VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Program for Analysis of Context-free Grammars

Program pro analýzu bezkontextových gramatik

Ruubesh Govindaraj

Diploma Thesis

Supervisor: doc. Ing. Sawa Zdeněk, Ph.D.

Ostrava, 2024

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Ref. no.VSB/23/112410
vsboee23037434

Diploma Thesis Assignment

Student: **Ruubesh Govindaraj**

Study Programme: N0613A140035 Computer Science

Title: Program for Analysis of Context-free Grammars
Program pro analýzu bezkontextových gramatik

The thesis language: English

Description:

The goal of the project is to create a program for manipulation with context-free grammars. The program should allow to perform different kinds of operations with grammars and testing some of their properties. In particular, the program should allow to create a derivation of a given word using a given grammar. In particular, it should allow an explicit manual creation of a derivation but also to allow to find out a derivation automatically. The program should display a derivation in two different ways - as a sequence of sentential forms and also in a graphical way displayed as a derivation tree.
The program should allow to detect and to remove useless nonterminals and rules. To get rid of epsilon-rules and simple rules, and to transform the grammars into Chomsky normal form and Greibach normal form. It should also allow to compute sets FIRST and FOLLOW, to compute LR states, to check if a grammar is of a given type (LR(1), LALR, SLR, LL, etc.), to detect different kinds of conflicts,etc.
The program should be implemented as a desktop application with a graphical user interface.
A part of the created program should be a collection of at least 10 example grammars that could be used to demonstrate the different functions of the program.

1. Study the necessary theory about context-free grammars.
2. Design and implement a program for manipulation with context-free grammars.
3. Demonstrate the working of your program on appropriate examples of grammars.

References:

[1] D. Grune, C.J.H. Jacobs. Parsing Techniques: A Practical Guide. Second Edition, Springer-Verlag, 2008.
[2] D.E. Knuth. On the translation of languages from left to right. Information and Control, 8(6), 1965, pp. 607–639.
[3] D.E. Knuth. Top-Down Syntax Analysis. Acta Informatica, vol. 1, Springer, 1971, pp. 79-110.
[4] F.L. DeRemer. Simple LR(k) Grammars. Communications of the ACM, 14(7), 1971, pp. 453-460.
[5] F.L. DeRemer, T. Pennello. Efficient computation of LALR(1) look-ahead sets. ACM Transactions on Programming Languages and Systems, 4(4), 1982, pp. 615–649, .

17. listopadu 2172/15
708 00 Ostrava-Poruba
Czech Republic

spojovatelka: +420 597 321 111
epodatelna: epodatelna@vsb.cz
ID datové schránky: d3kj88v

IČ: 61989100
DIČ: CZ61989100

email: studijni.fei@vsb.cz
www.fei.vsb.cz

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **doc. Ing. Zdeněk Sawa, Ph.D.**

Date of issue: 01.09.2023

Date of submission: 30.04.2024

Study programme guarantor: prof. RNDr. Václav Snášel, CSc.

In IS EDISON assigned: 07.11.2023 10:08:47

Abstrakt

Tato práce představuje rozšíření a vylepšení desktopové aplikace určené pro manipulaci s bezkontextovými gramatikami, která byla původně vyvinuta jako semestrální projekt v rámci magisterského studia informatiky. Původní aplikace sloužila jako praktický nástroj a nabízela uživatelsky přívětivé rozhraní pro zadávání, úpravy a konstrukci gramatik s funkcemi zahrnujícími sekvenční větné formy a derivační stromy v grafické podobě. Na tomto základě práce zavádí nové funkce, jejichž cílem je rozšířit analytické a transformační možnosti aplikace.

Rozšířená aplikace nyní zahrnuje pokročilé techniky optimalizace gramatik, včetně redukce daných gramatik odstraněním zbytečných neterminálů a nedosažitelných pravidel, jakož i odstranění epsilonových a jednoduchých pravidel. Kromě toho byly implementovány transformace do Chomského normální formy a Greibachové normální formy s podrobnými vysvětlivkami krok za krokem, které uživateli usnadňují pochopení. Uživatelé mají možnost ukládat transformované gramatiky do textových souborů, což usnadňuje další analýzu.

Kromě toho aplikace nyní nabízí robustní možnosti analýzy, včetně výpočtu množin FIRST a FOLLOW, které jsou nezbytné pro algoritmy parsování, a určení typu gramatiky (LR(0), LR(1), LL(1)). K dispozici jsou vizuální reprezentace položek LR, akcí a GOTO tabulek a mechanismy detekce konfliktů, které uživatelům pomáhají při identifikaci a řešení nejednoznačností při parsování.

Tím, že tato komplexní desktopová aplikace překlenuje propast mezi interakcí uživatele a manipulací s gramatikou, nabízí přístupnou platformu pro zkoumání a pochopení bezkontextových gramatik. Její nově přidané funkce nejen zvyšují uživatelský komfort, ale také prohlubují užitečnost aplikace pro účely výuky.

Klíčová slova

bezkontextové gramatiky; analýza gramatik; parsing; uživatelsky přívětivá aplikace

Abstract

This thesis presents the extension and enhancement of a desktop application designed for manipulating context-free grammars, originally developed as a semestral project for the Master of Computer Science degree. The initial application served as a practical tool, offering a user-friendly interface for grammar entry, modification, and derivation construction, with features including sequential

sentential forms and graphical derivation trees. Building upon this foundation, the thesis introduces new functionalities aimed at extending the analysis and transformation capabilities of the application.

The extended application now encompasses advanced grammar optimization techniques, including the reduction of given grammars by eliminating useless nonterminals and unreachable rules, as well as the removal of epsilon and unit rules. Additionally, transformations to Chomsky normal form and Greibach normal form have been implemented, with detailed step-by-step explanations provided to aid user comprehension. Users are empowered to save the transformed grammars to text files, facilitating further analysis.

Moreover, the application now offers robust analysis capabilities, including the computation of FIRST and FOLLOW sets, essential for parsing algorithms, and the determination of the type of grammar (LR(0), LR(1), LL(1)). Visual representations of LR items, action, and goto tables are provided, alongside conflict detection mechanisms to assist users in identifying and resolving parsing ambiguities.

By bridging the gap between user interaction and grammar manipulation, this comprehensive desktop application offers an accessible platform for exploring and understanding context-free grammars. Its newly added features not only enhance user experience but also deepen the application's utility in academic contexts.

Keywords

context-free grammars; analysis of grammars; parsing; user-friendly application

Acknowledgement

I would like to extend my heartfelt gratitude to doc. Ing. Zdeněk Sawa, Ph.D., for his invaluable assistance and unwavering support throughout the course of this work. Without his guidance and expertise, this thesis would not have been possible. I am deeply thankful for doc. Sawa generosity with his time, expertise, and resources.

Contents

List of Figures **9**

List of Tables **10**

1 Introduction **11**

1.1 Objective 11

1.2 Overview of the Application 11

1.3 Evolution of the Application 12

1.4 Outline of the Contents 13

2 Theoretical Foundations **14**

2.1 Context-Free Grammars (CFGs) 14

2.2 Transformations in Context-Free Grammars 17

2.3 Deterministic Parsing Methods 25

3 Application from User Point of View **35**

3.1 Comparison with an Existing System 35

3.2 Application Overview 36

3.3 Features and Functionality 36

3.4 Navigating the Graphical Interface 43

3.5 Grammar File Format 51

4 Design and Architecture **53**

4.1 Architecture Overview 53

4.2 Class Diagram 54

4.3 Evolution of the User Interface 58

4.4 Sequence Diagram for Constructing Derivations 58

4.5 Transformations and Detection of Grammar Types 59

5 Implementation Details **62**
5.1 How to Launch the Application 62
5.2 Programming Language and Framework 63
5.3 Grammar Representation 63
5.4 Reduction of Grammars 63
5.5 Removing Epsilon Productions 64
5.6 Elimination of Unit Productions 65
5.7 Computation of FIRST Sets 66
5.8 Computation of LR(0) Items 67

6 Conclusion **69**

Bibliography **71**

List of Figures

2.1 LR(0) Automaton for the Grammar 2.1 30
2.2 Two types of LR conflicts 32
2.3 LR(1) Automaton for the Grammar 2.2 33

3.1 Validation of grammar file 39
3.2 Selection Page 44
3.3 Edit Page 46
3.4 Derivation Page 48
3.5 Result window - LR parsing type 50
3.6 Result window - Transformation type 50

4.1 Representation of architecture 54
4.2 Class Diagram 57
4.3 Sequence Diagram for Constructing Derivations 60

List of Tables

2.1 LR(0) ACTION and GOTO tables for automaton of Figure 2.1 31
2.2 LR(1) ACTION and GOTO tables for automaton of Figure 2.3 34

Chapter 1

Introduction

In the realm of computational linguistics and formal language theory, Context-Free Grammars (CFGs) stand as foundational constructs, playing a crucial role in describing the syntactic structure of languages. The manipulation and analysis of these grammars are integral to various fields, including natural language processing, compiler design, and theoretical computer science. They form the backbone of many computational applications.

This desktop application emerges as a useful and user-friendly tool aimed at facilitating the exploration and manipulation of CFGs. Initially conceived as part of a semestral project for the Master of Computer Science degree, this application has now evolved into an extended version developed as a thesis. Serving as a practical implementation of theoretical concepts, it offers a hands-on approach to understanding and effectively working with context-free grammars.

1.1 Objective

The primary objective of this thesis project was to develop a desktop application for the manipulation and analysis of context-free grammars. The application aims to provide a user-friendly interface that empowers users, regardless of their familiarity with formal language theory, to interact with and manipulate CFGs effortlessly. By bridging the gap between theoretical concepts and practical application, the application seeks to facilitate exploration and understanding of CFGs in a hands-on manner.

1.2 Overview of the Application

The application offers a comprehensive suite of functionalities designed to streamline the editing, transformation, and analysis of CFGs. Users can seamlessly browse, modify, and save grammar files through a user-friendly interface, generate derivations of given words using specified grammars presented both as sentential forms and graphical derivation trees providing a deeper understand-

ing of grammar manipulation steps. It also has undo and redo functionalities to navigate between derivation steps, enhancing the overall user experience. It allows users to perform advanced transformation operations such as reducing grammars, removing epsilon and unit productions, and analyze grammars to determine their properties and characteristics. The application provides intuitive visualizations, step-by-step explanations, and interactive tools to enhance the user experience and facilitate deeper understanding. The application's versatility makes it suitable for educational purposes.

1.3 Evolution of the Application

In my semestral project, my application was designed to provide users with a user-friendly interface for effortlessly entering, modifying, and constructing derivations within context-free grammars (CFGs). One of the notable features of the application was its support for two distinct modes of displaying derivations—sequential sentential forms and graphical derivation trees. This allowed users to visualize and understand the structural transformations within the grammar.

Key functionalities of the application included a streamlined process for adjusting various elements of the grammar, including nonterminals, terminals, initial nonterminals, and production rules. Users could easily make modifications to these components directly within the grammar file, simplifying the editing process and promoting a seamless user experience.

Since then, the application has undergone significant enhancements and now offers an even more robust set of features. One notable improvement is the ability to import multiple grammar files from the user's computer, allowing for greater flexibility and convenience in working with various grammar specifications. The application also provides users with a straightforward interface for browsing and selecting grammar files, making it easy to view the contents of each file and choose the one to work with.

In addition to grammar manipulation, the application now includes advanced transformation functionalities. Users can perform a variety of transformations on the given grammar, including reduction of grammar, removal of epsilon productions, elimination of unit productions, and conversion to Chomsky Normal Form (CNF) and Greibach Normal Form (GNF). Importantly, each transformation step is accompanied by detailed explanations, helping users understand the underlying principles and implications of each transformation.

Moreover, the application has been equipped with powerful analysis tools for examining the properties and characteristics of CFGs. Users can compute sets FIRST and FOLLOW, generate LR states, and determine if a given grammar confirms to specific types such as LL(1), LR(0) or LR(1). The application also detects and highlights various conflicts encountered during analysis, such as shift/reduce conflicts and reduce/reduce conflicts. This feature-rich analysis functionality provides users with valuable insights into the structure and behavior of their grammars.

Overall, these enhancements have transformed the application into a versatile and indispensable tool for working with context-free grammars. Whether users are editing grammars, performing transformations, or conducting analysis, the application offers a comprehensive set of features to support their needs and foster a deeper understanding of CFGs.

1.4 Outline of the Contents

This document serves as a comprehensive guide to this desktop application. The document is organized into several chapters, each focusing on specific aspects of the application, ranging from the theoretical foundations to the practical implementation detail. Chapter 2 delves into the theoretical underpinnings of context-free grammars (CFGs) and parsing techniques. It explores the fundamental concepts and principles behind CFGs, including the formal definition, derivations, and parse trees. Additionally, this chapter discusses parsing methods and provides a classification of parsing techniques based on their directionality and determinism.

Chapter 3 provides an overview of the key functionalities offered by the application. It details the core features of the application, including grammar manipulation, transformation operations, analysis tools, and derivation generation. This chapter highlights the breadth of capabilities that the application provides to users.In chapter 4, the design and architecture of the application are elucidated. It outlines the overall structure of the application, including its components, modules, and interactions. The chapter discusses design considerations, such as user interface design, data structures, and algorithms, that have shaped the architecture of the application. Chapter 5 delves into the technical implementation of the application. It explores the programming languages, frameworks, and libraries used to develop the application. Chapter 6 of the this document concludes the study by summarizing the key findings, contributions, and implications of the application. It offers recommendations for future enhancements.

Chapter 2

Theoretical Foundations

In this chapter, we delve into the theoretical underpinnings of context-free grammars (CFGs), providing an in-depth exploration of their key components and properties. We begin by offering an overview of CFGs, highlighting their fundamental characteristics. Next, we delve into the transformation process within grammars and explore various parsing techniques. By gaining a solid understanding of these fundamental principles, users can confidently navigate and manipulate grammars within the application.

2.1 Context-Free Grammars (CFGs)

Context-free grammars are powerful method for describing languages. Such grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications.

Context-free grammars were first used in the study of human languages. One way of understanding the relationship of terms such as noun, verb, and preposition and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. Context-free grammars help us organize and understand these relationships.

An important application of context-free grammars occurs in the specification and compilation of programming languages. A grammar for a programming language often appears as a reference for people trying to learn the language syntax. Designers of compilers and interpreters for programming languages often start by obtaining a grammar for the language. Most compilers and interpreters contain a component called a parser that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. A number of methodologies facilitate the construction of a parser once a context-free grammar is available. Some tools even automatically generate the parser from the grammar.

Formally, a context-free grammar is a 4-tuple,

$$G = (\Pi, \Sigma, S, P)$$

where,

- Π is a finite set of nonterminal symbols (nonterminals)
- Σ is a finite set of terminal symbols (terminals), where $\Pi \cap \Sigma = \varnothing$
- $S \in \Pi$ is an initial nonterminal
- $P \subseteq \Pi \times (\Pi \cup \Sigma)^*$ is a finite set of rewrite rules

2.1.1 Derivation Tree and Sentential Form

Derivation in a CFG involves applying production rules iteratively to transform an initial nonterminal into a target string of terminals. At each step of the derivation, a nonterminal in the current string is replaced by the RHS of a production rule that matches its occurrence in the string. This process continues until no further derivations are possible, resulting in a string composed entirely of terminals.

The intermediate strings generated during the derivation process are called sentential forms. A sentential form may contain both terminals and nonterminals, representing various stages of the derivation. The final sentential form, composed only of terminals, is called a sentence of the language generated by the grammar.

Derivation trees provide a graphical representation of the derivation process in a CFG. Each node in the derivation tree represents either a terminal or a nonterminal, and the edges represent the application of production rules. Starting from the root node (corresponding to the initial nonterminal), traversing the tree from top to bottom yields the derivation of a sentence.

