

# Automata Project – Check if a word is derived from a context free grammar

Raviv Komem

February 2020

**Submitted By:** Raviv Komem

**Student ID:** 316217751

**Faculty:** Software Engineering

**Project Purpose:** Assignment for Automata course as part of the excellence program

**Year:** 2020

**Semester:** Winter Semester (A)

**Submission Date:** 20/02/2020 (TBD!)

**Supporting Lecturer:** Yoav Rodeh

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Modern Days Uses . . . . .	1
<b>2</b>	<b>Formal Definition</b>	<b>2</b>
2.1	Grammar Definition . . . . .	2
2.2	Production rule notation . . . . .	2
2.3	Rule application . . . . .	3
2.4	Repetitive rule application . . . . .	3
2.5	Context-free language . . . . .	3
<b>3</b>	<b>Examples of CFGs</b>	<b>3</b>
3.1	Words concatenated with their reverse . . . . .	3
3.2	Well-formed parentheses . . . . .	4
3.3	Matching pairs . . . . .	4
3.4	Distinct number of a's and b's . . . . .	5
<b>4</b>	<b>Derivations and syntax trees</b>	<b>5</b>
<b>5</b>	<b>Chomsky Normal Form - CNF</b>	<b>7</b>
5.1	Definition . . . . .	7
5.2	Converting a grammar to Chomsky normal form . . . . .	8
5.3	START: Eliminate the start symbol from right-hand sides . . . . .	8
5.4	TERM: Eliminate rules with nonsolitary terminals . . . . .	8
5.5	BIN: Eliminate right-hand sides with more than 2 nonterminals . . . . .	8
5.6	DEL: Eliminate $\epsilon$ -rules . . . . .	9
5.7	UNIT: Eliminate unit rules . . . . .	10
5.8	Order of transformations . . . . .	10
<b>6</b>	<b>CYK Algorithm</b>	<b>12</b>
6.1	Pseudo Code . . . . .	13
6.2	Improvements to the algorithm . . . . .	14
6.2.1	Relaxing of the grammar form . . . . .	14
6.2.2	Parser Algorithm . . . . .	16
<b>7</b>	<b>CYK using grammar in 2NF</b>	<b>17</b>
7.1	Calculate Nullable . . . . .	17
7.2	Constructing the Unit Relation . . . . .	18
7.3	Calculate $\tilde{U}_G^*(M)$ . . . . .	18
7.4	Algorithm . . . . .	18

<b>8</b>	<b>Implementation</b>	<b>19</b>
8.1	Breakdown of tasks . . . . .	19
8.2	Implementation language and environment . . . . .	20
8.3	Data Structures . . . . .	21
8.4	Program API - Application Program Interface . . . . .	21
<b>9</b>	<b>Input and output results of the code</b>	<b>23</b>
9.1	Grammar conversion . . . . .	23
9.1.1	Test 1 - Conversion success . . . . .	23
9.1.2	Test 2 - Conversion success . . . . .	24
9.2	Exponential algorithm to find all words in given CFG . . . . .	24
9.2.1	Java Implementation . . . . .	25
9.3	Word recognizing and parsing . . . . .	25
9.3.1	Test - Successful parsing . . . . .	26
9.3.2	Test - Unsuccessful parsing . . . . .	27
<b>10</b>	<b>Parsing using original grammar</b>	<b>27</b>
10.1	Backtracking the grammar conversion . . . . .	27

# 1 Introduction

## 1.1 Background

Since early times, linguists have described the grammars of languages in terms of their block structure, and described how sentences are recursively built up from smaller phrases, and eventually individual words or word elements. An essential property of these block structures is that logical units never overlap.

A context-free grammar provides a simple and mathematically precise mechanism for describing the methods by which phrases in some natural language are built from smaller blocks, capturing the "block structure" of sentences in a natural way. Its simplicity makes the formalism amenable to rigorous mathematical study. Important features of natural language syntax such as agreement and reference are not part of the context-free grammar, but the basic recursive structure of sentences, the way in which clauses nest inside other clauses, and the way in which lists of adjectives and adverbs are swallowed by nouns and verbs, is described exactly.

The formalism of context-free grammars was developed in the mid-1950s by Noam Chomsky,[1] and also their classification as a special type of formal grammar (which he called phrase-structure grammars). What Chomsky called a phrase structure grammar is also known now as a constituency grammar, whereby constituency grammars stand in contrast to dependency grammars. In Chomsky's generative grammar framework, the syntax of natural language was described by context-free rules combined with transformation rules.

Block structure was introduced into computer programming languages by the Algol project (1957–1960), which, as a consequence, also featured a context-free grammar to describe the resulting Algol syntax. This became a standard feature of computer languages, and the notation for grammars used in concrete descriptions of computer languages came to be known as Backus-Naur form, after two members of the Algol language design committee. The "block structure" aspect that context-free grammars capture is so fundamental to grammar that the terms syntax and grammar are often identified with context-free grammar rules, especially in computer science. Formal constraints not captured by the grammar are then considered to be part of the "semantics" of the language.

## 1.2 Modern Days Uses

Context free grammars have many uses in computer science and in our everyday life, here is a list of several applications of the context free grammars in our world:

- The formal definition means that context free grammars are computationally traceable, it is possible to write a computer program which determines whether sentences are grammatical or not.

- One of the common uses of grammars is to describe the structure of programming languages.
- They are used in an essential part of the Extensible Markup Language (XML) called the Document Type Definition.
- In most programming languages opening and closing of braces, curly brackets is taken care. It mainly track it and if any closing bracket is not there, it will throw error.
- They provide a convenient visual notation for the structure of sentences (the tree).

## 2 Formal Definition

### 2.1 Grammar Definition

Context Free Grammars, or in abbreviation CFG can be described as follows: A context free-grammar  $G$  is defined by the 4-tuple:

$$G = \{V, \Sigma, P, S\} \text{ where}$$

1.  $V$  is a finite set; each element  $v \in V$  is called a nonterminal character or a variable. Each variable represents a different type of phrase or clause in the sentence. Variables are also sometimes called syntactic categories. Each variable defines a sub-language of the language defined by  $G$ .
2.  $\Sigma$  is a finite set of terminals, disjoint from  $V$ , which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar  $G$ .
3.  $P$  is a set of rewrite rules of the form  $\alpha \rightarrow \beta$  where  $\alpha \in V$  and  $\beta \in (V \cup \Sigma)^*$ , where the asterisk represents the Kleene star operation. The members of  $P$  are called the (rewrite) rules or productions of the grammar.
4.  $S$  is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of  $V$ .

### 2.2 Production rule notation

A production rule in  $P$  is formalized mathematically as a pair  $(\alpha, \beta) \in P$ , where  $\alpha \in V$  is a nonterminal and  $\beta \in (V \cup \Sigma)^*$  is a string of variables and/or terminals; rather than using ordered pair notation, production rules are usually written using an arrow operator with  $\alpha$  as its left hand side and  $\beta$  as its right hand side:

$$\alpha \rightarrow \beta$$

It is allowed for  $\beta$  to be the empty string, and in this case it is customary to denote it by  $\epsilon$ . The form  $\alpha \rightarrow \epsilon$  is called an  $\epsilon$ -production.

