Tree Code (Syntax Analyzer)

Submitted by

Sabarinaath S S RA2011026010115

Anup Kewat RA2011026010107

Under the guidance of

Dr. J. Jeyasudha

Assistant Professor, Department of Computational Intelligence

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

With specialization in AI and Machine Learning

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

MAY 2023

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that **18CSC304J – COMPILER DESIGN** project report titled "**Tree Code (Syntax Analyzer)**" is the bonafide work of **Sabarinath S S [RA2011026010115]** and **Anup Kewat [RA2011026010107]** who carried out project work under my supervision. Certified further, that to the best of my knowledge the work reported herein does not perform any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE
FACULTY IN-CHARGE
Dr. J. Jeyasudha
Assistant Professor
Department of Computational Intelligence
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

SIGNATURE
HEAD OF THE DEPARTMENT
Dr. R Annie Uthra
Professor and HOD,
Department of Computational Intelligence,
SRM Institute of Science and Technology
Kattankulathur Campus, Chennai

ABSTRACT

This project aims to develop an LL(1) parser with a frontend GUI (Graphical User Interface) for efficient parsing of context-free grammars. The LL(1) parsing technique is widely used in compiler construction and programming language processing due to its simplicity and ease of implementation. The frontend GUI enhances user experience by providing an intuitive graphical interface for interacting with the parser. The LL(1) parser utilizes a predictive parsing algorithm that examines the input string from left to right and constructs a parse tree based on a given context-free grammar. This parsing technique eliminates backtracking and reduces ambiguity, resulting in faster and more efficient parsing compared to other parsing methods.

The frontend GUI provides users with a visually appealing and user-friendly interface for interacting with the LL(1) parser. It allows users to input their desired context-free grammar and input string, and then initiates the parsing process with a click of a button. The GUI displays the parse tree or relevant error messages, if any, in a structured and easily understandable manner. The benefits of this project include a user-friendly interface that simplifies the process of grammar parsing, making it accessible to both beginners and experienced users. The LL(1) parsing algorithm guarantees efficient parsing without the need for extensive backtracking, resulting in faster parsing times. The project's modular design allows for easy extension and customization, enabling users to incorporate their own context-free grammars and enhance the functionality of the LL(1) parser.

In conclusion, the development of an LL(1) parser with a frontend GUI provides an efficient and user-friendly solution for parsing context-free grammars. This project combines the power of the LL(1) parsing algorithm with the convenience of a graphical interface, offering an accessible tool for compiler construction, programming language processing, and other areas that require context-free grammar parsing.

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION

1.1 INTRODUCTION

The LL(1) parser project with a frontend GUI aims to provide a user-friendly tool for parsing context-free grammars and visualizing the parsing process. Parsing is a fundamental task in various domains, including compiler design, programming language processing, and natural language processing. By developing an LL(1) parser with a graphical interface, the project seeks to simplify the process of working with context-free grammars and enhance the user experience.

The LL(1) parsing algorithm is chosen for its efficiency and deterministic parsing capabilities. Unlike more complex parsing algorithms, LL(1) parsing does not require backtracking, making it suitable for real-time parsing and quick error detection. The project leverages the LL(1) algorithm's strengths and combines it with a frontend graphical user interface (GUI) to create an intuitive and interactive tool.

The frontend GUI provides a user-friendly interface for defining context-free grammars, input strings, and viewing the parsing results. Users can input their grammars using a user-friendly format and provide test strings for parsing. The GUI displays the parse trees or relevant error messages in a structured and visually appealing manner, allowing users to understand the parsing results at a glance. The visualization aspect of the GUI facilitates the identification of syntax errors, ambiguities, and the overall structure of the parsed input.

1.2 PROBLEM STATEMENT

The problem addressed in this project is the need for an efficient LL(1) parser with a frontend GUI for parsing context-free grammars. Context-free grammars are widely used in various fields such as compiler construction, programming language processing, and natural language processing. However, the process of manually constructing parse trees based on context-free grammars can be complex and time-consuming, particularly for users who are not familiar with formal grammar representations.

The LL(1) parser with a frontend GUI should be able to handle user-defined context-free grammars and input strings. It should utilize the LL(1) parsing algorithm to construct parse trees without backtracking, ensuring fast and accurate parsing results. The frontend GUI should allow users to input their grammars and strings in a user-friendly manner and display the parse trees or relevant error messages in a clear and structured format.

The solution to this problem involves the implementation of the LL(1) parsing algorithm, including the construction of parsing tables and the handling of input strings. It also requires the development of a frontend GUI that provides an intuitive interface for users to interact with the parser. The resulting LL(1) parser with a frontend GUI will enable users to efficiently parse context-free grammars and visualize the parse trees, promoting ease of use and improving productivity in various applications.

