REPORT ON

CD Mini Project

Carried out on

**DEVELOPMENT AND IMPLEMENTING LL(1) PARSER FOR “A Hypothetical Language”**

*Submitted to*

**NMAM INSTITUTE OF TECHNOLOGY, NITTE**

(An Autonomous Institution under VTU, Belagavi)

*In partial fulfilment of the requirements for the award of the*

Bachelor of Engineering in

Computer Science Engineering

*by*

**NNM22CS125** Prajith J Shetty

**NNM22CS174** Shrujan S Shetty

**NNM22CS210** Yashas UP

Submitted to,

**Dr. SANNIDHAN M S**

Associate Professor Gd. III





CERTIFICATE

*This is to certify that the students* ***NNM22CS125*** *Prajith J Shetty,* ***NNM22CS174*** *Shrujan S Shetty and* ***NNM22CS210*** *Yashas UP* *of semester VI B.E., of NMAM Institute of Technology, Nitte, have completed CD mini project on "Design and Implementation of a LL(1) Parser for* ***“A Hypothetical Language”*** *using LEX/YACC Paradigms during March 2025 – April 2025 fulfilling the partial requirements for the award of degree of Bachelor of Engineering in Computer Science* ***Engineering*** *at NMAM Institute of Technology, Nitte*

*Name and Signature of Mentor*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Signature of HOD*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Acknowledgement

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of people who made it possible because “Success is the abstract of hard work and perseverance, but steadfastness of all is encouraging guidance.” So, I acknowledge all those whose guidance and encouragement served as a beacon light and crowned my efforts with success.

I want to thank my college administration for providing a conducive environment and suitable facilities for this mini project. It is my great pleasure to thank my mentor, **Dr. Sannidhan M S,** for his continuous encouragement, guidance, and support throughout this project. I would like to thank our HOD, **Dr. Jyothi Shetty** for showing me the path and providing the inspiration required for taking the project to its completion

I would like to thank our principal, **Prof. Niranjan N. Chiplunkar,** firstly, for providing us with this unique opportunity to do the mini project in the 7th semester of Computer Science and Engineering.

Finally, I would like to thank the staff members of the CSE department, my parents, and friends for their honest opinions and suggestions throughout the course of our mini-project.

*Prajith J Shetty (NNM22CS125)*

*Shrujan S Shetty (NNM22CS174)*

*Yashas UP (NNM22CS210)*

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| Title | Page No. |
| Institute Certificate | 1-2 |
| Acknowledgement | 3 |
| Table of Contents | 4 |
| Abstract | 5 |
| Overview | 6-7 |
| Workflow Diagram | 8 |
| Implementation | 9-10 |
| Result | 11-16 |
| Consclusion & References | 17 |

Abstract

This project presents the design and implementation of a compiler front-end for a hypothetical programming language that supports basic arithmetic operations through a switch-case construct. Inspired by traditional procedural languages like C, the language features variable declarations, character-based operator selection, and conditional execution of arithmetic expressions using the + and - operators.

Implemented in Python, the compiler front-end focuses on the **lexical analysis** and **syntax parsing** stages, emulating the behavior of tools such as LEX and YACC. The system performs tokenization of the source code, computes essential compiler theory constructs such as **FIRST** and **FOLLOW** sets, and builds a **predictive parsing table** for syntax analysis. Python’s object-oriented and string processing capabilities are leveraged to construct a modular and extensible system that accurately parses and validates source code written in the hypothetical language.

The project demonstrates how high-level programming languages can be effectively utilized in compiler construction, particularly in educational settings. Key outcomes include successful token generation, grammar validation, and parse tree generation for simple arithmetic switch-case programs. The modular structure allows for future enhancements, such as semantic analysis and code generation. This work not only provides practical exposure to compiler design principles but also offers a simplified environment for learning core concepts of formal language theory and syntax-directed translation

1. Overview
   1. Project Overview

This project involves designing a compiler front-end for a simple, hypothetical language using Python. It focuses on lexical analysis and syntax parsing for basic arithmetic operations via a switch-case structure. The system demonstrates token generation, grammar parsing, and error detection for educational compiler design.

