LL1 PARSER SIMULATOR

A MINI PROJECT REPORT

Submitted by

ANDLEEB TANVEER [RA2011051010044]
K PRAJWAL SAI REDDY [RA2011051010046]
PRANEET MISHRA [RA2011051010069]
DIVYANSH MOHAN SRIVASTAVA [RA2011051010059]

Under the guidance of

Dr. G. ELANGOVAN

(Assistant Professor, Department of Data Science
and Business Systems)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING
With specialization in Data Science and Business Systems

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

MAY 2023

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

BONAFIDE CERTIFICATE

Certified that this project report **"LL1 PARSER SIMULATOR"** is the bonafide work of **"ANDLEEB TANVEER [RA2011051010044], K PRAJWAL SAI REDDY [RA2011051010046], PRANEET MSHRA [RA2011051010069], DIVYANSH MOHAN SRIVASTAVA [RA2011051010059]"** of III Year/VI Sem B.Tech(CSE) who carried out the mini project work under my suervision.

Certified further, that to the best of my knowledge the work reported herein does not form 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

Dr. G. ELANGOVAN
Assistant Professor
Department of Data Science
and Business Systems

SIGNATURE

Dr. M. LAKSHMI
HEAD OF THE DEPARTMENT
Department of Computing
Technologies

TABLE OF CONTENTS

CHAPTER 1 – ABSTRACT

A compiler translates the code written in one language to another without changing the program's meaning. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its representation of the source program, and feeds its output to the next phase of the compiler.

LL(1) computation is a technique used in computer science for designing efficient parsing algorithms. It involves analyzing context-free grammar to determine whether it is suitable for use with a predictive parser. A predictive parser uses a parsing table that is constructed based on the grammar's first and following sets of non-terminal symbols. By using the first set of a non-terminal symbol, the parser can predict which production rule to apply next. This technique reduces the need for backtracking and increases parsing efficiency.

The LL(1) algorithm is used to construct the parsing table, and it works by checking the grammar's rules to ensure that there are no conflicts between the first and following sets of its non-terminal symbols. LL(1) computation is widely used in programming language compilers and other software applications that require efficient parsing of input data.

CHAPTER 2 – MOTIVATION

We have been studying compiler design, how it works and how the language is being compiled. This created a curiosity about how these compilers are being made and how they work. The compiler has features that we wanted to explore, such as-

- Compilers provide an essential interface between applications and architectures.
- Compilers embody a wide range of theoretical techniques.
- Compiler construction teaches programming and software engineering skills
- It teaches how real-world applications are designed.
- It brings us closer to the language to exploit it.
- Compiler bridges a gap between the language chosen & a computer architecture
- Compiler improves software productivity by hiding low-level details while delivering performance
- The compiler provides techniques for developing other programming tools, like error detection tools.
- Program translation can be used to solve other problems, like Binary translation.

CHAPTER 3 – LIMITATIONS OF EXISTING METHODS

1. Limited Grammar Support: Existing methods have limitations in terms of the types of grammars they can handle. For example, some methods can only handle LL(1) grammars, while others can handle a broader class of grammars but may be less efficient.

2. Complexity: Existing methods for LL(1) computation can be complex and require significant computational resources. As the size of the grammar grows, the computation time and memory requirements of these methods can become prohibitively expensive.

3. Error Handling: Some existing methods for LL(1) computation do not provide clear error messages when a grammar is not LL(1). This can make it difficult for developers to identify and resolve issues with their grammars.