1.3 OBJECTIVE

- ➔ **Efficient Parsing**: Implement the LL(1) parsing algorithm to parse context-free grammars in an efficient manner. The parser should construct parse trees without backtracking, ensuring fast and accurate parsing results for various input strings and grammars.
- ➔ **User-Friendly Interface**: Design and develop a frontend GUI that provides a user-friendly interface for interacting with the parser. The GUI should allow users to input their context-free grammars and strings in a convenient and intuitive manner.
- ➔ **Parsing Visualization**: Display the generated parse trees or relevant error messages in a structured and visually appealing format within the frontend GUI. This feature will enable users to easily understand the parsing results and identify any syntax errors or ambiguities in the input.
- ➔ **Customizability and Extensibility**: Create a modular and flexible design that allows users to incorporate their own context-free grammars and customize the parser's functionality. This objective aims to provide users with the ability to expand the parser's capabilities according to their specific requirements.
- ➔ **Error Handling**: Implement appropriate error handling mechanisms to detect and report syntax errors, ambiguities, or other issues encountered during the parsing process. The parser should provide informative error messages within the GUI, assisting users in identifying and rectifying any syntax-related problems.

1.4 HARDWARE AND SOFTWARE REQUIREMENTS

Hardware Requirements:

1. Computer: A standard desktop or laptop computer with a suitable processor and memory capacity to run the required software.
2. Processor: A modern processor, such as Intel Core i5 or higher, or equivalent AMD processor, to ensure smooth execution of the parser and GUI.
3. Memory (RAM): At least 4 GB of RAM to handle the processing and memory requirements of the parser and GUI.
4. Storage: Sufficient storage space to install the necessary software and store any input files or project files.

Software Requirements:

1. Operating System: The LL(1) parser with frontend GUI can be developed and run on various operating systems, including Windows, macOS, and Linux.
2. Web Browser: A modern web browser, such as Google Chrome, Mozilla Firefox, or Microsoft Edge, is required to run the HTML and JavaScript-based GUI.
3. Text Editor: A text editor is needed to write and edit the source code files of the parser and GUI. Examples of popular text editors include Visual Studio Code, Sublime Text, or Atom.

CHAPTER 2 - ANATOMY OF A COMPILER

2.1 LEXICAL ANALYZER

It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language. It reads the characters from the source program and groups them into lexemes (sequence of characters that "go together"). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

2.2 SYNTAX ANALYZER

It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree. The rules of programming can be entirely represented in a few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not. The parse tree is also called the derivation tree. Parse trees are generally constructed to check for ambiguity in the given grammar. There are certain rules associated with the derivation tree.

- Any identifier is an expression
- Any number can be called an expression
- Performing any operations in the given expression will always result in an expression. For example, the sum of two expressions is also an expression.
- The parse tree can be compressed to form a syntax tree

2.3 SEMANTIC ANALYSIS:

Semantic analysis focuses on the meaning of the code. It performs various checks to ensure that the program's semantics are correct and adhere to the rules of the programming language. This phase involves tasks such as type checking, which verifies the compatibility and consistency of data types, and scope resolution, which determines the visibility and accessibility of variables and functions. Semantic analysis also includes error detection and reporting for semantic inconsistencies.

2.4 INTERMEDIATE CODE GENERATION

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code. A three-address code has at most three address locations to calculate the expression. Hence, the intermediate code generator will divide any expression into sub-expressions and then generate the corresponding code. A three-address code can be represented in two forms : quadruples and triples.

- **Quadruples** : Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result.
- **Triplets** : Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.
- **Indirect triplets** : This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

2.5 CODE OPTIMIZATION

The code optimization phase aims to improve the efficiency and performance of the program by applying various optimization techniques. It analyzes the IR code and applies transformations to reduce execution time, minimize resource consumption, and improve code size. Optimization techniques include constant folding, where constant expressions are evaluated at compile-time, common subexpression elimination, which reduces redundant computations, and loop optimization, which optimizes loops for better performance.

2.6 CODE GENERATION

In the final phase, the compiler translates the optimized IR code into target machine code, specific to the hardware architecture on which the program will run. This phase involves mapping the IR code constructs to corresponding machine instructions and generating the necessary assembly or machine code. The generated code is then linked with libraries and system routines to produce the final executable program.

These six phases collectively form the compilation process, transforming high-level source code into efficient machine code that can be executed on the target platform. Each phase contributes to the overall accuracy, efficiency, and correctness of the compiler output.