It is common to list all right-hand sides for the same left-hand side on the same line, using  $|$  (the pipe symbol) to separate them. Rules  $\alpha \rightarrow \beta_1$  and  $\alpha \rightarrow \beta_2$  can hence be written as  $\alpha \rightarrow \beta_1 \mid \beta_2$ . In this case,  $\beta_1$  and  $\beta_2$  is called the first and second alternative, respectively.

## 2.3 Rule application

For any strings  $u, v \in (V \cup \Sigma)^*$ , we say  $u$  directly yields  $v$ , written as  $u \Rightarrow v$ , if  $\exists (\alpha, \beta) \in P$  with  $\alpha \in V$  and  $u_1, u_2 \in (V \cup \Sigma)^*$  such that  $u = u_1 \alpha u_2$  and  $v = u_1 \beta u_2$ . Thus,  $v$  is a result of applying the rule  $(\alpha, \beta)$  to  $u$ .

## 2.4 Repetitive rule application

For any strings  $u, v \in (V \cup \Sigma)^*$ , we say  $u$  yields  $v$ , written as  $u \xRightarrow{*} v$ , if  $\exists k \geq 1 \mid \exists u_1, \dots, u_k \in (V \cup \Sigma)^*$  such that  $u = u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k = v$ . In this case, if  $k \geq 2$  (i.e.,  $u \neq v$ ), the relation  $u \xRightarrow{+} v$  holds. In other words,  $(\xRightarrow{*})$  and  $(\xRightarrow{+})$  are the reflexive transitive closure (allowing a word to yield itself) and the transitive closure (requiring at least one step) of  $(\Rightarrow)$ , respectively.

## 2.5 Context-free language

The language of a grammar  $G = (V, \Sigma, P, S)$  is the set

$$L(G) = \{w \in \Sigma^* : S \xRightarrow{*} w\}$$

A language  $L$  is said to be a context-free language (CFL), if there exists a CFG  $G$ , such that  $L = L(G)$ .

As taught in the course but will not be used during this work, non-deterministic pushdown automata recognize exactly the context-free languages.

# 3 Examples of CFGs

## 3.1 Words concatenated with their reverse

The grammar  $G = (\{S\}, \{a, b\}, P, S)$ , with productions

$$S \rightarrow aSa$$

$$S \rightarrow bSb$$

$$S \rightarrow \epsilon$$

is context-free. It also includes an  $\varepsilon$ -production. A typical derivation in this grammar is:

$$S \rightarrow aSa \rightarrow aaSaa \rightarrow aabSbaa \rightarrow aabbbaa.$$

This makes it clear that  $L(G) = \{ww^R : w \in \{a, b\}^*\}$ .<sup>1</sup>

### 3.2 Well-formed parentheses

The canonical example of a context-free grammar is parenthesis matching, which is representative of the general case. There are two terminal symbols "(" and ")" and one nonterminal symbol  $S$ . The production rules are:

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow ()$$

The first rule allows the  $S$  symbol to multiply; the second rule allows the  $S$  symbol to become enclosed by matching parentheses; and the third rule terminates the recursion.

### 3.3 Matching pairs

In a context-free grammar, we can pair up characters the way we do with brackets. The simplest example:

$$S \rightarrow aSb$$

$$S \rightarrow ab$$

This grammar generates the language  $\{a^n b^n : n \geq 1\}$ , which is not regular (according to the pumping lemma for regular languages).

The special character  $\varepsilon$  stands for the empty string. By changing the above grammar to:

$$S \rightarrow aSb | \varepsilon$$

we obtain a grammar generating the language  $\{a^n b^n : n \geq 0\}$  instead. This differs only in that it contains the empty string while the original grammar did not.

---

<sup>1</sup>If the productions  $S \rightarrow a$ ,  $S \rightarrow b$ , are added, a context-free grammar for the set of all palindromes over the alphabet  $\{a, b\}$  is obtained.

### 3.4 Distinct number of a's and b's

A context-free grammar for the language consisting of all strings over  $\{a, b\}$  containing an unequal number of  $a$ 's and  $b$ 's:

$$\begin{aligned}S &\rightarrow U|V \\U &\rightarrow TaU|TaT|UaT \\V &\rightarrow TbV|TbT|VbT \\T &\rightarrow aTbT|bTaT|\varepsilon\end{aligned}$$

Here, the nonterminal  $T$  can generate all strings with the same number of  $a$ 's as  $b$ 's, the nonterminal  $U$  generates all strings with more  $a$ 's than  $b$ 's and the nonterminal  $V$  generates all strings with fewer  $a$ 's than  $b$ 's. Omitting the third alternative in the rule for  $U$  and  $V$  doesn't restrict the grammar's language.

## 4 Derivations and syntax trees

A derivation of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs to the grammar's language. A derivation is fully determined by giving, for each step:

- the rule applied in that step
- the occurrence of its left-hand side to which it is applied

For clarity, the intermediate string is usually given as well. For instance, with the grammar:

1.  $S \rightarrow S + S$
2.  $S \rightarrow 1$
3.  $S \rightarrow a$

The string:

$1 + 1 + a$

can be derived with the derivation:

$S \rightarrow$  (rule 1 on the first  $S$ )

$S + S \rightarrow$  (rule 1 on the second  $S$ )

$S + S + S \rightarrow$  (rule 2 on the second  $S$ )

$S + 1 + S \rightarrow$  (rule 3 on the third  $S$ )

$S + 1 + a \rightarrow$  (rule 2 on the first  $S$ )

$1 + 1 + a$

Often, a strategy is followed that deterministically determines the next non-terminal to rewrite:



- in a leftmost derivation, it is always the leftmost nonterminal;
- in a rightmost derivation, it is always the rightmost nonterminal.

Given such a strategy, a derivation is completely determined by the sequence of rules applied. For instance, the leftmost derivation

$S \rightarrow$  (rule 1 on the first  $S$ )  
 $S + S \rightarrow$  (rule 2 on the first  $S$ )  
 $1 + S \rightarrow$  (rule 1 on the first  $S$ )  
 $1 + S + S \rightarrow$  (rule 2 on the first  $S$ )  
 $1 + 1 + S \rightarrow$  (rule 3 on the first  $S$ )  
 $1 + 1 + a$

can be summarized as:

rule1, rule2, rule1, rule2, rule3

The corresponding rightmost derivation is:

$S \rightarrow$  (rule 1 on the rightmost  $S$ )  
 $S + S \rightarrow$  (rule 1 on the rightmost  $S$ )  
 $S + S + S \rightarrow$  (rule 3 on the first  $S$ )  
 $S + S + a \rightarrow$  (rule 2 on the first  $S$ )  
 $S + 1 + a \rightarrow$  (rule 2 on the first  $S$ )  
 $1 + 1 + a$

can be summarized as:

rule1, rule1, rule3, rule2, rule2

The distinction between leftmost derivation and rightmost derivation is important because in most parsers the transformation of the input is defined by giving a piece of code for every grammar rule that is executed whenever the rule is applied. Therefore, it is important to know whether the parser determines a leftmost or a rightmost derivation because this determines the order in which the pieces of code will be executed.

A derivation also imposes in some sense a hierarchical structure on the string that is derived. For example, if the string "1 + 1 + a" is derived according to the leftmost derivation the structure of the string would be:

$$\{\{1\}_S + \{\{1\}_S + \{a\}_S\}_S\}_S$$

where  $\{\dots\}_S$  indicates a substring recognized as belonging to  $S$ . This hierarchy can also be seen as a tree.

