BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

GENERAL GRAMMARS: NORMAL FORMS WITH APPLICATIONS

OBECNÉ GRAMATIKY: NORMÁLNÍ FORMY A JEJICH APLIKACE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DOMINIKA KLOBUČNÍKOVÁ**
AUTOR PRÁCE

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

BRNO 2017

Zadání bakalářské práce/18090/2016/xklobu01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel:	**Klobučníková Dominika**
Obor:	Informační technologie
Téma:	**Obecné gramatiky: Normální formy a jejich aplikace**
	General Grammars: Normal Forms with Applications
Kategorie:	Teoretická informatika

Pokyny:

1. Seznamte se podrobně s obecnými gramatikami a jejich vlastnostmi. 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:

1. Meduna, A.: Automata and Languages, Springer, London, 2000
2. Rozenberg, G. and Salomaa, A. (eds.): Handbook of Formal Languages, Volume 1 through 3, Springer, 1997, ISBN 3-540-60649-1
3. Aho, A. V., Sethi, R., Ullman, J. D.: Compilers : principles, techniques, and tools, Addison-Wesley, 2nd ed., 2007, ISBN: 0321486811

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních 2 bodů zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Vedoucí:	**Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání:	1. listopadu 2016
Datum odevzdání:	17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstract

This thesis deals with the topic of unrestricted grammars, normal forms, and their applications. It focuses on context-sensitive grammars as their special cases. Based on the analysis of the set, an algorithm was designed using the principles of the Cocke-Younger-Kasami algorithm to make a decision of whether an input string is a sentence of a context-sensitive grammar. The final application, which implements this algorithm, works with context-sensitive grammars in the Penttonen normal form.

Abstrakt

Táto práca sa zaoberá problematikou obecných gramatík, normálnych foriem a ich aplikácií. Zameriava sa na kontextové gramatiky ako ich špeciálne prípady. Na základe analýzy tejto množiny bol navrhnutý algoritmus využívajúci princípy Cocke-Younger-Kasami algoritmu za účelom rozhodnutia, či zadaný reťazec je vetou jazyka definovaného kontextovou gramatikou. Výsledná aplikácia implementujúca toto riešenie je navrhnutá pre prácu s kontextovými gramatikami v Penttonenovej normálnej forme.

Keywords

formal languages, grammars, syntax analysis, normal forms, Kuroda, Penttonen, CYK, CKY, Cocke-Younger-Kasami

Klíčová slova

formálne jazyky, gramatiky, syntaktická analýza, normálne formy, Kuroda, Penttonen, CYK, CKY, Cocke-Younger-Kasami

Reference

KLOBUČNÍKOVÁ, Dominika. *General Grammars: Normal Forms with Applications.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. RNDr. Alexander Meduna, CSc.

General Grammars: Normal Forms with Applications

Declaration

Acknowledgements

I would like to express my gratitude to my supervisor Prof. RNDr. Alexander Meduna, CSc. His guidance helped me greatly during my work on this thesis.

Contents

1 Introduction **3**

2 Formal Grammars and Languages **4**
- 2.1 Alphabets and Words . . . 4
- 2.2 Languages . . . 5
- 2.3 Chomsky Hierarchy . . . 6
- 2.4 Unrestricted Grammars . . . 7
  - 2.4.1 Equivalence of Unrestricted Grammars and Turing Machines . . . 8
- 2.5 Context-sensitive Grammars . . . 9
  - 2.5.1 Equivalence of Context-sensitive Grammars and Linear-bounded Automata . . . 10
- 2.6 Normal Forms . . . 11
  - 2.6.1 Chomsky Normal Form . . . 11
  - 2.6.2 Normal Forms of Unrestricted Grammars . . . 11
  - 2.6.3 Normal Forms of Context-sensitive Grammars . . . 11
- 2.7 Construction of a Grammar in a Normal Form . . . 12
  - 2.7.1 Proof of Equivalence . . . 12
  - 2.7.2 Transformation Example . . . 13

3 Syntax Analysis **15**
- 3.1 Derivations and Syntax Tree . . . 15
  - 3.1.1 Grammar Ambiguity . . . 16
- 3.2 Top-Down Analysis . . . 16
  - 3.2.1 `First` and `Follow` Functions . . . 17
- 3.3 Bottom-Up Analysis . . . 17
  - 3.3.1 Handle Pruning . . . 17
  - 3.3.2 Shift-Reduce Parsing . . . 18
- 3.4 Parsing Methods Based on Normal Forms . . . 18
  - 3.4.1 Cocke-Younger-Kasami Algorithm . . . 18

4 Algorithm Design **20**
- 4.1 Extending the Cocke-Younger-Kasami Algorithm . . . 20
- 4.2 Supplementary Sets and Variables . . . 20
  - 4.2.1 Blacklist Queue . . . 21
  - 4.2.2 Ignored Coordinates and Predecessor Set . . . 21
  - 4.2.3 Version Set and Current Version . . . 21
- 4.3 Informal Description . . . 21
- 4.4 Proof of Correctness . . . 23

4.5 Algorithm Complexity 23
4.5.1 Time Complexity 23
4.5.2 Space Complexity 23
4.6 Pseudocode Representation 25
4.7 Example 26

5 Prototype Implementation **28**
5.1 Data Model 28
5.2 Grammar Adapter 28
5.2.1 Grammar Format 29
5.3 Input String Adapter 30
5.4 Version Control 30
5.4.1 Version Configuration 30
5.5 Parsing Core 31
5.5.1 `ParseMatrix` Initialization and Traversal 31
5.6 Rule Application 32
5.6.1 Context-sensitive Rule Application 32
5.6.2 Context-free Rule Application 32
5.7 Testing 33

6 Conclusion **35**

Bibliography **36**

Appendices **37**

A Prototype Information **38**
A.1 Parameters and Return Codes 38
A.2 Code Metrics 38

B Input Files Example **39**

C Contents of the Attached Medium **40**

D Poster **41**

Chapter 1

Introduction

Syntax analysis is an essential component of many disciplines focusing on string evaluation. In practice, it has mostly been limited to unambiguous, context-free grammars, as the less restricted families of grammars had proven both hard to use and too difficult to process. However, as the aforementioned areas of study have been progressing, the context-freeness offered by this group of grammars became no longer satisfactory, and a need to decompose the input string in a way reflecting natural language has arisen. Context-sensitive grammars, and therefore languages generated by them, present a suitable interlink between these groups, because they allow for context, which is an important aspect of human speech. Currently, only few parsing method capable of detecting context exist. However, none of them are capable of processing context-sensitive grammars as a whole. The aim of this thesis is to design and implement such an algorithm, which can deterministically decide whether the input string is a sentence generated by some grammar. To allow for maximum flexibility, the grammar is to be specified by the user.

The algorithm introduced in this thesis is based on the Cocke-Younger-Kasami (CYK) parsing algorithm for context-free grammars, [5] which works with grammars in the Chomsky normal form. To decide a string's correspondence to any grammar, it considers all reduction sequences possibly leading to the desired string. It works in a bottom-up way.

The presented algorithm works with grammars in the Penttonen normal form, as an expansion of the Chomsky normal form. Apart from the way CYK works, it also applies a set of restrictions to prevent inconsistencies of the syntax tree. If any nonterminal was used to apply both a context-sensitive and a context-free rule, it could lead to occurrences of nonterminals that would not appear in the syntax tree under normal conditions. In event of such context conflict, the syntax tree is split into two versions, of which each propagates a different rule. In the event of a failure of one such tree, a substitute is chosen and the process is repeated until the string is either accepted, or there are no other possible syntax trees to be examined and the string is rejected.

This algorithm offers the ability to parse an input string according to any user-specified context-sensitive grammar in the Penttonen normal form. Since every context-sensitive grammar can be converted to an equal grammar in this normal form, the algorithm therefore works for any such grammar. This design always halts even for cyclic and ambiguous grammars. Details of its functionality will be further described in the following chapters.

The algorithm was first presented as a part of Excel@FIT 2017.

Chapter 2

Formal Grammars and Languages

This chapter focuses on the explanation of the formal terms needed to understand the topic this thesis deals with. It starts by explaining basic terms such as alphabet, word, sentential form, language, and regular operations for said languages, and later moves on to the topic of Chomsky hierarchy as the most widely used testing ground for formal grammars. [7] It lists and briefly describes the types of grammars according to the hierarchy, focusing mostly on the unrestricted and context-sensitive grammars, as they are the focus of the thesis. The last part of this chapter focuses on normal forms of formal grammars, and the relationship between any grammar and a grammar in normal form.

2.1 Alphabets and Words

An *alphabet* is a finite nonempty set. An alphabet Σ consists of *symbols* or *letter*. A *string* or a *word* over an alphabet Σ is a finite sequence of length of a zero or more symbols, where any symbols can repeat more than once. A string of zero length is also called an *empty word*, and is denoted by ε. It is defined as follows: [4]

1. ε is a string over Σ,
2. if w is a string over Σ and $x \in \Sigma$, wx is a word over Σ.

The words ab, ba, abb, $aabba$ are words over the alphabet $\Sigma = \{a, b\}$. The set of all words over an alphabet Σ is denoted by Σ^*, which includes the empty word, ε, the set of all nonempty words is denoted by Σ^+. It is formally defined as:

$$\Sigma^+ = \Sigma^* - \{\varepsilon\}. \tag{2.1}$$

These sets are always infinite. [7] This thesis uses the symbol Σ to refer to alphabets. The *length* of a word, w, is the number of symbols the word contains, and is denoted by $|w|$. Each instance of a symbol of Σ is counted once. The length of an empty word, ε, is zero. Formally defined as follows: [4]

1. if $x = \varepsilon$, $|x| = 0$,
2. if $x = u_1u_2 \ldots u_n$, for some $n \geq 1$, and $u_i \in \Sigma$ for $i = 1 \ldots n$, $|x| = n$.

A word, u, is a *subword* of a word, w, if for some words, x_1 and x_2, the word w consists of the concatenation of these words; $w = x_1ux_2$. Any of these words can be empty. If x_1 is an empty word, u is also called the *prefix* of the word w. In case x_2 is a nonempty word, u is a *nontrivial* prefix of w. If x_2 is an empty word, u is the *suffix* of the word w. In case $x - 1$ is not empty, it is called the *nontrivial* suffix of w. This means that all subwords except for the word itself and ε are *nontrivial prefixes* and *suffixes*. Formally defined as follows:

1. $\text{prefix}(w) = \{x_1 : x_1 \text{ is a prefix of } w\}$,
2. $\text{suffix}(w) = \{x_2 : x_2 \text{ is a suffix of } w\}$,
3. $\text{subword}(w) = \{x : x \text{ is a subword of } w\}$.

For every word, w, the following properties always hold: [4]

1. $\text{prefix}(w) \subseteq \text{subword}(w)$,
2. $\text{suffix}(w) \subseteq \text{subword}(w)$,
3. $\{\varepsilon, w\} \subseteq \text{prefix}(w) \cap \text{suffix}(w) \cap \text{subword}(w)$.

2.2 Languages

Any subset Σ^* of an alphabet, Σ, is a *formal language*, L, over Σ; let $L \subseteq \Sigma^*$. [4] By this definition, both $\varnothing$ and $\{\varepsilon\}$ are languages over any alphabet. However, these languages are not equal. Such languages can be finite or infinite. The simplest form of defining a finite language is to list all of its words. However, infinite languages cannot be defined in such way, and have to be specified in other ways, such as grammars and automata. They will be discussed further in section 2.3.

There exist several operations over languages that allow for creation of new languages from existing ones. If treating languages as sets, some operations can be inherited from Boolean algebra: these are the operations of *union*, *intersection* and *complementation*. The operations are defined as follows: [4]

$$L_1 \cup L_2 = \{w : w \in L_1 \text{ or } w \in L_2\}, \tag{2.2}$$
$$L_1 \cap L_2 = \{w : w \in L_1 \text{ and } w \in L_2\}, \tag{2.3}$$
$$L_1 - L_2 = \{w : w \in L_1 \text{ and } w \notin L_2\}. \tag{2.4}$$

Several operations are extended to concern languages. The operation of *concatenation* is defined as follows: [7]

$$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}. \tag{2.5}$$

Power of a language, L^i, is extended to include $L^0 = \{\varepsilon\}$, and therefore it is defined as: [4]

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

The *closure* of a language, L^*, is defined as: [4]

$$L^* = \bigcup_{i=0}^{\infty} L^i \tag{2.6}$$

Analogically, the *positive closure* is defined as:

$$L^+ = \bigcup_{i=1}^{\infty} L^i. \tag{2.7}$$

Based on these definitions, the following properties hold for every language L:

1. $L^* = L^+ \cup \{\varepsilon\}$,
2. $L^+ = LL^* = L^*L$.