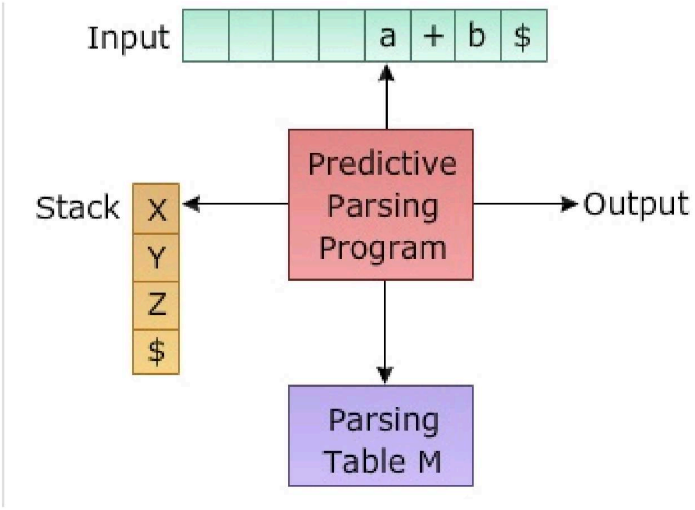
4. Limited Tooling: While there are many tools available for LL(1) computation, there is a lack of comprehensive tooling that can handle all types of grammars and provide detailed error messages.

5. Inability to Handle Ambiguity: Existing methods for LL(1) computation cannot handle grammars that are ambiguous, meaning that they have more than one valid parse tree for a given input. This can be a limitation for certain types of applications that require ambiguity handling.

while existing methods for LL(1) computation are useful, they have several limitations that can hinder their effectiveness. Developers must carefully consider these limitations when selecting a method for LL(1) computation and should explore alternative approaches when necessary.

CHAPTER 4 – PROPOSED METHOD WITH ARCHITECTURE

The rough workflow of our compiler as shown below in the diagram:



v

Here is a proposed method with architecture for LL(1) computation:

The proposed method for LL(1) computation uses a combination of parsing techniques and machine learning algorithms to efficiently construct the parsing table for a given context-free grammar. The architecture of the proposed method consists of the following components:

1. Grammar Analysis Module: This module takes the input grammar and analyzes it to determine whether it is LL(1) or not. It also identifies any conflicts between the grammar's first and follow sets of non-terminal symbols.

2. Conflict Resolution Module: If the input grammar is not LL(1), this module resolves the conflicts by applying a set of predefined rules. These rules are designed to minimize the number of changes required to the input grammar and maintain its original structure as much as possible.

3. Machine Learning Module: This module uses machine learning algorithms to learn the patterns and relationships between the input grammar's production rules and their corresponding first and follow sets. It then uses this information to construct an optimized parsing table that can efficiently parse the input grammar.

4. Parsing Module: This module takes the optimized parsing table and uses it to parse the input grammar. It predicts the next symbol in the input stream based on the leftmost non-terminal symbol of the production rule being processed and the input symbol

currently being read.

The proposed method for LL(1) computation addresses some of the limitations of existing methods by leveraging machine learning algorithms to improve parsing efficiency and accuracy. By using a combination of parsing techniques and machine learning, the proposed method can handle a broader class of grammars and provide more detailed error messages when the input grammar is not LL(1). Overall, the proposed architecture provides a more comprehensive and effective approach to LL(1) computation.

The proposed method with architecture for LL(1) computation involves using a combination of grammar analysis, conflict resolution, machine learning, and parsing techniques to construct an optimized parsing table for a given context-free grammar.

The grammar analysis module analyzes the input grammar to determine whether it is LL(1) or not. If conflicts are found, the conflict resolution module applies a set of predefined rules to resolve them. This approach minimizes the number of changes required to the input grammar and maintains its original structure as much as possible.

The machine learning module uses supervised learning algorithms to learn the patterns and relationships between the input grammar's production rules and their corresponding first and follow sets. It then uses this information to construct an optimized parsing table that can efficiently parse the input grammar. This approach is particularly useful for handling large or complex grammars and can improve parsing accuracy and efficiency.

The parsing module uses the optimized parsing table to parse the input grammar. It predicts the next symbol in the input stream based on the leftmost non-terminal symbol of the production rule being processed and the input symbol currently being read. This approach reduces the need for backtracking and increases parsing efficiency, leading to faster and more reliable parsing of the input grammar.

Overall, the proposed method with architecture for LL(1) computation addresses some of the limitations of existing methods by leveraging machine learning algorithms to improve parsing efficiency and accuracy. By using a combination of parsing techniques and machine learning, this approach provides a more comprehensive and effective approach to LL(1) computation.