Grammars are used for generating words. The following is an example of a context-free grammar, which we call G.

Example: $G = (\Pi, \Sigma, S, P)$ where $\Pi = A, B, C$, $\Sigma = a, b$, and P contains rules

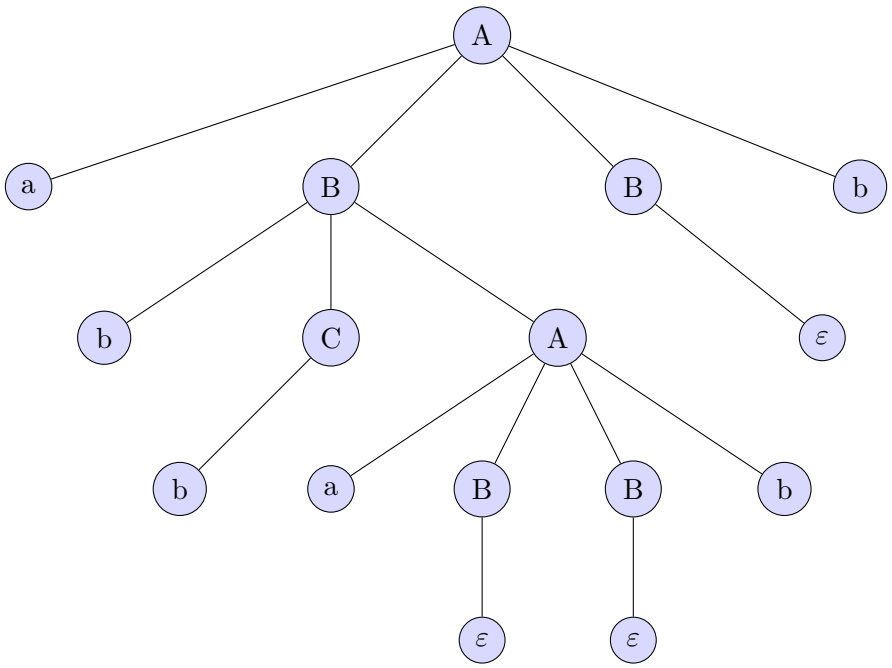
$$A \to aBBb \mid AaA$$
$$B \to \varepsilon \mid bCA$$
$$C \to AB \mid a \mid b$$

For example, the word *abbabb* can be in grammar G generated as follows:

Sentential Form:

$$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow$$
$$abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$$

Derivation Tree:



2.1.2 Context-Free Languages

The collection of languages associated with context-free grammars are called the context-free languages [1].

A language L is context-free if there exists some context-free grammar G such that $L = L(G)$. A language $L(G)$ generated by a grammar $G = (\Pi, \Sigma, S, P)$ is the set of all words over alphabet Σ that can be derived by some derivation from the initial nonterminal S using rules from P, i.e.,

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Note: w must be in Σ^*, the set of strings made from terminals. Strings involving nonterminals aren't in the language.

Example: We want to construct a grammar generating the language

$$L = \{a^n b^n \mid n >= 0\}$$

Grammar $G = (\Pi, \Sigma, S, P)$ where $\Pi = S$, $\Sigma = a, b$, and P contains

$$S \to \varepsilon \mid aSb$$

$S \Rightarrow \varepsilon$
$S \Rightarrow aSb \Rightarrow ab$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaaaSbbbb \Rightarrow aaaabbbb$
...

2.2 Transformations in Context-Free Grammars

The definition of context-free grammars allows for the development of a wide variety of grammars. However, it is common for certain productions within CFGs to be redundant or unnecessary. This redundancy arises because the definition of CFGs does not impose restrictions on the creation of such productions.

The process of simplifying CFGs involves removing these redundant productions while ensuring that the transformed grammar remains equivalent to the original grammar. Two grammars are considered equivalent if they generate the same language. The simplification of CFGs is a necessary step preceding their conversion into Normal Forms.

In this section, we explore the various types of redundant productions that may occur within CFGs and outline the procedures for identifying and removing them. Additionally, we delve into the conversion of CFGs into Normal Forms, discussing the step-by-step procedures involved in these transformations.

2.2.1 Reduction of Grammars

The productions that can never take part in derivation of any string, are called useless productions. Similarly, a nonterminal that can never take part in derivation of any string is called a useless nonterminal [2].

A context-free grammar $G = (\Pi, \Sigma, S, P)$ is reduced if for every $A \in \Pi$:

- there are some $u, v \in \Sigma^*$ such that $S \Rightarrow^* uAv$, and
- there is some $w \in \Sigma^*$ such that $A \Rightarrow^* w$.

If $S \Rightarrow^* uAv$ and $A \Rightarrow^* w$ where $u, v, w \in \Sigma^*$, then $S \Rightarrow^* uwv$, and so A is used in some derivation of word from Σ^*. On the other hand, if A is used in some derivation $S \Rightarrow^* z$ of a word $z \in \Sigma^*$, then

z can be divided into parts u, v, w such that $z = uwv$ and $S \Rightarrow^* uAv$ and $A \Rightarrow^* w$.

Obviously, every $A \in \Pi$ with the property that,

- there are no $u, v \in \Sigma^*$ such that $S \Rightarrow^* uAv$, or
- there is no $w \in \Sigma^*$ such that $A \Rightarrow^* w$,

can be safely removed from the grammar (together with all rules where it occurs) without affecting the generated language.

2.2.1.1 Example

Let G be a grammar and the production rules for the grammar is as follows,

$$
\begin{aligned}
S &\to AC \mid B \\
A &\to aC \mid AbA \\
B &\to Ba \mid BbA \mid DB \\
C &\to aa \mid aBC \\
D &\to aA \mid \varepsilon
\end{aligned}
$$

Construct a set T of all nonterminals that can generate a terminal word:

$$T = \{A \in \Pi \mid (\exists w \in \Sigma^*)(A \Rightarrow^* w)\}$$

$$
\begin{aligned}
T_0 &= \{C, D\} \\
T_1 &= \{C, D, A\} \\
T_2 &= \{C, D, A, S\}
\end{aligned}
$$

$$T = \{C, D, A, S\}$$

Remove from G all nonterminals from the set $\Pi - T$ together with all rules where they occur, Denote the resulting grammar $G' = (\Pi', \Sigma, S, P')$.

$$
\begin{aligned}
S &\to AC \\
A &\to aC \mid AbA \\
C &\to aa \\
D &\to aA \mid \varepsilon
\end{aligned}
$$

Construct the set D of all nonterminals that can be "reached" from the initial nonterminal S:

$$D = \{A \in \Pi' \mid (\exists \alpha, \beta \in (\Pi' \cup \Sigma)^*)(S \Rightarrow^* \alpha A \beta)\}$$

$$D_0 = \{S\}$$
$$D_1 = \{S, A, C\}$$

$$D = \{S, A, C\}$$

Remove from G' all nonterminals from the set $\Pi' - D$ together with all rules where they occur.

$$S \to AC$$
$$A \to aC \mid AbA$$
$$C \to aa$$

The resulting grammar is free from useless productions.

2.2.2 Removal of Epsilon Productions

The productions of type $A \to \varepsilon$ are called epsilon productions. It is possible for a grammar to contain null productions and yet not produce an empty string.

For every context-free grammar $G = (\Pi, \Sigma, S, P)$ there is a context-free grammar $G' = (\Pi', \Sigma, S', P')$ such that $L(G') = L(G)$ and either:

- G' does not contain ε-productions, or
- the only ε-production in G' is the rule $S' \to \varepsilon$ and S' does not occur on the right-hand side of any rule in G'.

2.2.2.1 Example

Let G be a grammar and the production rules for the grammar is as follows,

$$S \to ASA \mid aBC \mid b$$
$$A \to BD \mid aAB$$
$$B \to bB \mid \varepsilon$$
$$C \to AaA \mid b$$
$$D \to AD \mid BBB \mid a$$

Construct a set E of all nonterminals that can generate ε:

$$E_0 = \{B\}$$
$$E_1 = \{B, D\}$$
$$E_2 = \{B, D, A\}$$

$$E = \{B, D, A\}$$

Remove all epsilon rules and replace every other $A \to \alpha$ with a set of rules obtained by all possible rules of the form $A \to \alpha'$ where α' is obtained from α by possible omitting of (some) occurrences of nonterminals from set E.

$$S \to ASA \mid SA \mid AS \mid S \mid aBC \mid aC \mid b$$
$$A \to BD \mid B \mid D \mid aAB \mid aB \mid aA \mid a$$
$$B \to bB \mid b$$
$$C \to AaA \mid aA \mid Aa \mid a \mid b$$
$$D \to AD \mid D \mid A \mid BBB \mid BB \mid B \mid a$$

The resulting grammar is free from epsilon productions.

2.2.3 Elimination of Unit Productions

Rules of the form A $\to$ B where A, B $\in \Pi$ are called unit rules. The procedure to remove all unit rules from the grammar is,

1. Remove a unit rule $A \to B \in R$
2. For each rule $B \to u \in R$, add the rule $A \to u$ to R, unless $B \to u$ was a unit rule previously removed.
3. Repeat the above steps until all unit rules are eliminated.

Note: u is a string of nonterminals and terminals.

2.2.3.1 Example

$$X \to S$$
$$S \to ASA \mid aB \mid a \mid SA \mid AS$$
$$A \to B \mid S$$
$$B \to b$$

Removing $X \to S$:

$$X \to ASA \mid aB \mid a \mid SA \mid AS$$
$$S \to ASA \mid aB \mid a \mid SA \mid AS$$
$$A \to B \mid S$$
$$B \to b$$

Removing $A \to B$:

$$X \to ASA \mid aB \mid a \mid SA \mid AS$$
$$S \to ASA \mid aB \mid a \mid SA \mid AS$$
$$A \to b \mid S$$
$$B \to b$$

Removing $A \to S$:

$$X \to ASA \mid aB \mid a \mid SA \mid AS$$
$$S \to ASA \mid aB \mid a \mid SA \mid AS$$
$$A \to b \mid ASA \mid aB \mid a \mid SA \mid AS$$
$$B \to b$$

The grammar is now free from unit productions.

2.2.4 Conversion to Chomsky Normal Form (CNF)

A context-free grammar is in Chomsky normal form if every rule is of on of the following forms:

$$A \to BC$$
$$A \to a$$

where a is any terminal and A, B,and C are any nonterminals. In addition we permit the rule $S \to \varepsilon$, where S the initial nonterminal. In that case, nonterminal S cannot occur on the right-hand side of any rule.

For every context-free grammar G there is an equivalent context-free grammar G' in Chomsky normal form. The procedure to transform a context-free grammar to Chomsky Normal Form is as follow,

1. Decompose each rule $A \to \alpha$ where $|\alpha| \geq 3$ into a sequence of rules where each right-hand size has length 2.

2. Remove epsilon productions.

3. Remove unit productions.

4. For each terminal a occurring on the right-hand size of some rule $A \to \alpha$ where $|\alpha| = 2$ introduce a new nonterminal N_a, replace occurrences of a on such right-hand sides with N_a, and add $N_a \to a$ as a new rule.

2.2.4.1 Example

$$
\begin{aligned}
S &\to ASA \mid aB \\
A &\to B \mid S \\
B &\to b \mid \varepsilon
\end{aligned}
$$

After step 1, Decomposition of rules,

$$
\begin{aligned}
S &\to AZ \mid aB \\
Z &\to SA \\
A &\to B \mid S \\
B &\to b \mid \varepsilon
\end{aligned}
$$

After step 2, Epsilon productions removed,

$$
\begin{aligned}
S_0 &\to S \\
S &\to AZ \mid Z \mid aB \mid a \\
Z &\to SA \mid S \\
A &\to B \mid S \\
B &\to b
\end{aligned}
$$

After step 3, Unit productions removed,

$$
\begin{aligned}
S_0 &\to AZ \mid aB \mid a \mid SA \\
S &\to AZ \mid aB \mid a \mid SA \\
Z &\to SA \mid AZ \mid aB \mid a \\
A &\to b \mid AZ \mid aB \mid a \mid SA \\
B &\to b
\end{aligned}
$$

After step 4,

$$
\begin{aligned}
S_0 &\to AZ \mid YB \mid a \mid SA \\
S &\to AZ \mid YB \mid a \mid SA \\
Z &\to SA \mid AZ \mid YB \mid a \\
A &\to b \mid AZ \mid YB \mid a \mid SA \\
B &\to b \\
Y &\to a
\end{aligned}
$$

The grammar is now in Chomsky Normal Form.

2.2.5 Conversion to Greibach Normal Form (GNF)

There is another interesting normal form for grammars. Every nonempty language without ε is $L(G)$ for some grammar G each of whose productions is of the form:

$$A \to a\alpha$$

where a is a terminal and α is a string of zero or more nonterminals. Converting a grammar to this form is complex, even if we simplify the task by, say, starting with a Chomsky-Normal-Form grammar. Roughly, we expand the first nonterminal of each production, until we get a terminal. However, because there can be cycles, where we never reach a terminal, it is necessary to "short-circuit" the process, creating a production that introduces a terminal as the first symbol of the body and has nonterminals following it to generate all the sequences of nonterminals that might have been generated on the way to generation of that terminal.

This form, called Greibach Normal Form, after Sheila Greibach, who first gave a way to construct such grammars, has several interesting consequences. Since each use of a production introduces exactly one terminal into a sentential form, a string of length n has a derivation of exactly n steps [3]. Greibach normal form provides a justification of operator prefix-notation usually employed in algebra.

The procedure to transform a context-free grammar to Greibach Normal Form is as follows,

1. Remove epsilon productions.
2. Remove unit productions.
3. Change the names of non terminal symbols to A_1 till A_N and reorder the rules accordingly.
4. Check for every production rule if RHS has first symbol as nonterminal say A_j for the production of A_i, it is mandatory that i should be less than j. Not great and not even equal.

   If $i > j$ then replace the production rule of A_j at its place in A_i.

If $i = j$, it is the left recursion. Create a new state Z which has the symbols of the left recursive production, once followed by Z and once without Z, and change that production rule by removing that particular production and adding all other production once followed by Z [4].

5. Update each production rule by substituting the very first nonterminal symbol with its corresponding production until the rule adheres to the Greibach Normal Form.

2.2.5.1 Example

Assuming step 1 and step 2 are already done, the resulting grammar after removing epsilon and unit productions is,

$$A \rightarrow CD \mid BB$$
$$B \rightarrow b \mid AB$$
$$C \rightarrow b$$
$$D \rightarrow a$$

Step 3, Renaming the nonterminals,

$$A_1 = A$$
$$A_2 = C$$
$$A_3 = D$$
$$A_4 = B$$

Therefore, now the new production rules are,

$$A_1 \rightarrow A_2A_3|A_4A_4$$
$$A_2 \rightarrow b$$
$$A_3 \rightarrow a$$
$$A_4 \rightarrow b \mid A_1A_4$$

Step 4, Here for A_4, $4! < 1$, so now replace it with A_1's production rule.

$$A_1 \rightarrow A_2A_3|A_4A_4$$
$$A_2 \rightarrow b$$
$$A_3 \rightarrow a$$
$$A_4 \rightarrow b \mid A_2A_3A_4 \mid A_4A_4A_4$$

Again, for A_4, $4! < 2$, so now replace it with A_2's production rule.

$$A_1 \rightarrow A_2A_3|A_4A_4$$
$$A_2 \rightarrow b$$
$$A_3 \rightarrow a$$
$$A_4 \rightarrow b \mid bA_3A_4 \mid A_4A_4A_4$$

Here $A_4A_4A_4$ in production rule of A_4 is the example of left recursion. After removing the left recursion,

$$A_1 \rightarrow A_2A_3|A_4A_4$$
$$A_2 \rightarrow b$$
$$A_3 \rightarrow a$$
$$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$$
$$Z \rightarrow A_4A_4 \mid A_4A_4Z$$

Step 5, here we need to replace A_2 in production rule of A_1 and so on.

$$A_1 \rightarrow bA_3 \mid bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4$$
$$A_2 \rightarrow b$$
$$A_3 \rightarrow a$$
$$A_4 \rightarrow b \mid bA_3A_4 \mid bZ \mid bA_3A_4Z$$
$$Z \rightarrow bA_4 \mid bA_3A_4A_4 \mid bZA_4 \mid bA_3A_4ZA_4 \mid bA_4Z \mid bA_3A_4A_4Z \mid bZA_4Z \mid bA_3A_4ZA_4Z$$

The respective grammar is now in Greibach Normal Form, we can rename the nonterminals back to their original names.