2.3 Chomsky Hierarchy

The following section uses the notion of a *rewriting system.* A rewriting system is a set of production rules in the form of $v \longrightarrow w$, where v and W are words of some language, L, representing that an occurrence of v can be replaced by the subword w. A rewriting system can be used to transform words into other words, and thus to transform a language into a different language. An extended version of a rewriting system can be used to define languages [7]. Chomsky hierarchy is based on this principle – it introduces four types of formal languages and grammars:

1. *type 0* – unrestricted grammars,
2. *type 1* – context-sensitive grammars,
3. *type 2* – context-free grammars,
4. *type 3* – regular grammars.

The type 0 languages and grammars are the most general of the four, and are equal to *computability* [7]. The importance of this hierarchy lies in the fact that the generality of a type decreases with the increasing type. Nowadays, Chomsky hierarchy is not the only existing testing ground for language and grammar types, but it maintains its importance.

Each of the families of Chomsky hierarchy can be accepted by an automaton of power at least equal to that of the listed type:

1. *type 0* – Turing machine,
2. *type 1* – linear-bounded automaton,
3. *type 2* – pushdown automaton
4. *type 3* – finite state automaton.

"Rewriting system is an ordered pair, $M = (\Sigma, R)$, where Σ is an alphabet and R is a finite relation on Σ^*. Σ is called the total alphabet of M or, simply, M's alphabet. A member of R is called a rule of M, so R is referred to as M's set of rules." [5]

"A phrase-structure grammar or a type 0 Chomsky grammar is a construct $G = (N, T, S, P)$, N and T are disjoint alphabets, $S \in N$, and P is a finite set of ordered pairs (u, v), where u, $v \in (N \cup T)^*$." [7]

"Elements in N are referred to as nonterminals. T is the terminal alphabet. S is the start symbol and P is the set of productions or rewriting rules. Productions (u, v) are written $u \longrightarrow v$. The alphabet of G is $V = N \cup T$. The direct derivation relation induced by G is a binary relation between words over V, denoted $\Longrightarrow_G$, and defined as:

$$\alpha \Longrightarrow_G \beta \text{ if } \alpha = xuy, \beta = xvy \text{ and } (u \longrightarrow v) \in P, \tag{2.8}$$

where $\alpha, \beta, x, y \in V^*$." [7]

"The derivation relation induced by G, denoted $\Longrightarrow_G^*$, is the reflexive and transitive closure of the relation $\Longrightarrow_G$." [7]

"The language generated by G, denoted $L(G)$, is:

$$L(G) = \{w \mid w \in T^*, S \Longrightarrow_G^* w\}. \tag{2.9}$$

Note that $L(G)$ is a language over T, $(L(G) \subseteq T^*)$." [7]

Any string that is a member of $(\Sigma \cup N)^*$ is a sentential form of G. If such sentential form is a member of Σ^*, it is called a sentence.

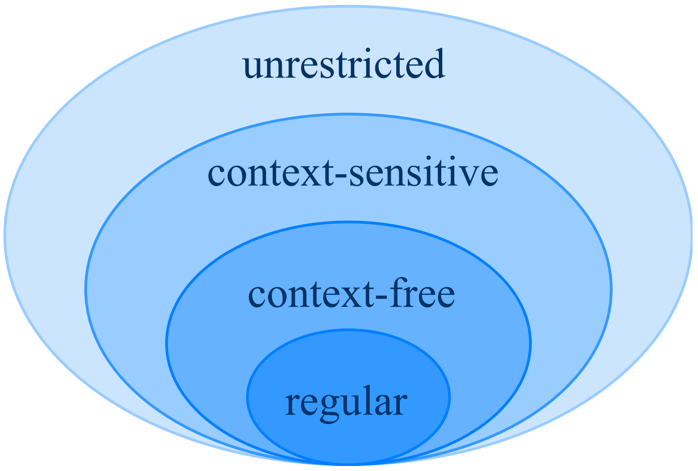


Figure 2.1: Chomsky hierarchy; this thesis uses it as its base of classification.

The closure properties of families of languages in Chomsky hierarchy are described by table 2.1.

Table 2.1: Closure properties of types of grammars as defined by Chomsky hierarchy [7] (pg. 30). UR stands for unrestricted, CS for context-sensitive, CF for context-free and RE for regular grammars.

	UR	CS	CF	RE
Union	Yes	Yes	Yes	Yes
Intersection	Yes	Yes	No	Yes
Complement	No	Yes	No	Yes
Kleene *	Yes	Yes	Yes	Yes

2.4 Unrestricted Grammars

Unrestricted grammars are the main type of the Chomsky hierarchy, as described at the beginning of this chapter. This set properly contains the set of context-sensitive grammars, which properly contains the set of context-free and regular grammars as depicted in figure 2.1.

An unrestricted grammar is an ordered quadruple

$$G = (N, T, P, S) \tag{2.10}$$

where:

- N is the alphabet of nonterminals,
- T is the alphabet of terminals, where $N \cap T = \varnothing$,
- $P \subseteq (N \cup T)^+ \times (N \cup T)^*$, which is a finite set of productions,
- S is the *start symbol* or the axiom. [4]

The members of P are referred to as productions, p, and are usually written as $u \longrightarrow v$, where u is a nonempty word that represents the left-hand side of the production rule, $lhs(p)$, and v represents the right-hand side of the rule, $rhs(p)$. Other than the left-hand side of a rule having to consist a nonempty word, the productions of grammars of this family are not restricted in any way. [4]

2.4.1 Equivalence of Unrestricted Grammars and Turing Machines

According to Church's Thesis [7], any unrestricted grammar, G, represents a procedure. Any procedure can be accepted by a Turing machine, and therefore every unrestricted grammar, G, is equal to some Turing machine, M. This can be proven by demonstrating conversion of both a grammar into a Turing machine, and a Turing machine into a grammar. Therefore, the family of unrestricted grammars is:

$$UR = \{L \mid \exists M, \text{Turing machine}, \ L = L(M)\}. \text{ [7]} \tag{2.11}$$

A Turing machine, M, is an ordered system:

$$\text{“}M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \tag{2.12}$$

where Q is a finite set of states, Σ is the input alphabet, Γ is the tape alphabet, $\Gamma \cup Q = \varnothing$ and $\Sigma \subset \Gamma$, $q_0 \in \ Q$ is the initial state, $B \in \Gamma - \Sigma$ is the blank symbol, $F \subseteq Q$ is the set of final states, δ is the transition function,

$$\delta : Q \times \Sigma \longrightarrow \mathcal{P}(Q \times \Sigma \times \{L, R\}).\text{''[7]} \tag{2.13}$$

Let M be a Turing machine, where $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, and an input word, $w\Sigma^*$. Given these, one of the following states will occur:

1. M halts after a finite number of moves in a state, $q \in Q$. If $q \in F$, the input word, w, is accepted.
2. M does not halt – in this case, the input word, w, is rejected.

A Turing machine that halts for every input represents the notion of an algorithm – this concept was introduced by the Church-Turing thesis. [7] However, given a Turing machine, M, and an input string, W, it is impossible to decide whether the machine M will halt when run with the input. [6]

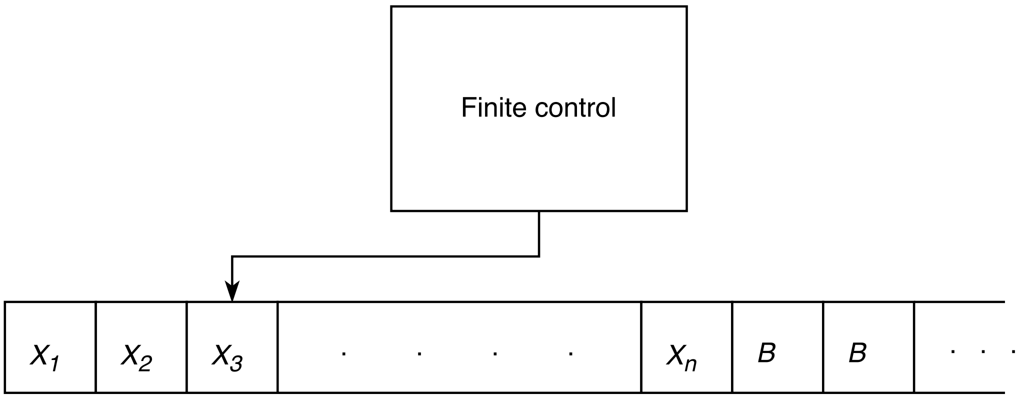


Figure 2.2: Power of a Turing machine is great enough to accept an unrestricted grammar.

Let G be an unrestricted grammar. Consider a string, w, where $w \in T^*$. Construct a three-tape Turing machine, M. First, M writes w on its first tape. Afterwards, it starts the computation by writing all production rules in P on the second tape. Then it writes the start symbol, S on the third tape. This tape records the sentential form of G during the current step of the computation. "From this point on, M iterates the following four-step computational cycle:

1. Nondeterministically select a position, i, in the current sentential form on the third tape.
2. Nondeterministically select a production, p, on the second tape.
3. If $lhs(p)$ appears on the third tape at positions i through $i + |lhs(p)| - 1$, replace $lhs(p)$ with $rhs(p)$; otherwise, **reject**.
4. If the first tape and the third tape coincide, **accept**; otherwise go to step 1." [4]

The algorithm of conversion of any Turing machine into an equal unrestricted grammar is considerably more complex than its counterpart. Please refer to [7] (pg. 172) and [4] (pg. 715 – 716) for its full description.

Based on these proofs, any language, L, is equal to a Turing machine, M, if it is generated by an unrestricted grammar. Formally,

$$L = L(M) \text{ if } L = L(G),\ G \in UR. \tag{2.14}$$

2.5 Context-sensitive Grammars

This section focuses on the family of context-sensitive grammars as a subset of unrestricted grammars. It discusses the differences between these sets, and the implied differences between the automata equal to these grammars.

"A context-sensitive (type 1) grammar is a type 0 grammar $G = (N, T, S, P)$ such that each production in P is of the form $\alpha X \beta \longrightarrow \alpha u \beta$, where $X \in N$, $\alpha, \beta, u \in (N \cup T)^*$, $u \neq \varepsilon$. In addition, P may contain the production $S \longrightarrow \varepsilon$ and in this case S does not occur on the right-hand side of any production of P." [7]

A language generated by a context-sensitive grammar is defined in the same way as a language generated by an unrestricted grammar:

$$CS = \{L \mid \exists G \text{ context-sensitive grammar such that } L = L(G)\}. \text{ [7]} \tag{2.15}$$

"A length-increasing (monotonous) grammar is a type 0 grammar $G = (N, T, S, P)$ such that for each production $(u \longrightarrow v) \in P$, $|u| \leq |v|\cdot$ In addition, P may contain the production $S \longrightarrow \varepsilon$, and in this case S does not occur on the right side of any production from P." [7]

The generative power of the set of context-sensitive grammars is equal to that of the set of monotonous grammars, thus they can be used interchangeably [7].

2.5.1 Equivalence of Context-sensitive Grammars and Linear-bounded Automata

Linear-bounded automata are a special subtype of Turing machine, whose input tape contains read-only markers that symbolize the beginning and the end of the tape. These symbols are usually marked as # and \$ respectively. This type of automata accepts exactly the family of context-sensitive languages. [7]

Other than this, their behavior is identical to that of normal Turing machines as described in section 2.4.1.

