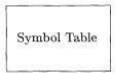
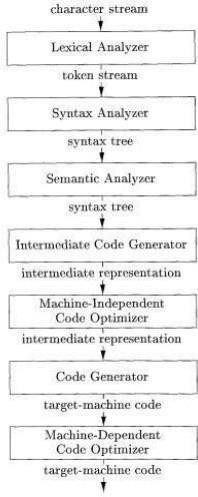
# INTRODUCTION

A compiler is a translator that converts the high-level language into the machine language. High- level language is written by a developer and machine language can be understood by the processor. Compiler is used to show errors to the programmer. The main purpose of compiler is to change the code written in one language without changing the meaning of the program. When you execute a program which is written in HLL programming language then it executes into two parts. In the first part, the source program compiled and translated into the object program (low level language). In the second part, object program translated into the target program through the assembler.

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.

**PHASES OF A COMPILER:**

If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in the figure below.

**Fig 1.0 – phases of compiler**

#### Analysis Part:

This part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

#### Synthesis Part:

This part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.

#### Lexical Analyzer:

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form.

(token-name, attribute-value)

#### Syntax Analyzer :

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

#### Semantic Analysis :

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

#### Intermediate Code Generation:

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

#### Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

#### Code Generation

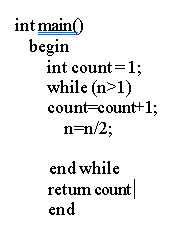
The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

#### Symbol Table

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

# PROBLEM STATEMENT

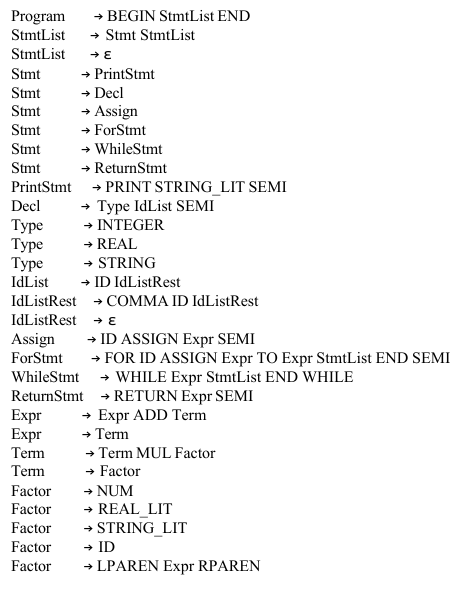
Number of digits in a binary number for a given decimal number This program computes the number of digits in the binary representation of a given decimal number n. It initializes a counter to 1 and repeatedly divides n by 2, incrementing the counter each time, until n becomes 1 or less. The number of divisions corresponds to the number of binary digits. Finally, it returns the count.

****

This project addresses the gap in practical tools for understanding how compilers process programming languages, particularly within academic settings. While students are often introduced to formal grammars and parsing theory, they seldom get the opportunity to work with complete, functional examples that demonstrate how compiler components operate together. To bridge this gap, the project implements a working compiler front-end for a pseudocode-based language using LALR (Look-Ahead LR) parsing techniques. The system starts with lexical analysis, converting raw source code into a stream of classified tokens. It then applies LALR parsing using manually defined grammar rules to carry out efficient shift-reduce parsing. The parser constructs compact LALR parsing tables from merged LR(1) item sets and processes the input accurately while maintaining the expressive power of full LR parsing.

In addition to precise syntax analysis, the system includes syntax error reporting and generates a visual parse tree to aid in debugging and understanding language structure. This design helps learners visualize how parsing decisions are made and how grammar rules are applied in real-time. By simulating realistic compiler behavior in a simplified, modular, and extensible format, the implementation serves as an ideal educational tool. It supports both learning and experimentation, making it especially beneficial for students and developers exploring the foundational concepts of compiler construction and LALR parsing.

**LALR grammar for the above problem statement is:**

****

# OBJECTIVES

The key goals of this project are as follows:

**Define Formal Grammar:**

* + Design a context-free grammar (CFG) that represents the syntax rules of a pseudocode

based programming language.

* + Ensure the grammar is compatible with LALR(1) parsing, which balances parsing powerand

table size by merging LR(1) states with identical cores.

**Lexical Analysis (Tokenizer):**

* + Implement a tokenizer that scans the source code and converts it into a stream of tokens

such as keywords, identifiers, literals, and symbols.