2.3 Deterministic Parsing Methods

In this section, we will focus on parsers that exhibit deterministic behavior, meaning that they always have a single possible choice at each parsing step. Deterministic parsers offer several advantages over non-deterministic ones: they are generally faster and eliminate ambiguity, and can construct the parse tree dynamically during parsing. A grammar is said to be unambiguous if every sentential form has a unique derivation tree [5]. However, there is a trade-off: deterministic parsing methods are applicable to a more restricted class of grammars compared to non-deterministic methods. Specifically, they can only be used with non-ambiguous grammars.

2.3.1 LL Parsing Techniques

The class of context-free grammars that can be parsed in a top-down manner without backtrack is of interest because the parsing can be done quickly and the type of syntax directed transductions which can be performed over such grammars by a deterministic pushdown machine is fairly general [6].

LL parsing is a top-down parsing technique that operates on leftmost derivations. The "LL" designation stands for "Left-to-right, Leftmost derivation," indicating the parsing strategy employed. In LL parsing, a parse tree is constructed recursively from the root to the leaves, following a predefined set of production rules.

2.3.1.1 LL(1)

"LL(1)" means that the grammar allows a deterministic parser that operates from Left to right, produces a Left-most derivation, using a look-ahead of one (1) symbol [7].

2.3.1.2 FIRST Sets

The FIRST set of a nonterminal symbol represents the set of terminal symbols that can begin any string derived from that nonterminal.

To compute the FIRST set for a nonterminal A, we consider all the production rules for A. For each production rule $A \to \alpha$, where α is a string of symbols (terminals and/or nonterminals), we determine the terminals that can appear as the first symbol in a derivation of α.

The process of computing FIRST sets involves considering various cases:

- If α starts with a terminal symbol t, then t is included in FIRST(A).
- If α starts with a nonterminal symbol B, then we include all symbols in FIRST(B) in FIRST(A), except for ε. Additionally, if ε is in FRIST(B), we continue to the next symbol in α until we encounter a terminal or a nonterminal that does not produce ε, adding its FIRST set to FIRST(A).
- If α can derive ε, then we include ε in FIRST(A) if α is the only production for A, or if all symbols in α can derive ε.

By iteratively applying these rules to all nonterminal symbols in the grammar, we can compute the FIRST sets for each nonterminal symbol.

2.3.1.3 FOLLOW Sets

The FOLLOW set of a nonterminal symbol represents the set of terminals that can immediately follow occurrences of that nonterminal in any sentential form.

To compute the FOLLOW set for a nonterminal A, we examine all the production rules in the grammar to identify occurrences of A. We then determine the terminals that can follow A in each of these contexts.

The process of computing FOLLOW sets involves considering various cases:

- If A is the start symbol of the grammar, we include the end-marker (\$) in FOLLOW(A), as it signifies the end of a derivation.
- For each production rule $B \to \alpha A \beta$, where α and β are strings of symbols (terminals and/or nonterminals), we include the FIRST set of β (excluding ε, if present) in FOLLOW(A). If β can derive ε or β is empty, we also include the symbols in FOLLOW(B) in FOLLOW(A).
- If A appears as the rightmost symbol in any production rule, we include the symbols in FOLLOW(B) in FOLLOW(A), where B is the nonterminal on the left side of the production rule.
- We continue this process iteratively until no new symbols can be added to any FOLLOW set.

By applying these rules to all nonterminals in the grammar, we can compute the FOLLOW sets for each nonterminal.

2.3.1.4 Using FIRST and FOLLOW Sets

These sets are crucial in LL(1) parsing, as they enable us to determine the correct production rule to apply based on the next input symbol and symbols that follow each nonterminal, ensuring deterministic parsing without backtracking.

Once we have the FIRST and FOLLOW sets of all the nonterminals in our grammar we can easily compute the FIRST sets of each production rule in the grammar, which will be used to detect conflicts and determine if the specified grammar is of type LL(1). By analyzing the properties mentioned in the section below, we can determine if a grammar is LL(1).

2.3.1.5 Properties of LL(1) Grammars

Suppose G is a grammar with no useless nonterminals. Let A be a nonterminal symbol whose rule is,

$$A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_m, \qquad m \geq 1$$

we have three conditions which are necessary for each rule of an LL(1) grammar:

1. first(α_1) ... first(α_m) are mutually disjoint, i.e. contain no common elements.
2. At most one of $\alpha_1, \ldots, \alpha_m$ can produce a null string.
3. If $\alpha_p \to^* \varepsilon$, the first($\alpha_p$) has no elements in common with follow(A), for $1 \leq q \leq m$, $q \neq p$.

These three conditions are sufficient to show that G is LL(1): For if we are to replace A by one of $\alpha_1, \ldots, \alpha_m$, the choice of which α_i to use is uniquely determined by examining the first character of the terminal string which ultimately is to be produced by A [8].

2.3.2 LR Parsing Techniques

LR parsing is a powerful parsing technique widely used in compiler construction and syntax analysis. LR parsing is based on a family of shift-reduce parsing algorithms that efficiently handle a broad class of grammars, including ambiguous and left-recursive grammars.

These parsers will attempt to construct the derivation tree "bottom-up"; i.e., from the leaves to the root. For historical reasons, these parsers are called LR parsers. The "L" stands for "left-to-right scan of the input", the "R" stands for "nghtmost derivation." We shall see that an LR parser operates by reconstructing the reverse of a rightmost derivation for the input [9].

2.3.2.1 LR(0)

LR(0) parsing is a bottom-up parsing technique that operates without any lookahead information. In LR(0) parsing, the parser makes parsing decisions solely based on the current state and the next input symbol, without considering any additional symbols ahead in the input stream.

LR(0) parsing involves constructing a parsing table (ACTION/GOTO table) that encodes the parsing decisions for each parsing state and input symbol combination. This parsing table is built using the LR(0) item sets, which represent the possible configurations of the parser as it processes the input.

2.3.2.2 LR(0) Items

LR(0) items are production rules used in LR(0) parsing to represent the current configuration of the parser as it processes input symbols. Each LR(0) item consists of a production rule with a dot (.) representing the current position of the parser within the rule. LR(0) items are crucial for constructing parsing tables and determining parsing actions in LR(0) parsing. LR(0) item sets are collections of LR(0) items that represent the possible configurations of the parser at various stages of parsing.

An LR(0) item for a production rule $A \to \beta$ is denoted as $A \to \alpha \cdot \beta$, where:

- A is a nonterminal representing the left-hand side of the production rule.
- α is a sequence of symbols that may appear before the dot (.).
- β is a sequence of symbols that may appear after the dot (.).

2.3.2.3 LR(0) Automaton

The components of the LR(0) automaton are called states and transitions. Nodes in the LR(0) automaton represent distinct parsing states. Each state corresponds to an LR(0) item set. Directed edges between states indicate transitions triggered by input symbols.

The steps involved in construction of the deterministic LR(0) automaton is as follows,

- **Augmented Grammar:** We create an augmented grammar by creating a new initial nonterminal and assigning the old initial nonterminal as the right-hand side of the new initial nonterminal.
- **Initial State:** The LR(0) automaton starts with an initial state corresponding to the LR(0) item set containing the start item of the augmented grammar.
- **Closure Operation:** Additional LR(0) items are derived by applying closure operations to existing item set. Closure operations involve expanding LR(0) item set to include all possible derivations of nonterminal symbols following the dot (.).
- **Transitions:** Transitions between states are determined based on the symbol that follow the dot(.) in the LR(0) items. The items in the item set that have the same symbol after the dot are added to the new state with the dot (.) moved after the symbol.

2.3.2.4 Example

We shall use the grammar below for this example:

$$\begin{aligned} S &\rightarrow AA \\ A &\rightarrow aA \mid b \end{aligned}$$

The resulting grammar after creating the augmented grammar is,

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow AA \\ A &\rightarrow aA \mid b \end{aligned} \tag{2.1}$$

We can now construct the LR(0) automaton.

2.3.2.5 ACTION and GOTO Tables

The ACTION and GOTO tables are used to determine the next action to take based on the current state and input symbol. These tables are essential components of the LR(0) parsing algorithm, enabling efficient and deterministic parsing of context-free grammars.

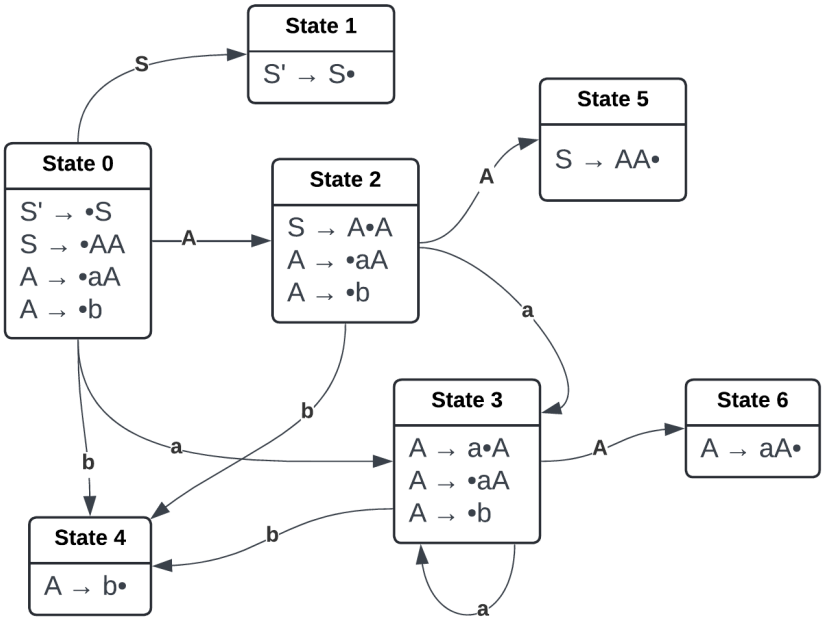


Figure 2.1: LR(0) Automaton for the Grammar 2.1

The ACTION table contains entries that specify the action to be taken and the GOTO table contains entries that specify the next state to transition to when the parser encounters a certain combination of state and nonterminal symbol.

There are three types of actions in the ACTION table:

1. **Shift:** This action instructs the parser to shift the input symbol onto the stack and transition to a new state.

2. **Reduce:** This action indicates that a reduction should be performed using a certain production rule.

3. **Accept:** This action signifies that the parsing process has successfully completed, and the input string has been recognized.

We can construct the LR(0) ACTION and GOTO table by using the LR(0) automaton we created.

The GOTO table for LR(0) parsing has symbols as its columns and states as its rows. The entries in the GOTO table are filled by reaching states to their corresponding state and symbol combinations.

The ACTION table for LR(0) parsing consists of only one "Action" column, with states as its rows. The entries in the ACTION table are determined based on the states in the LR(0) automaton. To add entries to the ACTION table we must consider various cases: In a state,

- If the dot (.) is in front of a symbol in an item, we add "Shift" to the corresponding state in the ACTION table. We do not add multiple "Shift" action to a same state.
- If the dot (.) is at the end of an item, we add the corresponding reduce rule in the grammar. If there are multiple reduce items we also add them to that corresponding state.

	Action
0	Shift
1	$S' \rightarrow S$
2	Shift
3	Shift
4	$A \rightarrow b$
5	$S \rightarrow AA$
6	$A \rightarrow aA$

	a	b	\$	A	S
0	3	4		2	1
1					
2	3	4		5	
3	3	4		6	
4					
5					
6					

Table 2.1: LR(0) ACTION and GOTO tables for automaton of Figure 2.1

2.3.2.6 LR(0) Conflicts

Despite its deterministic nature, LR(0) parsing can encounter conflicts. Detecting conflicts is the way to determine if the specified grammar is of type LR(0) or not.

In the ACTION table of an LR(0) parsing algorithm, each cell typically contains only one action corresponding to a particular parsing state and input symbol. However, when multiple actions are present within a single cell such as both a shift and reduce action or multiple reduce actions this indicates a conflict within the grammar. Specifically, the conflicts that arise are known as shift-reduce conflicts and reduce-reduce conflicts.

"When a grammar is not LR(0), one or more of the LR(0) parser's states must be "inconsistent", having either a "read-reduce" or a "reduce-reduce" conflict, or both. In the former case the parser cannot decide whether to read the next symbol of the input or to reduce a phrase on the stack. In the latter case the confusion is between distinct reductions" [10], in this sentence "read" represents the "shift" action. Figure 2.2 illustrates an example of the LR conflicts.

Figure 2.2: Two types of LR conflicts

2.3.2.7 LR(1)

LR(1) parsing is an extension of LR(0) parsing that takes into account one lookahead symbol when making parsing decisions. This additional lookahead symbol helps resolve many of the shift-reduce and reduce-reduce conflicts present in LR(0) grammars, making LR(1) parsing more powerful and versatile.

2.3.2.8 LR(1) Items

Each LR(1) item is annotated with a lookahead symbol. An LR(1) item is typically represented as follows:

$$A \to \alpha \cdot \beta, a$$

where:

- $A \to \alpha \cdot \beta$ represents a production rule with A being a nonterminal, α and β being strings of terminals and nonterminals, and the dot (.) indicating the current position in the production.
- a is the lookahead symbol, indicating the next input symbol expected after applying the production rule.

The construction procedure of the LR(1) automaton, ACTION table and GOTO table are same as for the LR(0) but in LR(1) we will also take the lookahead symbols in consideration.

2.3.2.9 LR(1) Automaton

LR(1) automaton states are constructed based on LR(1) items with unique lookahead symbols. States in LR(1) automaton may have multiple transitions on the same input symbol if they have different lookahead symbols.

Transitions between LR(1) automaton states are determined by considering lookahead symbols in addition to the current input symbols. Lookahead symbols influence the construction of transitions, allowing the automaton to distinguish between different expected inputs.