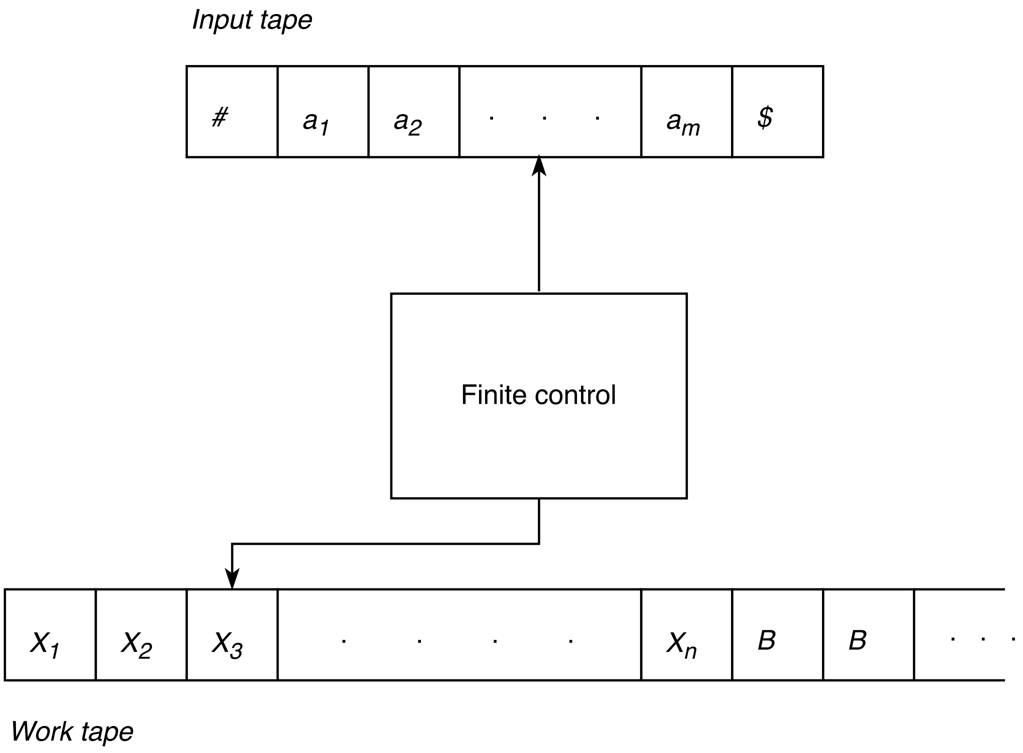


Figure 2.3: Linear-bound automaton is the special type of Turing machine that accepts the family of monotonous grammars.

2.6 Normal Forms

This section focuses on normal forms; they are crucial in proving the properties of grammars and their applications. [7] [4] The section briefly mentions the Chomsky normal form of context-free grammars, as it is the most commonly used form. This form serves as the base for the Kuroda and Penttonen normal forms for unrestricted grammars, which will be discussed further in the later parts of the section. Finally, an algorithm of conversion of any unrestricted grammar into a grammar in a normal form is presented.

2.6.1 Chomsky Normal Form

Let $G = (N, T, P, S)$ be a context-free grammar. G is in *weak* Chomsky normal form, if each rule that contains terminals has a right-hand side that is a member of $T \cup \{\varepsilon\}$, and each rule that contains only nonterminals has a right-hand side that is a member of $(N \cup T)^*$. It is in Chomsky normal form, if each rule of the latter type has a right-hand side whose length is equal to two. [7] Formally, a grammar, G, is in Chomsky normal form, if the right-hand side of production rule, $p \in P$, satisfies $rhs(p) \in (T \cup N^2)$. [4] "By this definition, a context-free grammar in Chomsky normal form has productions that satisfy these two forms:

1. $A \longrightarrow BC$, where $B, C \in N$,
2. $A \longrightarrow a$, where $a \in T$." [4]

2.6.2 Normal Forms of Unrestricted Grammars

This section normalizes production rules of an unrestricted grammar into the Kuroda normal form or its special case, the Penttonen normal form. These normal forms are similar to the Chomsky normal form presented in the previous section, and they can be used as its direct extension for unrestricted grammars.

Let $G = (N, T, P, S)$ be an unrestricted grammar. G is in Kuroda normal forms, if each of its production rules satisfies one of the following forms:

1. $AB \longrightarrow CD$, where $A, B, C, D \in N$,
2. $A \longrightarrow BC$, where $A, B, C \in N$,
3. $A \longrightarrow a$, where $A \in N, a \in T$,
4. $A \longrightarrow \varepsilon$, where $A \in N$. [4]

If every production rule in the first form satisfies the condition $A = C$, the grammar is in the one-sided normal form, introduced by Penttonen. [4] [7]

The algorithm of conversion of an unrestricted grammar to conform to a normal form is described in section 2.7.

2.6.3 Normal Forms of Context-sensitive Grammars

This section focuses on the Kuroda normal form and the Penttonen normal form of unrestricted grammars. It adapts them to the context-sensitive grammars – these are the grammars that do not contain the epsilon rules.

A grammar, G, is in Kuroda normal form, if each of its production rules, $p \in_G P$, is in one of the following forms:

1. $AB \longrightarrow CD$, where $A, B, C, D \in {}_GN$,
2. $A \longrightarrow CD$, where $A, C, D \in {}_GN$,
3. $A \longrightarrow a$, where $A \in {}_GN$ and $a \in {}_GT$. [7] [5]

For every grammar, G, there exists an equivalent grammar, G_{KNF}, which is in Kuroda normal form. The proof of this theorem is analogical to that described in section 2.7.

A grammar, G, is in Penttonen normal form, if each of its production rules is in one of the following forms:

1. $AB \longrightarrow AC$, where $A, B, C \in {}_GN$,
2. $A \longrightarrow CD$, where $A, C, D \in {}_GN$,
3. $A \longrightarrow a$, where $A \in {}_GN$ and $a \in {}_GT$. [7] [5]

Penttonen normal form is the special case of Kuroda normal form, and therefore for any context-sensitive grammar, G, there exists an effectively equal grammar, G_{PNF}, which is in Penttonen normal form.

2.7 Construction of a Grammar in a Normal Form

This section focuses on the proof that for any unrestricted grammar, $G = (N, T, P, S)$, there exists an equal grammar, $G_{KNF} = (N_{KNF}, T, P_{KNF}, S)$, in Kuroda normal form. It is proven by the algorithm of conversion presented in the following section. [4]

2.7.1 Proof of Equivalence

The algorithm begins by moving all nonterminals from N to N_{KNF}. Then it moves all productions, $p \in P$, that satisfy the Kuroda normal form, to P_{KNF}. Once these initial steps have been taken, it continues as follows:

1. In every rule, that contains a terminal, $a \in T$, replace the terminal by a new nonterminal, $A \in N_{KNF}$, and add the production rule $A \longrightarrow a$ to P_{KNF}.
2. Replace every production rule, $p \in P$, that is in form $A_1 \ldots A_n \longrightarrow B_1 \ldots B_m$, where $0 \leq n < m$, and therefore the left-hand side is at least as long as the right-hand side, with $A_1 \ldots A_n \longrightarrow B_1 \ldots B_m C_{m+1} \ldots C_n$. Nonterminals $C_{m+1} \ldots C_n$ represent occurrences of a new nonterminal, C, which finally derives into ε; $C \longrightarrow \varepsilon$.

   After this move, every rule, $p \in P$ has the right-hand side at least as long as the left-hand side; move every rule in P that satisfies the Kuroda normal form into P_{KNF}.
3. Replace any rule, $p \in P$, in form of $A \longrightarrow B$ by $A \longrightarrow BC$ in P_{KNF}, where C is a new nonterminal in N_{KNF}, and add the rule $C \longrightarrow \varepsilon$ to P_{KNF}.
4. For every context-free rule in the form $A \longrightarrow B_1 B_2 \ldots B_n$, where $n \geq 3$, add the following rules to P_{KNF}:

$$\begin{aligned} A &\longrightarrow B_1 \langle B_2 \ldots B_n \rangle \\ \langle B_2 \ldots B_n \rangle &\longrightarrow B_2 \langle B_3 \ldots B_n \rangle \\ &\vdots \\ \langle B_{n-1} \ldots B_n \rangle &\longrightarrow B_{n-1} B_n. \end{aligned} \tag{2.16}$$

Add the nonterminals $B_1\langle B_2 \dots B_n\rangle$ to $\langle B_{n-1} \dots B_n\rangle$ to N_{KNF}. Remove the original rule, $A \longrightarrow B_1\langle B_2 \dots B_n\rangle$ from P; thanks to this, every rule $p \in P$ satisfies $lhs(p) \in N \cup N^2$ and $rhs(p) \in N \cup N^2 \cup N^3$.

5. For every context-sensitive rule $A_1A_2 \dots A_n \longrightarrow B_1B_2 \dots B_n$, where $2 \leq n < m$, add a new rule, $A_1A_2 \longrightarrow B_1C$ to P_{KNF} and add the nonterminal C to N_{KNF}. If $|B2 \dots B_n| \leq 2$, add the rule $A_3C \longrightarrow B_2 \dots B_n$ to P_{KNF}. Otherwise, place the rule to P and repeat this step until $P = \varnothing$.

The transformed grammar satisfies the Kuroda normal form. Also, $L(G) = L(G_{KNF})$, and therefore the theorem holds. [4]

This algorithm can be applied to context-sensitive grammars in an analogical way. However, a change must be made in step 3, as the context-sensitive grammars exclude epsilon rules in most cases. In such a case, instead of adding $C \longrightarrow \varepsilon$ to P_{KNF}, replace $A \longrightarrow B$ with $A \longrightarrow BC$, where C is every nonterminal that can follow B in a production rule, and add this rule to P_{KNF}.

2.7.2 Transformation Example

Let G be an unrestricted grammar, where $G = (\{S, A, B, C, E\}, \{a, e\}, \{S \longrightarrow AAaBC, Aa \longrightarrow eCA, ABC \longrightarrow CE, E \longrightarrow ea, CC \longrightarrow a, Ae \longrightarrow e\}, S)$. This section describes the construction of a grammar, G_{KNF}, in Kuroda normal form, where $G_{KNF} = (N_{KNF}, T, P_{KNF}, S)$, such that $L(G) = L(G_{KNF})$.

Initialize the new grammar, G_{KNF} to $G = (\{S, A, B, C, E\}, \{a, e\}, \varnothing, S\}$. Because no rules of the original grammar satisfy Kuroda normal form, the set P_{KNF} is empty.

First, replace occurrences of all terminals, $a \in T$ with corresponding nonterminals, and add the rules describing these transformations to the set P_{KNF}. After this step, the sets P_{KNF} and N_{KNF} are defined as follows:

$$\begin{aligned} N_{KNF} &= \{a', e', E\}, \\ P_{KNF} &= \{a' \longrightarrow a, e' \longrightarrow e, E \longrightarrow e'a'\}. \end{aligned} \tag{2.17}$$

Subsequently, in each rule, $p \in P$, where the left-hand side is longer than the right-hand side, extend the right-hand side by $|lhs(p)| - |rhs(p)|$ instances of a new nonterminal, X. Add the nonterminal to N_{KNF}, as well as the rule $X \longrightarrow \varepsilon$ to P_{KNF}.

$$\begin{aligned} P &= \{S \longrightarrow AAaBC, Aa' \longrightarrow e'CA, ABC \longrightarrow CEX\}, \\ N_{KNF} &= \{S, a', e', E, X, C, A, B\}, \\ P_{KNF} &= \{a' \longrightarrow a, e' \longrightarrow e, E \longrightarrow e'a', X \longrightarrow \varepsilon, CC \longrightarrow a'X, Ae' \longrightarrow e'X\}. \end{aligned} \tag{2.18}$$

In this step, transform all context-free rules, $p \in P$, whose right-hand side is longer than two. Keep replacing substrings of the right-hand side by auxiliary nonterminals until the set of production rules satisfies the Kuroda normal form.

$$\begin{aligned} S &\longrightarrow A\langle Aa'BC\rangle \\ \langle Aa'BC\rangle &\longrightarrow A\langle a'BC\rangle \\ \langle a'BC\rangle &\longrightarrow a'\langle BC\rangle \\ \langle BC\rangle &\longrightarrow BC \end{aligned} \tag{2.19}$$

After this step, the examined sets are defined as follows:

$$\begin{aligned}P =& \{ABC \ \longrightarrow \ CEX\},\\ N_{KNF} =& \{S, a^{'}, e^{'}, E, X, C, A, B, \langle Aa^{'}BC\rangle, \langle a^{'}BC\rangle, \langle BC\rangle\},\\ P_{KNF} =& \{a^{'} \longrightarrow a, e^{'} \longrightarrow e, E \longrightarrow e^{'}a^{'}, X \longrightarrow \varepsilon, CC \ \longrightarrow \ a^{'}X, Ae^{'} \ \longrightarrow \ e^{'}X,\\ & S \longrightarrow A\langle Aa^{'}BC\rangle, \langle Aa^{'}BC\rangle \longrightarrow A\langle a^{'}BC\rangle, \langle a^{'}BC\rangle \longrightarrow a^{'}\langle BC\rangle,\\ & \langle BC\rangle \longrightarrow BC\}\end{aligned} \tag{2.20}$$