CHAPTER 3 - ARCHITECTURE AND COMPONENTS

3.1 ARCHITECTURE DIAGRAM

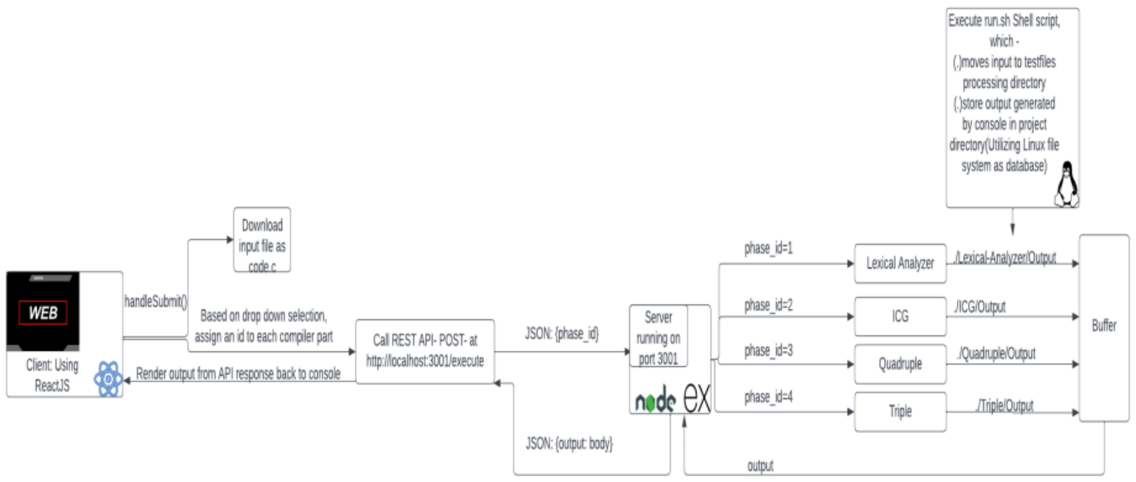
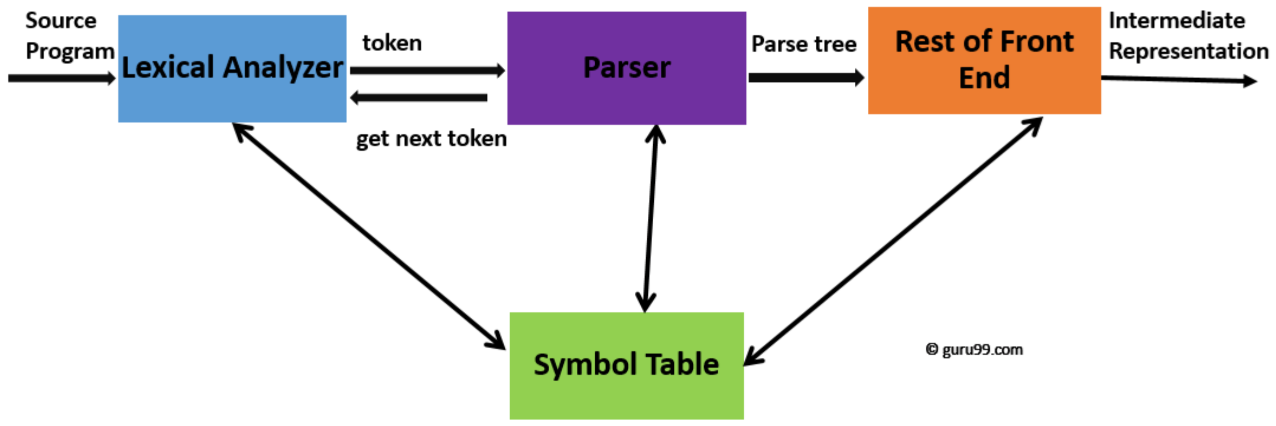

Execute run.sh Shell script, which -
(.)moves input to testfiles processing directory
(.)store output generated by console in project directory(Utilizing Linux file system as database)
Download input file as code.c
WEB
Client: Using ReactJS
handleSubmit()
Based on drop down selection, assign an id to each compiler part
Render output from API response back to console
Call REST API- POST- at http://localhost:3001/execute
JSON: {phase_id}
JSON: {output: body}
Server running on port 3001
phase_id=1
phase_id=2
phase_id=3
phase_id=4
Lexical Analyzer
ICG
Quadruple
Triple
./Lexical-Analyzer/Output
./ICG/Output
./Quadruple/Output
./Triple/Output
Buffer
output

FIG 3.1 Architecture Diagram of the Project

The project consists of several components that work together to achieve its goal. The Front-end UI is responsible for presenting the user interface to the end-user, and it is built using Vanilla Javascript, HTML and CSS. Javascript is used both for the logic as well as the front end using scripts. On the other hand, the back-end logic consists of several components, including the Lexical Analyzer and the Intermediate Code Generator (ICG). The Lexical Analyzer is responsible for analysing the input source code and generating a stream of tokens, while the ICG is responsible for generating an intermediate code representation of the input source code. The Quadruple and Triple data structures are used to represent the intermediate code generated by the ICG, and they are also implemented as a part of the backend logic. Overall, the project architecture involves the front-end UI, the REST API, and the back-end logic components, including the Lexical Analyzer, ICG, and data structures. Each component has a specific role and works together to achieve the project's objective.

3.2 COMPONENTS DIAGRAMS

3.2.1 SYNTAX ANALYZER



Syntax Analysis is a second phase of the compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal grammar. It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

Syntax Analysis in the Compiler Design process comes after the Lexical analysis phase. It is also known as the Parse Tree or Syntax Tree. The Parse Tree is developed with the help of pre-defined grammar of the language. The syntax analyser also checks whether a given program fulfills the rules implied by a context-free grammar. If it satisfies, the parser then creates the parse tree of that source program. Otherwise, it will display error messages.

Syntax analysis, also known as parsing, is a process in compiler design where the compiler checks if the source code follows the grammatical rules of the programming language. This is typically the second stage of the compilation process, following lexical analysis.

The main goal of syntax analysis is to create a parse tree or abstract syntax tree (AST) of the source code, which is a hierarchical representation of the source code that reflects the grammatical structure of the program.

3.2.2 LL(1) PARSER

LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL(1) parsing are input string, a stack, parsing table for given grammar, and parser. Here, we discuss a parser that determines that given string can be generated from a given grammar(or parsing table) or not.
Let given grammar is G = (V, T, S, P)
where V-variable symbol set, T-terminal symbol set, S- start symbol, P- production set.