2.3.2.10 Example

Consider the grammar,

$$S \to CC$$
$$C \to cC$$
$$C \to d$$

create augmented grammar,

$$\begin{aligned} S' &\to SS \quad \to CC \\ C &\to cC \\ C &\to d \end{aligned} \tag{2.2}$$

construct LR(1) automaton,

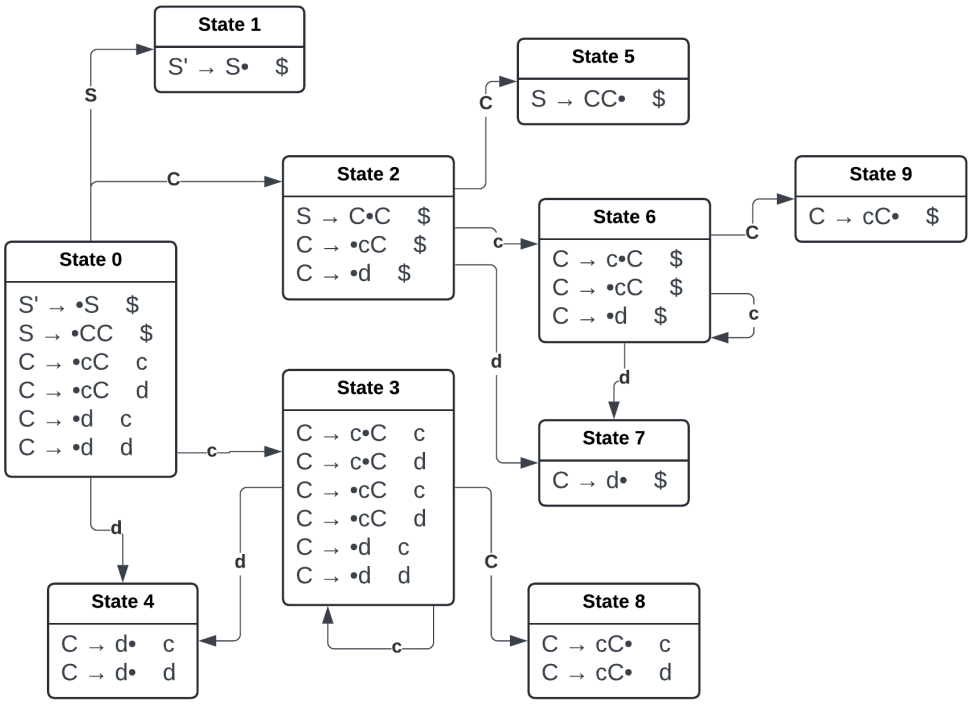


Figure 2.3: LR(1) Automaton for the Grammar 2.2

2.3.2.11 LR(1) ACTION and GOTO Tables

The LR(1) parsing tables includes entries for LR(1) items, incorporating lookahead symbols into parsing actions such as shift, reduce, and goto.

Parsing actions in LR(1) table are determined by considering both the current input symbol and the lookahead symbol associated with the LR(1) item.

	c	d	\$
0	Shift	Shift	
1			$S' \to S$
2	Shift	Shift	
3	Shift	Shift	
4	$C \to d$	$C \to d$	
5			$S \to CC$
6	Shift	Shift	
7			$C \to d$
8	$C \to cC$	$C \to cC$	
9			$C \to cC$

(a) ACTION Table

	c	d	\$	S	C
0	3	4		1	2
1					
2	6	7			5
3	3	4			8
4					
5					
6	6	7			9
7					
8					
9					

(b) GOTO Table

Table 2.2: LR(1) ACTION and GOTO tables for automaton of Figure 2.3

Chapter 3

Application from User Point of View

In this chapter, we provide an insightful exploration of the application from the user's perspective. We delve into the application's features, functionality, user interface design, and basic operations, offering users a comprehensive guide to effectively utilize the application to its fullest potential.

3.1 Comparison with an Existing System

Comparing my application with ANTLR [11] reveals distinct differences in their focus, target audience, and usability. While ANTLR is a powerful tool widely used in industry settings for developing domain-specific languages (DSLs), compilers, and interpreters, my application prioritizes user-friendliness, accessibility, and educational value.

Firstly, my application offers a streamlined interface specifically designed for editing, transforming and analyzing grammar files. Unlike ANTLR, which requires users to learn the ANTLR DSL for defining grammars, my application provides a user-friendly platform where users can choose grammar files from their computers and edit grammar specifications within the interface. This approach significantly reduces the learning curve, making it accessible to students, educators, and enthusiasts interested in exploring grammar manipulation and analysis.

Moreover, my application implements algorithms for reducing grammars, removing epsilon rules, and detecting grammar types such as LL(1), LR(0), and LR(1). These functionalities are accompanied by detailed step-by-step explanations of the transformation processes, enhancing transparency and usability. In contrast, ANTLR may require additional effort from users to implement similar functionalities using its parsing framework, which may be more suitable for developers and language processing professionals.

Overall, while ANTLR excels in providing advanced parsing and code generation features for developing language processing tools, my application stands out for its user-friendly interface, educational value, and accessibility to beginners and enthusiasts interested in grammar manipulation

and analysis. By prioritizing usability and transparency, my application offers a lightweight and accessible alternative to more complex language processing tools like ANTLR.

3.2 Application Overview

This application is carefully designed to help users learn and explore grammar manipulation, transformation, and parsing. Whether you're a student studying language theory or simply curious about computer science, the application offers a user-friendly and educational way to play with grammars and learn new concepts.

At its core, the application provides a user-friendly interface that simplifies the process of working with context-free grammars (CFGs). Users can effortlessly create, modify, and analyze grammars through an intuitive graphical interface, making it ideal for educational purposes and self-paced learning.

The application allows users to interactively create derivations using a given grammar. Derivations are displayed in both graphical tree and sentential form, providing users with a visual representation of the structural transformations within the grammar.

One of the key features of the application is its ability to handle different types of grammars and parsing techniques. Whether users are interested in LL parsing, LR parsing, or other parsing methods, the application offers a range of tools and functionalities to support their exploration.

Furthermore, the application supports various operations on grammars, such as grammar reduction, epsilon production removal, unit production elimination, and conversion to Chomsky Normal Form (CNF) or Greibach Normal Form (GNF). These operations are accompanied by detailed explanations and step-by-step guides to help users understand the underlying principles and implications.

Additionally, the application provides analysis tools for examining the properties and characteristics of CFGs, including computing FIRST and FOLLOW sets, generating LR states, and detecting conflicts such as shift/reduce and reduce/reduce conflicts. These analysis tools are essential for gaining insights into the structure and behavior of grammars, further enhancing the learning experience.

Overall, the application serves as a comprehensive and accessible platform for exploring formal language theory and gaining experience with grammars and parsing techniques.

3.3 Features and Functionality

This section delves into the diverse array of features and functionalities offered by the application which makes it versatile and indispensable tool for educational purposes.

3.3.1 Grammar Editing

Users can effortlessly modify grammar specifications within the selected grammar file using the intuitive editing tools available in the graphical user interface (GUI). Entry boxes, dropdown menus, and buttons within the editing frame facilitate seamless addition or removal of nonterminals or terminals, modification of production rules, and alteration of the initial nonterminal. Users can conveniently save all changes directly back to the original grammar file, and the updated grammar is dynamically displayed in the GUI for easy reference.

To ensure the integrity and correctness of grammars, the application incorporates built-in validation mechanisms. These mechanisms serve to verify that grammars are correctly formatted and that operations such as adding or removing symbols and rules are performed safely. Specifically, the validation mechanisms check whether:

- The nonterminal or terminal being added by the user already exists in the grammar.
- All symbols in the newly added rule are present in the set of nonterminals and terminals.
- If a nonterminal or terminal is removed, any production rules dependent on the removed symbol are also eliminated.

By implementing these validations, the application provides users with a smooth and error-free experience when working with grammars.

3.3.2 Grammar Files Management and Selection

The application offers seamless management of multiple grammar files, providing users with convenient tools for organizing and selecting grammar resources.

3.3.2.1 Effortless Management of Multiple Grammar Files

Users can effortlessly add multiple grammar files to a dedicated list box within the application's interface. This feature allows users to maintain an organized collection of grammar resources, facilitating easy access and navigation.

3.3.2.2 Convenient Viewing of Grammar File Contents

The intuitive interface of the application enables users to view the contents of each grammar file directly within the application. By selecting a grammar file from the list box, users can conveniently access and review its specifications without the need for external text editors or viewers.

3.3.2.3 Streamlined Selection for Operations

Users can seamlessly choose a grammar file from the list box to perform various operations such as editing, constructing derivations, transformations, and analysis. This streamlined selection process enhances user efficiency and workflow management.

3.3.2.4 Built-in Validation for Correct Grammar Format

The application incorporates built-in validation mechanisms to ensure the correctness of grammar files before performing operations. When a user attempts to perform operations on a selected grammar file, the application checks the format of the file. If the format is incorrect, an error message is displayed, providing users with immediate feedback and guidance for rectification, see Figure 3.1.

3.3.2.5 Error Handling for Missing Grammar File Selection

Additionally, the application includes error handling for scenarios where users attempt to perform operations without selecting a grammar file. In such cases, an error message is displayed in the GUI, prompting users to select a grammar file before proceeding with the operation.

By offering seamless management of grammar files, convenient viewing options, streamlined selection for operations, and robust error handling, the application ensures a user-friendly and efficient experience in working with diverse grammar specifications.

3.3.3 Interactive Derivation Construction

The application empowers users to interactively construct derivations of words based on a specified grammar, providing a dynamic and intuitive approach to exploring language generation. Users can step through the derivation process, observing each transformation of the input string in real-time.

3.3.3.1 Display Options for Derivation Steps

During the derivation process, users have the option to view the step-by-step transformations in two distinct formats: graphical derivation tree and sentential form. In each derivation step, the newly applied production rule and the nonterminal that has been rewritten are highlighted with square brackets in the sentential form, providing users with clear visual cues of the changes made.

Once the derivation is complete and no more nonterminals remain in the string, users are presented with the generated word along with the complete derivation tree and sentential form. This comprehensive display provides users with a holistic view of the derivation process and the resulting word.

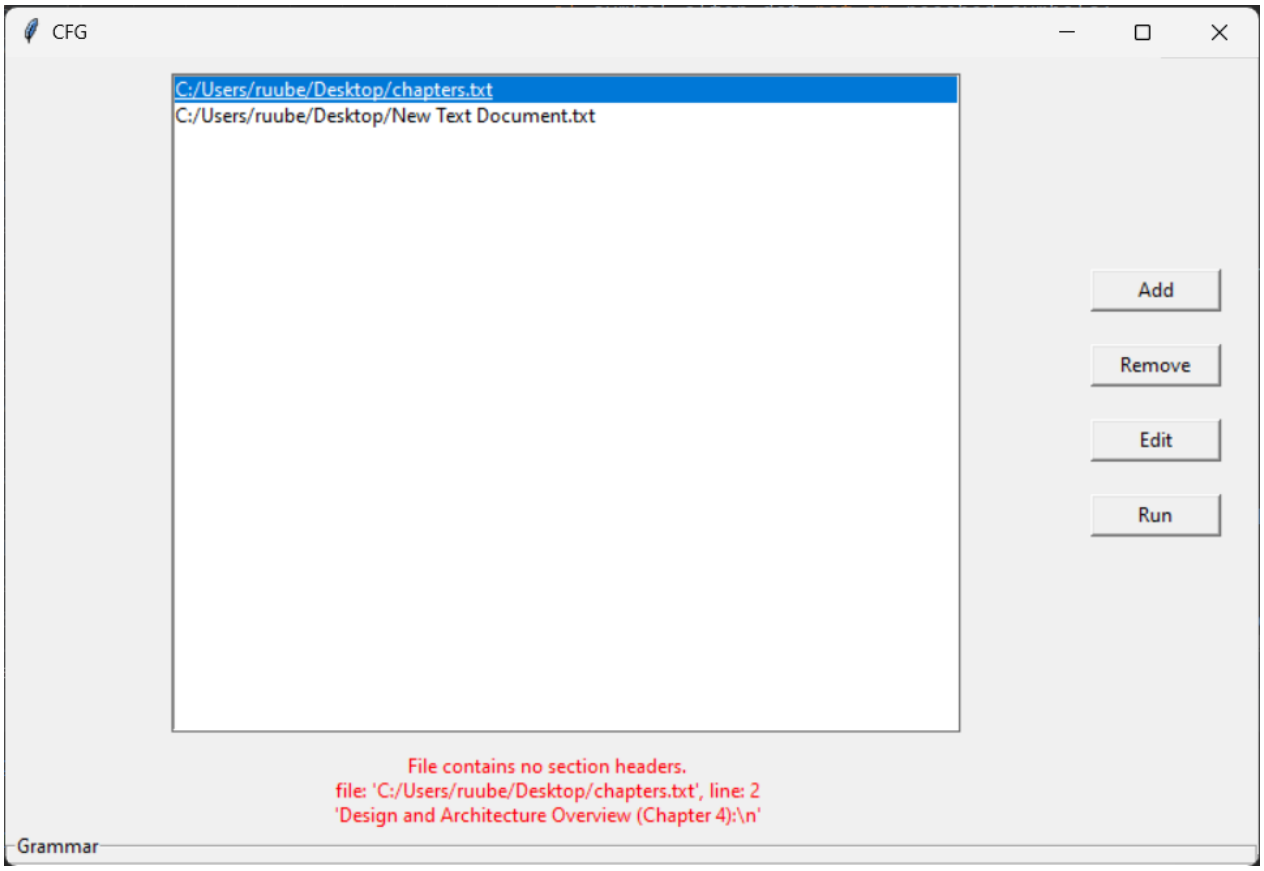


Figure 3.1: Validation of grammar file

3.3.3.2 Highlighting Multiple Occurrences of Nonterminals

In cases where multiple occurrences of the same nonterminal can be expanded in a derivation step, the application provides intuitive highlighting in the graphical derivation tree. Each occurrence of the nonterminal is highlighted in pink, with corresponding numbers displayed below each node to signify its position in the derivation step. This feature enables users to easily distinguish between multiple occurrences and make informed decisions on which nonterminal to expand next.

3.3.3.3 Undo and Redo Functionality

To further enhance user flexibility and control, the application includes undo and redo buttons, allowing users to navigate through previous derivation steps or redo specific manipulations. This feature enables users to experiment with different derivation paths and backtrack if necessary, ensuring a seamless and error-free exploration of language generation.

By offering interactive derivation construction with display options and convenient navigation controls, the application empowers users to explore and understand language generation processes with ease and flexibility.

3.3.4 Detailed Transformation Tools

The application provides users with a robust suite of transformation functionalities, enabling comprehensive manipulation and refinement of grammar structures. These transformation tools empower users to optimize grammars for various purposes while maintaining semantic equivalence.

3.3.4.1 Reduction of Grammars

The reduction feature allows users to simplify grammars by eliminating redundant or unnecessary productions. By identifying and removing redundant rules, users can streamline grammars without altering their language generation capabilities.

3.3.4.2 Removal of Epsilon Productions

Epsilon productions are systematically removed from grammars to enhance clarity and streamline parsing processes. The application facilitates the seamless elimination of epsilon productions while ensuring that the resulting grammar remains equivalent to the original grammar in terms of language generation.

3.3.4.3 Elimination of Unit Productions

Unit productions are systematically eliminated to enhance the transparency and simplicity of grammars. By removing unit productions, users can transform complex grammars into more structured and manageable forms, facilitating easier parsing and analysis.

3.3.4.4 Conversion to Chomsky Normal Form (CNF) and Greibach Normal Form (GNF)

The application offers advanced functionality for converting grammars to Chomsky Normal Form (CNF) and Greibach Normal Form (GNF). These canonical forms are widely used in formal language theory and parsing algorithms, making the transformed grammars more amenable to analysis and processing.

3.3.4.5 Comprehensive Transformation Guidance

The application goes beyond basic transformation functionalities by providing users with an immersive and interactive transformation experience. When performing transformations on a grammar, users are presented with a comprehensive popup window that guides them through each step of the transformation process. This popup window consists of three frames - the *grammar frame*, the *transformation frame*, and the *explanation frame* - and two buttons to navigate to the next or previous step in the transformation process.

In the grammar frame, users can view the original or grammar specification in the previous transformation step. This frame provides users with a reference point to compare the changes made during the transformation process.

The transformation frame displays the step-by-step process of the transformation, showcasing the resulting grammar after each transformation step. Users can navigate through the transformation steps using intuitive buttons provided within the window.

In the explanation frame, users can find detailed explanations for each transformation step. This includes information on why and how each step was performed, helping users understand the rationale behind the transformation process. Additionally, production rules involved in the current transformation step are highlighted in the grammar displayed in the grammar frame, providing users with visual cues to correlate explanations with specific grammar elements.