Note however that both parse trees can be obtained by both leftmost and rightmost derivations. For example, the last tree can be obtained with the leftmost derivation as follows:

$$\begin{aligned}
 S &\rightarrow S + S \text{ (1)} \\
 &\rightarrow S + S + S \text{ (1)} \\
 &\rightarrow 1 + S + S \text{ (2)}
 \end{aligned}$$

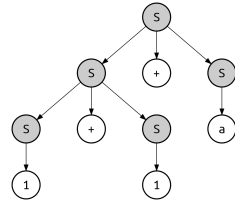


Figure 1: Example of leftmost tree

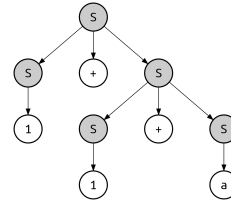


Figure 2: Example of rightmost tree

$$\begin{aligned} &\rightarrow 1 + 1 + S \text{ (2)} \\ &\rightarrow 1 + 1 + a \text{ (3)} \end{aligned}$$

If a string in the language of the grammar has more than one parsing tree, then the grammar is said to be an ambiguous grammar. Such grammars are usually hard to parse because the parser cannot always decide which grammar rule it has to apply. Usually, ambiguity is a feature of the grammar, not the language, and an unambiguous grammar can be found that generates the same context-free language. However, there are certain languages that can only be generated by ambiguous grammars; such languages are called inherently ambiguous languages.

## 5 Chomsky Normal Form - CNF

### 5.1 Definition

In formal language theory, a context-free grammar  $G$  is said to be in Chomsky normal form (first described by Noam Chomsky)[3] if all of its production rules are of the form:

$$\begin{aligned} &A \rightarrow BC \\ &\textbf{OR} \\ &A \rightarrow a \\ &\textbf{OR} \\ &S \rightarrow \varepsilon \end{aligned}$$

where  $A$ ,  $B$ , and  $C$  are nonterminal symbols,  $a$  is a terminal symbol (a symbol that represents a constant value),  $S$  is the start symbol, and  $\varepsilon$  denotes the empty string. Also, neither  $B$  nor  $C$  may be the start symbol, and the third production rule can only appear if  $\varepsilon$  is in  $L(G)$ , the language produced by the context-free grammar  $G$ .

Every grammar in Chomsky normal form is context-free, and conversely, every context-free grammar can be transformed into an equivalent one<sup>2</sup> which

<sup>2</sup>That is, one that produces the same language

is in Chomsky normal form and has a size no larger than the square of the original grammar's size.

## 5.2 Converting a grammar to Chomsky normal form

To convert a grammar to Chomsky normal form, a sequence of simple transformations is applied in a certain order; this is described in most textbooks on automata theory. The presentation here follows Hopcroft, Ullman (1979)[1], but is adapted to use the transformation names from Lange, Leiß (2009)[2]<sup>3</sup>. Each of the following transformations establishes one of the properties required for Chomsky normal form.

## 5.3 START: Eliminate the start symbol from right-hand sides

Introduce a new start symbol  $S_0$ , and a new rule:

$$S_0 \rightarrow S$$

where  $S$  is the previous start symbol. This does not change the grammar's produced language, and  $S_0$  will not occur on any rule's right-hand side.<sup>4</sup>

## 5.4 TERM: Eliminate rules with nonsolitary terminals

To eliminate each rule:

$$A \rightarrow X_1 \dots a \dots X_n$$

with a terminal symbol  $a$  being not the only symbol on the right-hand side, introduce, for every such terminal, a new nonterminal symbol  $N_a$ , and a new rule:

$$N_a \rightarrow a$$

Change every rule:

$$A \rightarrow X_1 \dots a \dots X_n$$

to

$$A \rightarrow X_1 \dots N_a \dots X_n$$

If several terminal symbols occur on the right-hand side, simultaneously replace each of them by its associated nonterminal symbol. This does not change the grammar's produced language.

## 5.5 BIN: Eliminate right-hand sides with more than 2 nonterminals

Replace each rule:

$$A \rightarrow X_1 X_2 \dots X_n$$

with more than 2 nonterminals  $X_1, \dots, X_n$  by rules

$$A \rightarrow X_1 A_1,$$

---

<sup>3</sup>For example, Hopcroft, Ullman (1979) merged TERM and BIN into a single transformation.

<sup>4</sup>This step is only required if the original start symbol  $S$  is in any of the rule's right hand side.

$A_1 \rightarrow X_2 A_2,$

$\dots,$

$A_{n-2} \rightarrow X_{n-1} X_n,$

where  $A_i$  are new nonterminal symbols. Again, this does not change the grammar's produced language.[1]

## 5.6 DEL: Eliminate $\varepsilon$ -rules

An  $\varepsilon$ -rule is a rule of the form

$A \rightarrow \varepsilon,$

where  $A$  is not  $S_0$ , the grammar's start symbol.

To eliminate all rules of this form, first determine the set of all nonterminals that derive  $\varepsilon$ . Hopcroft and Ullman (1979) call such nonterminals nullable, and compute them as follows:

If a rule  $A \rightarrow \varepsilon$  exists, then  $A$  is nullable.

If a rule  $A \rightarrow X_1 \dots X_n$  exists, and every single  $X_i$  is nullable, then  $A$  is nullable, too.

Obtain an intermediate grammar by replacing each rule

$A \rightarrow X_1 \dots X_n$

by all versions with some nullable  $X_i$  omitted. By deleting in this grammar each  $\varepsilon$ -rule, unless its left-hand side is the start symbol, the transformed grammar is obtained.[1]

For example, in the following grammar, with start symbol  $S_0$ ,

$S_0 \rightarrow AbB|C$

$B \rightarrow AA|AC$

$C \rightarrow b|c$

$A \rightarrow a|\varepsilon$

the nonterminal  $A$ , and hence also  $B$ , is nullable, while neither  $C$  nor  $S_0$  is. Hence the following intermediate grammar is obtained:<sup>5</sup>

$S_0 \rightarrow AbB|\cancel{A}bB|Ab\cancel{B}|\cancel{A}b\cancel{B}|C$

$B \rightarrow AA|\cancel{A}A|A\cancel{A}|\cancel{A}\varepsilon A|AC|\cancel{A}C$

$C \rightarrow b|c$

$A \rightarrow a|\varepsilon$

In this grammar, all  $\varepsilon$ -rules have been "inlined at the call site".<sup>6</sup> In the next step, they can hence be deleted, yielding the grammar:

$S_0 \rightarrow AbB|Ab|bB|b|C$

$B \rightarrow AA|A|AC|C$

$C \rightarrow b|c$

<sup>5</sup>indicating a kept and omitted nonterminal  $N$  by  $N$  and  $\cancel{N}$ , respectively

<sup>6</sup>If the grammar had a rule  $S_0 \rightarrow \varepsilon$ , it could not be "inlined", since it had no "call sites". Therefore it could not be deleted in the next step.

$$A \rightarrow a$$

This grammar produces the same language as the original example grammar, viz.  $\{ab, aba, abaa, abab, abac, abb, abc, b, bab, bac, bb, bc, c\}$ , but has no  $\varepsilon$ -rules.