* 1. Problem statement

Q: Design compiler for the following hypothetical languages

X: integer ;

Procedure foo( b : integer )

b := 13;

If x = 12 and b = 13 then

printf( “by copy-in copy-out” );

elseif x = 13 and b = 13 then

printf( “by address” );

else

printf( “A mystery” );

end if;

end foo

* 1. Syntax Analysis

Syntax analysis is all about discovering structure in code. It determines whether a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not. Syntax analysis is based on the rules based on the specific programming language by constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language.

Obtain tokens from the lexical analyser

Implemented FIRST and FOLLOW set computations for grammar rules.

Built an LL(1) parse table for predictive parsing.

Checks if the expression is syntactically correct or not

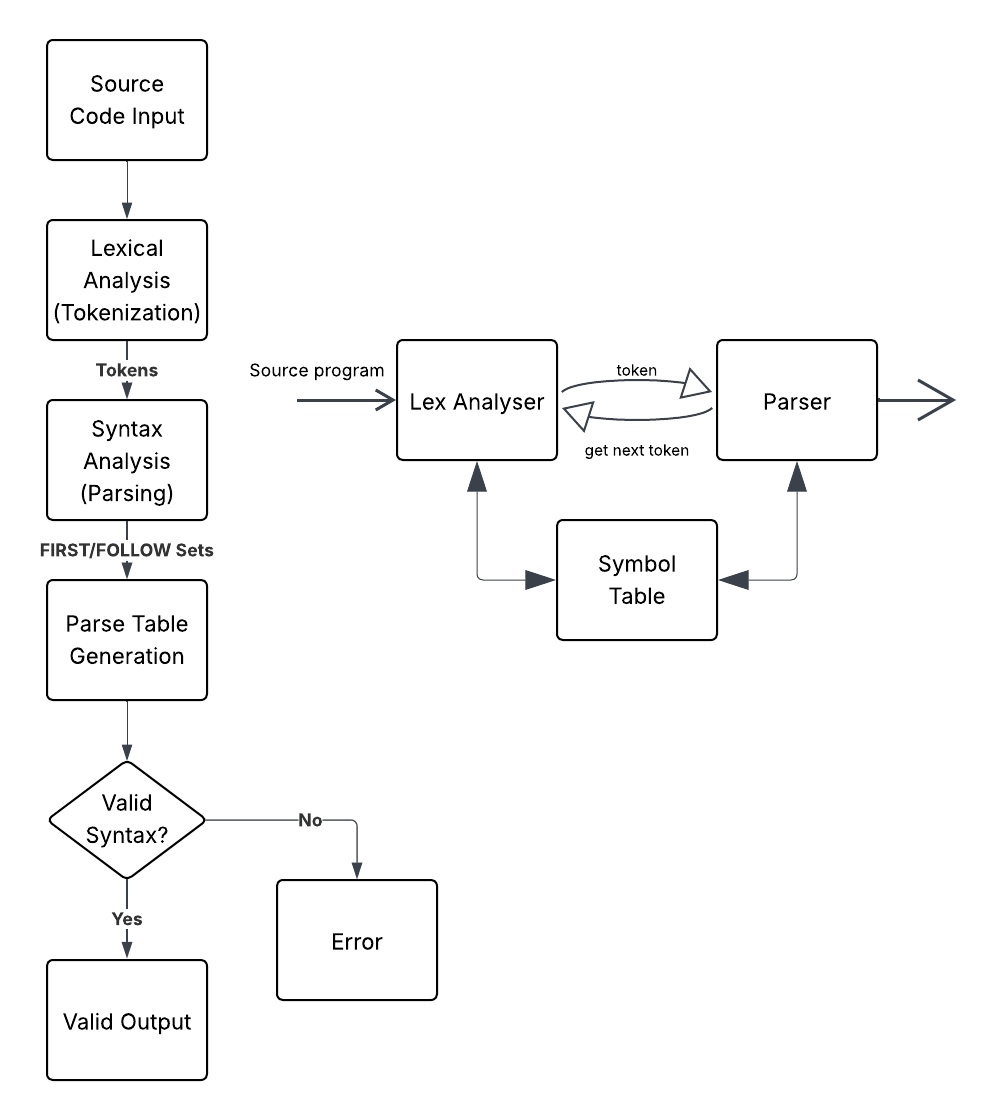
* 1. Parser

Parser: Parser is that phase of the compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer. The parser obtains a string of tokens from the lexical analyser and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