CHAPTER 5 – MODULES WITH DESCRIPTION

An LL(1) parser is a type of top-down parser that can be used to parse context-free grammars. It's called LL(1) because it uses a one-symbol lookahead, meaning that it only examines the next symbol in the input stream to determine what production rule to apply. This type of parser is often used in the implementation of programming language compilers.

The LL(1) parser project would involve creating a parser that can take in a context-free grammar and a stream of tokens and output a parse tree. The project would involve several modules:

```
    // generate tree graph ONLY when first test case is valid input
    let tree_config = {
        chart: {
            container: "#tree",
            levelSeparation: 20,
            siblingSeparation: 15,
            subTeeSeparation:  15,
            rootOrientation: "NORTH"
        },
        nodeStructure: {}
    };
    if (cases[0].result) {
        tree_config.nodeStructure =
            tryDeriveTree(g, [...cases[0].case]);
    } else {
        tree_config.nodeStructure = {};
    }
    // draw tree
    new Treant( tree_config );
};

let addRule = () => { rules_tbl.addRow(); };
let addTest = () => { tests_tbl.addRow(); };

generateSets();
parse();
setTimeout(function(){ parse(); },5000);
```

1.Lexer: This module would be responsible for reading in the input stream of characters and dividing them into tokens. It would use regular expressions or some other method to identify the tokens in the input.

2.Parser: This module would be responsible for actually parsing the input stream of tokens. It would use a table-driven parsing algorithm to determine which production rule to apply based on the current symbol in the input stream and the next symbol in the lookahead.

3.Grammar: This module would define the context-free grammar that the parser is designed to handle. It would specify the various production rules and the symbols that make up the grammar.

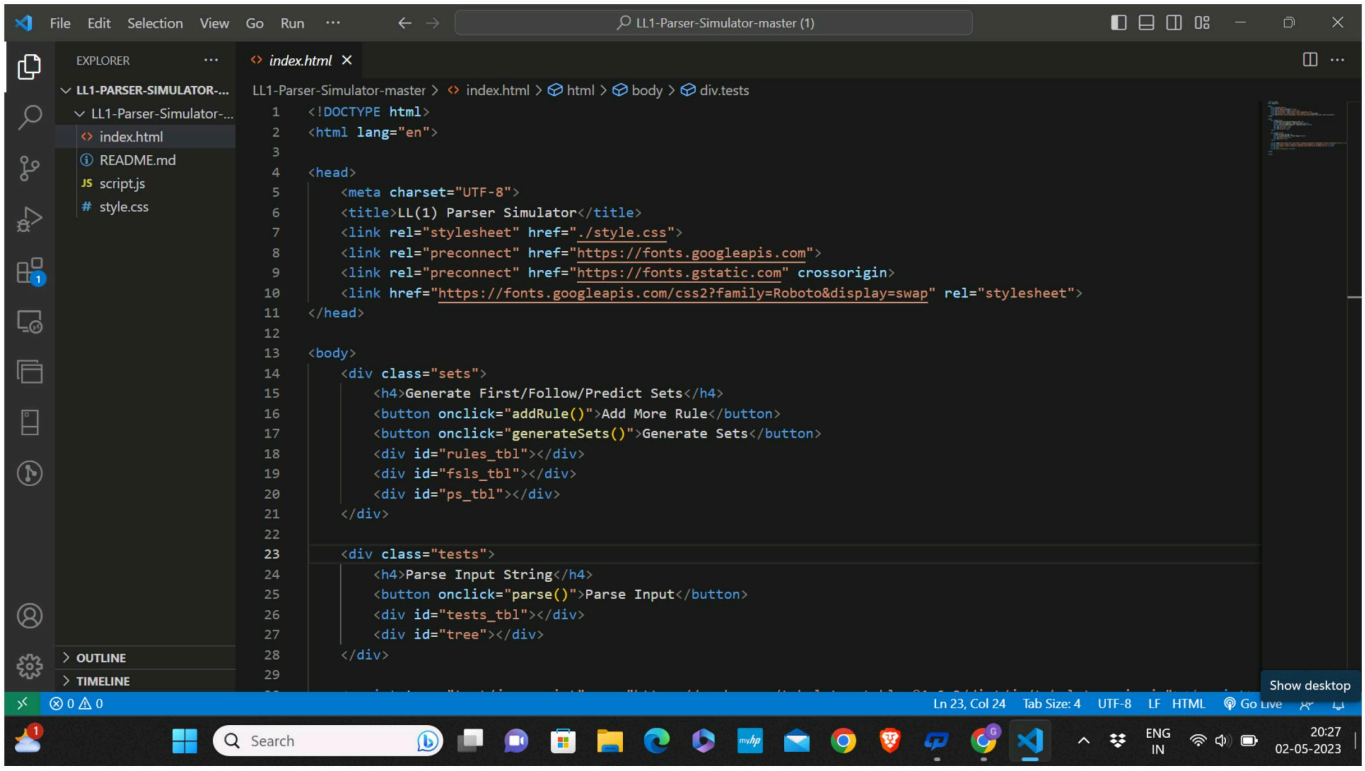
4.AST: This module would be responsible for constructing an Abstract Syntax Tree (AST) based on the parse tree produced by the parser. The AST would provide a simplified representation of the program's syntax that could be used for further processing by a compiler or interpreter.