## 5.7 UNIT: Eliminate unit rules

A unit rule is a rule of the form

$$A \rightarrow B,$$

where  $A, B$  are nonterminal symbols. To remove it, for each rule  $B \rightarrow X_1 \dots X_n$ ,

where  $X_1 \dots X_n$  is a string of nonterminals and terminals, add rule

$A \rightarrow X_1 \dots X_n$  unless this is a unit rule which has already been (or is being) removed.

## 5.8 Order of transformations

When choosing the order in which the above transformations are to be applied, it has to be considered that some transformations may destroy the result achieved by other ones. For example, **START** will re-introduce a unit rule if it is applied after **UNIT**. The table shows which orderings are admitted.

Figure 3: Mutual preservation of transformation results

Transformation $X$ always preserves (✓) resp. may destroy (✗) the result of $Y$ :					
$X \backslash Y$	START	TERM	BIN	DEL	UNIT
START		✓	✓	✗	✗
TERM	✓		✗	✓	✓
BIN	✓	✓		✓	✓
DEL	✓	✓	✓		✗
UNIT	✓	✓	✓	(✓)*	
*UNIT preserves the result of DEL if START had been called before.					

Picture taken from xxx

Moreover, the worst-case bloat in grammar size<sup>7</sup> depends on the transformation order. Using  $|G|$  to denote the size of the original grammar  $G$ , the

<sup>7</sup>i.e. written length, measured in symbols

size blow-up in the worst case may range from  $|G|^2$  to  $2^{2|G|}$ , depending on the transformation algorithm used.[2] The blow-up in grammar size depends on the order between **DEL** and **BIN**. It may be exponential when **DEL** is done first, but is linear otherwise. **UNIT** can incur a quadratic blow-up in the size of the grammar.[2] The orderings **START**, **TERM**, **BIN**, **DEL**, **UNIT** and **START**, **BIN**, **DEL**, **UNIT**, **TERM** lead to the least (i.e. quadratic) blow-up.

## 6 CYK Algorithm

In computer science, the **Cocke-Younger-Kasami** algorithm is a parsing algorithm for context-free grammars, named after its inventors, John Cocke, Daniel Younger and Tadao Kasami. It employs bottom-up parsing and dynamic programming.

The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF). However any context-free grammar may be transformed to a CNF grammar expressing the same language[4].

The importance of the CYK algorithm stems from its high efficiency in certain situations. Using Big O notation, the worst case running time of CYK is  $\mathcal{O}(n^3 \cdot |G|)$ , where  $n$  is the length of the parsed string and  $|G|$  is the size of the CNF grammar  $G$ . [1] This makes it one of the most efficient parsing algorithms in terms of worst-case asymptotic complexity, although other algorithms exist with better average running time in many practical scenarios.

## 6.1 Pseudo Code

The code can be described as follows:

---

**Algorithm 1** CYK Algorithm

---

**Require:** String  $w$ , CFG  $G$

**Ensure:**  $w \in L(G)$

**let** the input be a string  $w$  consisting of  $n$  characters:  $a_1 \dots a_n$ .

**let** the grammar  $G$  be in CNF form

**let** the  $G$  contain  $r$  nonterminal symbols  $R_1 \dots R_r$ , with start symbol  $R_1$ .

**let**  $P[n, n, r]$  be three dimensional array of booleans. Initialize all elements of  $P$  to false

Comment: Starting with checking only length 1

**for** Each  $s = 1$  to  $n$  **do**

**for** Each unit production  $R_v \rightarrow a_v$  **do**

$P[1, s, v] \leftarrow true$

**end for**

**end for**

Comment: Using the length 1 calculation to reach all other possible lengths

**for** Each  $l = 2$  to  $n$  (Length of span) **do**

**for** Each  $s = 1$  to  $n-l+1$  (Start of span) **do**

**for** Each  $p = 1$  to  $l - 1$  (Partition of span) **do**

**for** Each production  $R_a \rightarrow R_b R_c$  **do**

**if**  $P[p, s, b]$  and  $P[l - p, s + p, c]$  **then**

$P[l, s, a] \leftarrow true$

**end if**

**end for**

**end for**

**end for**

**end for**

**if**  $P[n, 1, 1]$  is true **then**

$W$  is member of language ( $W \in L(G)$ )

**else**

$W$  is not member of language ( $W \notin L(G)$ )

**end if**

---

Note that the algorithm described here only accepts context free grammars in CNF, however as described in the previous algorithms every grammar can be transformed into CNF grammar which share the same language. Also this algorithm is only a "Recognizer" algorithm, which means he can only identify whether a word is part of the grammar language or not, it can not parse the word and display the rules applied to get from the start symbol  $S_0$  to the word  $W$ .



## 6.2 Improvements to the algorithm

### 6.2.1 Relaxing of the grammar form

The membership and parsing problems for context-free languages (CFL) are of major importance in compiler design, bioinformatics, and computational linguistics. The algorithm due to Cocke, Younger and Kasami, often called the CYK algorithm, is the most well-known and therefore commonly taught algorithm that solves the word problem for context-free grammars (CFG), and with a minor extension, the parsing problem as well. It is also a very prominent example for an algorithm using dynamic programming, a design principle for recursive algorithms that become efficient by storing the values of intermediate calculations. It is therefore fair to say that the CYK algorithm is one of the most important algorithms in undergraduate syllabi<sup>8</sup> for students in subjects like computer science, (bio)informatics, and computational linguistics.

One would expect that textbooks and course notes reflect this importance by good presentations of efficient versions of CYK. However, this is not the case. In particular, many textbooks present the necessary transformation of a context-free grammar into Chomsky normal form (CNF) in a suboptimal way leading to an avoidable exponential blow-up of the grammar. They neither explain nor discuss whether this can be avoided or not. Complexity analyses often concern the actual CYK procedure only and neglect the pretransformation. Some of the textbooks give vague reasons for choosing CNF, indicating ease of presentation and proof. A few state that CNF is chosen to achieve a cubic running time, not mentioning that only the restriction to binary rules is responsible for that.

In the literature on parsing, one can find a number of variations of CYK that differ in the restrictions on the input grammar. The most liberal one that suffices to make CYK run in time  $O(|w|^3)$  is that the grammar is bilinear, i.e. in each rule  $A \rightarrow \alpha$  there are at most two nonterminal occurrences in  $\alpha$ , see the so-called C-parser [7]. Others relax the CNF restriction by admitting unit rules, i.e. rules  $A \rightarrow \alpha$  where  $\alpha$  is a nonterminal [8].

In their paper [2], Martin Lange and Hans Leiß describe all the suggested grammar normal forms in the literature and their hierarchy.