* 1. LL1 Grammar/Rules for the problem statement



2.Workflow Diagram



3.Implementation

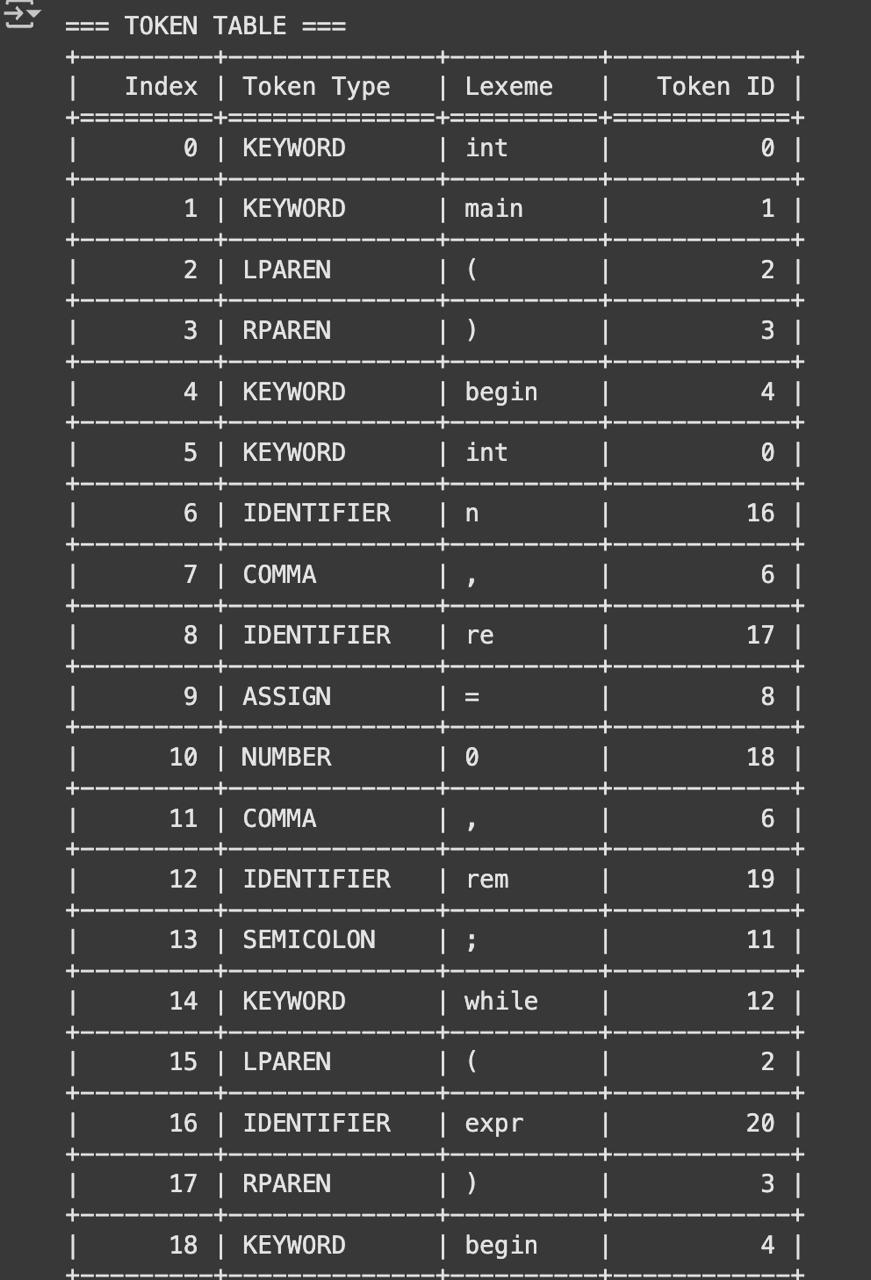
* **import ply.yacc as yacc**
* Imports the PLY parser generator under the alias yacc.
* **Used for**: Defining grammar rules and generating the parser using yacc.yacc().
* **import sys**
* Imports Python’s built-in system module.
* **Used for**: Accessing command-line arguments with sys.argv.
* **from lexer import tokens**
* Imports token definitions from a user-defined lexer.py.
* **Used for**: Supplying token names needed to match grammar patterns in parsing.
* **from tabulate import tabulate**
* Imports the tabulate function for pretty-printing tables.
* **Used for**: Displaying First Sets, Follow Sets, Parsing Table, and Parsing Actions in table format.
* **symbol\_table = {}**
* Initializes an empty dictionary.
* **Used for**: Storing declared variables and their values during execution
* **parsed\_program = None**
* Initializes a placeholder for the final parse tree.
* **Used for**: Storing the parse result if parsing is successful.
* **action\_table = []**
* Initializes an empty list.
* **Used for**: Recording each step of the parsing simulation (stack contents + description).
* **Grammar Rule Functions (p\_\*)**
* Defined with names like p\_program, p\_declarations, etc.
* **Used for**: Representing grammar rules and building the parse tree or updating the symbol table.
* **parser = yacc.yacc(start='program')**
* Builds the parser using all the defined p\_\* functions.