* + Use regular expressions to detect patterns and extract meaningful lexemes efficiently.

**Grammar Processing:**

* + Compute FIRST and FOLLOW sets for all non-terminal symbols to support grammar analysis.
  + Generate the LALR parsing table (ACTION and GOTO tables) by merging compatible LR(1) item sets, ensuring compact and efficient parsing for a wide range of grammars.

**LALR-based Shift-Reduce Parser:**

* + Use the LALR parsing table to carry out shift and reduce operations on the input token stream.
  + Dynamically construct a parse tree during parsing, associating grammar rules with the appropriate node structure.

**Error Detection:**

* + Provide clear and helpful syntax error messages, including the position and type of error, to support quick debugging and learning.
  + Handle unexpected or invalid tokens gracefully during parsing.

**Parse Tree Visualization:**

* + Generate a text-based ASCII parse tree to illustrate how the input is parsed step-by-step.
  + Optionally support graphical visualization using tools like Graphviz to render parse

trees as images (e.g., PNG format) for better comprehension.

**Educational Usability:**

* + Develop the system as an educational tool for learning key concepts such as tokenization,

parsing, and parse tree generation.

* + Make the architecture modular and extensible, so students can easily modify grammar rules,

input code, or extend features for experimentation.

# HARDWARE / SOFTWARE Requirements

**Hardware Requirements:**

* Minimum of 4 GB RAM.
* Processor with a clock speed of at least 2 GHz.

**Software Requirements:**

* Python 3.7 or above: The programming language used for implementation.
* graphviz: Used to generate graphical representations of the parse tree.
* numpy: Used for table and data management during parsing.

rich: Used to enhance console output with color and formatting for better readability

# METHODOLOGY

The compiler is structured into five main modules:

**Tokenization**:

* + Converts raw pseudocode into a list of tokens using regular expressions.
  + Recognizes token types such as identifiers, keywords, numbers, operators, and delimiters.
  + Each token includes additional metadata like its type, value, and position in the source code.

**Grammar Processing:**

* + Defines a context-free grammar for the custom pseudocode language.
  + Computes FIRST and FOLLOW sets for each non-terminal in the grammar.
  + Constructs the canonical collection of LR(1) items and generates the LALR parsing tables

**Parsing:**

* + Implements a shift-reduce parser that operates based on the LALR Action and Goto tables.
  + Performs state transitions on a parsing stack using LR(1) items.
  + Accurately detects syntax errors, indicating problematic tokens and their positions.

**Final Result:**

* + If it is Success then it means that the parsing was successful else it will give error and generate the Final result as Failure

# IMPLEMENTATION

The implementation process is structured around modular design, with each component having a distinct role in the compilation pipeline:

**Tokenization Module:**

* + This module uses Python’s re library (regular expressions) to define token patterns.
  + The tokenizer reads the source code and converts it into a stream of tokens, each tagged with its type (e.g., keyword, identifier, operator).

**Grammar Definition and Table Construction:**

* + A grammar file or in-code structure defines all the production rules.
  + The system calculates FIRST and FOLLOW sets for each non-terminal to understand which symbols can begin or follow particular constructs.
  + The LALR parsing table is then built, resolving possible shift-reduce or reduce- reduce conflicts based on lookahead tokens.

**LALR(1) Parser:**

* + Implements a shift-reduce engine that reads the token stream and uses the LALR(1) table to parse the code.
  + The parser uses a stack to hold symbols and states and performs actions based on the table entries (shift, reduce, accept, error).

**Parsing steps:**

* + Every shift and reduce is shown in this steps. Every parsing steps.
  + At the end it indicate the parsing status, whether it is successfully parsed or not.

# RESULTS AND DISCUSSIONS

The final system performs all stages of lexical and syntax analysis successfully. For well-formed input, it:

• Correctly tokenizes the input, listing all recognized tokens and their types.

• Parses the token stream using the constructed LALR table, applying shift-reduce parsing as needed.

• Outputs parsing steps in a structured tabular format, displaying actions like shift, reduce, and error handling.

• Builds an accurate parse tree that reflects the syntactic structure of the input program.

• Displays the parse tree in both ASCII format and graphical PNG image format for easy inspection.