---

<sup>8</sup>Unfortunately CYK algorithm is not part of the syllabi in Automata course taught in ORT Braude College

Figure 4: Grammars normal form table

Name	Rule format
CNF <sup>-ε</sup>	$A \rightarrow BC \mid a$
CNF	$A \rightarrow BC \mid a, S \rightarrow \varepsilon$
CNF <sup>+ε</sup>	$A \rightarrow BC \mid a \mid \varepsilon$
S2F	$A \rightarrow \alpha \text{ where }  \alpha  = 2$
C2F	$A \rightarrow BC \mid B \mid a, S \rightarrow \varepsilon$
2NF	$A \rightarrow \alpha \text{ where }  \alpha  \leq 2$
2LF	$A \rightarrow uBvCw \mid uBv \mid v$

Figure 5: Grammars hierarchy

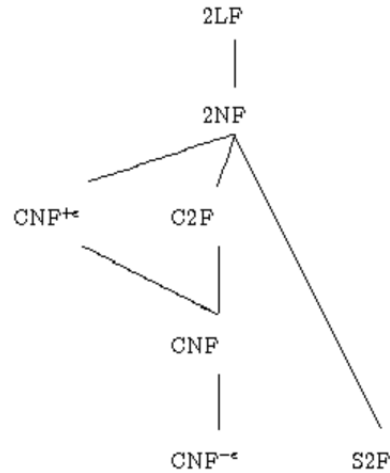


Figure 4 defines the normal forms that are being used in textbooks and shows how they relate to each other. In the column "rule format",  $A, B, C$  are arbitrary nonterminals,  $S$  is the start symbol that may not occur on a right-hand side,  $a$  is a terminal symbol,  $u, v, w$  are strings of terminal symbols, and  $\alpha$  is a sentential form.

#### Which normal form to choose?

All normal forms mentioned in Figure 4 allow for a time complexity of CYK that is cubic in the size of the input word and linear in the size of the grammar. Therefore, the question which of the possible normal forms to choose ought to be determined by two factors:

- (i) complexity of the grammar transformation, in particular the size of the transformed grammar.
- (ii) ease of presentation and proofs.

The advantage of using 2NF or C2F over CNF in CYK is not reflected well in the literature. According to Martin Lange and Hans Leiß they are only aware of one textbook which presents the CYK algorithm for grammars in C2F - a specialised book on parsing [8] - and a PhD thesis [9]. It seemed to have been unnoticed until Martin Lange and Hans Leiß paper that one can use even 2NF instead of C2F. They consider 2NF as an advantage over C2F, since not only unit rules, but also deletion rules are convenient in grammar writing; the latter are often used as a means to admit optional constituents.

Therefore an 2NF may be a superior form because in comparison to other normal forms, the time required to convert a CFG to 2NF form is linear in the grammar size, while in all other forms (except for C2F) the time required is cubic. And on the other factor the proofs and the grammar appearance are similar to those of the CNF, with some minor adjustments.

### 6.2.2 Parser Algorithm

The algorithm described in sub-section 6.1 is only a "Recognizer" algorithm, and sometimes it is required not only to know if a word is part of the grammar language, but also to know the steps and rules used in order to reach that word with the given grammar. We call such algorithm as "Parser Algorithm".

CYK can become "Parser Algorithm" with some minor adjustments:

- Change the table to be as follows:  $T[n, n]$ , where  $n$  is the length of the word, and  $T[i, j]$  contains all the non-terminals that can derive that sub-string of the word starting at index  $i$  with length  $j$ .
- For example, given the input  $w = 'abb'$  and the following grammar:

$$S \rightarrow A|B$$

$$A \rightarrow a$$

$$B \rightarrow b$$

we will get that  $T[1, 1] = \{A\}$ ,  $T[2, 1] = T[3, 1] = \{B\}$

- In order to backtrack the algorithm and find the rules used at any given point it is also necessary to contain backpointers of the rules used.
- We can then obtain a parse tree by traversing the productions from left to right, starting with every  $S$ -production in  $T[1, n]$ .

#### Example:

I will describe the appearance of the table, given the following input:

String  $w = aaabbb$

CFG  $G$  (In CNF Form):

$$S \rightarrow AB|AD$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$D \rightarrow SB$$

$S \rightarrow A_{1,1}D_{2,5}$					
	$D \rightarrow S_{2,4}B_{6,1}$				
	$S \rightarrow A_{2,1}D_{3,3}$				
		$D \rightarrow S_{3,2}B_{5,1}$			
		$S \rightarrow A_{3,1}B_{4,1}$			
$A \rightarrow a$	$A \rightarrow a$	$A \rightarrow a$	$B \rightarrow b$	$B \rightarrow b$	$B \rightarrow b$

Now that we calculated every cell in the table we can backtrack and find the rules used in order to reach the given word, in this example it is very easy to backtrack because there is no ambiguity.

The rules are (Using left to right):

$$S \rightarrow AD \rightarrow aD \rightarrow aSB \rightarrow aADB \rightarrow aaDB \rightarrow aaSBB \rightarrow aaABBB \rightarrow aaaBBB \Rightarrow aaabbb$$

## 7 CYK using grammar in 2NF

As mentioned in the previous section CYK algorithm can be improved by allowing it to work on grammars in 2NF, in this section I will describe the additions needed to be made in order for CYK to work on grammars using 2NF. The advantages however are clear, converting any CFG to CNF may cause the size of the grammar to increase exponentially, while to convert the same grammar to 2NF you only need to apply **BIN** rule, which in worst case scenario will cause a linear increase in size, therefore  $O(G') = O(G)$ .

### 7.1 Calculate Nullable

Let Nullable be the set of all the grammar variables that can be derived to  $\varepsilon$ , we will mark it using  $\mathcal{E}_G$ . This set can be calculated using  $O(G)$  time, i.e linear time complexity. Here is an example of the algorithm that can be used, given a grammar  $G$  in 2NF:

```
Nullable( $G$ ) =  
1      nullable :=  $\emptyset$   
2      todo :=  $\emptyset$   
3      for all  $A \in N$  do  
4          occurs( $A$ ) :=  $\emptyset$   
5      for all  $A \rightarrow B$  do  
6          occurs( $B$ ) := occurs( $B$ )  $\cup$   $\{A\}$   
7      for all  $A \rightarrow BC$  do  
8          occurs( $B$ ) := occurs( $B$ )  $\cup$   $\{\langle A, C \rangle\}$   
9          occurs( $C$ ) := occurs( $C$ )  $\cup$   $\{\langle A, B \rangle\}$   
10     for all  $A \rightarrow \varepsilon$  do  
11         nullable := nullable  $\cup$   $\{A\}$   
12         todo := todo  $\cup$   $\{A\}$   
13     while todo  $\neq \emptyset$  do  
14         remove some  $B$  from todo  
15         for all  $A, \langle A, C \rangle \in$  occurs( $B$ ) with  $C \in$  nullable do  
16             if  $A \notin$  nullable then  
17                 nullable := nullable  $\cup$   $\{A\}$   
18                 todo := todo  $\cup$   $\{A\}$   
19     return nullable
```

Figure 6: Algorithm Nullable for the linear-time computation of  $\mathcal{E}_G$ .

## 7.2 Constructing the Unit Relation

Using the now computed  $\mathcal{E}_G$ , we can also compute the unit relation efficiently. Let  $G$  be a grammar in 2NF. The unit relation can be described as follows: for every rule  $A \rightarrow y$ ,  $A \rightarrow By$  and  $A \rightarrow yB$  with  $B \in \mathcal{E}_G$  we will get  $(A, y) \in U_G$ . The inverse relation will be  $\tilde{U}_G = \{(y, A) | (A, y) \in U_G\}$ . We view  $\tilde{U}_G$  as a relation on  $V$  and call  $I_G = (V, \tilde{U}_G)$  the inverse unit graph of  $G$ . The inverse unit relation can be calculated in  $O(G)$  time, by simply going over all rules and adding the edge  $(A, y)$  for the appropriate rules as described above.