* **Used for**: Starting the parsing process with program as the root non-terminal.
* **add\_action\_entry()**
* Helper function to log manual parsing steps.
* **Used for**: Adding stack state and action descriptions to the action\_table.
* **execute()**
* Runs the final parse tree after successful parsing.
* **Used for**: Executing the program logic, especially the switch-case flow.
* **run\_switch()**
* Executes the logic inside the switch\_stmt.
* **Used for**: Taking user inputs, evaluating selected case, and printing the result.
* **first\_sets()**
* Returns and prints First sets of non-terminals.
* **Used for**: Constructing the parsing table and understanding parser decisions.
* **follow\_sets()**
* Returns and prints Follow sets of non-terminals.
* **Used for**: Aiding parsing table construction and error recovery.
* **generate\_parsing\_table\_with\_productions()**
* Uses First and Follow sets to build the parsing table.
* **Used for**: Displaying production rules mapped to non-terminal and terminal pairs.
* **simulate\_parsing(filename)**
* Orchestrates the entire parsing flow.
* **Used for**: Reading source code, simulating parsing steps, invoking parser, and executing parsed program.
* **generate\_manual\_parse\_sequence(code)**
* Manually simulates parsing stack actions.
* **Used for**: Demonstrating how parsing would happen step-by-step for learning/debugging.
* **if \_\_name\_\_ == '\_\_main\_\_'**
* Script's entry point when run directly.
* **Used for**: Triggering parsing table generation and parsing of input file passed via command line.

