REPORT ON

CD Mini Project Carried
out on

DEVELOPMENT AND IMPLEMENTING LALR PARSER FOR "Binary Digit Count from Decimals"

Submitted to

NMAM INSTITUTE OF TECHNOLOGY, NITTE
(An Autonomous Institution under VTU, Belagavi)

In partial fulfillment of the requirements for the award of the

Degree of Bachelor of Engineering in Computer Science Engineering

By

ADALINE CONOSA D'SILVA- NNM22CS214

K HARSHA S HAVALDAR-NNM23CS505

RAKSHITH P R - NNM23CS507

Submitted to,

Dr. Asmitha Poojari Assistant
Professor [GD-III]
Department of Computer Science and Engineering

NITTE EDUCATION TRUST | NMAM INSTITUTE OF TECHNOLOGY

NITTE EDUCATION TRUST | NMAM INSTITUTE OF TECHNOLOGY

CERTIFICATE

This is to certify that the Ms. ADALINE CONOSA D'SILVA bearing USN Of NNM22CS214 and Mr. K HARSHA S HAVALDAR bearing USN of NNM23CS505 and Mr. RAKSHITH P R bearing USN of NNM23CS507 of VI semester B.E., a Bonafide student of NMAM Institute of Technology, Nitte, has completed CD mini project on " DEVELOPMENT AND IMPLEMENTING LALR PARSER FOR Binary Digit Count from Decimals" during January 2025 - May 2025 fulfilling the partial requirements for the award of degree of Bachelor of Engineering in ***Computer Science and 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 steadfast 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 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.

I would like to thank my college administration for providing a conductive environment and also suitable facilities for this mini 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. It is my great pleasure to thank my mentor **Dr. Asmitha Poojari** for her continuous encouragement, guidance, and support throughout this project.

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

ADALINE CONOSA D’SILVA (NNM22CS214)
K HARSHA S HAVALDAR (NNM23CS505)
RAKSHITH P R (NNM23CS507)

ABSTRACT

This project presents the design and implementation of a compiler front-end using Python, focusing on the lexical analysis and syntax parsing phases. The system emulates the functionality of traditional LEX and YACC tools, implementing token generation, computation of FIRST and FOLLOW sets, and parse table construction. Using Python's string processing capabilities and object-oriented features, we developed a robust lexical analyzer that efficiently breaks source code into valid tokens, followed by a parser that validates syntax against defined grammar rules.

The implementation demonstrates how high-level languages like Python can effectively handle compiler construction tasks typically associated with lower-level tools. Key achievements include successful tokenization of input programs, proper computation of compiler theory constructs (FIRST, FOLLOW, and parse tables), and accurate syntax validation. The project serves as both an educational resource for understanding compiler design principles and a foundation for building more advanced language processing tools.

This work provides valuable insights into the practical application of formal language theory while showcasing Python's versatility in systems programming domains. The modular design allows for future extensions to include semantic analysis and code generation phases.

TABLE OF CONTENTS

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.

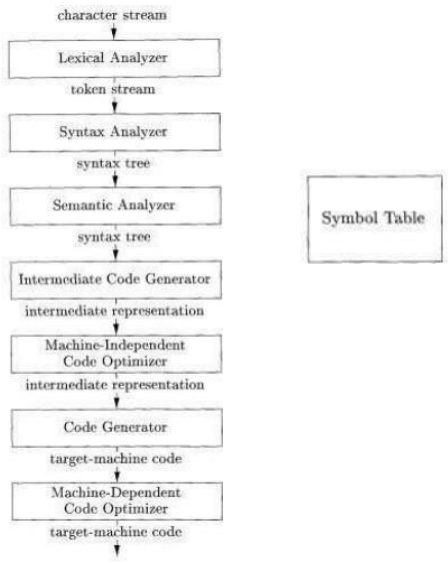

Fig 1.0 – phases of compiler

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.

```
int main()
  begin
     int count = 1;
     while (n>1)
     count=count+1;
        n=n/2;

     end while
     return count
     end
```