## 7.3 Calculate $\tilde{U}_G^*(M)$

$\tilde{U}_G^*(M)$  consists of all nodes  $x \in V$  that are reachable in  $I_G$  from some node  $y \in M$ . The set of all nodes in a graph reachable from a given set of nodes can be computed in time and space  $O(n + m)$  where  $n$  is the number of all nodes and  $m$  the number of all edges. This simply uses depth- or breadth-first search[10].

## 7.4 Algorithm

Input is CFG  $G$  in 2NF, and it's inverse graph  $I_G$  <sup>9</sup>

```

CYK( $G, \tilde{U}_G, w$ ) =
1   for  $i = 1, \dots, n$  do
2        $T_{i,i} := \tilde{U}_G^*(\{a_i\})$ 
3   for  $j = 2, \dots, n$  do
4       for  $i = j - 1, \dots, 1$  do
5            $T'_{i,j} := \emptyset$ 
6           for  $h = i, \dots, j - 1$  do
7               for all  $A \rightarrow yz$ 
8                   if  $y \in T_{i,h}$  and  $z \in T_{h+1,j}$  then
9                        $T'_{i,j} := T'_{i,j} \cup \{A\}$ 
10           $T_{i,j} := \tilde{U}_G^*(T'_{i,j})$ 
11   if  $S \in T_{1,n}$  then return yes else return no

```

Figure 7: Algorithm CYK adjusted for grammars in 2NF

<sup>9</sup>Can be changed such that we calculate  $I_G$  in linear time inside the algorithm itself

## 8 Implementation

In my assignments I was tasked with using and implementing a program that will receive CFG and a word, and return if the word can be derived from the CFG, and if so also display the derivation sequence of the word using the grammar rules. Before I could jump to implementation I had to search and understand all the fundamentals regarding the task at hand. Therefore in the previous sections I have discussed all required steps in order to complete and fulfill the task.

First I needed to understand how to describe a context free grammar and what it represents clearly, in order to create an object that will behave the same as context free grammar, and also to check the validity of the given input according to the formal definition described.

Next I will have to use the CYK algorithm in order to recognize if the given word is part of the grammar language, however the first step in using CYK algorithm is to transform the grammar into an equivalent grammar in CNF. I have decided to go by the following transformation order: **START**, **DEL**, **UNIT**, **TERM** and **BIN** (As discussed previously this order always preserve the result of the transformation).

Now that my grammar is in CNF, I can apply the CYK algorithm, therefore I had to implement it using the pseudo code I have described in the previous section. And also I extended the algorithm to be more than just recognizer but also a parser algorithm, which means that now I can not only check if a word is part of the grammar language, but also to display a possible derivation sequence of the word using the grammar rules.

In order to create unit tests for my code, I was also required to create an exponential algorithm that will run on the original grammar and the result will be a subset of the grammar language with words that generated using predefined finite number of rewrite rules. Using this algorithm with a very large constant number will allow me to check if a word can be derived from the grammar with very high accuracy. Therefore it can be used a way to test the implementation of the CYK algorithm.

### 8.1 Breakdown of tasks

The following tasks will have to be completed:

1. Create an object that represent CFG.
2. Convert a context free grammar to chomsky normal form.
3. Check if a word is part of the grammar language.
4. Find a derivation sequence for the word, if possible.
5. Write exponential algorithm to find all the words in the grammar language, that can be derived from the start symbol  $S_0$  with up to  $M$  rules applied.

## 8.2 Implementation language and environment

In order to complete the assignment I was given free hand to choose between any programming language and development environment in order to implement my solution. When deciding which language to use I focused on the following parameters:

- Familiarity with the language, while it is great and always fun to learn new programming languages, this project was not given in order to advance my knowledge regarding programming languages. I need to use code that I am familiar with and have easier time to implement the given algorithms.
- Rich libraries that allows implementing complicated algorithms with ease.
- Coding approach that seem suitable for easier implementation of the tasks.

After weighing my options I have decided to use **Java** programming language, using eclipse IDE (Integrated development environment). Because the language is object oriented which in my opinion will offer great value when trying to implement the algorithms. Also I am quite familiar with one year of experience using it.

### 8.3 Data Structures

First of all I would need data structure to contain any context free grammar, because I use Java, it should be implemented as an object.

Figure 8: Core variables used to in order to implement  $G = \{V, \Sigma, P, S\}$

```
private Set<String> variableSet; // V
private Set<Character> terminalsSet; // Sigma
private Multimap<String, String> rewriteRules; // P
private String S0; // Start variable
```

- **V implementation** - I have decided that the non-terminal variables  $V$  shall be stored as a set of strings, that is to avoid duplication of non-terminal elements. And also to allow for non-terminals with length greater or equal to 1.<sup>10</sup>
- **$\Sigma$  implementation** - For similar reasons I have decided that  $\Sigma$  should be represented by set object, however this is a set of characters to indicate that the length must be 1.
- **P implementation** - I have decided to implement the rewrite rules, as a Multi-map<sup>11</sup> where every string is a key to multiple other strings. The keys of the multi-map are  $\forall v \in V$ , and the values are all the rules that are applicable to that specific non-terminal.
- **Start Symbol -  $S_0$  implementation** - This is simply a string that should be part of the variable set and represent the starting symbol.

As proof of concept I have decided to implement all the related methods inside the object itself, however it should be noted that proper coding standards may require several of those function to be inside a different object that offers will offer API for context free grammar operations.

### 8.4 Program API - Application Program Interface

As taken from the application generated javadoc, the object will contain the following methods for all the operations necessary to complete the given task.

For further explanation regarding the implementation the source code will be attached with this document, I think most of the function are rather self explanatory.

<sup>10</sup>For example it will allow me to use combination of letters and numerical to represent non terminals

<sup>11</sup>Using google guava package 28.2



Figure 9: CFG Object API

Method and Description
<code>breakLongRules()</code> Break long rules.
<code>breakToVarArray(java.lang.String varOnlyRule)</code> Break to var array.
<code>cleanNullCharacters()</code> Clean null characters.
<code>covertToChomanskyNormalForm()</code> Covert to chomansky normal form.
<code>findParseOfWord(java.lang.String w)</code> Find parse of word.
<code>generateAllWordWithLength(int length)</code> Generate all word with length.
<code>getAllBreakRules(java.lang.String rule)</code> Gets the all break rules.
<code>handleInitialNonTerminal()</code> Handle initial non terminal.
<code>isTerminal(java.lang.String s)</code> Checks if is terminal.
<code>isValidGrammar()</code> Few simple checks if the grammar is valid or not.
<code>isVariable(java.lang.String s)</code> Checks if is variable.
<code>removedTerminalNonTerminalMix()</code> Removed terminal non terminal mix.
<code>removeNullableVariables()</code> Removes the nullable variables.
<code>removeUnderiableVariables()</code> Removes the underiable variables.
<code>removeUnitProductions()</code> Removes the unit productions.
<code>removeUselessProductionVariables()</code> Removes the useless production variables.
<code>simplifyGrammar()</code> Simplify grammar.
<code>toString()</code>

## 9 Input and output results of the code

In order to display the correctness of the written code I have come up with several input sets to make sure the code work as intended. Most of the results are given in the console, so it can not be tested automatically, however it is good enough for "black box" testing.