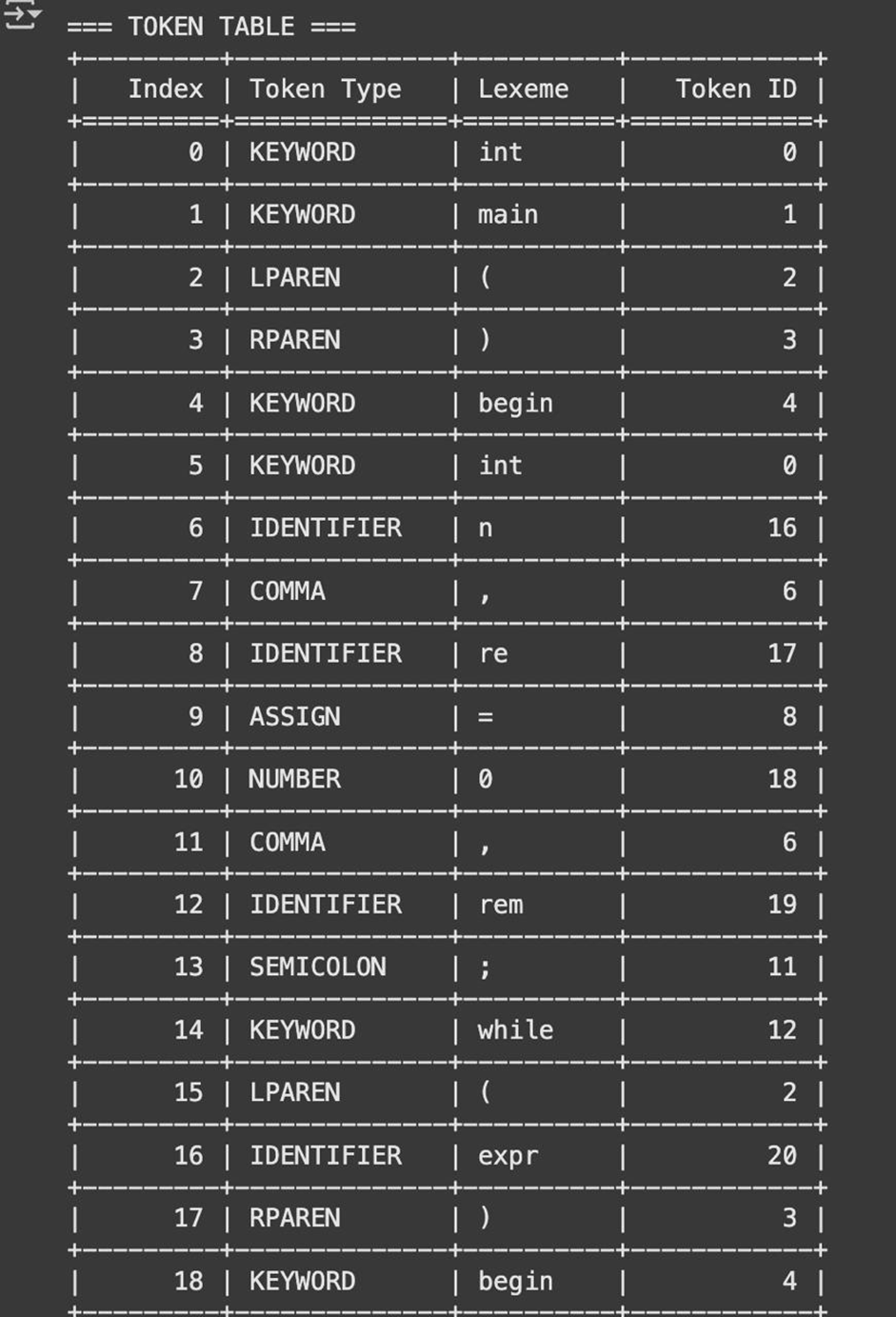
4.Result

4.1 Input File ( Source Program )



4.2.1 Tokenization Output from input file



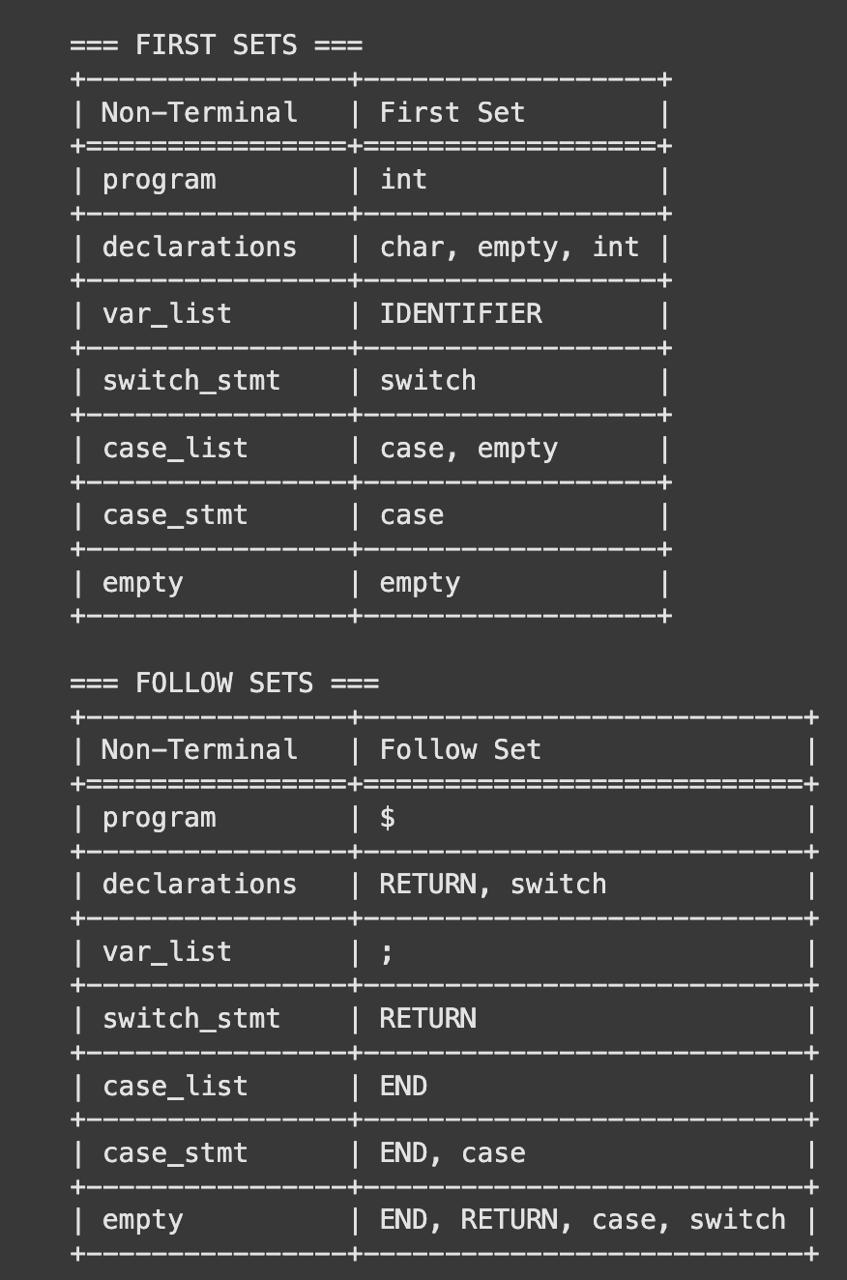




4.2.2 Token ID Map



4.3 First | Follow

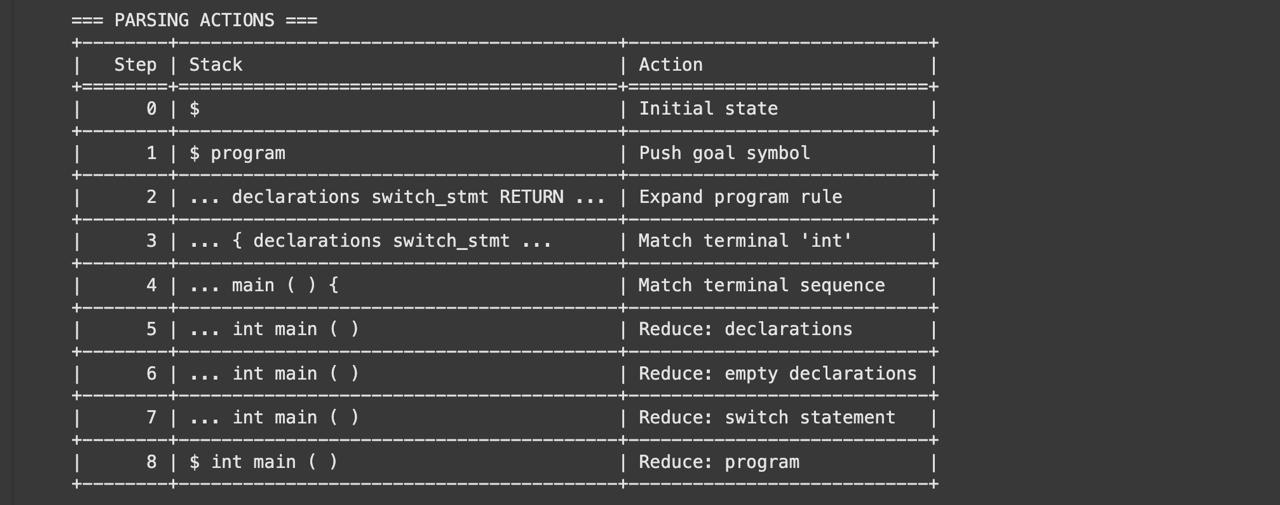


* 1. Parser

4.4.1 Parsing Table Generation



4.4.2 Parsing Action



5.Conclusion

This project successfully demonstrates the design and implementation of a compiler front-end for a simple, hypothetical programming language using Python. By focusing on lexical analysis and LL(1) syntax parsing, the project illustrates how fundamental compiler concepts such as tokenization, grammar definition, FIRST and FOLLOW sets, and predictive parsing can be applied in practice. The modular and extensible design lays a solid foundation for future enhancements, including semantic analysis and code generation. Overall, this project provides a hands-on understanding of compiler construction and the practical application of formal language theory.

6.References

https://www.javatpoint.com/compiler-tutorial https://www.tutorialspoint.com/compiler\_design/index.htm https://www.geeksforgeeks.org/compiler-design-tutorials/