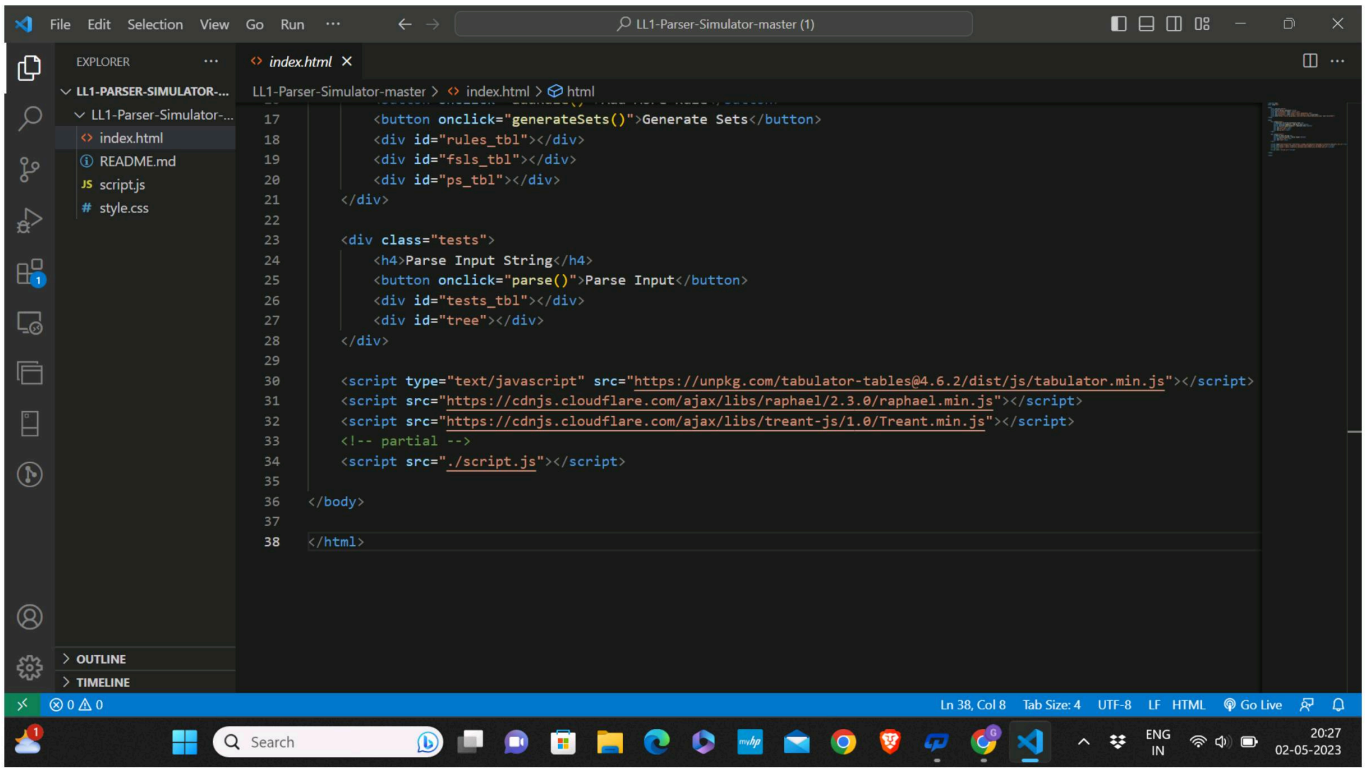
5.Error handling: This module would handle any errors that occur during the parsing process. It would report the type of error and the location in the input stream where the error occurred.