### 9.1 Grammar conversion

#### 9.1.1 Test 1 - Conversion success

The input grammar is:

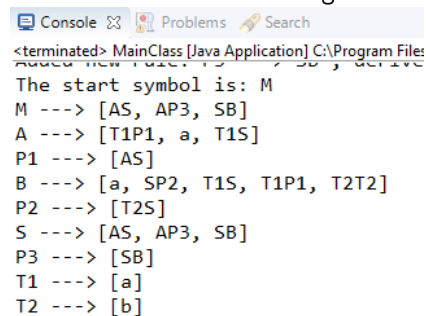
$$\begin{aligned} S &\rightarrow ASB \\ A &\rightarrow aAS \mid a \mid \varepsilon \\ B &\rightarrow SbS \mid A \mid bb \end{aligned}$$

The expected result should be the following grammar: <sup>12</sup>

$$\begin{aligned} S_0 &\rightarrow AS \mid PB \mid SB \\ S &\rightarrow AS \mid QB \mid SB \\ A &\rightarrow RS \mid XS \mid a \\ B &\rightarrow TS \mid VV \mid US \mid XS \mid a \\ X &\rightarrow a \\ Y &\rightarrow b \\ V &\rightarrow b \\ P &\rightarrow AS \\ Q &\rightarrow AS \\ R &\rightarrow XA \\ T &\rightarrow SY \\ U &\rightarrow XA \end{aligned}$$

The actual result is:

Figure 10: Test 1 actual result



```
<terminated> MainClass [Java Application] C:\Program Files
Added new rule: S -> AS, SB, derive
The start symbol is: M
M ---> [AS, AP3, SB]
A ---> [T1P1, a, T1S]
P1 ---> [AS]
B ---> [a, SP2, T1S, T1P1, T2T2]
P2 ---> [T2S]
S ---> [AS, AP3, SB]
P3 ---> [SB]
T1 ---> [a]
T2 ---> [b]
```

<sup>12</sup>The input and output was taken from geeksforgeeks

While on first glance it may not seem as if we got the correct result, upon further investigation it is indeed the exact same grammar, the difference is perhaps because of different order of CNF conversion as described in section 5 regarding the CNF conversion steps.

### 9.1.2 Test 2 - Conversion success

The input grammar is:

$$S \rightarrow ASA \mid aB$$

$$A \rightarrow B \mid S$$

$$B \rightarrow b \mid \varepsilon$$

The expected result should be the following grammar:<sup>13</sup>

$$S_0 \rightarrow AA_1 \mid AS \mid SA \mid XB \mid a$$

$$S \rightarrow AA_1 \mid AS \mid SA \mid XB \mid a$$

$$A \rightarrow b \mid AA_1 \mid AS \mid SA \mid XB \mid a$$

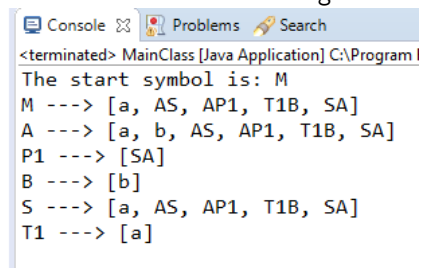
$$B \rightarrow b$$

$$A_1 \rightarrow SA$$

$$X \rightarrow a$$

The actual result is:

Figure 11: Test 2 actual result



```

<terminated> MainClass [Java Application] C:\Program I
The start symbol is: M
M ---> [a, AS, AP1, T1B, SA]
A ---> [a, b, AS, AP1, T1B, SA]
P1 ---> [SA]
B ---> [b]
S ---> [a, AS, AP1, T1B, SA]
T1 ---> [a]

```

As we can see this is exactly the same grammar as the expected result, with the difference being that  $S_0$  is represented by  $M$ ,  $A_1$  represented by  $P_1$  and  $X$  is represented by  $T_1$ .

## 9.2 Exponential algorithm to find all words in given CFG

This algorithm was developed by me, I did not use any literature regarding the subject and it was a very nice practice and experience. The idea behind the algorithm is to use a scanning method very similar to BFS (Breadth-first search) that is in use for graphs.

<sup>13</sup>Used Yoav example

I will have a set *Incomplete – Rules* of strings where  $String\ s \in \{V \cup \Sigma\}^*$  and a set of all the words generated by the language, meaning *Incomplete – Rules*  $\cap \Sigma^*$  (i.e all the strings that contain only non terminals).

The beginning will be with the start symbol  $S_0$  and then we will run for  $X$  number of steps, where step meaning that we will go over all the strings in the set and advance them all by one derive rule.

The start symbol is the root of the search, and from there we advance to all the possible combination of strings we can create.

#### How to advance the string?

I have decided to parse the words from Left To Right, because it is easier for use and implementation, after finite amount of steps it may cause significant difference in the language generated compared to other parsing methods (for example right to left), however I try to mimic the whole language, meaning activated very large number of steps, so it is unlikely we will miss any words.

The code will go as follows:

For every string  $s$  in *Incomplete – Rules*

Remove string  $s$  from the set

Break  $s$  to 3 parts –  $X, Y, Z$

Where  $X \in \Sigma^*$  (i.e it is a sequence of only terminals),  $Y \in V$  (i.e it is non terminal) and  $Z \in \{V \cup \Sigma\}^*$

For every  $\alpha$  such that  $(Y, \alpha) \in P$

Enter the new string  $X\alpha Z$  to *Incomplete – Rules*

If the string could not broken in such a way it means it is a sequence of only terminals, meaning it is a valid word.

### 9.2.1 Java Implementation

In my implementation I receive an integer input that represents the required number of steps we want to apply to the grammar, and to receive a set (To avoid duplicates) of all the valid words of the grammar. The implementation can be seen in the attached source code, inside the method:

*generateLengthSteps(Int length)*

### 9.3 Word recognizing and parsing

For this testing I will use the the grammar described in section "Parser Algorithm" (6.2.2)

$S \rightarrow AB|AD$

$A \rightarrow a$

$B \rightarrow b$

$D \rightarrow SB$

Using the algorithm described in the section above, this is a subset of the  $L(G)$ , as received from the Java program:

Figure 12:  $L(G)$  subset

```
Word #1 aaaaaaaaaabbbbbbbbbbb
Word #2 ab
Word #3 aaaaaaaaaabbbbbbbbbbb
Word #4 aaaaaaaaaabbbbbbbbbbb
Word #5 aaaaaaaabbbbbbb
Word #6 aaaaabbbb
Word #7 aaaaaaaaaabbbbbbbbbbb
Word #8 aaabbb
Word #9 aaaaaabbbbbbb
Word #10 aabb
Word #11 aaaaaaaaaabbbbbbbbbbb
Word #12 aaaaaaaaaabbbbbbbbbbb
Word #13 aaaaaaaaaabbbbbbbbbbb
Word #14 aaaaaaabbbbbbb
Word #15 aaaaabbbb
Word #16 aaaabbbb
Word #17 aaaaaaabbbbbbb
Word #18 aaaaaaaaaabbbbbbbbbbb
```

### 9.3.1 Test - Successful parsing

**Input:** *aaabbb*

As shown this word can be generated using the grammar, this is an example of how the implementation works with this input of word and grammar.