3.3.4.6 Enhanced User Experience

To accommodate lengthy transformation steps, a scrollbar is provided in the transformation frame, ensuring that users can access all transformation details conveniently.

3.3.4.7 Saving Transformed Grammar

The application allows users to save the transformed grammar to a new grammar file, enabling seamless integration of transformed grammars into their projects or analyses.

By providing an immersive transformation experience with detailed explanations, visual cues, and convenient navigation options, the application empowers users to perform complex grammar transformations with confidence and clarity.

3.3.5 Analysis Tools for Detection of Grammar Types

The Analysis Tools provided by the application offer users a powerful toolkit for in-depth analysis of grammars. These tools enable users to gain comprehensive insights into the structure and properties of grammars, facilitating a deeper understanding of formal language theory and parsing algorithms.

3.3.5.1 Computation of FIRST and FOLLOW Sets

The application provides users with the capability to compute the FIRST and FOLLOW sets of nonterminals within a grammar, essential components in parsing techniques such as LL(1) and LR(1) parsing. These sets are instrumental in predicting terminal symbols that can appear at the beginning of strings derived from nonterminals (FIRST set) and those that can immediately follow nonterminals in derivations (FOLLOW set).

By computing FIRST and FOLLOW sets, users gain insights into the predictability and lookahead behavior of their grammars. This understanding is invaluable in optimizing parsing algorithms

and designing efficient parsers. Additionally, the application visually represents the nodes and transitions within the grammar, illustrating how terminals in the FIRST set of nonterminals are reached during parsing. This visual representation enhances users' comprehension of grammar structures and parsing processes, facilitating informed decisions in grammar design and optimization.

3.3.5.2 Detection of Grammar Types

Users can leverage the application's capabilities to detect the type of a grammar, including LL(1), LR(0), and LR(1) grammars. This feature enables users to classify grammars based on their parsing properties, facilitating the selection of appropriate parsing techniques and algorithms for grammar processing. Additionally, when detecting if a grammar is LL(1), the application displays which rule to apply according to the next input symbol in the input string along with the computed FIRST and FOLLOW sets, providing valuable insights into LL(1) parsing decisions.

3.3.5.3 Conflict Detection

The analysis tools provided by the application include conflict detection functionality, allowing users to identify and resolve parsing conflicts such as shift/reduce and reduce/reduce conflicts. These conflicts arise in LR parsing due to ambiguities or limitations in the grammar's parsing behavior. By detecting conflicts, users can refine their grammars to improve parsing efficiency and accuracy.

3.3.5.4 Generation of LR States

The application allows users to generate LR states from a given grammar, providing a detailed view of the parsing configurations and transitions in an LR parsing table. By visualizing the LR states, users can gain insights into the parsing process and identify potential parsing conflicts or ambiguities. Moreover, when detecting LR grammars, the application constructs and displays the ACTION table and GOTO table along with LR states. If there are conflicts, they are highlighted in red in the ACTION table, enabling users to identify and address parsing conflicts effectively.

3.3.5.5 Display of Analysis Results

The analysis results, including computed sets, LR items, and conflict highlights, are presented to users in a separate window just like the one used for transformation functions but detecting LR grammar functions the window is slightly different by displaying tables in the interface such as ACTION table, GOTO table and LR items. Detailed insights and explanations accompany the analysis results, providing users with contextual understanding and guidance on interpreting the analysis findings effectively.

Overall, the Advanced Analysis Tools offered by the application empower users to conduct thorough analysis and optimization of grammars, enhancing their proficiency in formal language

theory and parsing techniques. With intuitive functionalities and informative insights, users can explore the intricacies of grammars and parsing algorithms with confidence and clarity.

3.4 Navigating the Graphical Interface

The application's graphical interface is designed for user convenience, featuring three main pages: the *Selection Page*, the *Edit Page*, and the *Derivation Page*. These pages streamline grammar file management, editing, and derivation construction. Additionally, when users perform transformation or analysis operations, a separate window is created which guides them through the process. This structured interface enhances usability and ensures an intuitive user experience.

3.4.1 Selection Page

Upon launching the application, users are greeted with the *Selection Page* which we can see below in Figure 3.2. This page features a user-friendly interface designed to facilitate effortless management of grammar files. Users can easily add multiple grammar files to the listbox using the *Add* button and remove selected files using the *Remove* button.

The *listbox* displays the added grammar files, allowing users to conveniently select and perform actions. Below the *listbox* and control buttons, a *grammar frame* provides a space to display the contents of the selected grammar file. This enables users to quickly review the specifications of each grammar.

Additionally, the *Selection Page* offers seamless navigation to other key functionalities of the application. Users can transition to the *Edit Page* by clicking the *Edit* button to make modifications to grammar files. Similarly, clicking the *Run* button allows users to proceed to the *Derivation Page* to perform derivations.

Overall, the intuitive interface of the Selection Page enhances the user experience by streamlining the process of accessing and interacting with grammar resources.

3.4.1.1 Grammar Files Management and Selection

The initial step in utilizing any feature within the application begins on the *Selection Page* by selecting a grammar file. This operation is conducted within the user-friendly interface of the *Selection Page*, featuring a listbox where users can conveniently manage multiple grammar files.

To add grammar files, users simply click the *Add* button, triggering the appearance of a *file dialog box*. Within this dialog box, users navigate their computer directories to select desired grammar files. Here, users have the flexibility to select either a single grammar file or multiple files by holding the Alt key while clicking. Upon selection, users finalize their choice by clicking *Open* seamlessly adding the chosen files to the *listbox* on the *Selection Page*.

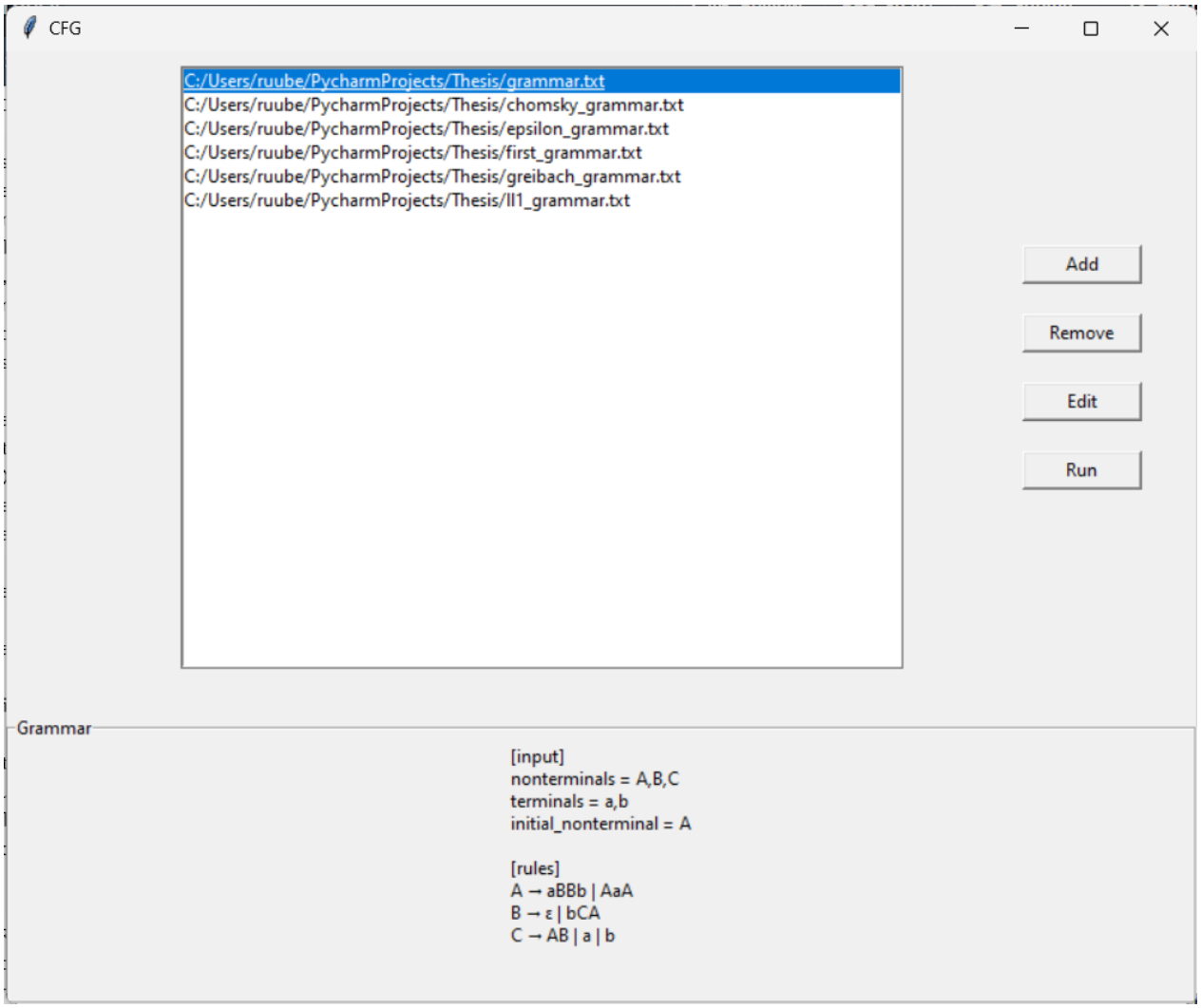


Figure 3.2: Selection Page

Once added, users can effortlessly inspect the specifications of each grammar file directly within the *listbox* interface. By selecting a specific grammar file, its contents are dynamically displayed in the *grammar frame*, see Figure 3.2, allowing users to review and assess each file's specifications. This intuitive design facilitates efficient decision-making regarding which grammar file to operate on.

Should the need arise to remove a grammar file, users can do so with ease. By selecting the desired file within the *listbox* and clicking the *Remove* button, the selected grammar file is promptly removed from the list, streamlining the organization of grammar resources.

3.4.2 Edit Page

The Edit Page serves as a comprehensive toolkit for users to manipulate, transform, and analyze the specifications of the selected grammar file. Organized into three distinct frames-*Edit Frame*, *Transform Frame*, and *Grammar Frame*-the page offers users intuitive controls and functionalities for efficient grammar operations, see Figure 3.3.

3.4.2.1 Edit Frame

This frame provides users with a range of editing tools, including entry boxes, dropdown boxes, and buttons, facilitating the addition and removal of nonterminals and terminals. Users can also change the initial nonterminal and modify production rules directly within this frame, enabling seamless customization of grammar specifications.

3.4.2.2 Transform Frame

In this frame, users can access a variety of transformation options and analysis tools. Buttons are available for performing transformations such as grammar reduction, removal of epsilon rules, elimination of unit rules, and conversion to Chomsky Normal Form or Greibach Normal Form. Additionally, users can initiate analyses such as computing FIRST and FOLLOW sets, checking for LL(1), LR(0), and LR(1) properties. Each transformation or analysis option is conveniently accessible via dedicated buttons within this frame.

3.4.2.3 Grammar Frame

The *Grammar Frame* dynamically displays the specifications of the grammar file, providing users with real-time updates and references as they make changes. This interactive display enhances user understanding and facilitates seamless navigation through the grammar specifications.

3.4.2.4 Navigation and Interaction

Users can easily navigate back to the *Selection Page* by clicking the *Back* button within the *Edit Frame*, ensuring smooth transitions between different sections of the application. Additionally, clicking any button within the *Transform Frame* opens a separate window dedicated to displaying the specific transformation or analysis process, enabling focused interaction and exploration.

3.4.3 Editing Grammar Specifications

Once the user has selected a grammar file and navigated to the *Edit Page*, the interface provides intuitive tools for editing the grammar of the selected file. Upon arrival, the specifications from the file are automatically loaded into the initial nonterminal entry box, rules dropdown menu, and its entry box, see Figure 3.3. This comprehensive functionality encompasses various key operations,

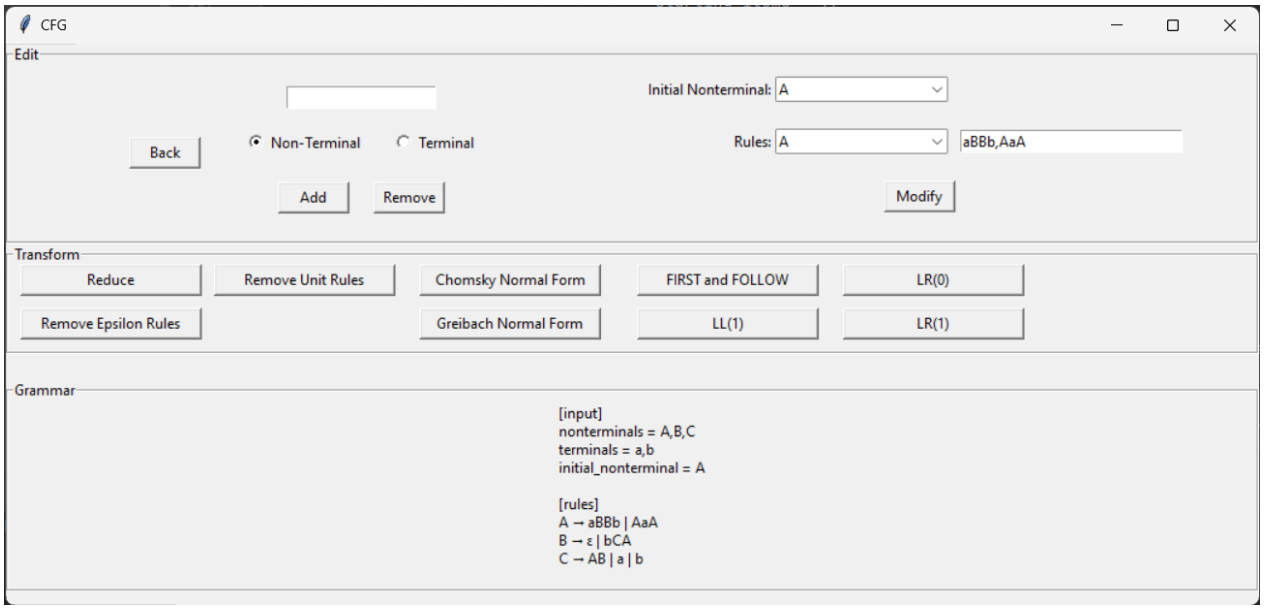


Figure 3.3: Edit Page

such as adding and removing terminals or nonterminals, changing the initial non-terminal, and modifying production rules.

3.4.3.1 Adding and Removing of Terminals and Nonterminals

Adding and removing terminals and nonterminals is straightforward. Users can simply enter the symbol they wish to add or remove in the provided *entry box*. Below this box, two radio buttons are available: one for selecting nonterminals and the other for terminals. Users can choose the appropriate option and then click the *Add* button to include the symbol as either a nonterminal or terminal. Conversely, clicking the *Remove* button removes the symbol from the respective category.

In cases where users need to add a substring or string that references an existing nonterminal or terminal, they can enclose the substring or string within angle brackets ('<' and '>'). This feature enables users to effectively manage complex grammar specifications with ease.

3.4.3.2 Changing the Initial Non-terminal

The initial non-terminal serves as the starting point for grammar derivation, influencing the structure and generation of language constructs. Users can easily customize this aspect by accessing the *Edit Page* and selecting the desired initial non-terminal from the provided dropdown menu. Upon selection, users can save this modification with a simple click of the *Modify* button, ensuring seamless integration into the grammar file.

3.4.3.3 Modifying the Production Rules

Production rules define the transformation of non-terminals into strings of terminals and non-terminals, shaping the language generated by the grammar. The application allows users to modify them as needed within the interface. Within the *Edit Page*, users can select specific production rules from the *rules dropdown menu*, view their corresponding right-hand sides, and make modifications directly in its respective *entry box*. Upon completion, users can save these modifications with the *Modify* button, facilitating efficient refinement of grammar specifications.