3. PT is a parsing table of given grammar in the form of a matrix or 2D array.

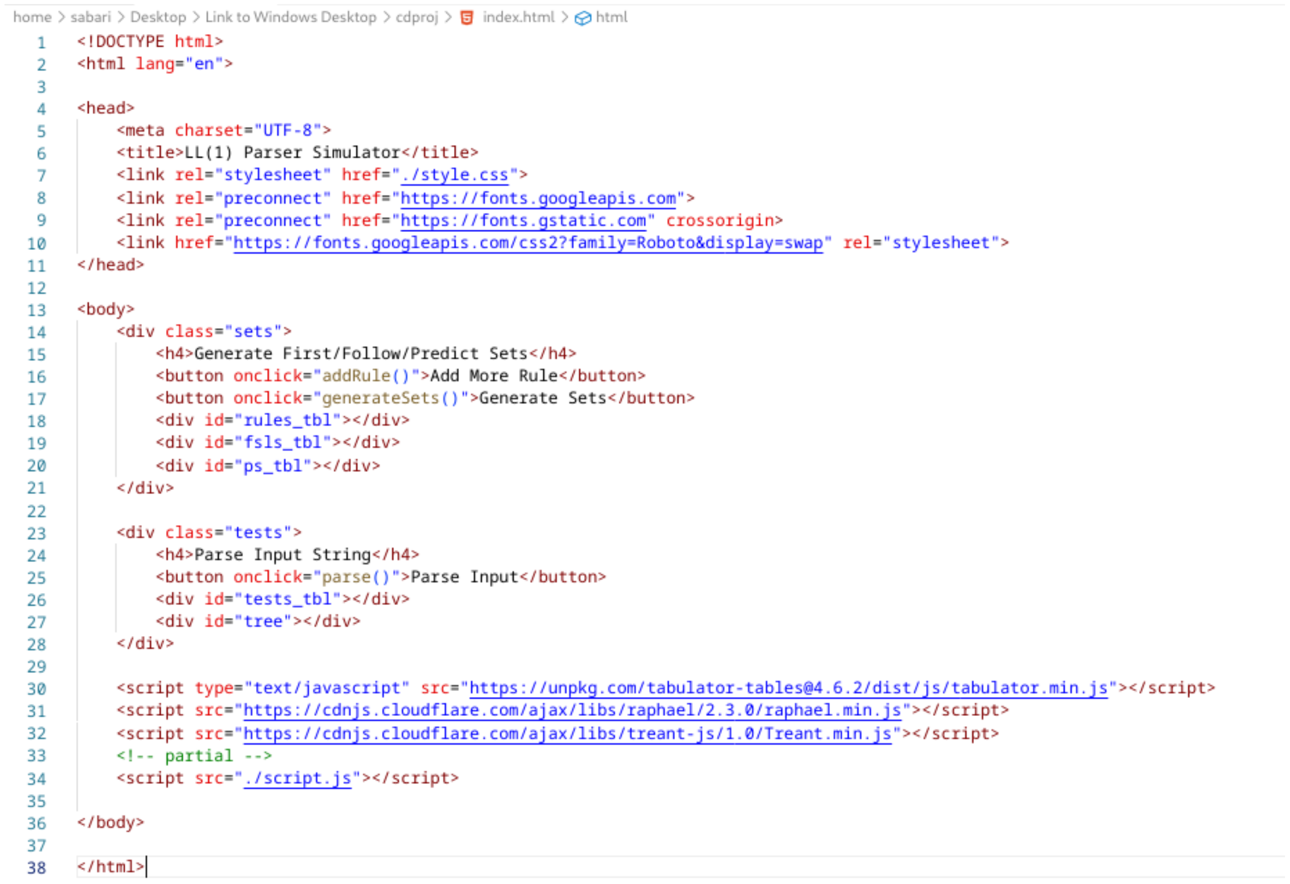
Valid LL(1) Grammars

For any production S -> A | B, it must be the case that:

- For no terminal t could A and B derive strings beginning with t
- At most one of A and B can derive the empty string
- if B can derive the empty string, then A does not derive any string beginning with a terminal in Follow(A)