In the final step, introduce pairs of nonterminals from the beginnings of left-hand sides of context-sensitive rules, such that they derive into auxiliary nonterminals that emulate the original rules. The right-hand side of both examined production rules is of the length three, and therefore a rule preventing a loss of the left-hand side nonterminal is constructed.

$$\begin{aligned}Aa^{'} &\longrightarrow e^{'}Y\\ Y &\longrightarrow CA\\ AB &\longrightarrow CZ\\ ZC &\longrightarrow EX\end{aligned} \tag{2.21}$$

Assuming rules satisfying the Kuroda normal form have been moved to P_{KNF} after each step, the sets are defined as follows:

$$\begin{aligned}P =& \varnothing,\\ N_{KNF} =& \{S, a^{'}, e^{'}, E, X, C, A, B, \langle Aa^{'}BC\rangle, \langle a^{'}BC\rangle, \langle BC\rangle, Y, Z\},\\ P_{KNF} =& \{a^{'} \longrightarrow a, e^{'} \longrightarrow e, E \longrightarrow e^{'}a^{'}, X \longrightarrow \varepsilon, CC \ \longrightarrow \ a^{'}X, Ae^{'} \ \longrightarrow \ e^{'}X,\\ & S \longrightarrow A\langle Aa^{'}BC\rangle, \langle Aa^{'}BC\rangle \longrightarrow A\langle a^{'}BC\rangle, \langle a^{'}BC\rangle \longrightarrow a^{'}\langle BC\rangle,\\ & \langle BC\rangle \longrightarrow BC, Aa^{'} \longrightarrow e^{'}Y, Y \longrightarrow CA, AB \longrightarrow CZ, ZC \longrightarrow EX\}.\end{aligned} \tag{2.22}$$

The grammar constructed by this algorithm satisfies the Kuroda normal form, and it is equal to the original grammar; $L(G) = L(G_{KNF})$.

Chapter 3

Syntax Analysis

Syntax analysis, or often called parsing, is one of the most important phases of the code compilation. It takes input data often in the form of tokens previously generated by the lexical analyzer, also called the scanner, and generates an output in the form of a syntax tree.

A lexer is a finite state automaton that processes an input string, breaks it down into lexemes, and outputs corresponding tokens based on a predefined set of transition rules. In many cases, the term of a 'token' is interchangeable with 'terminal'. [1]

Syntax analysis determines the syntactic structure of the input data structure based on grammar rules, and detects possible grammar errors. These rules can be defined in several ways, such as natural speech, mathematical formula, or a grammar, usually in some normal form, as described in section 2.3. [4] The output of this phase is a syntactic structure, often a syntax tree or other structure capable of holding hierarchical data; this structure is constructed by selection of suitable grammar rules and their application.

This phase is usually followed by semantic analysis. It takes an input in the form of the data structure previously constructed during syntax analysis, and checks if it satisfies the semantic conventions of the examined language, such as data type compatibility and variable initialization. This phase is important especially when parsing context-free languages, as these are unable to maintain any context-related data. However, context-sensitive languages are suited to join these two phases because of their ability to process context-sensitive rules, as described in section 2.4. [1]

There exist three main types of syntax analysis methods – these are the top-down parsers, the bottom-up parsers, and the universal parsers. "Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar" [1]; section 3.4.1 describes the former in a more detailed way. Common methods used in compilers can be generally classified as either top-down or bottom-up. All methods described in the following sections use context-free grammars.

3.1 Derivations and Syntax Tree

A syntax tree, also called a parse tree, can be constructed by treating the production of a grammar as rewriting rules. It begins by rewriting the start symbol by the right-hand side of one of its derivations. This thesis uses the notion of derivations and sentential forms as presented in section 2.3.

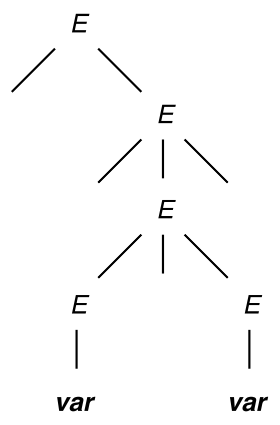


Figure 3.1: Syntax tree of the expression '$-(var + var)$' based on the grammar $E \longrightarrow E + E \mid -E \mid (E) \mid var$.

A sentential form can consist of a non-negative number of terminals and nonterminals. Parsers generally choose a nonterminal to derive in the next step of parse tree construction in one of the following ways:

1. "In leftmost derivations, the leftmost nonterminal in each sentential is always chosen. If $\alpha \Longrightarrow \beta$ is a step in which the leftmost nonterminal in α is replaced, we write $\alpha \underset{lm}{\Longrightarrow} \beta$.
2. In rightmost derivations, the rightmost nonterminal is always chosen; we write $\alpha \underset{rm}{\Longrightarrow} \beta$ in this case" [1]

The parse tree is a tree, whose each node represents the application of a rewriting rule. The node itself is called after the name of the nonterminal on the left-hand side of the applied rule. It ignores the order in which the derivations were applied, and therefore there can exist many derivations that produce the same syntax tree. The leaves of this tree consist of terminals and nonterminals that form a sentential form. [1]

3.1.1 Grammar Ambiguity

A grammar that produces more than a single parse tree is called ambiguous. Most parsers require an unambiguous grammar to be able to deterministically determine which production is supposed to be applied next.

Parsers that are able to use these rules use a set of disambiguating rules that eliminate unsuitable trees.

3.2 Top-Down Analysis

A top-down parser verifies the syntactical correctness of an input string by building its parse tree. It starts from the root, which carries the start symbol, and applies suitable productions depth-first. Equivalently, it works by creating the left-most derivation of the input string.

At each step of such a derivation, the parser needs to select the rule to be applied for the currently deepest nonterminal, E. Once the production is selected, the parser tries to find the path that leads to the input string. This is often achieved by using recursive-descent parsers that use backtracking to choose the production that properly matches the stream of tokens produced by the scanner. [1]

Recursive-descent parsers often use LL grammar-based parsers. These parsers use a predictive set of tokens to deterministically choose a production in every step of the derivation. [5]

3.2.1 `First` and `Follow` Functions

Construction of both top-down and bottom-up parsers is aided by the `First` and `Follow` functions defined for any grammar, G. These sets help to determine which production to choose based on the next token.

`First`(w), where w is a word consisting of any number of syntactical symbols, is the set of all terminals that can be at the beginning of words derived from w. If $w \Longrightarrow^* \varepsilon$, ε is also a member of `First`(w). The set is used to choose between possible productions, as a symbol, a can be in at most of these sets.

`Follow`(A), where A is a nonterminal, is a set of terminals that can appear immediately after A in a sentential form of G. Moreover, in case A can be the last nonterminal of some sentential form, the appointed end-marker, \$ is also a member of `Follow`(A).

$LL(k)$ parsers are a family of predictive parsers that do not need backtracking. These parsers scan the input string from left to right and always derive the leftmost nonterminal; k is the number of lookahead symbols used to determine the rule at every step of the parsing process. This class covers most programming languages, however, a grammar must not be ambiguous or left-recursive to be usable by a $LL(k)$ parser. [1]

3.3 Bottom-Up Analysis

A bottom-up parse represents the construction of a parse tree beginning from its leaves, the input string, and working up towards the root, which represents the start symbol. In terms of derivation, bottom-up parsing is equal to the rightmost derivation, as described in section 3.1. Alternatively, bottom-up parsing can be viewed as a process of reducing the input string into the start symbol. At each step of this process – a reduction–a substring matching the right hand side of a reduction is reduced into the left-hand side of the said production. [1]

Each step of a bottom-up parsing process represents either a shift, or a reduction. These operations are used in a general parsing method called shift-reduce parsing – it is used mainly by LR parsers; these, however, will not be described into further detail, as they are difficult to construct without the aid of parser generators.

3.3.1 Handle Pruning

"Bottom-up parsing during a left-to-right scan of the input constructs a right- most derivation in reverse." [1] A handle is a substring of some sentential form that matches the right-hand side of a production, and can be reduced to construct a step of rightmost derivation in reverse. In case the parsed grammar is ambiguous, there might exist more than one handle. In the opposite case, every right-sentential form has a single handle. [1]

3.3.2 Shift-Reduce Parsing

"Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed." [1] The handle is always on the top of the stack before being recognized as a handle.

During a left-to-right scan of the input string, the parser shifts zero of more symbols onto its stack. When it is ready to reduce a string of symbols on the stack, it reduces the string into the left-hand side of the corresponding production. This is repeated until either an error is detected, or the input string is empty and the stack contains the start symbol – in such case, the parser accepts the string. [1]

"While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine." [1]

There exist grammars even in the family of context-free grammars that cannot be parsed by a shift-reduce parser. In case a parser of such grammar knows the entire stack and k characters of the input string, and it is unable to determine whether to shift or reduce – a shift/reduce conflict, or what reduction to make – reduce/reduce conflict. None of these grammars are in the LR grammar class, which is often implemented using this kind of parser.

Another possible source of conflicts is the situation when a handle has been recognized on the stack, but the contents of both the stack and the input string are not sufficient to decide which production to apply; this could be solved by letting the lexer access semantic data.

3.4 Parsing Methods Based on Normal Forms

Parsing methods described in the previous sections are applicable only for certain subsets of context-free grammars. There exist universal parsing algorithms that work any grammar, such as Earley algorithm [1] and Cocke-Younger-Kasami algorithm. An example of this type of parsing algorithms is the class of parsing methods that work with grammars in normal forms. [5] Any grammar can be converted into an equal grammar in a normal form, [4] and therefore parsing methods based on them are universal. [5]

3.4.1 Cocke-Younger-Kasami Algorithm

Let G be a context-free grammar in Chomsky normal form, as defined in section 2.6.1, and an input string, $w = a_1a_2 \dots a_n$, where $a_i \in T$, and $1 \leq i \leq n$. The Cocke-Younger-Kasami algorithm is used to determine whether w is a sentence of grammar G in a bottom-up way.

The algorithm works by constructing sets of nonterminals, $CYK[i, j]$, where $1 \leq i \leq j \leq n$, where a nonterminal, A, can be a member of $CYK[i, j]$ only if $A \Longrightarrow^* a_i \dots a_j$, or in other words, if it is possible to reduce $a_i \dots a_j$ into A. As a special case, w is a sentence of G, if the start symbol, $S \in CYK[1,n]$.

The algorithm initializes these sets by adding a nonterminal, A, to set $CYK[i,i]$ if there exists a production in the form of $A \longrightarrow a_i$, where $1 \leq i \leq n$. Then, anytime there exist nonterminals $B \in CYK[i,j]$, $C \in CYK[j+1,k]$ such that there exists a production $A \longrightarrow BC$, the nonterminal A is added to the set $CYK[i,k]$. This is because $B \Longrightarrow^* a_i \dots a_j$ and $C \Longrightarrow^* a_{j+1} \dots a_k$, which implies that $A \Longrightarrow^* BC$, because

$$\begin{aligned} A &\Longrightarrow BC \\ &\Longrightarrow a_i \dots a_j C \\ &\Longrightarrow a_i \dots a_j a_{j+1} \dots a_k.[5] \end{aligned} \tag{3.1}$$

Once no set can be extended in this way, the algorithm checks whether $S \in CYK[1,n]$ to verify that $S \Longrightarrow^* a_1 \dots a_n$, and therefore w is a sentence of the grammar G. If so, the string is accepted; otherwise, the string is not a sentence, and the algorithm announces its rejection. In the worst case scenario, the time complexity reaches $O(n^3|G|)$, where n is the length of the input string, and $|G|$ is the size of the grammar. [3]

The following algorithm presents the pseudo-code of the Cocke-Younger-Kasami parsing method; taken from [5] (pg. 120 – 121).