3.4.4 Perform Transformations

After selecting a grammar file and accessing the *Edit Page*, users encounter dedicated buttons representing each transformation function within the *transform frame* of the *Edit Page*. Clicking any transformation button opens a separate window dedicated to that specific transformation. Within this window, users are presented with navigation options to facilitate the progression through the transformation steps. A '$\leftarrow$' button allows users to navigate to the previous step of the transformation, while a '$\rightarrow$' button enables movement to the subsequent step in the transformation process, see Figure 3.6.

Additionally, users are provided with options to save the transformed grammar for future use. Clicking the *Save* button triggers a file dialog box, allowing users to specify a name for the file and save the transformed grammar to their desired location. Conversely, clicking the *Close* button terminates the transformation window, providing a streamlined approach to managing transformation processes.

3.4.5 Analyzing Grammars

In the *Transform frame* of the *Edit Page*, users can find analysis buttons alongside the transformation buttons. These analysis functions serve to dissect the grammar structure, offering insights into its properties and behavior. Similar to transformations, clicking these buttons initiates a process that opens a dedicated window tailored to the specific analysis task.

For operations such as computing FIRST and FOLLOW sets and determining LL(1) grammar properties, the window follows a familiar design, presenting information relevant to the analysis process. However, for LR(0) and LR(1) analysis, the window adopts a different layout and interaction paradigm.

In the LR(0) and LR(1) analysis window, users are greeted with two tables: the *ACTION table* and the *GOTO table*. Unlike transformation windows, there are no interactive buttons provided here. Instead, users can interact directly with the tables by selecting a row (state). Upon selection, a table dynamically appears below the *GOTO table*, displaying LR items associated with the chosen state, see Figure 3.5. This intuitive interaction enables users to explore LR parsing details efficiently, facilitating deeper understanding and analysis of grammar structures.

3.4.6 Derivation Page

The *Derivation Page* offers users a dynamic environment for interactively constructing derivations and visualizing the step-by-step process. Comprising three distinct frames—*Execute Frame*, *Derivation Tree Frame*, and *Sentential Form Frame*—as we can see in the Figure 3.4 the page provides users with intuitive tools and displays for efficient derivation execution and analysis.

3.4.6.1 Execute Frame

Central to the *Derivation Page* is the *Execute Frame*, which houses the interactive tools for performing derivations. Here, users can initiate and control derivation processes, navigate through derivation steps using undo and redo functionalities, and interact with the derivation in real-time. The *Execute Frame* empowers users to engage with derivations dynamically, facilitating seamless exploration and experimentation. Users can easily navigate back to the *Selection Page* by clicking the *Back* button within this frame.

3.4.6.2 Derivation Tree Frame and Sentential Form Frame

The *Derivation Tree Frame* and *Sentential Form Frame* serve as visual aids, dynamically displaying the graphical derivation tree and sentential form of the ongoing derivation, respectively. These frames offer users real-time insights into the structure and progression of the derivation, enhancing understanding and facilitating effective analysis.

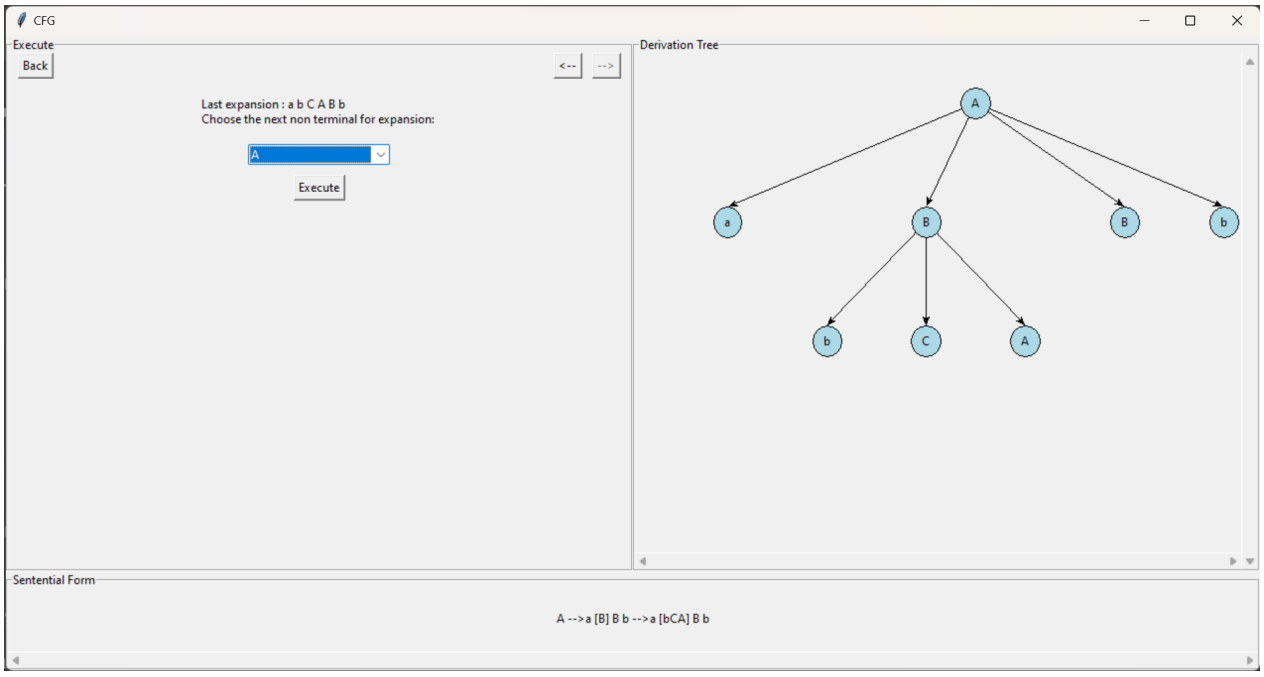


Figure 3.4: Derivation Page

3.4.7 Constructing Derivations

Select a grammar file and navigate to the *Derivation Page*. In the *Derivation Page*, users can initiate the derivation process by selecting the initial nonterminal from the *combo box* and clicking the *Execute* button. Subsequently, users can interactively progress through the derivation steps by selecting the desired options from the *combo box* and clicking *Execute* until the desired result is achieved.

Derivations are presented in both sentential form and graphical tree formats, providing visual representation of each step in the derivation process (refer to Figure 3.4).

The undo function allows users to revert to the previous derivation step by clicking the '$\leftarrow$' button, facilitating easy correction of errors or changes in manipulation.

Users can redo specific manipulations using the redo function, enabling effortless revisiting of desired derivation steps by clicking the '$\rightarrow$' button.

Clicking the *Back* button navigates users back to the *Selection Page*.

3.4.8 Window for Displaying Results

When users initiate transformation or analysis operations within the application, a dedicated *result window* is generated to provide a focused environment for conducting these tasks. The result window comes in two types, each tailored to specific types of operations: LR parsing analyses and other transformations or analyses.

3.4.8.1 LR Parsing Window

In the LR parsing analyses window, the window features a *grammar frame* alongside several frames dedicated to displaying critical components such as the ACTION table, GOTO table, and LR items, see Figure 3.5. Each frame is equipped with scrollbars for easy navigation, allowing users to explore detailed information about specific states within the tables. Beneath the *grammar frame*, the results of the analysis are presented, providing users with comprehensive insights into the parsing process.

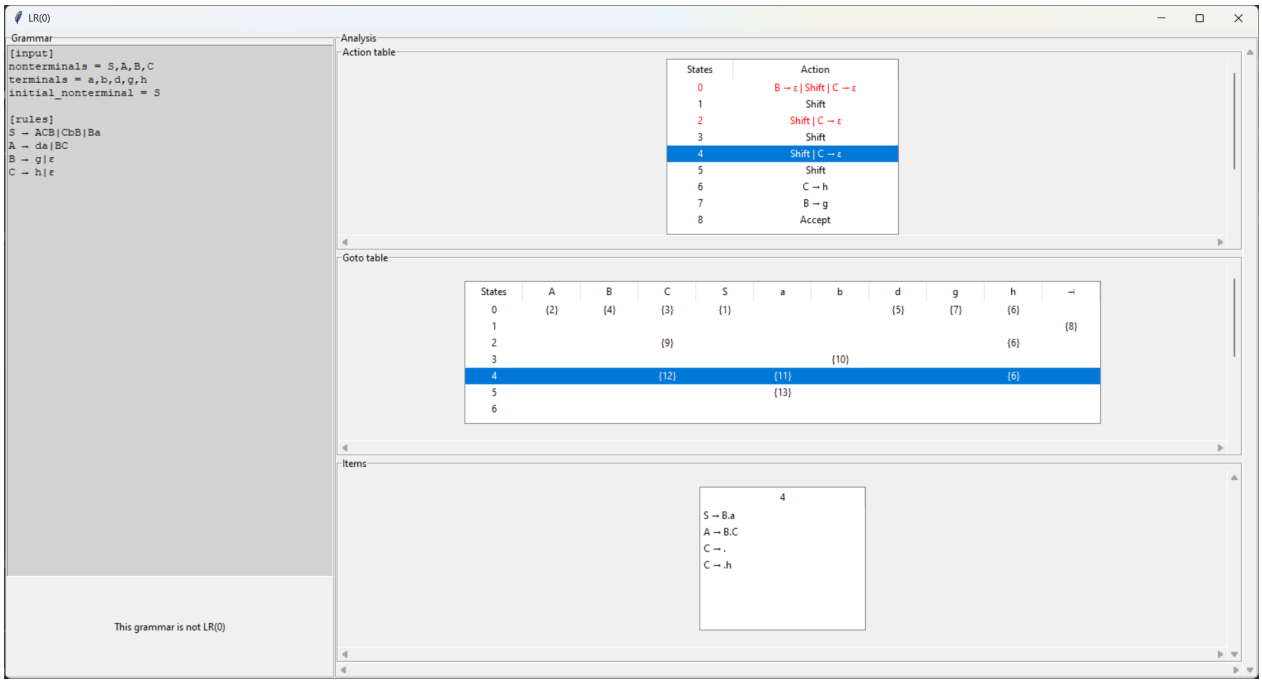


Figure 3.5: Result window - LR parsing type

3.4.8.2 Transformation window

As we can see in the Figure 3.6. The transformation type of the *result window* consists of three frames: the *grammar frame*, *transformation frame*, and *explanation frame*. Here, users can visualize the transformation steps performed and understand the rationale behind each step through detailed explanations. The *grammar frame* displays the grammar specification of the selected file, while the *transformation frame* showcases the transformation process in a step-by-step manner. The *explanation frame* provides insights into the reasons behind each transformation step, enhancing user's understanding of the process.

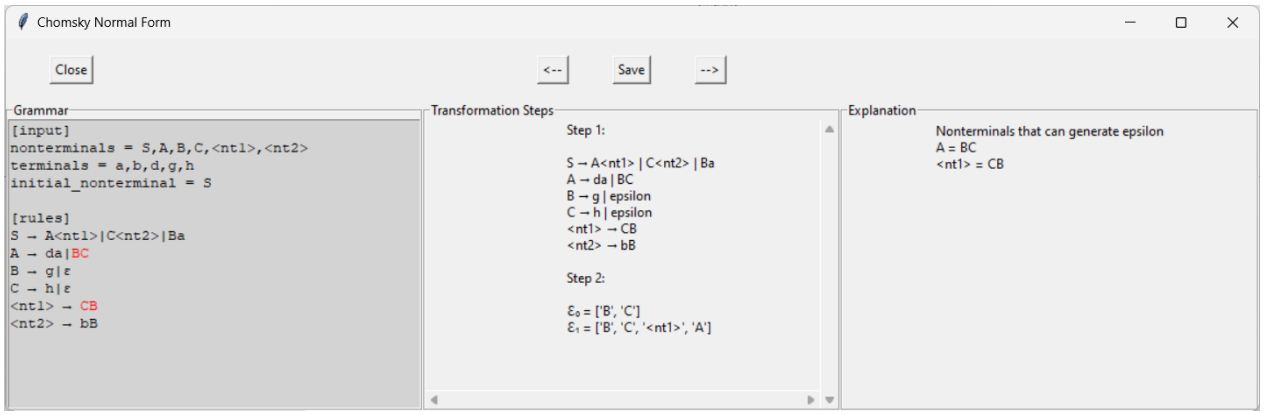


Figure 3.6: Result window - Transformation type

3.4.8.3 Interactive Functionality

Both types of the *result window* offer interactive functionality to enhance user experience and facilitate efficient workflow. For transformations, users can navigate through the transformation steps using intuitive buttons provided above the frames. Additionally, users have the option to save the transformed grammar specification to a new grammar file directly from the *result window*. To close the window and return to the main interface, users can simply click the *close* button within the frame.

By providing focused environments tailored to specific operations, the result window enhances user's ability to conduct detailed analyses and transformations within the application. The interactive features and intuitive design ensure a seamless and productive user experience.

3.5 Grammar File Format

The grammar file format serves as the foundation for inputting and manipulating context-free grammars within the application. The grammar file must be a text file i.e., (`.txt`) extension. It consists of two distinct sections.

3.5.0.1 [input] Section

The [input] section outlines essential components of the grammar:
nonterminals: Defines a list of nonterminal symbols used in the grammar. Each nonterminal is separated by a comma.
terminals: Defines a list of terminal symbols used in the grammar. Each terminal is separated by a comma.
initial_nonterminal: Specifies the starting or initial nonterminal symbol for the grammar.

Note: Nonterminal symbols and Terminal symbols cannot be same. If the user wishes to add a terminal or nonterminal symbol which is a subset of another input symbol or combination of two or more symbols, he/she can put the desired symbol between angular brackets '<' and '>'.

3.5.0.2 [rules] Section

The [rules] section represents the production rules for the nonterminals in the grammar. Each nonterminal's rules are specified individually. Each nonterminal symbol is defined with its associated production rules. The left-hand side (LHS) represents the nonterminal symbol, followed by the equal sign ('='), and then its production rules. Multiple production rules for a nonterminal are separated by commas.

3.5.1 Example

```
[input]
nonterminals = A,B,C
terminals = a,b
initial_nonterminal = A

[rules]
A = aBBb,AaA
B = epsilon,bCA
C = AB,a,b
```

Example grammar files can be found in the `"grammar files"` folder within `CFGAnalyzer.zip` file. These example files can be used to explore the features and functionality of the application.

Chapter 4

Design and Architecture

In software development, design and architecture are foundational elements that profoundly influence the effectiveness and viability of an application. Design involves the creative and strategic process of conceptualizing, planning, and defining the structure and behavior of a software system. Conversely, architecture concentrates on the high-level organization and arrangement of components within the system, serving as a roadmap for its construction.

This section delves into the design and architecture of the application, highlighting their critical roles in shaping its functionalities.

4.1 Architecture Overview

The application's architecture is designed following the Model-View-ViewModel (MVVM) pattern, ensuring a clear separation of concerns and promoting modularity. This architecture enhances maintainability, scalability, and testability of the application. Figure 4.1 provides a visual representation of the architecture.

4.1.1 Model Layer

The Model layer encapsulates the data representation and manipulation components. It is responsible for storing and managing grammar specifications, including rules, terminals, nonterminals, and associated operations. Within the application, the model layer is implemented in the `cfg.py` file.

4.1.2 View Layer

The View layer comprises the graphical user interface (GUI) components, implemented using the *Tkinter* library for Python. It provides the visual presentation of grammar specifications, editing tools, interactive derivation construction, display of derivations, transformation tools, and analy-

sis tools. The View layer, implemented in the `window.py` file, handles the visual aspects of the application.