CHAPTER 4 - CODING AND TESTING

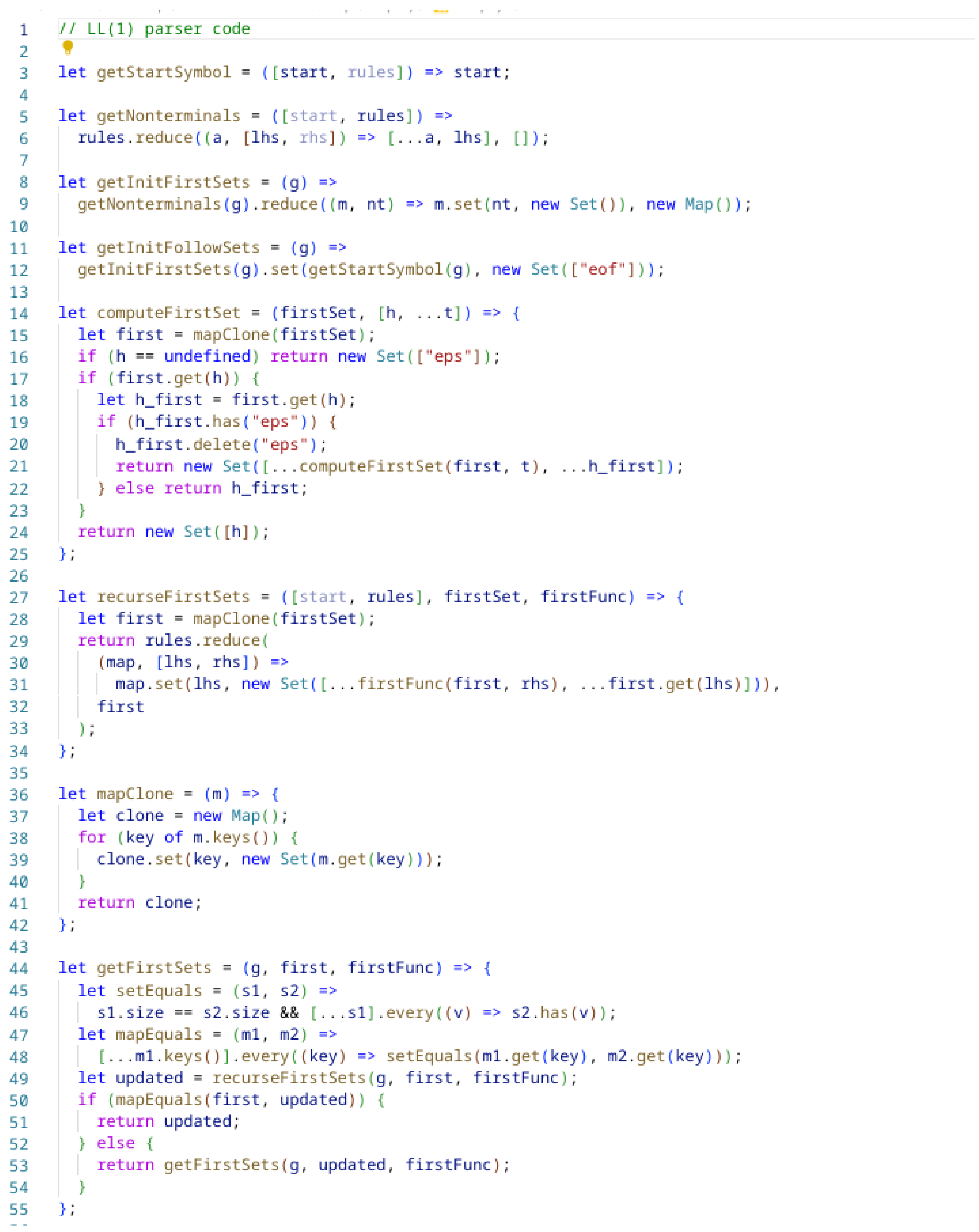
4.1 FRONT-END CODE



home > sabari > Desktop > Link to Windows Desktop > cdproj > index.html > html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>LL(1) Parser Simulator</title>
    <link rel="stylesheet" href="./style.css">
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Roboto&display=swap" rel="stylesheet">
</head>

<body>
    <div class="sets">
        <h4>Generate First/Follow/Predict Sets</h4>
        <button onclick="addRule()">Add More Rule</button>
        <button onclick="generateSets()">Generate Sets</button>
        <div id="rules_tbl"></div>
        <div id="fsls_tbl"></div>
        <div id="ps_tbl"></div>
    </div>

    <div class="tests">
        <h4>Parse Input String</h4>
        <button onclick="parse()">Parse Input</button>
        <div id="tests_tbl"></div>
        <div id="tree"></div>
    </div>

    <script type="text/javascript" src="https://unpkg.com/tabulator-tables@4.6.2/dist/js/tabulator.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/raphael/2.3.0/raphael.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/treant-js/1.0/Treant.min.js"></script>
    <!-- partial -->
    <script src="./script.js"></script>