**Output:**

Figure 13: Successful parsing result

The Word: 'aaabbb' can be derived from the grammar

The parse tree is:

```
M ---> AD
---> aD
---> aSB
---> aADB
---> aaDB
---> aaSBB
---> aaABBB
---> aaaBBB
---> aaabBB
---> aaabbB
---> aaabbb
```

### 9.3.2 Test - Unsuccessful parsing

It may be interesting to test the program using a word  $W$  where  $W \notin L(G)$ . To find a word that is not part of the grammar language we should examine the grammar and words it can generate. As we can see from the subset of the grammar language, all the words generated from the grammar, follow a specific rule. Word  $W = \{a^i b^i | i \geq 1\}$ ,<sup>14</sup> therefore let's test the program using  $W = bbaa$ :

Figure 14: Successful parsing result

```
# Added new start non-terminal M because previous one was located in the rhs
# Removed rule: M --> S , Because this is unit production
# Added rule: M --> AB , To replace unit production
# Added rule: M --> AD , To replace unit production
The start symbol is: M
M --> [AB, AD]
A --> [a]
B --> [b]
S --> [AB, AD]
D --> [SB]

The word: 'bbaa' can not be derived from the language
```

## 10 Parsing using original grammar

Currently I displayed the ability to parse a given word using grammar in CNF form, and in-case the grammar is not in CNF form the program will first transform it into a grammar with the same language, and then parse the word using the new grammar. However it may be beneficial to understand the word parsing using the original grammar. That raise several problems as context free grammars are not limited and may contain  $\epsilon$ -rules, rules with length of more than 2, and mixed rules with terminals and non-terminals.

I have not implemented this part of the assignment, but I did give a lot of thought in the process of backtracking and undoing the CNF conversion.

### 10.1 Backtracking the grammar conversion

This section required the generated table from the CYK algorithm using the converted grammar in the CNF form, we already know how to construct the table and work with it from the previous sections therefore I will skip the explanation part.

This process has several steps, and the purpose of each step is to undo the CNF conversion of the grammar. For the undoing of the conversion we will use the extended table, where every cell also contains all the rules and their

<sup>14</sup>It can be proven that it exactly the language, we will skip this part

locations in bottom parts of the table. We will distinguish every rule in the cell as follows:  $A_{i,l} = [A, i, l]$ .

**Undo the CNF conversion:**

- Replace every  $A_a \rightarrow a$  (Where  $A_a$  is a new rule added in the conversion process) in the chart with  $a$  and replace every occurrence of  $A$  in a production with  $a$ .
- For all  $l, i \in [1..n]$ : If  $A \rightarrow \alpha D_{i_D, l_D} \in C_{i,l}$  such that  $D$  is a new symbol introduced in the CNF transformation and  $D \rightarrow \beta \in C_{i_D, l_D}$ , then replace  $A \rightarrow \alpha D_{i_D, l_D}$  with  $A \rightarrow \alpha\beta$  in  $C_{i,l}$ .
- Finally remove all  $D \rightarrow Y$  with  $D$  being a new symbol introduced in the CNF transformation from the chart.

**Undo the elimination of unit productions:**

- For every  $A \rightarrow \beta$  in  $C_{i,l}$  that has been added in removing of the unit productions to replace  $B \rightarrow \beta'$  ( $\beta'$  is  $\beta$  without chart indices): replace  $A$  with  $B$  in this entry in  $C_{i,l}$ .
- For every unit production  $A \rightarrow B$  in the original grammar and for every  $B \rightarrow \beta \in C_{i,l}$ : add  $A \rightarrow B_{i,l}$  to  $C_{i,l}$ . Repeat this until chart does not change any more.

**Undo the elimination of  $\epsilon$ -productions:**

- Add a row with  $l = 0$  and a column with  $i = n + 1$  to the chart.
- Fill row 0 as in the general case using the original CFG grammar (tabulating productions).
- For every  $A \rightarrow \beta$  in  $C_{i,l}$  that has been added in removing the  $\epsilon$ -productions: add the deleted nonterminals to  $\beta$  with the position of the preceding non-terminal as starting position (or  $i$  if it is the first in the rhs) and with length 0.

**Observation:** Information on the obligatory presence of terminals might get lost in the CNF transformation. Consider the following grammar  $G$ :

$$S \rightarrow aSb|ab$$

The transformation to CNF form will lead us to  $G'$ :

$$S \rightarrow C_a C_b | C_a S_B$$

$$C_a \rightarrow a$$

$$C_b \rightarrow b$$

$$S_B \rightarrow S C_b$$

$S \rightarrow aSb$  (requires an  $a$ , an  $S$  and a  $b$ )  $\rightsquigarrow S \rightarrow C_a S_B$  (requires an  $a$  and  $S$  and a  $b$ ) and  $S_B \rightarrow S C_b$  (requires an  $S$  and a  $b$ )

Consider an input 'babb':

- In a CYK parser with the original grammar, we would derive  $[S, 2, 2]$  and  $[b, 4, 1]$  but we could not apply  $S \rightarrow aSb$ .
- In the CNF grammar, we would have  $[S, 2, 2]$  and  $[C_b, 4, 1]$  and then we could apply  $S_B \rightarrow SC_b$  and obtain  $[S_B, 2, 3]$  even though the only way to continue with  $S_B$  in a rhs is with  $S \rightarrow C_a S_B$  which is not possible since the first terminal is not an  $a$ .

**Solution: add an additional check:**

- Every new non-terminal  $D$  introduced for BIN step, stands (deterministically) for some substring  $\beta$  of a rhs  $\alpha\beta$ . Ex:  $S_B$  in our sample grammar stands for  $Sb$ .
- Every terminal in this rhs to the left of  $\beta$ , i.e., every terminal in  $\alpha$  must necessarily be present to the left for a deduction of a  $D$  that leads to a goal item. Ex:  $S_B$  can only lead to a goal item if to its left we have an  $a$ .
- **Terminal Filter:** During CNF transformation, for every nonterminal  $D$  introduced during the BIN step, record the sets of terminals in the rhs to the left of the part covered by  $D$ . During parsing, check for the presence of these terminals to the left of the deduced  $D$  item.



## References

- [1] Hopcroft, John E.; Ullman, Jeffrey D. (1979), Introduction to Automata Theory, Languages, and Computation, Addison-Wesley. Chapter 4: Context-Free Grammars, pp. 77–106; Chapter 6: Properties of Context-Free Languages, pp. 125–137.
- [2] To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm.
- [3] Chomsky, Noam (1959). "On Certain Formal Properties of Grammars". *Information and Control*. 2 (2): 137–167. doi:10.1016/S019-9958(59)90362-6
- [4] Sipser, Michael (1997). Introduction to the Theory of Computation (1st ed.).
- [5] Younger, Daniel H. (February 1967). "Recognition and parsing of context-free languages in time  $n^3$ ". *Inform. Control*. 10 (2): 189–208. doi:10.1016/s0019-9958(67)80007-x
- [6] E. Rich. Automata, Computability, and Complexity: Theory and Applications. Prentice-Hall, 2007.
- [7] R. Leermakers. How to cover a grammar. In 27th Annual Meeting of the ACL, Proceedings of the Conference, pages 135-142, Vancouver, Canada, June 1989. Association for Computational Linguistics.
- [8] S. Sippu and E. Soisalon-Soininen. Parsing Theory. Vol.I: Languages and Parsing. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1988
- [9] M.-J. Nederhof. Linguistic Parsing and Program Transformation. PhD thesis, Proefschrift Nijmegen, Wiskunde en Informatica, 1994..
- [10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. MIT Press, McGraw-Hill, Cambridge, London, 1990