4.1.3 ViewModel Layer

Serving as an intermediary between the Model and View layers, the ViewModel layer manages user interactions and facilitates communication between the GUI and underlying application functionalities. This layer, implemented in the `functions.py` file, orchestrates the behavior of the application and ensures seamless interaction between the user interface and the data model.

By adhering to the MVVM pattern, the application achieves a modular and maintainable architecture, enabling efficient development and extensibility. This architecture fosters flexibility and scalability, allowing for future enhancements and modifications with minimal impact on existing functionality.

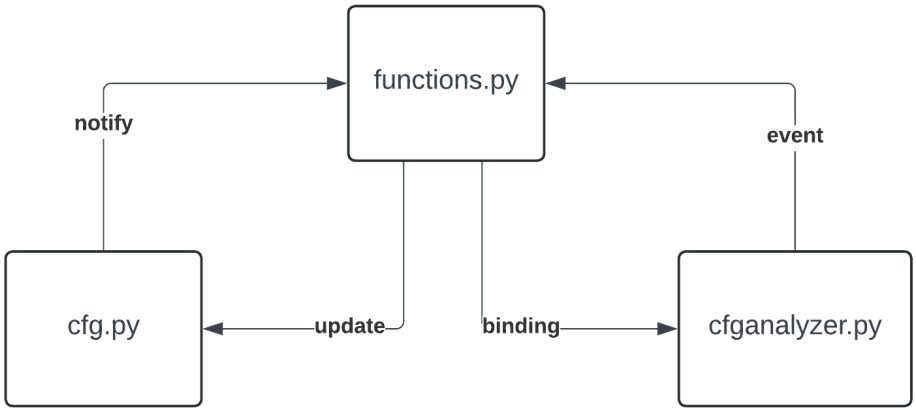


Figure 4.1: Representation of architecture

4.2 Class Diagram

The application is designed with a modular structure utilizing object-oriented principles. The class diagram below Figure 4.2 represents the main components and their relationships within the system.

4.2.1 CFG Class

At the heart of the application lies the *CFG* class, which serves as the cornerstone for handling context-free grammars and their manipulations. This class encapsulates essential functionalities crucial for grammar operations.

As depicted in Figure 4.2, the *CFG* class forms composition relationships with two other key classes: the *Stack* class and the *TreeNode* class. These relationships signify the integral role of the *CFG* class in orchestrating interactions with these auxiliary components, facilitating seamless grammar operations and derivations.

4.2.1.1 Attributes

- **rules:** A dictionary storing the production rules.
- **nonterminals:** A list of nonterminals in the grammar.
- **terminals:** A list of terminals in the grammar.
- **initial_nonterminal:** The starting nonterminal symbol.
- **stack:** An instance of the *Stack* class used in grammar manipulation.
- **stack_tree:** An instance of the *Stack* class specifically used for constructing derivation trees.

4.2.1.2 Methods

Several methods for managing rules, creating derivation steps, reading/writing configurations, adding/removing values and rules, and expanding nonterminals in the grammar.

4.2.2 Stack Class

The *Stack* class represents a stack data structure used for handling sequential sentential forms in the manipulation of context-free grammars.

4.2.2.1 Attriubutes

- **data:** A list that stores the elements of the stack.
- **index:** An integer representing the current index within the stack.

4.2.2.2 Methods

- **push():** Adds an item to the stack.
- **undo():** Reverts to the previous state (sentential form or tree) in the manipulation process.
- **redo():** Reapplies a previously undone step in the manipulation.
- **current():** Retrieves the current state from the stack.
- **printst():** Prints the sentential form at a given index.

4.2.3 TreeNode Class

The *TreeNode* class represents nodes in a tree data structure used for constructing derivation trees from context-free grammars.

4.2.3.1 Attriubutes

- **data:** The data stored in the node.
- **children:** A list of child nodes.
- **parent:** The parent node in the tree.

4.2.3.2 Methods

- **get_level():** Retrieves the level of the node within the tree.
- **print_tree():** Prints the tree structure at a given index
- **add_child():** Adds a child node to the current node.

4.2.4 Transform Class, LLParser Class, LRParser Class

The *Transform class* is responsible for implementing various transformations on the given context-free grammar (CFG). These transformations include reduction, elimination of epsilon and unit rules, as well as conversion to Chomsky normal form and Greibach normal form.

The *LLParser* class provides essential functionalities for analyzing context-free grammars (CFGs). Its methods are primarily focused on computing the FIRST and FOLLOW sets, crucial for various analyses and grammar property checks. Additionally, the class has methods to determine whether a given grammar conforms to the LL(1) property, a significant aspect in parser construction.

The *LRParser class* is dedicated to handling LR parsing operations and analyses. It encompasses essential methods for computing LR(0) and LR(1) items, as well as generating ACTION and GOTO tables crucial for LR parsing. These items are essential for determining the LR(0) and LR(1) properties of a grammar.

All of these classes have abstraction relationship with the *CFG* class. This abstraction allows these classes to interact seamlessly with *CFG* objects, accessing necessary grammar data and performing transformations and analysis of grammar types effectively.

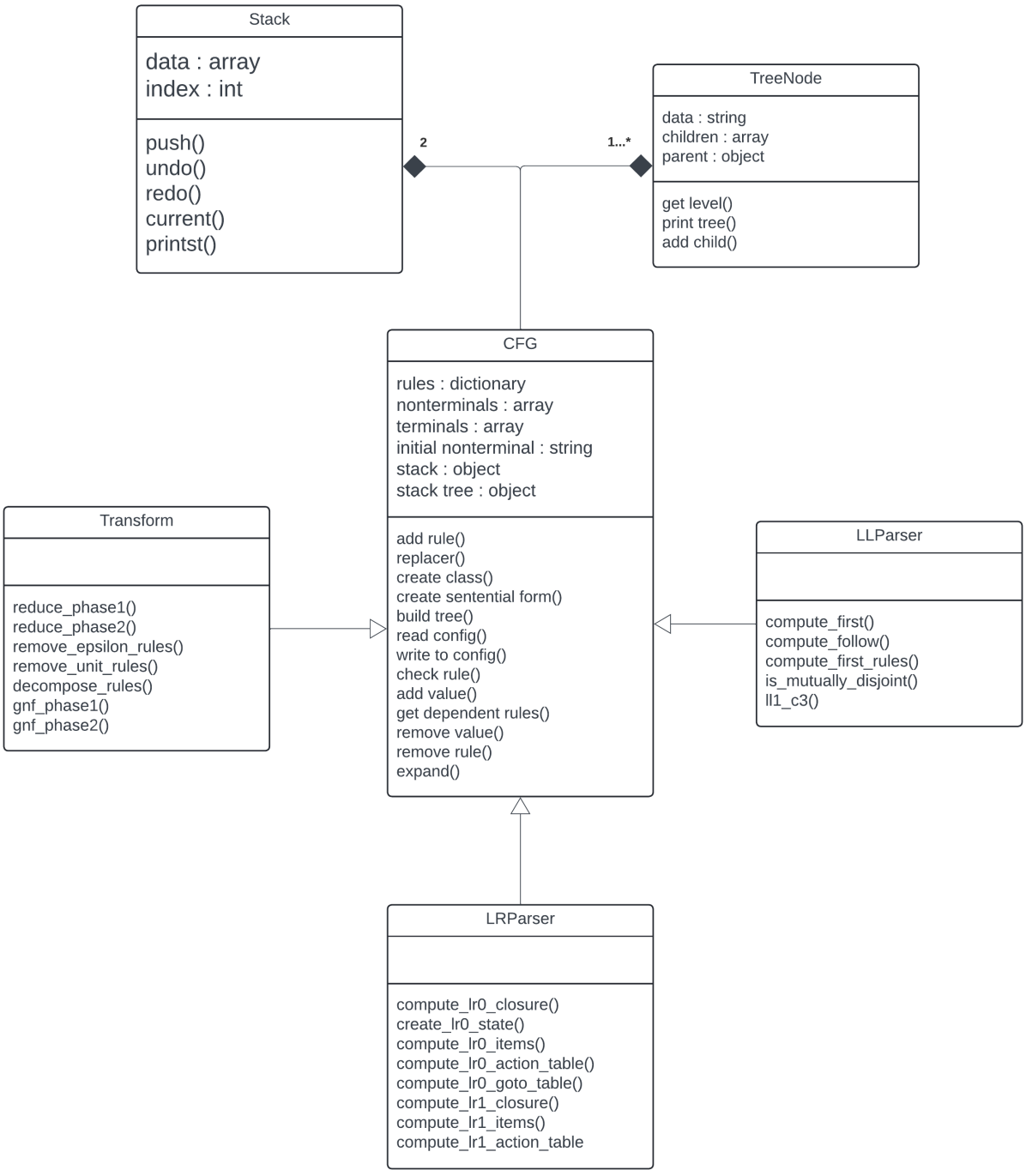


Figure 4.2: Class Diagram

4.3 Evolution of the User Interface

The evolution of the application's interface and functionality represents a significant improvement in user experience and efficiency. In the initial version of the semestral project, users were constrained to browsing and selecting one grammar file at a time using the file dialog box, which could be cumbersome and inefficient, especially when dealing with multiple grammar files.

To address this limitation, the application now features a dedicated selection page, allowing users to add multiple grammar files to a listbox for easy access. This enhancement streamlines the process of viewing grammar file contents, empowering users to quickly select and decide whether to perform derivations, editing, transformations, or analysis.

Moreover, the edit page has been refined to include comprehensive editing tools and buttons for performing transformations and detecting grammar types. These functionalities now generate separate dedicated windows to display the process of the operation performed. This approach enables users to conduct multiple operations concurrently, with each operation presented in its own window for enhanced clarity and usability.

Overall, these updates reflect a commitment to improving user experience by optimizing workflow efficiency and providing intuitive tools for working with context-free grammars.

4.4 Sequence Diagram for Constructing Derivations

The sequence diagram illustrates the flow of interactions among the *User*, *Interface*, and *CFG* objects in the context of manipulating context-free grammars.

4.4.1 Participants

User: Represents the end-user interacting with the application.
Interface: Manages communication between the *User* and the *CFG* object, handling user inputs and displaying outputs.
CFG: Represents the core context-free grammar manipulation functionalities.

4.4.2 Interactions

4.4.2.1 User-Interface Interaction

The *User* interacts with the *Interface*, providing inputs such as choosing the next expansion for a specified nonterminal, specifying the occurrence of a nonterminal to expand, and choosing the next nonterminal for expansion.

4.4.2.2 Interface-CFG Interaction

The *Interface* communicates with the *CFG* object to perform grammar manipulations based on the user-provided inputs. It triggers the *CFG* to expand the specified nonterminal in the grammar, updating the sentential form and derivation tree accordingly.

4.4.2.3 Sentential Form and Graphical Tree Display

Upon manipulation by the *CFG*, the *Interface* receives the updated sentential form and derivation tree. The *Interface* displays the sentential form and graphical tree representations to the *User*, allowing visual feedback on the grammar manipulation.

4.4.2.4 Looping Iteration

This process iterates until there are no more nonterminals to expand in the string, maintaining an interactive loop for manipulation and display.

4.5 Transformations and Detection of Grammar Types

In the application, the transformation and analysis functionalities play a pivotal role in manipulating and understanding context-free grammars (CFGs). When a user triggers a transformation or analysis operation through the interface, the corresponding function within the `functions.py` module, serving as the ViewModel, is invoked. This function orchestrates the generation of necessary input data and orchestrates the execution of the specified operation by calling methods in the *Transform*, *LLParser*, or *LRParser* classes, located within the `cfg.py` file, which serves as the Model component of the architecture.

4.5.1 Operation Execution Flow

- **Input Data Generation:** The ViewModel function generates the input data required for the operation, such as the CFG specifications or grammar rules.
- **Method Invocation:** The function calls the appropriate methods in the *Transform*, *LLParser*, or *LRParser* classes to execute the transformation or analysis operation.
- **Stack Maintenance:** Throughout the operation, the application maintains a stack data structure to track the transformation steps and provide explanations for each step. This stack enables users to easily follow the transformation process and understand the changes made to the grammar.
- **Dedicated Window Display:** The information recorded in the stack is utilized to populate a dedicated window created specifically for the transformation or analysis operation. This

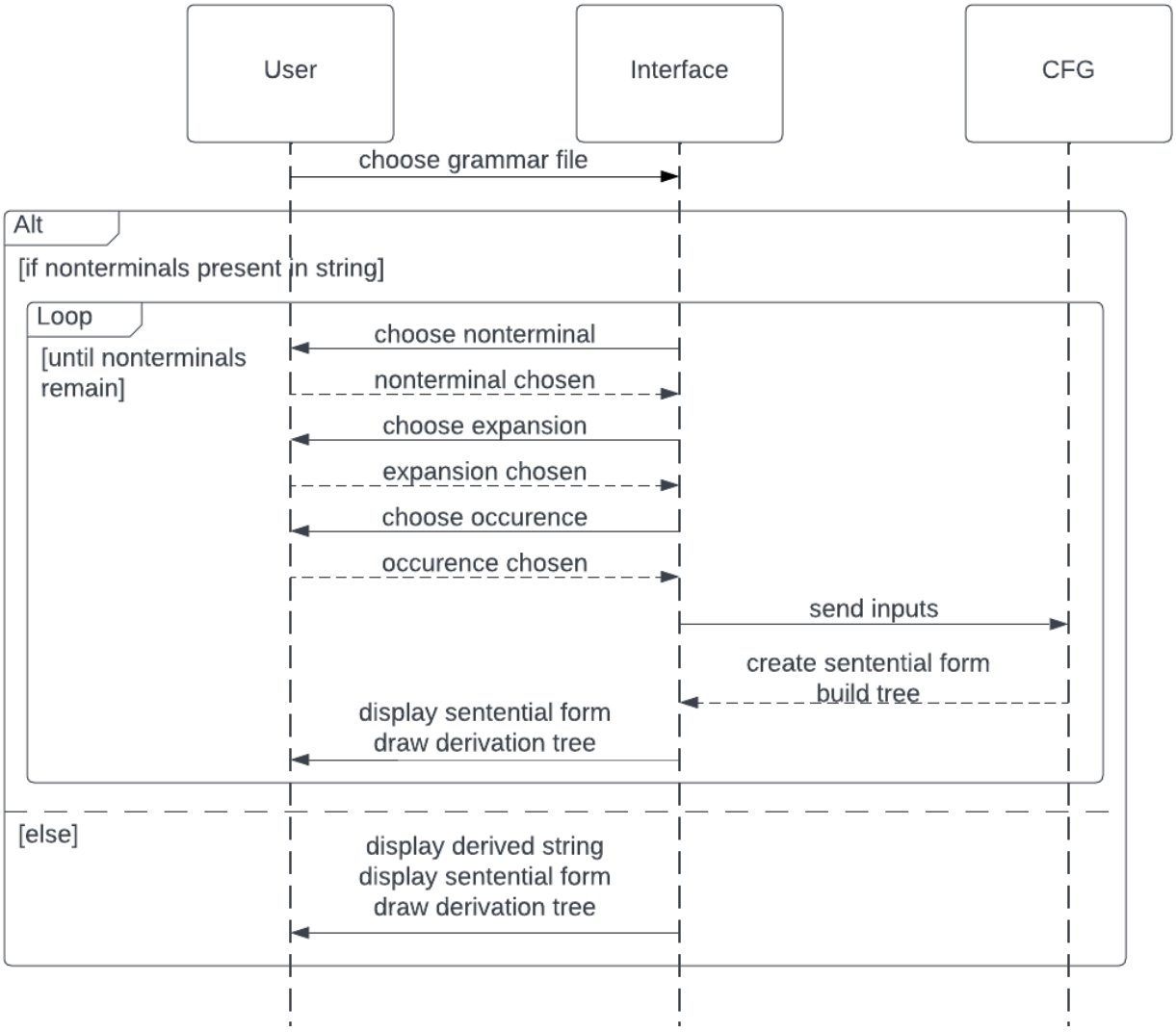


Figure 4.3: Sequence Diagram for Constructing Derivations

window provides users with a clear and accessible view of the transformation steps and their explanations.