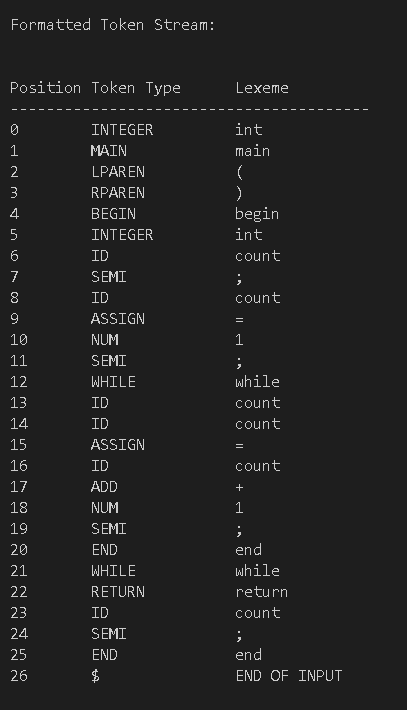
**Discussion Points:**

• The grammar was carefully designed to be LALR compatible, ensuring that parsing conflicts are avoided and the system can handle a wide range of syntactic structures.

• Error handling is functional but basic; syntax errors are detected, and parsing is stopped with descriptive error messages, including token positions, to help users identify and correct issues.

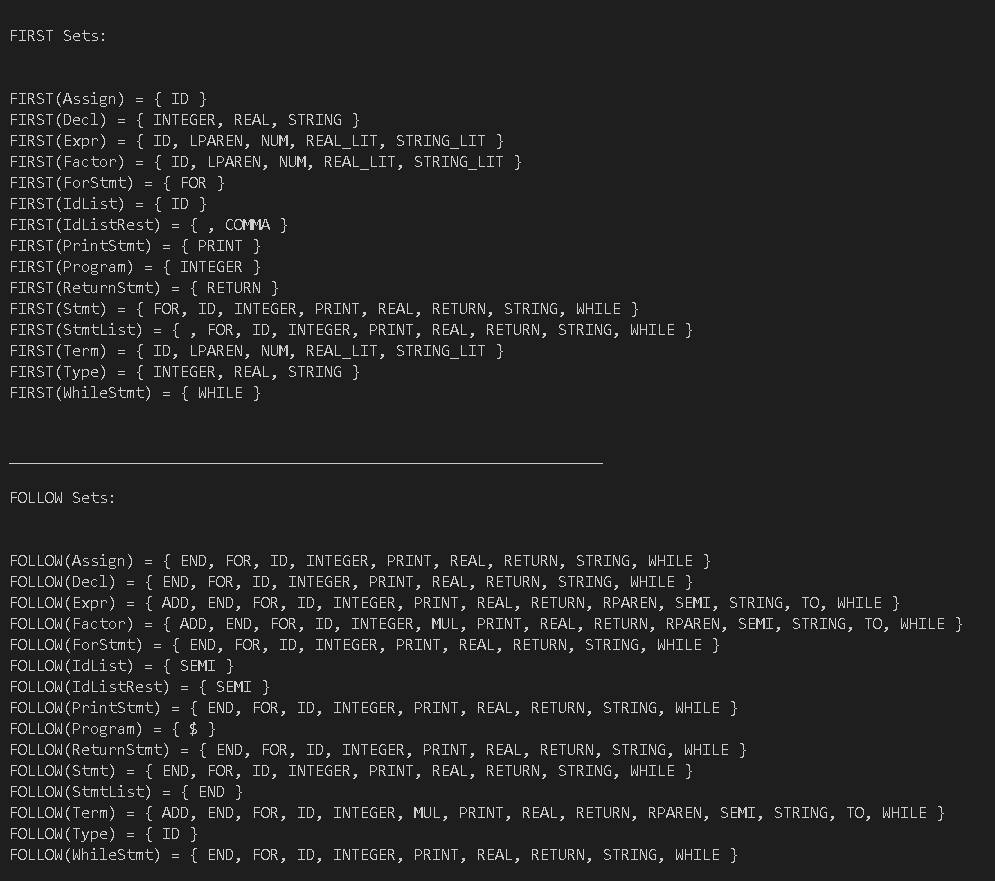
• For larger programs, the graphical tree becomes increasingly valuable, as the ASCII version can become difficult to interpret when the program's complexity increases.