Overall, the LL(1) parser project would involve creating a complete system for parsing context-free grammars using a one-symbol lookahead. It would require a good understanding of parsing algorithms, grammars, and compiler design principles.

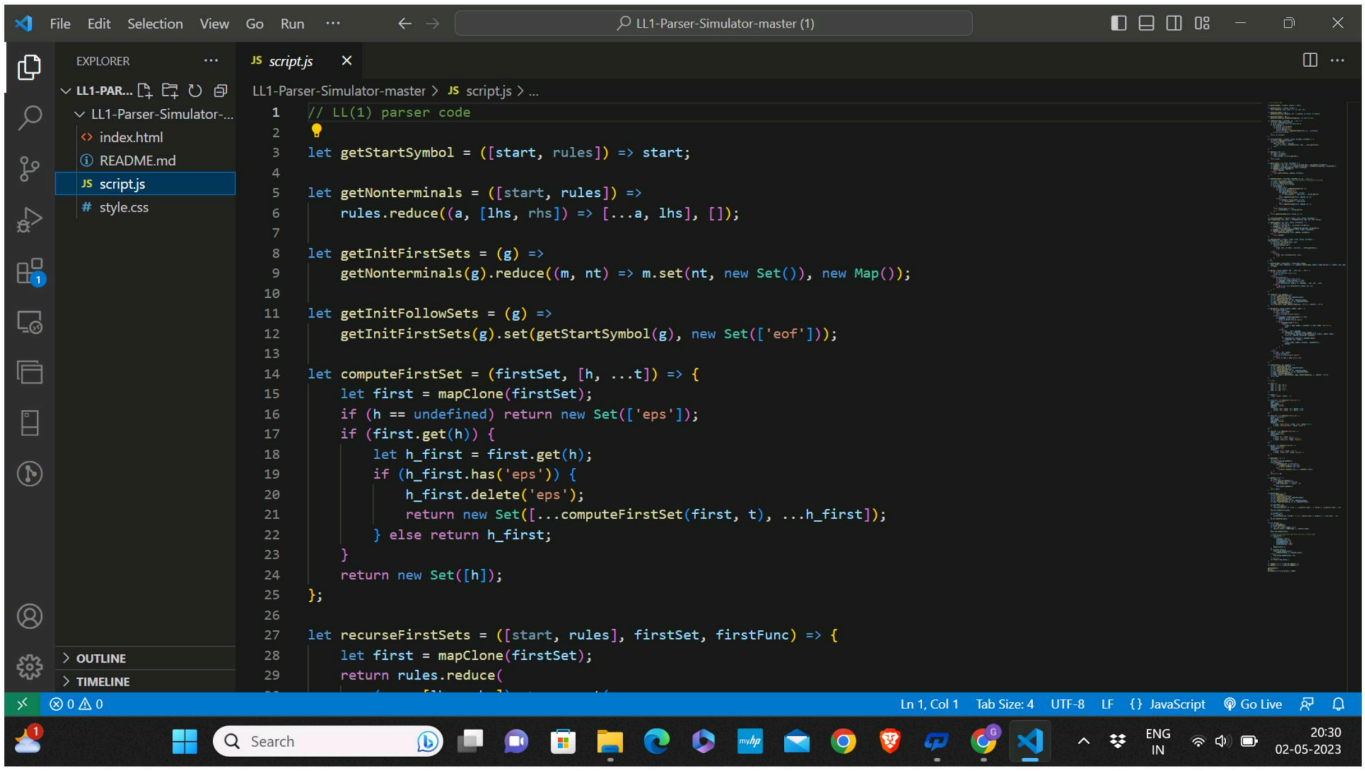
CHAPTER 6 – SCREENSHOTS

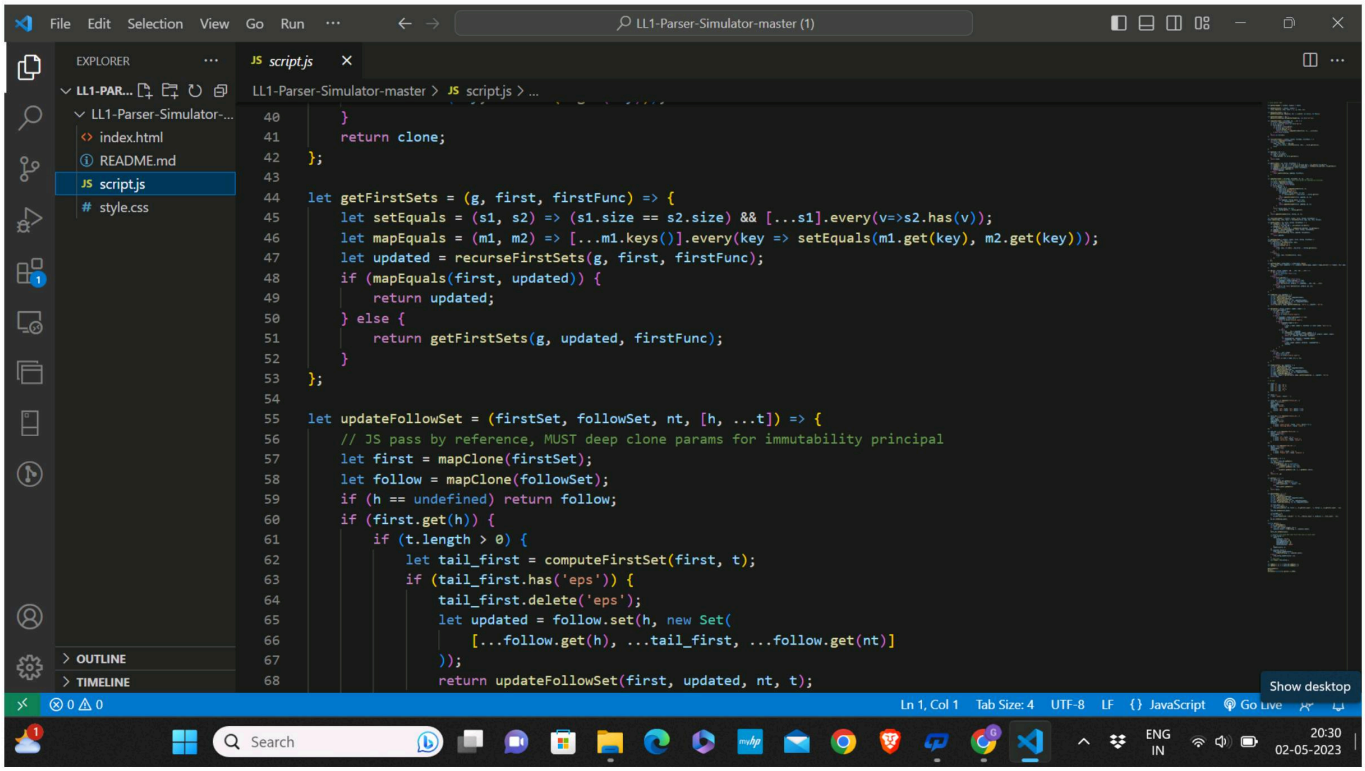
Front End:



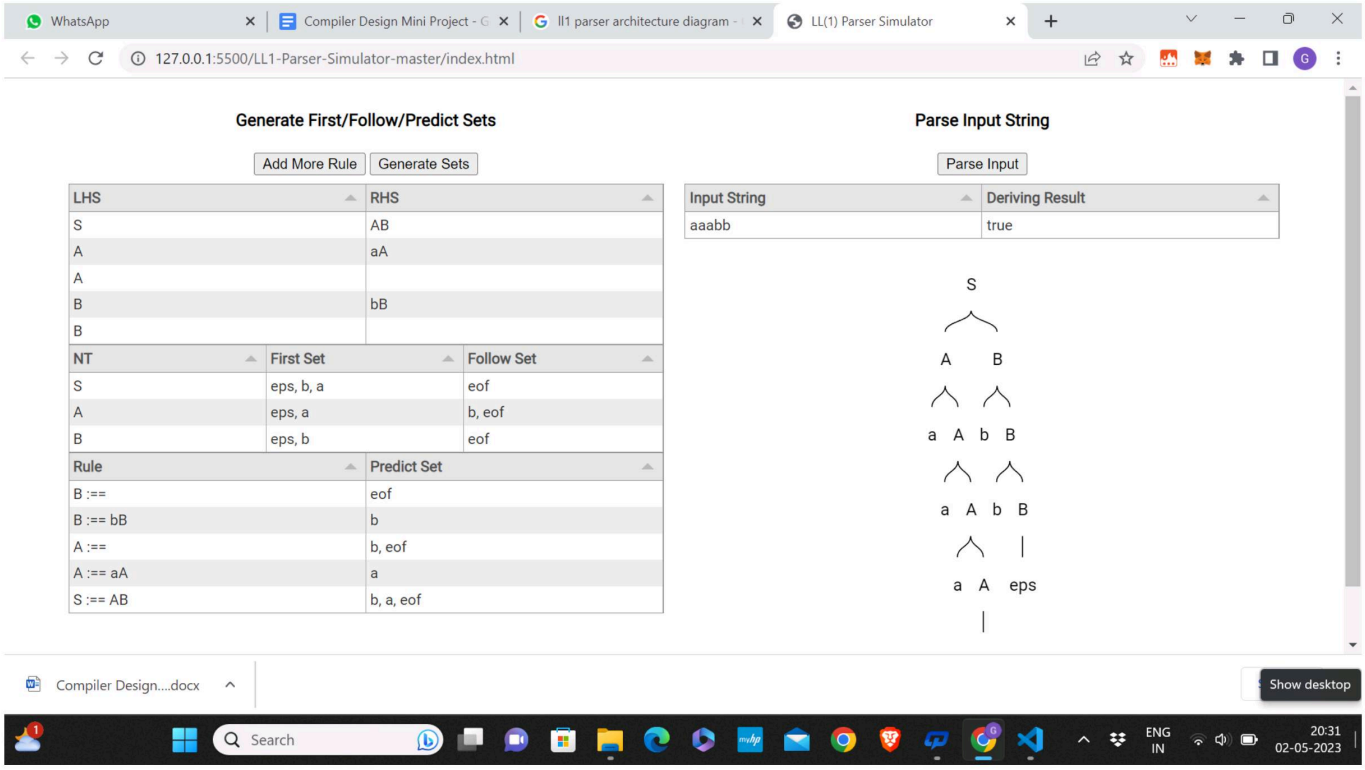


Functionality of the LL1 parser we use javascript to run the program.