For the detection of LR grammars, the dedicated window displays ACTION and GOTO tables. Users can interact with these tables by clicking on individual items to view their LR items, facilitating a deeper understanding.

For other transformation and analysis operations, the window includes navigation buttons that allow users to traverse through the recorded transformation steps. This user-friendly interface design ensures that users can easily access and comprehend the transformation process.

4.5.2 Integration with Architecture

These functionalities seamlessly integrate with the overall architecture of the application. Leveraging the abstraction provided by the *CFG* class, the transformation and analysis functionalities access grammar data and perform operations effectively. The recorded transformation steps enhance transparency and usability, providing users with valuable insights into the manipulation and analysis of CFGs.

In Chapter 5, we will delve deeper into the implementation details of these functions, providing a comprehensive understanding of their workings and capabilities.

Chapter 5

Implementation Details

5.1 How to Launch the Application

5.1.1 Installation

To use the application, you need to have Python installed on your system. The recommended version is Python 3.9 or above. If you don't have Python installed, you can download it from the official Python website: https://www.python.org/downloads/

The application does not require any additional packages or dependencies beyond the standard Python library.

5.1.2 Running the application

Once Python is installed on your system, you can run the application using the command line interface.

On Windows, open a command prompt and navigate to the directory where the application files are located. Then, use the following command:

```
python app.py
```

On Linux, open a terminal and navigate to the directory where the application files are located. Then, use the following command:

```
python3 app.py
```

After executing the appropriate command, the application will launch, and you can start using it to edit, transform and analyze grammar files.

You can find example grammar files in the `"grammar files"` folder within the `CFGAnalyzer.zip` file. These example files can be used to explore the features and functionality of the application.

5.2 Programming Language and Framework

The application is primarily developed using *Python 3.9* [12], harnessing its versatility and extensive libraries for GUI development. The GUI aspects are constructed using the *Tkinter* library [13], a robust and commonly used toolkit for creating graphical interfaces in Python applications.

For working with regular expressions, such as searching and replacing nonterminals with their expansions in a string, the application utilizes the *re* module [14] in Python. Additionally, the *configparser* module [15] is employed to read from and write to grammar files.

To provide a portable way of utilizing operating system dependent functionality, the application leverages the *os* module [16]. Moreover, the *itertools* module [17] is used for finding combinations, enhancing the application's functionality.

5.3 Grammar Representation

5.3.1 Terminal and Nonterminal Storage

Python lists are employed to store terminals, nonterminals. These structures allow access and manipulation of grammar components.

5.3.2 Rule Representation

Rules are stored as key-value pairs within dictionaries, associating nonterminals with their corresponding production rules. This data structure enables easy retrieval and modification of rules during grammar manipulation.

The keys of the dictionary is the nonterminals and the values are their rules stored in nested lists. For example,

The production rules,

$$S \rightarrow aBA \mid bB$$

is represented as,

$$dict\{S : [[a, B, A], [b, B]]\}$$

5.4 Reduction of Grammars

The reduction process begins by retrieving the grammar specifications and configurations. The grammar specifications are stored in a Grammar object, instantiated from the *CFG* class, while the configurations are stored in a Config object created using the *configparser* module.

5.4.1 Phase 1: Identification of Nonterminals Generating Terminal Words

An empty set, *set_t*, is initialized. This set, along with the grammar and config objects, is passed as arguments to the *reduce_phase*1() method in the Transform class. In this phase, the method iterates through all production rules to identify nonterminals capable of generating terminal words. If a nonterminal is found to generate terminal words, it is added to *set_t*. Additionally, if the right-hand side of a production rule comprises a combination of terminals and nonterminals present in *set_t*, or, consists solely of nonterminals in *set_t*, the corresponding left-hand side (nonterminal) is also added to *set_t*. This process continues recursively until no more nonterminals can be added to *set_t*.

5.4.2 Phase 2: Identification of Reachable Nonterminals

Before proceeding to Phase 2, another empty set, *set_d*, is initialized. Along with other arguments, *set_d* is passed to the *reduce_phase*2() method. The initial nonterminal is added to *set_d*, signifying that the grammar's derivation must start from this nonterminal. In Phase 2, the application identifies nonterminals reachable from the initial nonterminal. Similar to Phase 1, the program iterates through all production rules to find nonterminals reachable from the initial nonterminal and adds them to *set_d*. This recursive process continues until no further changes occur in *set_d*.

5.4.3 Result

After completing both phases, nonterminals not present in *set_d* are removed, along with the production rules where they appear. The resulting grammar is now in reduced form.

5.5 Removing Epsilon Productions

The *remove_epsilon_rules*() function is responsible for identifying nonterminals that can generate epsilon. An empty set *set_e* is initiated and passed as argument with other arguments.

5.5.1 Parameters

- **file:** The grammar file being processed.
- **config:** Instance of *configparser* containing grammar specifications.
- *set_e*: A set containing nonterminals capable of generating epsilon.

5.5.2 Algorithm for Identifying Nonterminals Generating Epsilon

1. Initialize *not_set_e* as the set difference between the set of all terminals and nonterminals and the set of nonterminals generating epsilon (*set_e*).

2. Create a copy of *set_e* called *set_temp*.

3. Iterate through each nonterminal *nt* in the grammar rules.

4. For each production rule associated with *nt*, check if it generates epsilon.

    - If it generates epsilon, add *nt* to *set_e*.
    - If it does not generate epsilon, check if it contains nonterminals not in *set_e*. If it does not contain such nonterminals, add *nt* to *set_e*.

5. If *set_temp* has changed, recursively call *remove_epsilon_rules*() with updated arguments.

5.5.3 Algorithm for Generating New Production Rules without Epsilon

After identifying nonterminals that can generate epsilon, the next step is to generate new production rules without epsilon occurrences.

1. Initialize an empty dictionary *new_rules* to store new production rules.

2. Iterate through each nonterminal in the grammar rules.

3. For each production rule associated with the current nonterminal:

    - Identify the indices of nonterminals present in *set_e* within the rule.
    - Using *combinations*() method from *itertools*, for each combination of these indices, create a new production rule by removing nonterminals at these positions.
    - Store the new production rules in the *new_rules* dictionary.

4. Update the configuration object *config* with the new production rules for each nonterminal.

5.5.4 Result

The grammar rules are modified to exclude epsilon occurrences. The new production rules are stored in the configuration object *config*, ready for further processing or analysis.

5.6 Elimination of Unit Productions

The objective is to eliminate unit rules and update the grammar accordingly. *remove_unit_rules*() method is responsible for identifying unit rules in the grammar and removing them.

5.6.1 Algorithm for Removing Unit Productions

1. Initialize an empty dictionary $transform\_sets$ to store sets of unit rules reachable from each nonterminal.
2. Iterate through each nonterminal in the grammar rules.
3. For each nonterminal, Create an empty set set_nt to store unit rules reachable from the current nonterminal and add the nonterminal to set_nt.
4. Call the $remove\_unit\_rules()$ method:
   - Iterate through each nonterminal in the set_nt set
   - For each nonterminal, iterate through its production rules.
   - If a production rule is also a nonterminal in the grammar, add it to the set_nt set.
   - If the set_nt set has been modified, recursively call the $remove\_unit\_rules$ method with the updated parameters.
5. Store the updated set_nt in the $transform\_sets$ dictionary.
6. After processing all nonterminals, the $transform\_sets$ dictionary contains sets of nonterminals reachable from each nonterminal.
7. Iterate through each key-value pair in the $transform\_sets$ dictionary.
8. For each key-value pair, Initialize an empty list new_rules to store updated production rules.
9. For each nonterminal in the set of reachable nonterminals, Iterate through each production rule of the nonterminal. If the production rule is not a nonterminal and not already in the new_rules list, add it to new_rules.
10. Update the grammar configuration with the updated production rules for the current nonterminal.

5.6.2 Result

After processing all nonterminals, the grammar configuration is updated with the new rules and the unit rules are successfully removed from the grammar.

5.7 Computation of FIRST Sets

The $compute\_first()$ method is responsible for computing the FIRST sets for all nonterminals in the given grammar. The algorithm iterates through each nonterminal and its production rules to determine the FIRST set for each nonterminal.

5.7.1 Algorithm for Computing FIRST Sets

1. Initialize an empty dictionary $first\_dict$ to store the computed FIRST sets.
2. Iterate through each nonterminal in the grammar and Initialize an empty set for the FIRST set of the current nonterminal.
3. Perform the following steps iteratively until no further updates are made:
    - Iterate through each nonterminal and its production rules.
    - Retrieve the first item in the rule.
    - If the first item is a terminal, add it to the FIRST set of the current nonterminal and mark the computation as updated.
    - If the FIRST set of the first item contains epsilon, explore further items in the rule until a terminal or non-epsilon nonterminal is encountered.
    - If it is the last symbol in the production rule and it also can generate epsilon and epsilon is not already in the FIRST set of the current nonterminal, add epsilon to the FIRST set and mark the computation as updated.
    - Return the computed FIRST sets ($first\_dict$)

5.8 Computation of LR(0) Items

Computation of LR(0) items involve computing closure for initial items in a state and dynamically creates classes and objects for each state. The *compute_lr0_items* method in *LRParser* class is responsible for performing this computation.

5.8.1 Algorithm for Computing LR(0) Items

1. Initialize an empty dictionary *states_dict* to store states of the parsing automaton.
2. Dynamically create a class with the attributes of two dictionaries: *items* and *transitions* for storing the items and transitions of the automaton.
3. Add an object of the created class to the *states_dict* with the current state number as the key.
4. Retrieve the initial production rule of the grammar using the grammar's initial nonterminal.
5. Copy the initial production rule and explicitly insert a dot (.) at the beginning of rule list to denote the current position of the parsing pointer.

6. Add the modified initial production rule to the items attribute of the object representing the current state, using the grammar's initial nonterminal as the key.

7. Compute the closure of the items in items attribute of the object representing the current state.

8. Compute goto of the items in the state and add the transition symbol as the key and arriving state number as the value to the transitions attribute of the object representing the current state.

9. Iterate until the current position of the parsing pointer has reached the end of items in all possible states and no new data is added in the *states_dict*.

5.8.2 Result

After computing all the LR(0) items, the *states_dict* dictionary has all the states of the automaton as its keys and the objects representing each state as its values respectively. Each object in the *states_dict* contains *items* dictionary storing the left-hand side of the items as the keys and right-hand sides stored in nested lists as the values. Outgoing transitions from the state are stored in the *transitions* dictionary.

Chapter 6

Conclusion

In this work, I have developed a desktop application, a comprehensive tool designed for the manipulation and analysis of context-free grammars. Throughout this project, I have delved into various facets of the application, from its theoretical underpinnings to its implementation and functionality. The objective was to develop a versatile and user-friendly application that empowers users to explore, understand, and manipulate context-free grammars effectively.

In the initial chapters, I laid the foundation by discussing the theoretical background of context-free grammars (CFGs) and their significance in various domains like compiler construction and theoretical computer science. I elucidated the fundamental concepts such as terminals, nonterminals, production rules, derivation trees and performing operations on context-free grammars providing the necessary theoretical groundwork for the subsequent chapters.

Moving forward, I transitioned into the design and architecture of the application, outlining the structural components, architectural patterns, and design principles that underpin the application. Leveraging the Model-View-ViewModel (MVVM) architecture, the application achieves a modular and extensible design, facilitating seamless integration of new features and functionalities.

One of the key aspect of the application lies in its intuitive user interface, which provides a seamless and engaging experience for users. Built using the Tkinter library, the GUI of the application offers a range of interactive tools and visualizations, enabling users to manipulate grammars, analyze derivation trees, and perform grammar transformations with ease.

Furthermore, the application incorporates parsing algorithms and grammar analysis tools, including LL-parser and LR-parser functionalities. These algorithms, implemented using Python's versatile libraries, empower users to analyze the properties of context-free grammars, detect grammar types, and construct parsing tables.

Looking ahead, there are several avenues for future enhancements and improvements to the application. These include optimizing algorithms and integrating additional grammar analysis tools such as SLR, LALR and CLR parsing.

In conclusion, the application stands as a versatile and user-friendly platform catering to students and enthusiasts alike, offering a comprehensive toolkit for exploring and analyzing grammar manipulation. Through its intuitive interface and robust functionality, the application empowers users to delve into the intricate world of context-free grammars with ease and confidence. Whether for educational purposes or personal exploration, the application serves as a valuable resource for individuals seeking to deepen their understanding of grammar manipulation and analysis by providing an accessible and indispensable tool for learners and practitioners alike.

Bibliography

1. SIPSER, Michael. *Introduction to the Theory of Computation*. Cengage Learning, 2012-06.
2. GEEKSFORGEEKS. *Simplifying context free grammars*. 2022-12. Available also from: `https://www.geeksforgeeks.org/simplifying-context-free-grammars/`.
3. HOPCROFT, J.E. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 2008. Always Learning. ISBN 9788131720479. Available also from: `https://books.google.cz/books?id=tzttuN4gsVgC`.
4. GEEKSFORGEEKS. *Converting context free grammar to Greibach normal form*. 2022-12. Available also from: `https://www.geeksforgeeks.org/converting-context-free-grammar-greibach-normal-form/?ref=lbp`.
5. KNUTH, Donald Ervin. On the Translation of Languages from Left to Right. *Inf. Control.* 1965, vol. 8, pp. 607–639. Available also from: `https://api.semanticscholar.org/CorpusID:14648496`.
6. ROSENKRANTZ, Daniel J.; STEARNS, Richard Edwin. Properties of deterministic top down grammars. *Proceedings of the first annual ACM symposium on Theory of computing*. 1969. Available also from: `https://api.semanticscholar.org/CorpusID:30515666`.
7. GRUNE, Dick; JACOBS, Ceriel J.H. *Parsing techniques: A Practical Guide*. Springer Science and Business Media, 2007-10.
8. KNUTH, Donald Ervin. Top-down syntax analysis. *Acta Informatica*. 1971, vol. 1, pp. 79–110. Available also from: `https://api.semanticscholar.org/CorpusID:206773993`.
9. AHO, Alfred V.; JOHNSON, Stephen C. LR Parsing. *ACM Computing Surveys (CSUR)*. 1974, vol. 6, pp. 99–124. Available also from: `https://api.semanticscholar.org/CorpusID:3254307`.
10. DEREMER, Frank; PENNELLO, Thomas J. Efficient Computation of LALR(1) Look-Ahead Sets. *ACM Trans. Program. Lang. Syst.* 1982, vol. 4, pp. 615–649. Available also from: `https://api.semanticscholar.org/CorpusID:52833742`.
11. PARR, Terence. *ANTLR (ANother Tool for Language Recognition)*. [N.d.]. Available also from: `https://www.antlr.org/`.

12. *Python 3.12.1 documentation*. [N.d.]. Available also from: https://docs.python.org/3/.

13. *tkinter — Python interface to Tcl/Tk*. [N.d.]. Available also from: https://docs.python.org/3/library/tkinter.html.

14. *re — Regular expression operations*. [N.d.]. Available also from: https://docs.python.org/3/library/re.html.

15. *configparser — Configuration file parser*. [N.d.]. Available also from: https://docs.python.org/3/library/configparser.html.

16. *os — Miscellaneous operating system interfaces*. [N.d.]. Available also from: https://docs.python.org/3/library/os.html.

17. *itertools — Functions creating iterators for efficient looping*. [N.d.]. Available also from: https://docs.python.org/3/library/itertools.html.