Files

This appendix demonstrates the syntax of the gramma configuration files. First, a general example is given using all configurable fields; these do not have to be used at the same time. Subsequently, an example of multigenerative grammar system configuration, using multiple grammars, is given. The configuration files are written in the YAML mark-up language.

General Configuration File Example

```yaml
---
gtype: cs | sc | eol | epil
delimiter: ' '
grammar:
  nonterminals: ['S', 'A', 'B', 'C']
  terminals: ['a', 'b', 'c']
  start: 'S'
  environment: [1, 1]
  productions:
    - lhs: ['S']
      rhs: ['A', 'B']
    - lhs: ['S', 'A']
      rhs: ['B', 'C']
    - lhs: ['S']
      rhs: ['A', 'B']
      lenv: ['B']
      renv: ['C']
    # continued
...
```

Multigenerative Grammar System Configuration Example

```yaml
---
gtype: mg
delimiter: ' '
string_delim: '\n'
grammars:
  - nonterminals: ['S', 'A', 'B', 'C']
    terminals: ['a', 'b', 'c']
    start: 'S'
    productions:
      - lhs: ['S']
        rhs: ['A', 'C']
      # continued
  - nonterminals: ['S', 'A', 'B', 'C']
    terminals: ['a', 'b', 'c']
    start: 'S'
    productions:
      - lhs: ['S']
        rhs: ['A', 'B']
     # continued
  - nonterminals: ['S', 'A', 'B', 'C']
    terminals: ['a', 'b', 'c']
    start: 'S'
    productions:
      - lhs: ['S']
        rhs: ['B', 'C']
      # continued
control:
  - ['A', 'S', 'S']
  - ['S', 'C', 'B']
...
```

Appendix D

Contents of the Attached Storage Medium

The attached medium contains the following directories and files:

- `bin/` – directory containing the `parserium` executable and example grammar configuration files,
- `doc/` – directory containing the LaTeX source codes and all other files needed to compile this document,
- `pdf/` – directory containing the electronic and print-ready version of this document,
- `src/` – directory containing the program source codes and used libraries,
- `README` – text file containing instructions for compilation and execution of the program.