**LEXEMES :**

****

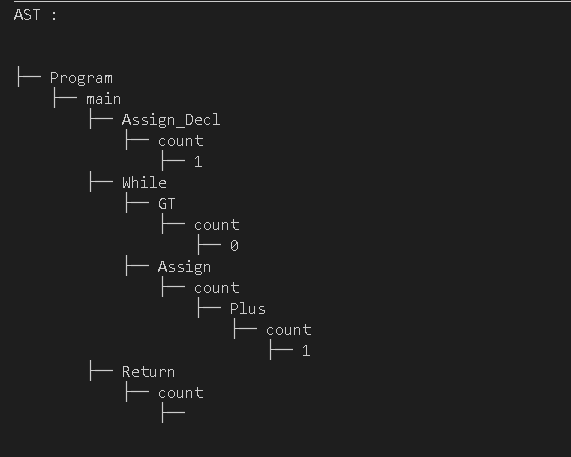
**Fig 1.1 – lexemes**

**FIRST & FOLLOW SET :**



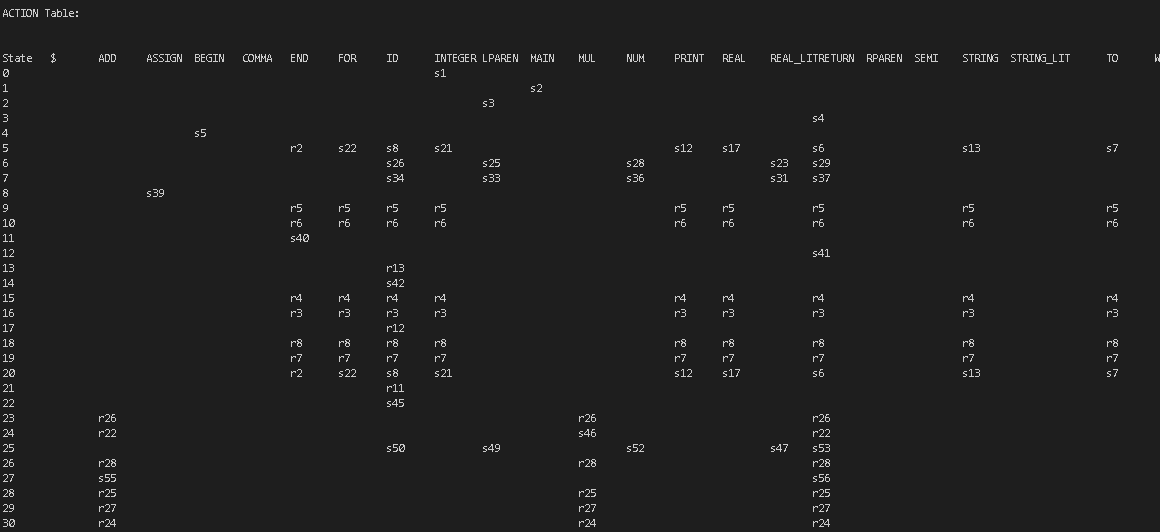