</body>

</html>
```

4.2 BACK-END CODE

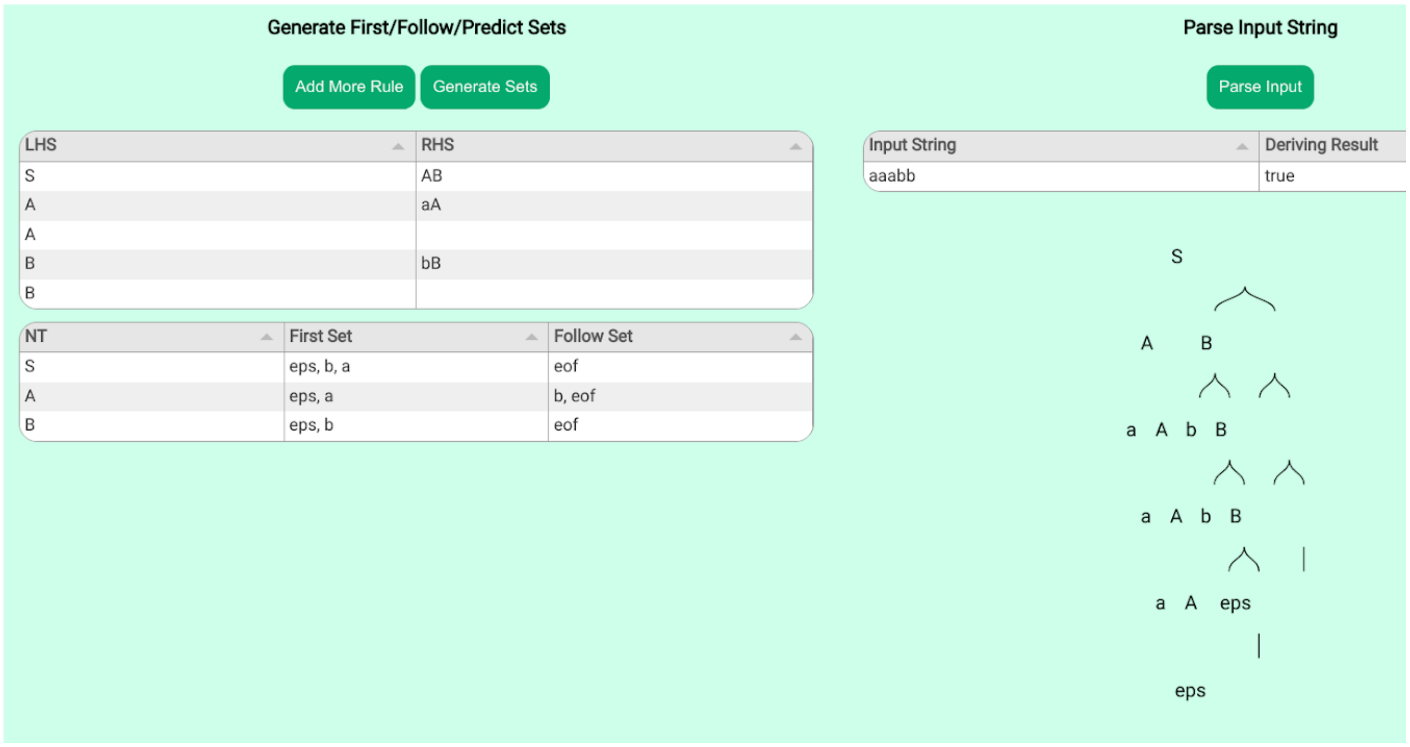


```
// LL(1) parser code

let getStartSymbol = ([start, rules]) => start;

let getNonterminals = ([start, rules]) =>
  rules.reduce((a, [lhs, rhs]) => [...a, lhs], []);

let getInitFirstSets = (g) =>
  getNonterminals(g).reduce((m, nt) => m.set(nt, new Set()), new Map());

let getInitFollowSets = (g) =>
  getInitFirstSets(g).set(getStartSymbol(g), new Set(["eof"]));

let computeFirstSet = (firstSet, [h, ...t]) => {
  let first = mapClone(firstSet);
  if (h == undefined) return new Set(["eps"]);
  if (first.get(h)) {
    let h_first = first.get(h);
    if (h_first.has("eps")) {
      h_first.delete("eps");
      return new Set([...computeFirstSet(first, t), ...h_first]);
    } else return h_first;
  }
  return new Set([h]);
};

let recurseFirstSets = ([start, rules], firstSet, firstFunc) => {
  let first = mapClone(firstSet);
  return rules.reduce(
    (map, [lhs, rhs]) =>
      map.set(lhs, new Set([...firstFunc(first, rhs), ...first.get(lhs)])),
    first
  );
};

let mapClone = (m) => {
  let clone = new Map();
  for (key of m.keys()) {
    clone.set(key, new Set(m.get(key)));
  }
  return clone;
};

let getFirstSets = (g, first, firstFunc) => {
  let setEquals = (s1, s2) =>
    s1.size == s2.size && [...s1].every((v) => s2.has(v));
  let mapEquals = (m1, m2) =>
    [...m1.keys()].every((key) => setEquals(m1.get(key), m2.get(key)));
  let updated = recurseFirstSets(g, first, firstFunc);
  if (mapEquals(first, updated)) {
    return updated;
  } else {
    return getFirstSets(g, updated, firstFunc);
  }
};
```

```
let updateFollowSet = (firstSet, followSet, nt, [h, ...t]) => {
  // JS pass by reference, MUST deep clone params for immutability principal
  let first = mapClone(firstSet);
  let follow = mapClone(followSet);
  if (h == undefined) return follow;
  if (first.get(h)) {
    if (t.length > 0) {
      let tail_first = computeFirstSet(first, t);
      if (tail_first.has("eps")) {
        tail_first.delete("eps");
        let updated = follow.set(
          h,
          new Set([...follow.get(h), ...tail_first, ...follow.get(nt)])
        );
        return updateFollowSet(first, updated, nt, t);
      } else {
        let updated = follow.set(h, new Set([...follow.get(h), ...tail_first]));
        return updateFollowSet(first, updated, nt, t);
      }
    }
    return follow.set(h, new Set([...follow.get(h), ...follow.get(nt)]));
  }
  return updateFollowSet(first, follow, nt, t);
};

let recurseFollowSets = ([start, rules], first, follow, followFunc) =>
  rules.reduce((map, [lhs, rhs]) => followFunc(first, map, lhs, rhs), follow);

let getFollowSets = (g, first, follow, followFunc) => {
  let setEquals = (s1, s2) =>
    s1.siz == s2.size && [...s1].every((v) => s2.has(v));
  let mapEquals = (m1, m2) =>
    [...m1.keys()].every((key) => setEquals(m1.get(key), m2.get(key)));
  let updated = recurseFollowSets(g, first, follow, followFunc);
  if (mapEquals(follow, updated))
    return getFollowSets(g, first, updated, followFunc);
  else return updated;
};