Algorithm 1 Cocke-Younger-Kasami Parsing Algorithm

```
Input:   a grammar, G = $(N, T, P, S)$, in the Chomsky normal form;
         $w = a_1 a_2 \dots a_n$ 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: for $i = 1$ to $n$ do
 3:     if $A \longrightarrow a_i \in P$ then
 4:         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;
```

Chapter 4

Algorithm Design

The aim of this chapter is to design an algorithm capable of parsing context-sensitive grammars. Most parsing methods do not deal with context, and if so, they merely highlight places where context might be important. [2]

The algorithm presented in this chapter is based on the Cocke-Younger-Kasami parsing algorithm for context-free grammars, which was described in section 3.4.1. However, the original algorithm works with grammars in Chomsky normal form, which exists only for context-free grammars. As its extension, the presented algorithm works with Penttonen normal form, which allows context-sensitive rules in the form $AB \longrightarrow AC$, as well as all forms of rules presented by Chomsky normal form.

4.1 Extending the Cocke-Younger-Kasami Algorithm

The Cocke-Younger-Kasami algorithm works by constructing sets of nonterminals that could appear in different nodes of the syntax tree. This can possibly cause context conflict; in the presented version of the algorithm, applying a context-sensitive rule means replacing a nonterminal by a different nonterminal at the same position.

In a production $AB \longrightarrow AC$, the nonterminal C would be added to the set at coordinates of the nonterminal B. If the nonterminal C was used to reduce a production with its closest right neighbour, the syntax tree would become inconsistent, which could lead to the acceptance of a sentential form that was not a sentence. However, if the algorithm ignored all nonterminals reduced from context-sensitive rules, its power would decrease to that of the original Cocke-Younger-Kasami algorithm, which would lead to rejecting of all sentential forms whose reduction lead to at least one context-sensitive nonterminal in their syntax tree. The extended version of the algorithm uses several additional sets and variables to handle these collisions. These will be described in the following sections.

4.2 Supplementary Sets and Variables

This section describes the additional sets and variables needed to handle context as an extension of the Cocke-Younger-Kasami algorithm. It explains the purpose of every such variable, and describes the way its value is acquired.

4.2.1 Blacklist Queue

The presented algorithm works with an upper diagonal matrix, which represents its syntax tree. A set of nonterminals is located at each of its non-null entries. The algorithm iterates through these sets in a deterministic way, and therefore can appoint each entry's closest right neighbour.

Every time a context-sensitive production has been reduced for a specific pair of sets, the algorithm checks if the modified set knows its right neighbour. If not, its coordinates are added to the *Blacklist Queue*. This set holds coordinates of all sets that might cause a context conflict, because they contain nonterminals reduced from context-sensitive productions.

4.2.2 Ignored Coordinates and Predecessor Set

Next time the set is in the role of the first nonterminal of the examined pair, its coordinates are removed from the *Blacklist Queue*, and its *ignored coordinates* are set to those of the other set of the pair. When examining a pair of sets where the second is ignored by the first, the context-sensitive nonterminals of the former set are excluded from the comparison; in case a rule is to be reduced, the syntax tree splits into two versions – one that respects the context-sensitive rules of the former set, but ignores any rules reduced in this step, and the latter, that does the opposite. This ensures that all versions of the syntax tree are investigated and the syntax tree remains consistent during the parsing.

When splitting the syntax tree into two, the second version removes all context-sensitive nonterminals from the set at the coordinates of the former examined set. For each nonterminal that is deleted, it deletes all its related nonterminals. To be able to do this, each nonterminal has its own *Predecessor* set, which contains references to all its predecessors – since the algorithm works in a bottom-up way, these are the nonterminals that have been reduced from the original nonterminal.

4.2.3 Version Set and Current Version

The *Version Set* is a global set used during the entire run of the algorithm. It contains all versions of the parse matrix the algorithm has generated so far, of which are mutually exclusive. At any time there is a *Current Version*, which is the version of the parse matrix the algorithm is currently analyzing and modifying.

At the beginning of the algorithm, there exists only a single matrix initialized using $A \longrightarrow a$ productions. Every time a context conflict is detected, a new version is initialized; at the end of the *Current Version* analysis, the new version containing resolving all found conflicts is added to the *Blacklist Queue.*

Once no set of the *Current Version* can be extended, success of the version is examined in a way identical to the Cocke-Younger-Kasami algorithm. If the version fails, it is removed from the *Version Set* and a new *Current Version* is appointed. If there are no more versions to appoint, the algorithm rejects the input string.

4.3 Informal Description

Given a grammar, $G = (N, T, P, S)$, in Penttonen normal form, and a string $w = a_1 a_2 \dots a_n$ where $a_i \in T$, and $1 \leq i \leq n$, for some $n \geq 1$, this algorithm decides, whether w is a sentence of $L(G)$ in a bottom-up way. For its work, it uses an upper triangular matrix of sets, CV – *CurrentVersion*, where $CV[i, j]$, $1 \leq i \leq j \leq n$.

Every member of $CV[i, j]$ has its own BLQ set – every *BlacklistQueue* contains coordinates of all entries of the corresponding matrix that contain context-sensitive nonterminals, but whose ignored coordinates have not been set yet. As mentioned above, every entry of the CV matrix contains a set of nonterminals that are likely to occur in this node of the parse tree. For every such nonterminal A, there exists a *Predecessor* set, P_A, containing references to all of its preceding nonterminals. Any nonterminal can be marked as context-sensitive, which indicates that it originated from a context-sensitive production.

The algorithm starts the parsing process by scanning the input string, adding nonterminal A to $CV[i, i]$, if $A \longrightarrow a_i \in {}_GR$. For every such nonterminal, an empty set of its predecessors is constructed. Afterwards, the algorithm iterates through the matrix constructing the sets $CV[i, j]$. For every pair $CV[i, j]$ and $CV[j + 1, k]$, it checks whether $CV[i, j]$ is a member of BLQ, which indicates that the set contains context-sensitive nonterminals, but its ignored coordinates have not been appointed yet. These are the coordinates of the first right neighbour of $CV[i, j]$, whose nonterminals must not be used in combination with the context-sensitive nonterminals of the examined set – this is to ensure the consistency of the parse tree. If the ignored coordinates have not been set yet, those of $CV[j + 1, k]$ are used, and $CV[i, j]$ is removed from BLQ.

The behaviour of the following step is dependent on whether the ignored coordinates of $CV[i, j]$ are equal to $CV[j + 1, k]$. If not, the algorithm proceeds as follows. Any nonterminal B that satisfies $A \in CV[i, j]$, $C \in CV[j + 1, k]$, $AB \longrightarrow AC \in P$, is added to $CV[j + 1, k]$. This implies that C $\Longrightarrow^* a_{j+1} \dots\ a_k$, and therefore B $\Longrightarrow^* a_{j+1} \dots a_k$, as shown in section 3.4.1. An empty set P_B, used to refer to the nonterminal's predecessors, is constructed. A reference to this nonterminal is then added to sets P_A and P_C, in case either of the original nonterminals will be deleted in the event of version splitting. This process is repeated until $CV[j+1, k]$ cannot be extended anymore. If any context-sensitive nonterminals were added and the set's ignored coordinates have not been set yet, a reference to this set is added to BLQ at the end of the step.

Any nonterminal A is then added to the set $CV[i, k]$, if it satisfies $B \in CV[i, j]$, $C \in CV[j + 1, k]$, $A \longrightarrow BC \in P$, because $B \Longrightarrow^* a_i \dots a_j$, $C \Longrightarrow^* a_{j+1} \dots a_k$, and therefore $A \Longrightarrow^* a_i \dots a_k$. An empty set of predecessors, P_A, is created for each added nonterminal. A reference to the nonterminal is added to sets P_B and P_C.

If the ignored coordinates are equal to $CV[j + 1, k]$, a copy of the matrix is created before the first satisfied rule is applied. All context-sensitive nonterminals of $CV[i, j]$, including their predecessors, are deleted from the copy using predecessor sets, P_A for each nonterminal A, to recursively detect all nonterminal's predecessors. All of the subsequently added nonterminals are then saved to the copy instead of CV. Once the scan is completed, this copy is added to V.

Once no set can be extended, it is examined whether the starting symbol, $S \in CV[1, n]$, so $S \Rightarrow^* a_1 \dots a_n$. If the check is passed, the algorithm announces **ACCEPT**. Otherwise, CV is removed from V, a different member is appointed as the new CV, and the parsing is continued for this version of the matrix. If V is empty, and therefore no new CV can be appointed, there are no alternative syntax trees to be constructed. In such case, the algorithm announces **REJECT**.

4.4 Proof of Correctness

Sets N, T and P of the examined grammar, G, are final. There can exist at most $|P| + 1$ versions. For each entry of every version of the parse matrix, at most $|P|$ productions can be reduced. Therefore, the algorithm terminates for any input.

As this algorithm is based on the Cocke-Younger-Kasami algorithm, its correctness can be assumed. [5] The presented algorithm does not change any of its key parts, only extends them. These changes do not affect the rule application, with the exception of the matrix splitting, which does not limit the parsing in any way, only splits it into independent phases.

4.5 Algorithm Complexity

Complexity of the presented algorithm is greater than that of the Cocke-Younger-Kasami algorithm. This is a result of its ability to apply context-sensitive rules, which causes greater time and space requirements for the algorithm.

4.5.1 Time Complexity

In the worst case scenario, the algorithm produces $|P| + 1$ versions of the parse matrix – the initial version, and $|P|$ additional versions created because of the context conflict detection. If only one of these versions is created during each loop of the main `repeat until` cycle, the loop is repeated $|P| + 1$ times – the initial loop and $|P|$ repetitions. During each of them, the algorithm has to repeat the three `for` cycles that iterate through the triangle matrix in the appropriate order. For each of the entries, $|P|$ production have to be tested. Therefore, the time complexity is as follows:

$$t = (|P| + 1)^2 \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} \sum_{k=0}^{i-1} |P| \tag{4.1}$$

$$t = \frac{n(n^2 - 1)\ |P|(|P| + 1)^2}{6} \tag{4.2}$$

$$O(t) = O(n^3 \cdot |P|^3) \tag{4.3}$$

4.5.2 Space Complexity

In the case described in the previous section, the algorithm generates a total of $|P| + 1$ matrices – an initial matrix and a copy of the matrix for each of the generated versions. Each entry of these matrices can hold a total of $|P|$ derived nonterminals. The space complexity reaches:

$$s = (|G| + 1) \sum_{i=1}^{n} |G| i \tag{4.4}$$

$$s = \frac{(|G| + 1)(n - |G| + 1)(|G| + n)}{2} \tag{4.5}$$

$$O(s) = O(n^2 \cdot |G|^3) \tag{4.6}$$

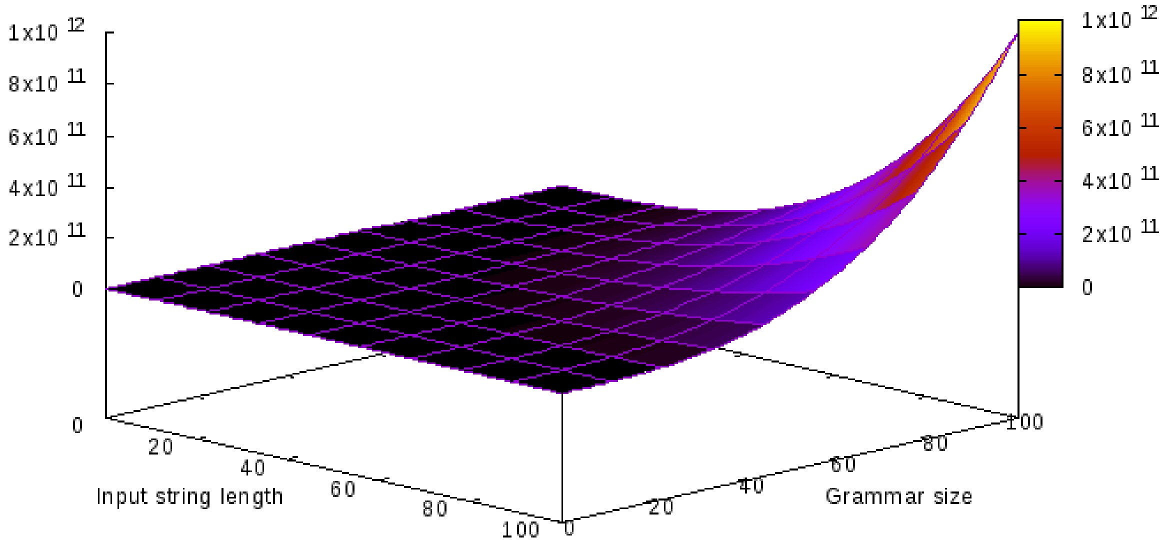


Figure 4.1: Time complexity of the presented algorithm is estimated at $O(n^3 \cdot |G|^3)$.