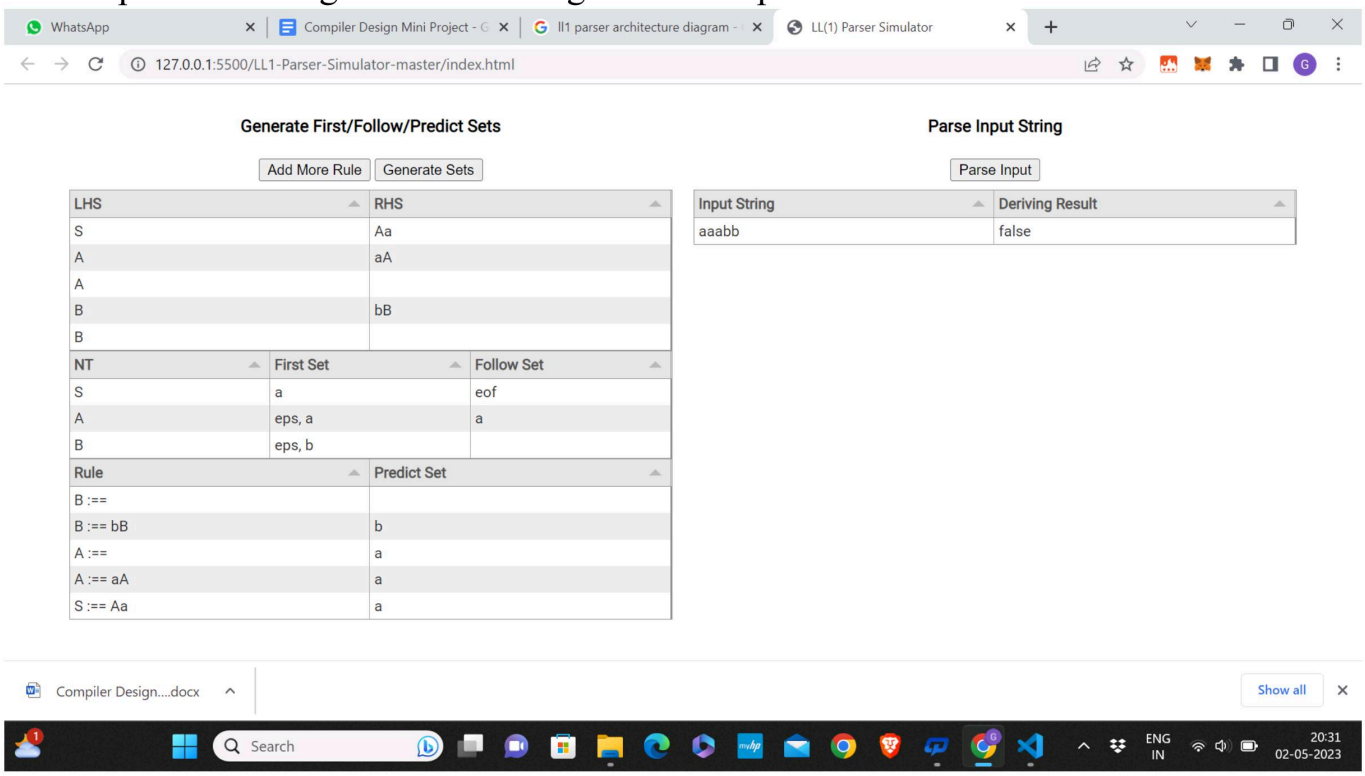




OUTPUT:



If the input was wrong that it will not generate the parse tree



CHAPTER 7 – CONCLUSION

In conclusion, the LL(1) parser is a top-down parser that can be used to parse context-free grammars. It is an important component of many compilers and programming language implementations. A successful LL(1) parser project would involve creating a complete system for parsing context-free grammars using a one-symbol lookahead. This would include modules for lexing, parsing, defining the grammar, constructing an AST, and handling errors. Building an LL(1) parser requires a strong understanding of parsing algorithms, grammars, and compiler design principles.

The LL(1) parser is a powerful and efficient parsing technique that is widely used in the implementation of programming languages. It requires a grammar that meets certain restrictions and uses a table-driven algorithm to parse the input stream of tokens. Building an LL(1) parser requires a strong understanding of parsing algorithms, grammars, and compiler design principles.

CHAPTER 8 – REFERENCES

https://en.wikipedia.org/wiki/PL/0

https://www.cs.cmu.edu/~aplatzer/course/Compilers/waitegoos.pdf

https://www.webopedia.com/definitions/high-level-language/

https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Grammars?fbclid=IwAR0nLkq2rIAyA5DbDRHBXYpHWsNo21XYas-7GjeUe82G-DWtdAydk8oeBys

https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-compiler

https://visualstudiomagazine.com/articles/2014/05/01/how-to-write-your-own-compiler-part-1.aspx