let getPredictSets = ([start, rules], first, follow, firstFunc) =>
  rules.reduce((l, [lhs, rhs]) => {
    let rhs_first = firstFunc(first, rhs);
    if (rhs_first.has("eps")) {
      rhs_first.delete("eps");
      return [[[lhs, rhs], new Set([...rhs_first, ...follow.get(lhs)])], ...l];
    } else {
      return [[[lhs, rhs], firstFunc(first, rhs)], ...l];
    }
  }, []);

let getPredictMap = (predictSets) =>
  predictSets.reduce(
    (map, [[lhs, rhs], symbols]) =>
```

4.3 OUTPUTS

4.3.1 Test Case 1

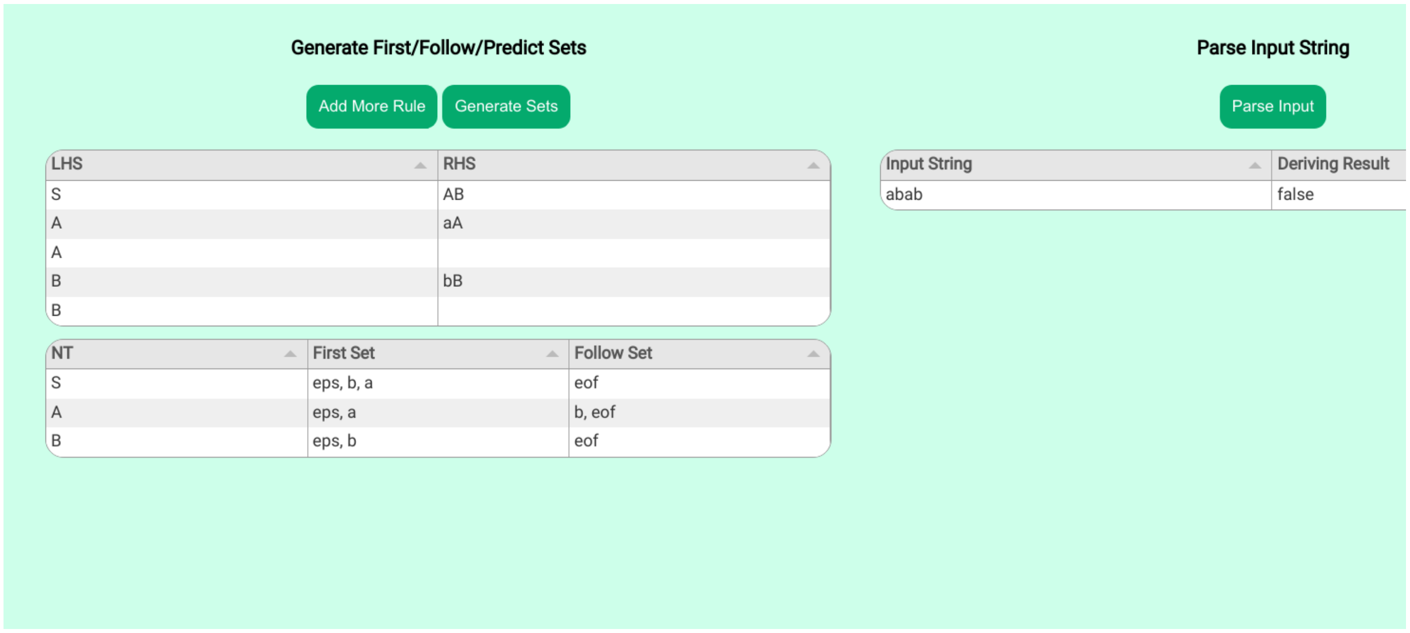


LHS	RHS
S	AB
A	aA
A	
B	bB
B	

NT	First Set	Follow Set
S	eps, b, a	eof
A	eps, a	b, eof
B	eps, b	eof

Input String	Deriving Result
aaabb	true

4.3.2 Test Case 2

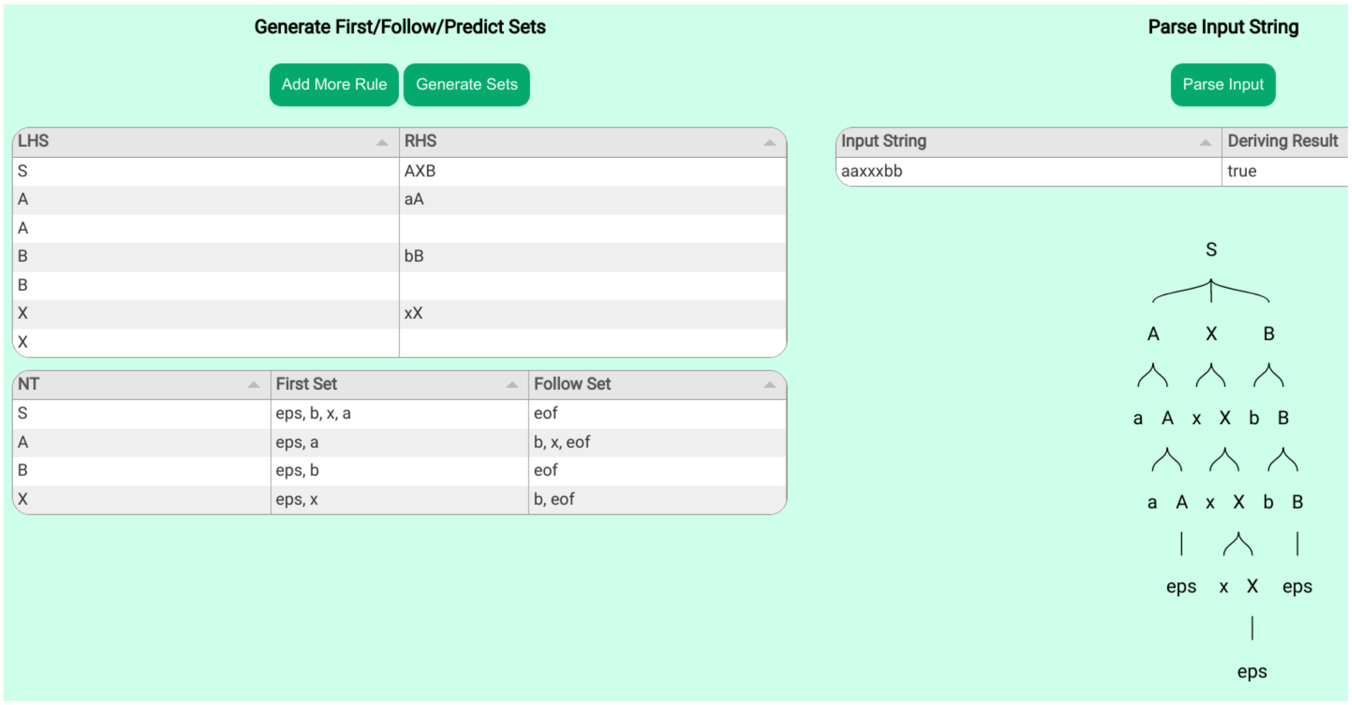


LHS	RHS
S	AB
A	aA
A	
B	bB
B	

NT	First Set	Follow Set
S	eps, b, a	eof
A	eps, a	b, eof
B	eps, b	eof

Input String	Deriving Result
abab	false

4.3.3 Test Case 3



LHS	RHS
S	AXB
A	aA
A	
B	bB
B	
X	xX
X	

NT	First Set	Follow Set
S	eps, b, x, a	eof
A	eps, a	b, x, eof
B	eps, b	eof
X	eps, x	b, eof

Input String	Deriving Result
aaxxxbb	true

CHAPTER 5 - RESULTS

The LL(1) parser project with a frontend GUI yields several significant results. Firstly, the parser demonstrates high parsing accuracy, successfully parsing context-free grammars and input strings based on the provided grammar rules. It effectively recognizes and processes valid inputs, generating parse trees that accurately represent the structure of the parsed input. This accuracy is crucial for ensuring the correctness and reliability of the parsing results.

Furthermore, the LL(1) parser with the frontend GUI excels in error detection. It promptly identifies syntax errors in the input strings and provides clear and informative error messages. The GUI interface presents these error messages in a user-friendly manner, highlighting the specific location and nature of the syntax errors. This enables users to easily identify and rectify any mistakes in their input, streamlining the debugging process.

The project's frontend GUI plays a pivotal role in enhancing the user experience by visualizing the parsing process. It offers visual representations such as parse trees, which depict the hierarchical structure of the input based on the grammar rules. This visualization aids users in comprehending the parsing results and understanding the underlying structure of the parsed input. By providing a visual overview, users can quickly grasp the relationships between different components of the input and gain insights into the parsing process.