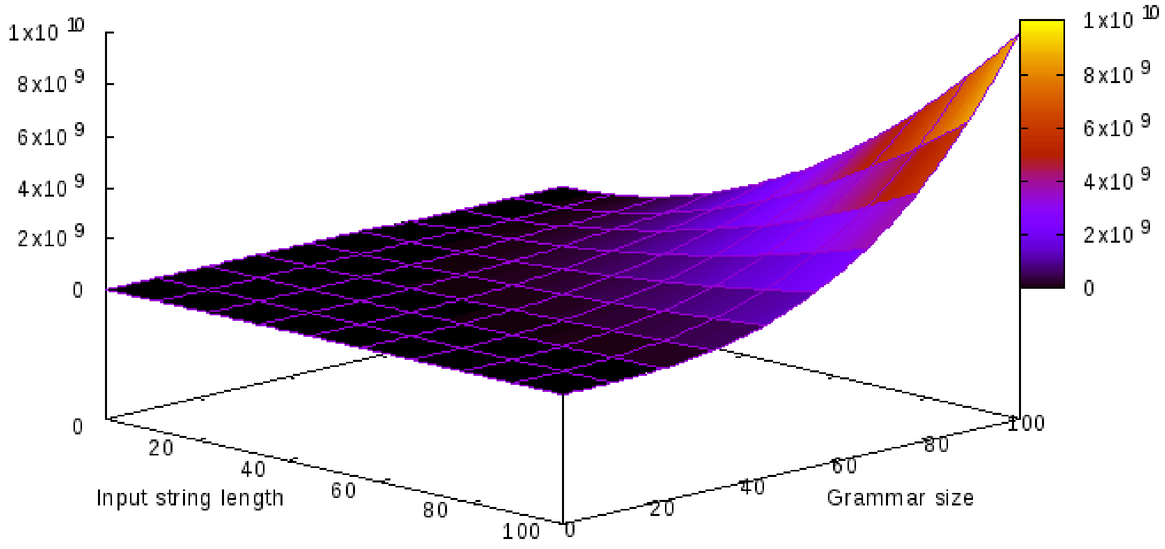


Figure 4.2: Space complexity of algorithm reaches $O(n^2 \cdot |G|^3)$ in the worst case scenario.

4.6 Pseudocode Representation

The following section formalizes the presented algorithm in the form of pseudocode. The sets used, and their roles, are further described in the section 4.2. Algorithm 2 describes the main loops that are responsible for the parse matrix traversal, and the final verification of presence of the start symbol.

Algorithm 3 focuses on the production matching and application. It describes the circumstances under which new versions are created, and how their values are set.

Algorithm 2 Cocke-Younger-Kasami Algorithm Adapted to the Penttonen Normal Form

Input: a grammar, G $= (N, T, P, S)$, in Penttonen normal form;
$w = a_1 a_2 \dots a_n$ 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 set $V = \varnothing$;
2: introduce matrix of sets $CV$, where $CV[i,j] = \varnothing$ for $1 \leq i \leq j \leq n$, and add $CV$ to $V$;
3: introduce set $BLQ = \varnothing$ belonging to $CV$;
4: for $i = 1$ to $n$ do
5:     if $A \longrightarrow a_i \in P$ then
6:         add A to $CV[i,i]$;
7:         introduce set $P_A = \varnothing$ holding references to $A$'s predecessors;
8: parse_loop:
9: repeat
10:     for $level = 1$ to $n - 1$ do
11:         for $i = 1$ to $n - level$ do
12:             set $k$ to $i + level$;
13:             for $offset = 0$ to $level - 1$ do
14:                 set $j$ to $i + offset$;
15:                 ApplyRules$(i, j, k)$;
16: until no change;
17: if S $\in CV[1,n]$ then
18:     ACCEPT
19: else
20:     remove $CV$ from $V$;
21:     if $V \neq \varnothing$ then
22:         pick an element of $V$ and set as $CV$;
23:     else REJECT;
24: goto parse_loop;
```

The indices acquired in algorithm 2 are used to locate sets representing the neighbouring substrings of the input string, and to subsequently appoint coordinates of the destination set of nonterminals for this reduction. The process is further described in algorithm 3. The variables i, j, k are listed as the procedure parameters, because their value affects which sets are tested. The algorithms consider all listed sets to be global.

Algorithm 3 Rule Application