**Fig 1.2– first & follow set**

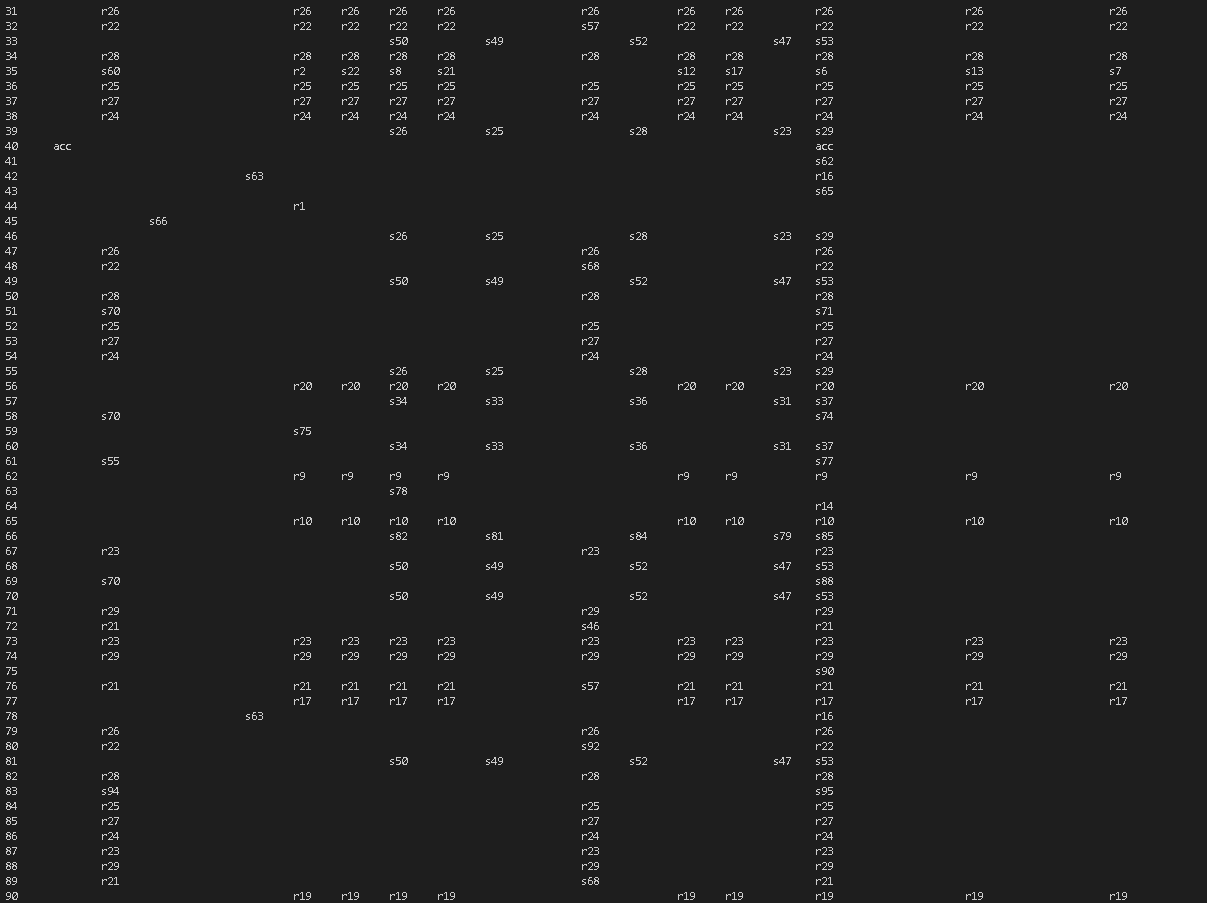
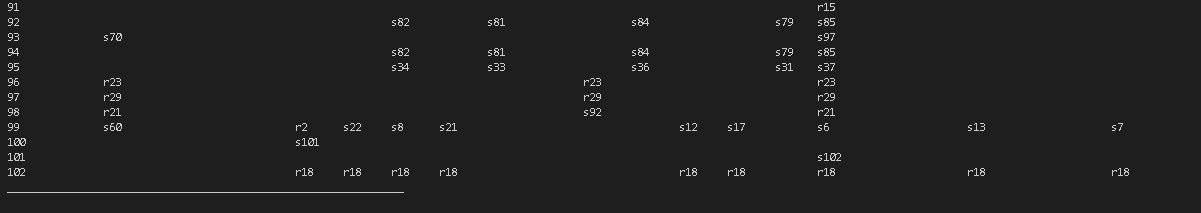
**AST :**