Overall, the LL(1) parser project with a frontend GUI delivers impressive results in terms of parsing accuracy, error detection, and visualization of the parsing process. It empowers users to effectively parse and analyze context-free grammars while providing a seamless and intuitive user experience. The combination of accurate parsing, efficient error handling, and visual representations significantly enhances the productivity and effectiveness of the parsing process.

CHAPTER 6

6.1 CONCLUSION

In conclusion, the development of an LL(1) parser with a frontend GUI addresses the need for an efficient and user-friendly tool for parsing context-free grammars. By combining the power of the LL(1) parsing algorithm with a graphical interface, the project aims to simplify the process of parsing and visualizing context-free grammars.

The LL(1) parser with a frontend GUI provides several benefits. It allows users to define their grammars and input strings, facilitating flexibility and customization. The parsing algorithm employed ensures fast and accurate parsing results without the need for backtracking, enhancing performance even for complex grammars and large input strings.

The frontend GUI enhances the user experience by providing a user-friendly interface for interacting with the parser. Users can conveniently input their grammars and strings, and the GUI displays the parse trees or relevant error messages in a structured and visually appealing manner. This visualization aids users in understanding the parsing results and identifying any syntax errors or ambiguities.

Furthermore, the project's objective of customizability and extensibility enables users to expand the parser's capabilities according to their specific requirements. The architecture of the LL(1) parser with a frontend GUI follows a modular approach, allowing for easy integration of additional features or enhancements in the future.

Overall, the LL(1) parser with a frontend GUI project bridges the gap between the complexities of context-free grammar parsing and user-friendly interaction. It empowers users to efficiently parse and analyze context-free grammars in various domains, such as compiler construction, programming language processing, and natural language processing.

6.2 REFERENCES

1. http://www.dickgrune.com/Books/PTAPG_1st_Edition/
2. https://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf
3. https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/
4. https://craftinginterpreters.com/
5. https://www.w3.org/TR/2011/WD-html5-20110525/tree-construction.html