```
procedure ApplyRules(i,j,k)
    if $CV[i,j] \in BLQ$ then
        set its ignored coordinates to $[j+1,k]$;
        remove $CV[i,j]$ from $BLQ$;
    if $A \in CV[i,j]$, $C \in CV[j+1,k]$, $rhs(\mathrm{p}) = AC$ for some $A, C \in N$, $\mathrm{p} \in P$ then
        if coordinates ignored by $CV[i,j]$ are not equal to $[j+1,k]$ then
            if $AB \longrightarrow AC \in P$ for some $B \in N$ then
                add $B$ to $CV[j+1,k]$ and mark it as context-sensitive;
                if ignored coordinates of $CV[j+1,k]$ are not set then
                    add $CV[j+1,k]$ to $BLQ$;
            if $B \longrightarrow AC \in P$ for some $B \in N$ then
                add $B$ to $CV[i,k]$;
            introduce set $P_B = \varnothing$ holding references to $B$'s predecessors;
            add a reference to the nonterminal $B$ to sets $P_A$ and $P_C$;
        else
            if this instance of $A$ is context-sensitive then
                skip this nonterminal;
            create a copy of $CV$, its $BLQ$, and all sets of predecessors;
            remove all predecessors of context-sensitive nonterminals in $copy[i,j]$;
            remove all context-sensitive nonterminals from $copy[i,j]$;
            if $AB \longrightarrow AC \in P$ for some $B \in N$ then
                add $B$ to $copy[j+1,k]$ and mark it as context-sensitive;
                if ignored coordinates of $copy[j+1,k]$ are not set then
                    add $copy[j+1,k]$ to the $copy$'s $BLQ$;
            if $B \longrightarrow AC \in P$ for some $B \in N$ then
                add $B$ to $copy[i,k]$;
            introduce set $P_B = \varnothing$ holding references to $B$'s predecessors;
            add a reference to the nonterminal $B$ to sets $P_A$ and $P_C$;
            add the copy to $V$;
```

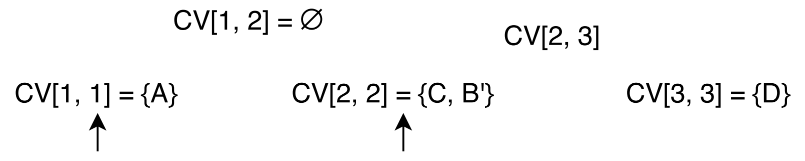
4.7 Example

The following example aims to demonstrate the work of the algorithm presented in this chapter. It describes the parsing process of the input string, $w =$*words are fun*, according to the grammar, $G = (N, T, P, S)$, where $N = \{S, A, B, C, D, E, F\}$, $T = \{fun, are, words\}$, and $P = \{C \longrightarrow are, D \longrightarrow fun, A \longrightarrow words, S \longrightarrow AF, AB \longrightarrow AC, E \longrightarrow CD, F \longrightarrow CD\}$. In this example, diagonals of the parse matrix are treated as tiers, and context-sensitive nonterminals are marked with an apostrophe.

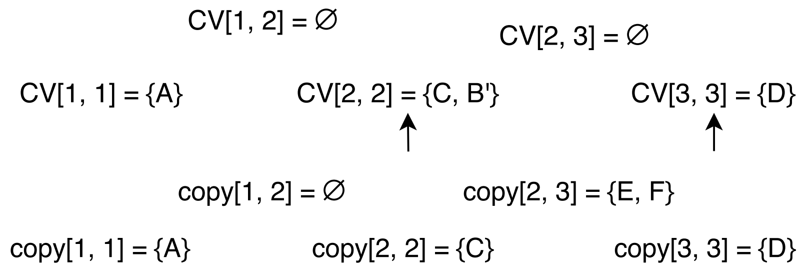
First, the input string is scanned, and nonterminal sets reflecting it are constructed. After this step, $BLQ = \varnothing$.

CV[1, 1] = {A}	CV[2, 2] = {C}	CV[3, 3] = {D}
words	**are**	**fun**

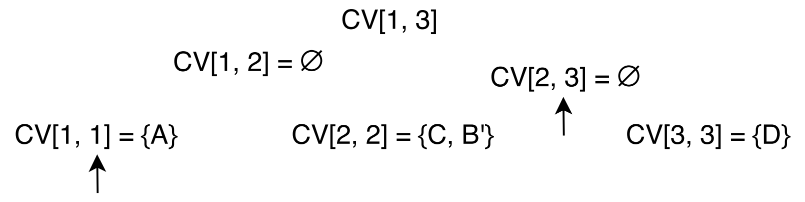
Then, the first two sets of nonterminals are evaluated. The rule $AB \longrightarrow AC$ is applied, nonterminal B is added and marked as context-sensitive. Because coordinates ignored by $CV[2,2]$ have not been set yet, the set is added to BLQ. After this step, $BLQ = \{[2,2]\}$.

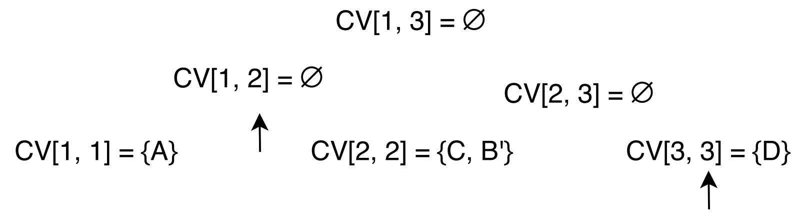


Since $CV[2,2]$ is used as the left operand in this pair and it is a member of BLQ, the coordinates $[3,3]$ are set as ignored by the set. This pair is then evaluated – before the rules $E \longrightarrow CD$ and $F \longrightarrow CD$ can be applied, a copy of CV is created, and B is deleted from this copy. Subsequently, the rules are applied. Finally, the copy is added to V. For CV, $BLQ = \varnothing$, Ignored coordinates: $CV[2,2] \to [3,3]$. For *copy*, $BLQ = \varnothing$, Ignored coordinates: $copy[2,2] \to copy[3,3]$.

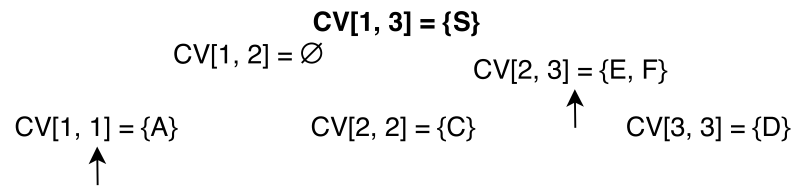


Sets in the second tier are evaluated, and since there are no more changes to be made, it is tested whether $S \in CV[1,3]$. As this check fails, CV is abandoned. It is removed from V, and the previously created copy is appointed as the new CV. $BLQ = \varnothing$, Ignored coordinates: $CV[2,2] \to CV[3,3]$.





This matrix is evaluated in an identical manner as before. The rule $S \longrightarrow AF$ is applied for nonterminals $A \in CV[1,1]$ and $F \in CV[2,3]$. Once no new members can be added to any of the sets, it is confirmed that $S \in CV[1,3]$, and the algorithm announces **ACCEPT**.



Chapter 5

Prototype Implementation

This chapter focuses on the working prototype which implements the algorithm presented in the previous chapter. The prototype was implemented in C++ using the C++11 standard. The language was chosen because of the variety of standard containers and pointer operations it offers.

The final program is a console application; it is platform-independent, as it only uses data structures and methods that are available as a part of the Standard Template Library. The examined grammar and input string are both user-specifiable.

The application expects input data in the form of two configuration files – first, the file holding rules of the grammar that will be used to parse the string, and second, the file containing the sentential form whose grammatical correctness will be verified.

If the program has successfully completed the parsing, it prints a single line to console indicating the input string's correspondence to the chosen grammar.

5.1 Data Model

The program is composed of several modules. Each of these modules has a distinct role during the application run, such as an input string and grammar processing, version administration and grammar rule applications.

Figure 5.1 reflects the application composition; a single instance of the `Parser` class owns instances of the three main modules and coordinates their communication. These modules will be described in detail in the following sections.

5.2 Grammar Adapter

The `GrammarAdapter` class is responsible for processing and usage of the grammar specified by user at launch. It parses the file set by a command line argument, and constructs an object containing an equivalent grammar. This grammar is later used during the analysis of the parse matrix.

After confirming that the file exists and is readable, the adapter reads it line by line. It analyses every line, and extracts the symbols it recognizes according to rules specified in section 5.2.1. Based on the number of the present symbols on each side of the rule, it checks whether all symbols are of the correct type – e.g. the left-hand side of the examined rule does not contain `Terminals`.

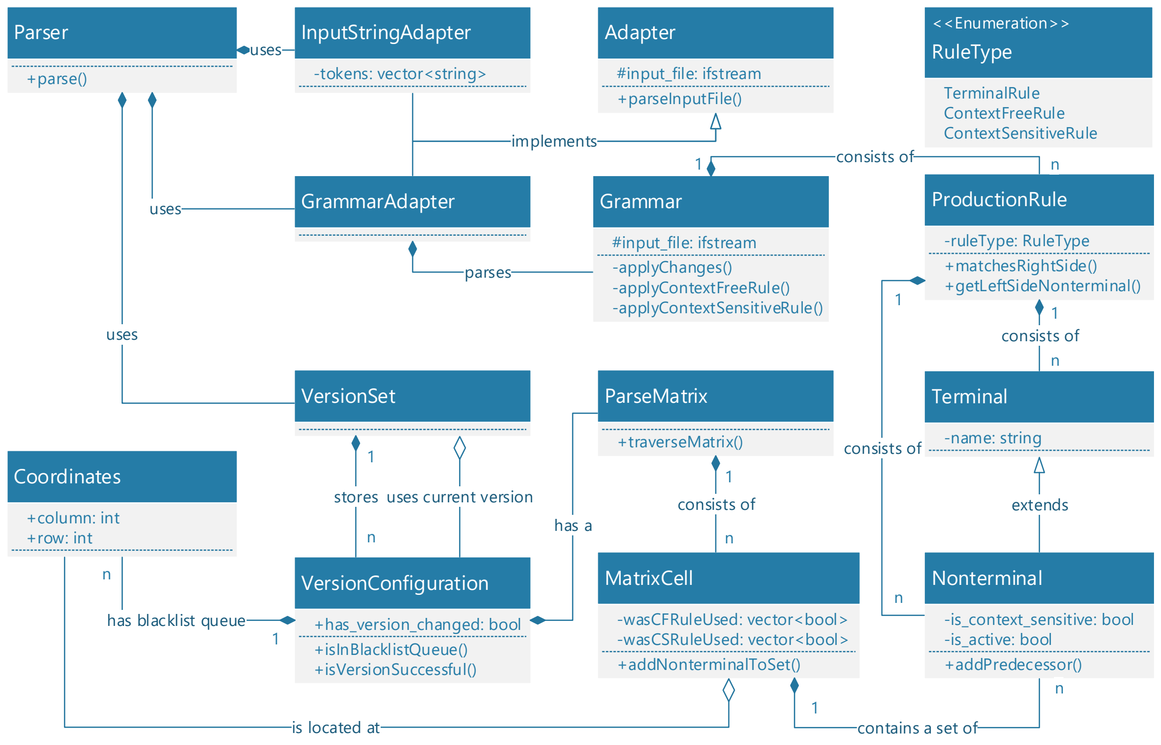


Figure 5.1: Diagrams of classes as implemented in the working prototype. The two main classes are `GrammarAdapter` and `VersionSet`; these are encapsulated by a single object that oversees the entire program run.

The symbols can be of two types: `Terminals`, whose only member variable is their name, and the derived class, `Nonterminals`, that contain data on own context-sensitivity and activity. Each `Nonterminal` contains a `vector<Predecessor>` that saves the position of all instances derived from it. All symbols are active by default.

Once all symbols have been extracted and verified, an instance of `ProductionRule` is created, and the extracted nonterminals are assigned to it. Based on the number of extracted symbols, the type of the rule is set; a rule can be of one of the following types:

1. *context-sensitive rule* – rules used at the beginning of examination of every pair of nonterminal sets.

2. *context-free rule* – rules used at the end of examination of set pairs.

3. *terminal rule* – rules used during the initialization of the first version of *ParseMatrix*.

This system of rule classification will be further described in section 5.5.

The complete rule is then added to the corresponding rule `vector` in the adapter's instance of `Grammar` to prevent additional type checks every time a pair is compared, as these would cause delays in the parse matrix examination.

5.2.1 Grammar Format

The adapter uses the general rule of at most one production on every line. The left-hand side and the right-hand side of every rule are divided by a colon; the form of the rules themselves reflect the Penttonen normal form, which was described in section 2.6.3. A rule can be in one of the following forms:

1. `<A><B>:  <A><C>` – the context-sensitive rules
2. `<A>:  <B><C>` – the context-free rules
3. `<A>:  a` – the terminal rules

The name of every nonterminal has to be surrounded by angle brackets. In case a symbol is not surrounded, the adapter treats it as a terminal. There can be any number of whitespace excluding the end of line in nonterminal names and between symbols; end of terminals is marked by the first occurrence of any whitespace character. In case the user wants to use any of the control symbols, i.e. colon or an angle bracket, they can be escaped by a backslash.

5.3 Input String Adapter

The `InputStringAdapter` class implements the same virtual base class as the `GrammarAdapter` class. It scans the input string specified by the user and creates a `vector` of tokens.

After verifying the access to the input file specified as a command line arguments, the adapter reads the input file word by word, splitting at a whitespace. This is achieved using the standard input file stream class, `ifstream`.

The extracted words are then pushed to the adapter's token `vector<string>`. The `vector` is then used during the initial `ParseMatrix` construction to provide data about the length of the input string, and therefore about the dimension of the matrix, and to offer the potential right-hand side of the terminal productions.

5.4 Version Control

This module is responsible for managing the different versions of the parse matrix.

The main class in this module is the `VersionSet` class. It is responsible for keeping data about the currently examined version, storing possible versions that have not been analyzed yet, and finally, accepting or rejecting the input string.

5.4.1 Version Configuration

The various versions managed by the `VersionSet` are stored in a `vector` of the `VersionConfiguration` class instances.

The `VersionConfiguration` class consists of an instance of `ParseMatrix` and a `vector<Coordinates>` – the `_blacklist_queue`; it contains positions of all matrix entries whose ignored coordinates have not been set yet. The `vector` contains `Coordinates` instead of pointers for the sake of references staying up-to-date in case of version splitting.

Throughout its run, the program uses a single instance of the `VersionSet`. This instance represents a reference to the currently analyzed instance of `VersionConfiguration`; this instance serves the purpose of being the `CurrentVersion`, as described in algorithm 2. As a contrast to the original algorithm, the instance is removed from the `VersionSet` before being analyzed to allow for a simpler input string rejection test.

When a context conflict is detected during the analysis of the `CurrentVersion` as described in section 5.5, the object is copied, the copy is modified and at the end of the analysis, the modified copy is pushed to the version `vector`.

5.5 Parsing Core

The parse matrix, which is modified during the analysis of an instance of `VersionConfiguration`, is represented by an instance of the `ParseMatrix` class.

The class consists of a `map` of `MatrixCells`, ordered by `Coordinates`. A `MatrixCell` contains a `vector<Nonterminal>`, which represents the nonterminal set that is used during the rule application, as described in algorithm 3. The prototype uses `vectors` instead of C++'s standard `set` container, as its usage would mean complications when using the `Nonterminalss`' `vector<Predecessor>`, as described in section 5.6. Instead, it implements a `namespace` of function templates that allow treating `vectors` in a way equivalent to `sets` and offering extra functionality.

Each instance is aware of its location on the map, and keeps track of `Coordinates` of the next cell on the matrix diagonal, as this cell could cause a context conflict.

The `ParseMatrix` class is responsible for the comparison of the `MatrixCells` in an order that ensures the cells' `_ignoredCoordinates` consistency, and application of `ProductionRules` of the `Grammar` generated at the beginning of the program run. This class is also responsible for triggering the version splitting in case of a context conflict detection, as described in section 5.6.

5.5.1 `ParseMatrix` Initialization and Traversal

Once the `Adapters` have constructed all objects based on the user data, the `Grammar::constructInitialSets` method is called. This method iterates through the main diagonal of the initial `ParseMatrix` instance, and applies terminal rules. It does so by comparing each of the tokens with the right-hand side of all such rules, and adding a `Nonterminal` representing the left-hand side of the matching `ProductionRule` to the `MatrixCell` as the corresponding `Coordinates`, as described in algorithm 2.

The `ParseMatrix::traverseMatrix` method is responsible for iterating through the `map` in the order required by *parse_loop* of algorithm 2. It starts by processing sets on the main diagonal, advancing further right. On each diagonal, it processes `MatrixCells` starting from the topmost cell, working its way down. The `i`, `j` and `k` variables represent indices used to appoint the cells to be examined are acquired as follows, where the `length` variable represents the number of tokens extracted by the `InputStringAdapter` at the beginning of the program run:

```
for (depth = 1; depth < length; depth++) {
    for (int i = 0; i < length - depth; i++) {
        int k = i + depth;
        for (int offset = 0; offset < depth; offset++) {
            int j = i + offset;
            // rule application
        }
    }
}
```

This cycle keeps repeating as long as any of the `MatrixCells` has changed, because addition of context-sensitive `Nonterminals` does not have to be linear.

5.6 Rule Application

For every pair of `MatrixCells` appointed by the traversal cycle shown in section 5.5.1, all rules containing only `Nonterminals` are checked, and in case of right-hand side match, applied.

Once there are no rules to be applied for this `ParseMatrix`, the success of this `Version-Configuration` is tested.

5.6.1 Context-sensitive Rule Application

As the first step of the pair examination, the context-sensitive rules are applied. They are applied first, because a match of the right-hand side of such a rule may lead to creation of `Nonterminals` that could be used to reduce *context-free* rules, and therefore the program applies them immediately instead of having to wait for the following iteration of the main cycle described in section 5.5.1, which might lead to decreased run time.

The rules are checked by examining every possible combination of `Nonterminals` in the examined `vectors` and comparing it to the right-hand side of all rules of the corresponding phase – in this case, the *context-sensitive* rules. When a match is found, the `Grammar` acquires the changing left-hand side `Nonterminal` of the corresponding `ProductionRule`, and rule is marked as applied for the modified `MatrixCell` to prevent infinite production rule loops. Since this algorithm uses the Penttonen normal form, only one `Nonterminal` changes regardless of the type of the rule.

If the second `MatrixCell` of the pair is ignored by the first one, the new `Nonterminal` is added to the copy instead, and an inactive version of it is added to the `CurrentVersion`; in this case, the `ProductionRule` is marked as applied in both instances of `VersionCon-figuration` to prevent matrix inconsistencies.

In case a `Nonterminal` is to be added to the copy of the `CurrentVersio`, but the copy has not been initialized, the `Grammar::initializeNewVersion` is evoked. This method creates a copy of the `CurrentVersion` and sets all context-sensitive `Nonterminals` of the currently examined `MatrixCell` to inactive, as well as all `Nonterminals` of its `_predecessors`. This is equal to deleting the nonterminals described in algorithm 3; since the C++ `vector` dynamically reallocates itself, the prototype uses the index of the `Predecessor` in its designated `MatrixCell` instead of pointers to prevent inconsistent memory reference. Because of this, the `Nonterminals` must not be deleted and are set inactive instead. Inactive `Nonterminals` are ignored during the pair examinations.

5.6.2 Context-free Rule Application

As the second step of the pair examination, the context-free rules are applied.

The process is analogical to that described in section 5.6.1. The combinations of `Nonterminals` are examined and compared to the right-hand side of context-free `Production-Rules`.

When a match is detected, the corresponding left-hand side `Nonterminal` is added to the set at `Coordinates{i, k}`. If the second `MatrixCell` is ignored, the symbol is added to the copy instead, and an inactive version is added to `CurrentVersion`. The `Production-`

`Rule` is marked as used in both versions to prevent duplicate application of the same rule, and therefore to prevent possible creation of new `VersionCongurations` that are identical to already existing versions.

5.7 Testing

The program was tested using a total of forty-two pairs of grammars and corresponding input strings. The tests were run for existing files, with a mixture of both accepted and rejected input strings with a total success rate of ~ 40.5 %.

These results confirm the assumptions about time and space complexity of the algorithm presented in section 4.5, as the values show a steady growth when increasing one of the parameters, and an exponential growth when values of both are high.

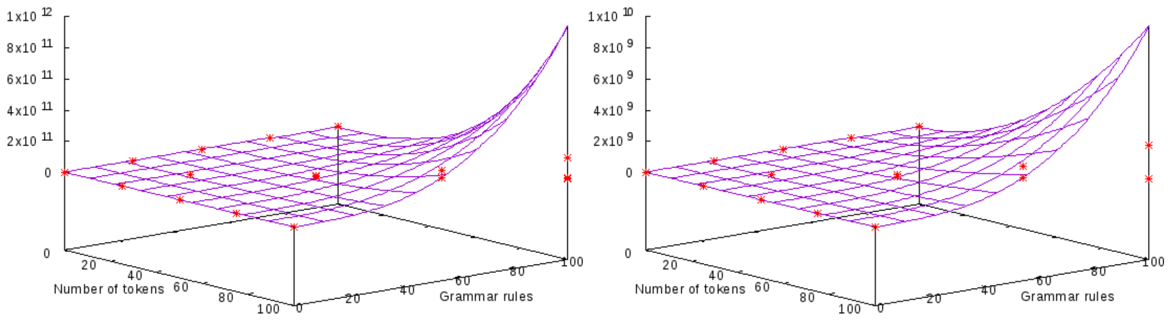
The table 5.1 shows the number of instructions executed based on the length of the input string and number of production rules a grammar contains. The table 5.2 shows the memory consumption for these combinations. The experimental data was acquired using the `Valgrind` utilities `Callgrind` and `Memcheck`. The visualization of this data compared to the analytical values is available in figures 5.2a and 5.2b respectively.

Table 5.1: Experimental values acquired from the time complexity testing. The values shown in the last four columns represented number of instructions executed; n represents the number of tokens, and $|G|$ represents number of `ProductionRules` the grammar contains.

n	$\|G\|$	**Average**	**Set 1**	**Set 2**	**Set 3**
0	0	2270000	2270000	2270000	2270000
0	25	43700000	43700000	43700000	43700000
0	50	220800000	220800000	220800000	220800000
0	75	553566666.7	617000000	617000000	426700000
0	100	1218000000	1328000000	1328000000	998000000
25	0	2780000	2780000	2780000	2780000
25	25	165566666.7	232500000	122500000	141700000
50	0	3226666.67	3200000	3280000	3200000
50	50	3532333333	7774000000	1248000000	1575000000
75	0	3750000	3780000	3830000	3640000
75	75	21086666667	51374000000	4911000000	6975000000
100	0	4210000	4250000	4380000	4000000
100	100	60819000000	149120000000	13310000000	20027000000

Table 5.2: Experimental values acquired from the space complexity testing. The values shown in the last four columns represented memory consumption in bytes. n represents the number of tokens, and $|G|$ represents number of `ProductionRules` the grammar contains.

n	$\|G\|$	**Average**	**Set 1**	**Set 2**	**Set 3**
0	0	85100	91300	72700	91300
0	25	693350	693400		693300
0	50	2500000	2500000		2500000
0	75	4850000	5500000		4200000
0	100	9066666.67	9700000	9700000	7800000
25	0	99933.33	100000	100800	99000
25	25	1900000	3100000	1200000	1400000
50	0	109000	109000	110000	108000
50	50	45866666.67	116000000	8900000	12700000
75	0	119066.67	121000	120200	116000
75	75	294900000	790000000	43700000	51000000
100	0	128266.67	128000	133000	123800
100	100	847666666.7	2259000000	140000000	144000000



(a) Experimental time complexity (b) Experimental space complexity

Figure 5.2: Visual representation of experimental space complexity data from tables 5.1 and 5.2 respectively. The purple grid represents the analytical values assumed in section 4.5, red points represent the actual data.

Chapter 6

Conclusion

This thesis examines general properties of formal grammars, and languages generated by them. Using the Chomsky hierarchy as the base for its research, it examines the unrestricted grammars, their equivalence to Turing machines, and focuses on context-sensitive grammars as their proper subset.

It inspects normal forms as systems for the formalization of grammar productions into the designated form, which facilitate the grammar application. The Kuroda normal form of unrestricted grammars, which extends the Chomsky normal form, and its special case, the Penttonen normal form, are introduced. These are modified to satisfy the context-sensitive grammars, and an algorithm of construction of a grammar in normal form is presented.

The purpose of syntax analysis, and its existing types, are explored. The thesis mentions both the widely used context-free analyses, as well as the context-sensitive alternative that integrates semantic checks. The three main types of parsing – top-down, bottom-up and universal – are described.

Based on this research, an algorithm capable of parsing context-sensitive grammars is designed. This algorithm extends the functionality of the Cocke-Younger-Kasami parsing method, and adapts it to accept ambiguous context-sensitive grammars in the Penttonen normal form. It successfully introduces a number of support systems that prevent syntax tree inconsistencies and handle possible context conflicts while ensuring all possible versions of parse tree are examined.

A prototype implementing this algorithm was created in C++. It uses a system of file adapters that allow for any user-specified grammar and input string as long as the grammar conforms to the Penttonen normal form. The testing confirmed that the complexity of the program is similar to its analytical values – almost linear in case either of the key parameter values is low, increasing almost cubically in case both of the parameter values are high.

This thesis focuses on a topic that has not been thoroughly researched yet, therefore it aims to serve mainly as a base for further research. In the future, I would like to expand the algorithm to accept grammars in additional normal forms. This could allow for a more extensive application, as the process of construction of a grammar in the Penttonen normal form causes an exponential growth in the grammar size.

Bibliography

[1] Aho, A. V.: *Compilers: principles, techniques & tools*. Addison Wesley. second edition. 2007. ISBN 0321486811.

[2] debguy: *bnf2xml*. [Online; visited 14/05/2017].
Retrieved from: `https://sourceforge.net/projects/bnf2xml/`

[3] Hopcroft, J. E.; Motwani, R.; Ullman, J. D.: *Introduction to automata theory, languages, and computation*. Pearson; Addison-Wesley. third edition. 2007. ISBN 1420063235.

[4] Meduna, A.: *Automata and languages: theory and applications*. Springer. 2000 edition. 2000. ISBN 1852330740.

[5] Meduna, A.: *Elements of compiler design*. Auerbach Publications. first edition. 2008. ISBN 1420063235.

[6] Rozenberg, G.; Salomaa, A.: *Handbook of formal languages: Linear modelling, background and application*. vol. 2. Springer Verlag. 1997 edition. 1997. ISBN 3540606483.

[7] Rozenberg, G.; Salomaa, A.: *Handbook of formal languages: word, language, grammar*. vol. 1. Springer. first edition. 1997. ISBN 9783642638633.

Appendices

Appendix A

Prototype Information

This appendix contains information about the source code and usage details.

A.1 Parameters and Return Codes

The program is a console application; it accepts input in the form of command line parameters at the time of launch. These parameters are used to specify the grammar and input string files the program is going to use. The program accepts following arguments:

1. `--help` or `-h` – prints the help message and terminates the program,
2. `--grammar` or `-g` – sets the relative path to the grammar file,
3. `--input` or `-i` – sets the relative path to the grammar file.

If neither of the parameters is used, the program parses an example string and grammar shown in appendix B. However, if either is used, both parameters need to be specified; a missing parameter results in the program terminating with the error code `ARGUMENT_ERROR (2)`.

An example of a correct program launch would be:

- `pnf_parser`
- `pnf_parser --help`
- `pnf_parser --grammar=ex_grammar_01 -i=ex_input_string_01`

If either adapters detect a problem while accessing the input files, the program terminates with the error code `PARSE_ERROR (1)`.

In case all of the previous actions have been executed successfully, the program returns `0` regardless of string acceptance result.

A.2 Code Metrics

Lines of code: 1922
Number of files: 25
Total size of source code: 69.3 kB
Executable file size: 299.8 kB

Appendix B

Input Files Example

This appendix presents the grammar and the input string used in section 4.7 written in the form acceptable by the working prototype. Corresponding files can be found on the attached media as `ex_grammar_01` and `ex_input_string_01`. This grammar is meant to demonstrate both the context conflict detection and the possible ambiguity of accepter grammar.

Grammar (`ex_grammar_01`)

```
<S>:  <A><F>
<A><B>:  <A><C>
<E>:  <C><D>
<F>:  <C><D>
<C>:  are
<D>:  fun
<A>:  words
```

Input String (`ex_input_string_01`)

```
words are fun
```

Appendix C

Contents of the Attached Medium

The attached medium contains the following directories and files:

- `bin/` – the directory containing the `pnf_parser` executable and example files,
- `doc/` – the directory containing LaTeX sources codes, and files needed to generate the documentation,
- `pdf/` – the directory containing both versions of the `pdf` documentation,
- `src/` – the directory containing the `*.h` and `*.cpp` source codes,
- `README` – the text file containing instructions for compilation and execution of the program.

Appendix D

Poster

An article about the presented algorithm was published as a part of Excel@FIT 2017. The following poster was used to represent it during the public exhibition of participant works.

CYK Algorithm Adapted to the Penttonen Normal Form

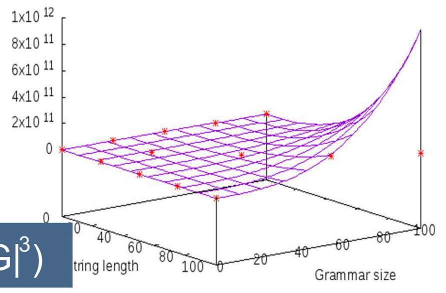
#41

Why? And how?

- as of now, no parser capable of processing context-sensitive grammars exists
- algorithm based on the CYK algorithm for context-free grammars in Chomsky normal form
- integrates context-sensitive rules in the form of AB → AC introduced by Penttonen normal form
- uses a versioning system to manage alternate parsing matrices created as the result of context conflict
- makes an unambiguous decision of whether the input string is a sentence of user-defined grammar

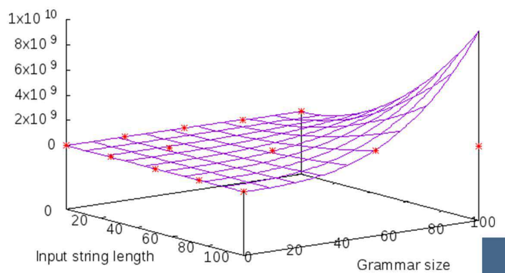
How many resources does it take?

- time complexity is similar to that of the original CYK algorithm, multiplied by the number of possible versions



$O(n^3|G|^3)$

- space complexity is affected mainly by the size of the grammar, as it affects both the number of possible versions as well as number of symbols in a set



$O(n^2|G|^3)$

How exactly?

- in case of a context-sensitive rule reduction, it adds the reduced nonterminal to the same set as the original symbol
- the set remembers the first right neighbour it is compared with to detect context collisions in time

CV[1,1] = {A}	CV[2,2] = {C}	CV[3,3] = {D}
he	*likes*	*rainbows*

- if a context collision is detected, the version is split into two
- the original version keeps the context-sensitive symbols, and the new version gets all subsequently reduces symbols

CV[1,2] = {} CV[2,3]

CV[1,1] = {A} CV[2,2] = {C, **B**} CV[3,3] = {D}

copy[1,2] = {} copy[2,3] = {E, F}

copy[1,1] = {A} copy[2,2] = {C} copy[3,3] = {D}

- in case of failure of currently used matrix, the version is abandoned, and a copy is chosen in its place

CV[1,3] = {S}

CV[1,2] = {} CV[2,3] = {E, F}

CV[1,1] = {A} CV[2,2] = {C} CV[3,3] = {D}

- if any such version succeeds, the string is **accepted**, otherwise it is **rejected.**

Dominika Klobučníková

Excel@FIT 2017