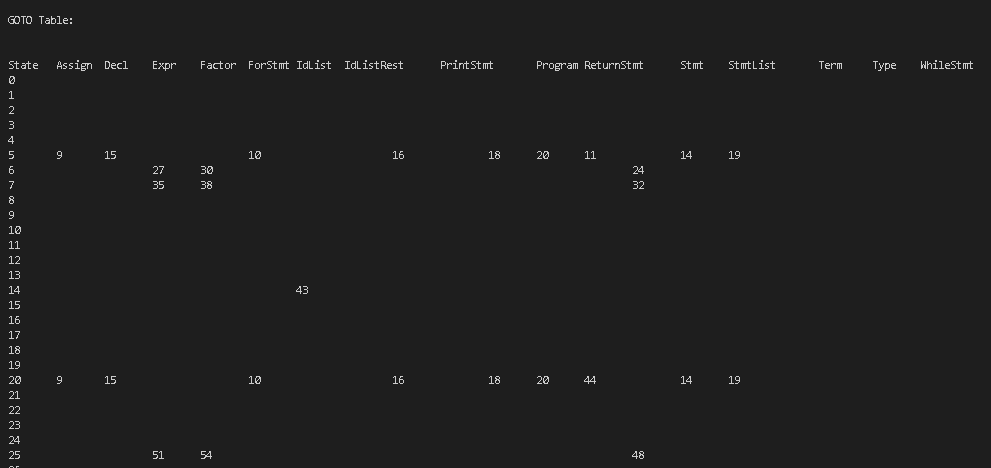
**Fig 1.3 – AST**

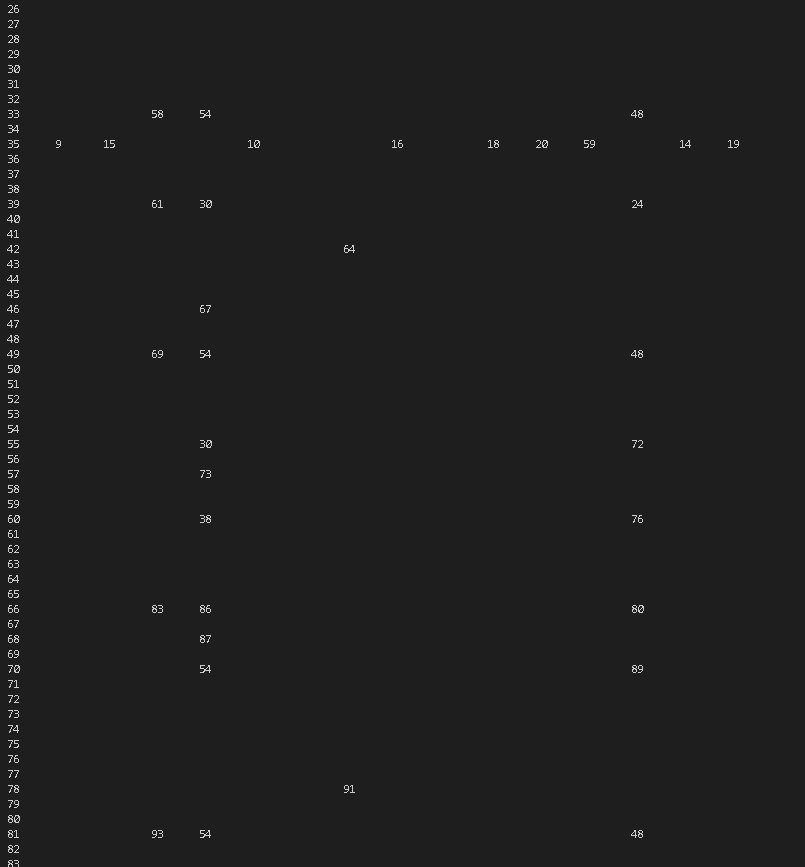
**LALR PARSING TABLE (ACTION) :**



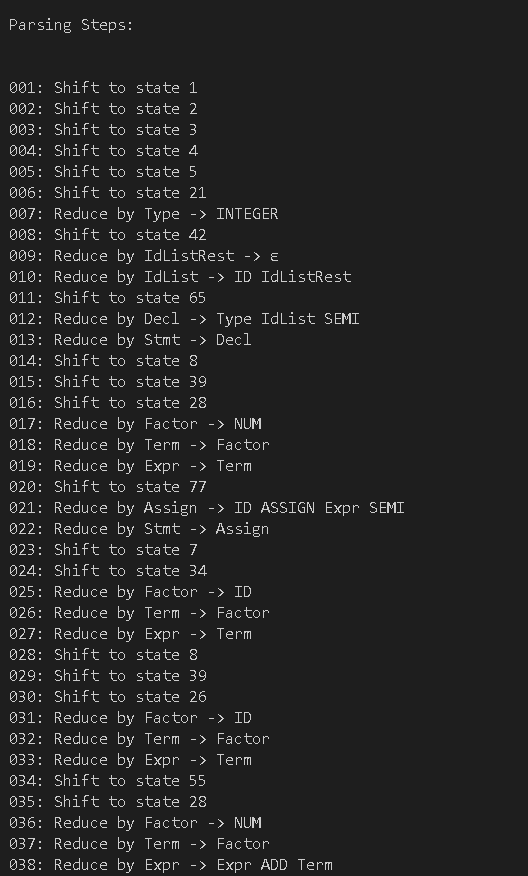


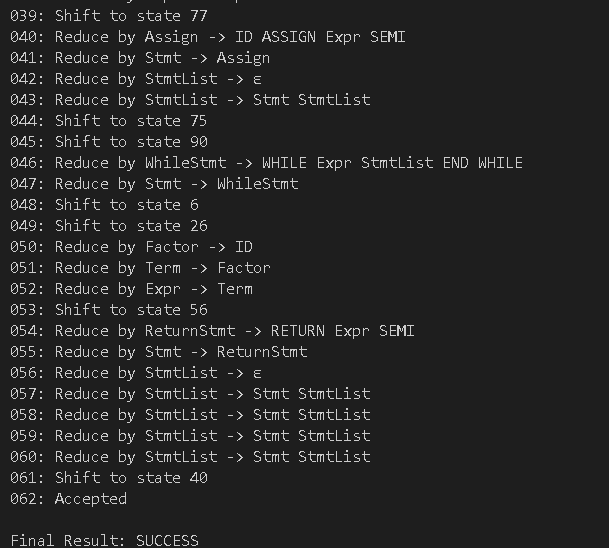
**Fig 1.4 – action table**

**LALR PARSING TABLE (GOTO) :**

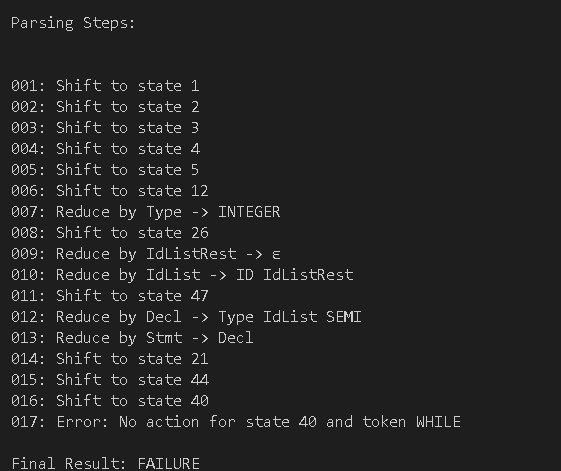
****

**Fig 1.5 – goto table**

**PARCING STEPS : for valid input**



**Fig 1.6 – successful parsing**

**PARCING STEPS : for invalid input**

**Fig 1.7 – unsuccessful parsing**

# CONCLUSION AND FUTURE SCOPE

**Conclusion:**

The LALR compiler for pseudocode effectively demonstrates key compiler design concepts, focusing on lexical analysis and syntax analysis. By implementing a tokenizer, grammar processor, LALR parser, and visualization engine, the project showcases a working model of the front-end phase of a compiler. This system serves as a valuable educational tool, allowing students and developers to explore the inner workings of a compiler in a controlled and simplified environment. This system serves as a valuable educational resource, helping students and developers understand the inner workings of a compiler in a controlled and simplified environment.

**Future Scope:**

**Semantic Analysis:**

* + Introduce scope resolution, type checking, and symbol table management to ensure the program's correctness beyond syntax.

**Intermediate Code Generation:**

* + Generate intermediate representations, such as three-address code, to enable further optimization and translation to target languages.

**Code Optimization and Backend Support:**

* + Extend the system to generate actual target code (e.g., Python, C, or assembly).
  + Implement optimization passes to enhance the efficiency of the generated code.

**Error Recovery:**

Implement error recovery techniques to handle syntax errors gracefully and continue parsing, enabling the detection of multiple issues in a single pass.

**IDE Plugin or GUI Interface:**

* + Integrate with a graphical user interface (GUI) or develop an IDE plugin that allows real- time code parsing, syntax highlighting, and parse tree visualization.

**Extended Grammar Support:**

○ Expand the grammar to include more advanced language features, such as loops, conditionals functions, and data structures, to support a broader range of programming constructs.

By expanding in these directions, this project can evolve from a learning tool into a full-fledged educational compiler platform.

# REFERENCES

[1] Wikipedia:LALR Parser

[2] Graphviz Tool:https://www.graphviz.org/

[3] Python Documentation:https://docs.python.org

Aho, Lam, Sethi, Ullman – Compilers: Principles, Techniques, and Tools (The Dragon Book).