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:

Program → BEGIN StmtList END
StmtList → Stmt StmtList
StmtList → ε
Stmt → PrintStmt
Stmt → Decl
Stmt → Assign
Stmt → ForStmt
Stmt → WhileStmt
Stmt → ReturnStmt
PrintStmt → PRINT STRING_LIT SEMI
Decl → Type IdList SEMI
Type → INTEGER
Type → REAL
Type → STRING
IdList → ID IdListRest
IdListRest → COMMA ID IdListRest
IdListRest → ε
Assign → ID ASSIGN Expr SEMI
ForStmt → FOR ID ASSIGN Expr TO Expr StmtList END SEMI
WhileStmt → WHILE Expr StmtList END WHILE
ReturnStmt → RETURN Expr SEMI
Expr → Expr ADD Term
Expr → Term
Term → Term MUL Factor
Term → Factor
Factor → NUM
Factor → REAL_LIT
Factor → STRING_LIT
Factor → ID
Factor → LPAREN Expr RPAREN

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 :

```
Formatted Token Stream:


Position Token Type      Lexeme
----------------------------------------
0        INTEGER         int
1        MAIN            main
2        LPAREN          (
3        RPAREN          )
4        BEGIN           begin
5        INTEGER         int
6        ID              count
7        SEMI            ;
8        ID              count
9        ASSIGN          =
10       NUM             1
11       SEMI            ;
12       WHILE           while
13       ID              count
14       ID              count
15       ASSIGN          =
16       ID              count
17       ADD             +
18       NUM             1
19       SEMI            ;
20       END             end
21       WHILE           while
22       RETURN          return
23       ID              count
24       SEMI            ;
25       END             end
26       $               END OF INPUT
```

Fig 1.1 – lexemes

FIRST & FOLLOW SET :

```
FIRST Sets:


FIRST(Assign) = { ID }
FIRST(Decl) = { INTEGER, REAL, STRING }
FIRST(Expr) = { ID, LPAREN, NUM, REAL_LIT, STRING_LIT }
FIRST(Factor) = { ID, LPAREN, NUM, REAL_LIT, STRING_LIT }
FIRST(ForStmt) = { FOR }
FIRST(IdList) = { ID }
FIRST(IdListRest) = { , COMMA }
FIRST(PrintStmt) = { PRINT }
FIRST(Program) = { INTEGER }
FIRST(ReturnStmt) = { RETURN }
FIRST(Stmt) = { FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
FIRST(StmtList) = { , FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
FIRST(Term) = { ID, LPAREN, NUM, REAL_LIT, STRING_LIT }
FIRST(Type) = { INTEGER, REAL, STRING }
FIRST(WhileStmt) = { WHILE }



_____________________________________________________________

FOLLOW Sets:


FOLLOW(Assign) = { END, FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
FOLLOW(Decl) = { END, FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
FOLLOW(Expr) = { ADD, END, FOR, ID, INTEGER, PRINT, REAL, RETURN, RPAREN, SEMI, STRING, TO, WHILE }
FOLLOW(Factor) = { ADD, END, FOR, ID, INTEGER, MUL, PRINT, REAL, RETURN, RPAREN, SEMI, STRING, TO, WHILE }
FOLLOW(ForStmt) = { END, FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
FOLLOW(IdList) = { SEMI }
FOLLOW(IdListRest) = { SEMI }
FOLLOW(PrintStmt) = { END, FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
FOLLOW(Program) = { $ }
FOLLOW(ReturnStmt) = { END, FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
FOLLOW(Stmt) = { END, FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
FOLLOW(StmtList) = { END }
FOLLOW(Term) = { ADD, END, FOR, ID, INTEGER, MUL, PRINT, REAL, RETURN, RPAREN, SEMI, STRING, TO, WHILE }
FOLLOW(Type) = { ID }
FOLLOW(WhileStmt) = { END, FOR, ID, INTEGER, PRINT, REAL, RETURN, STRING, WHILE }
```

Fig 1.2– first & follow set

AST :

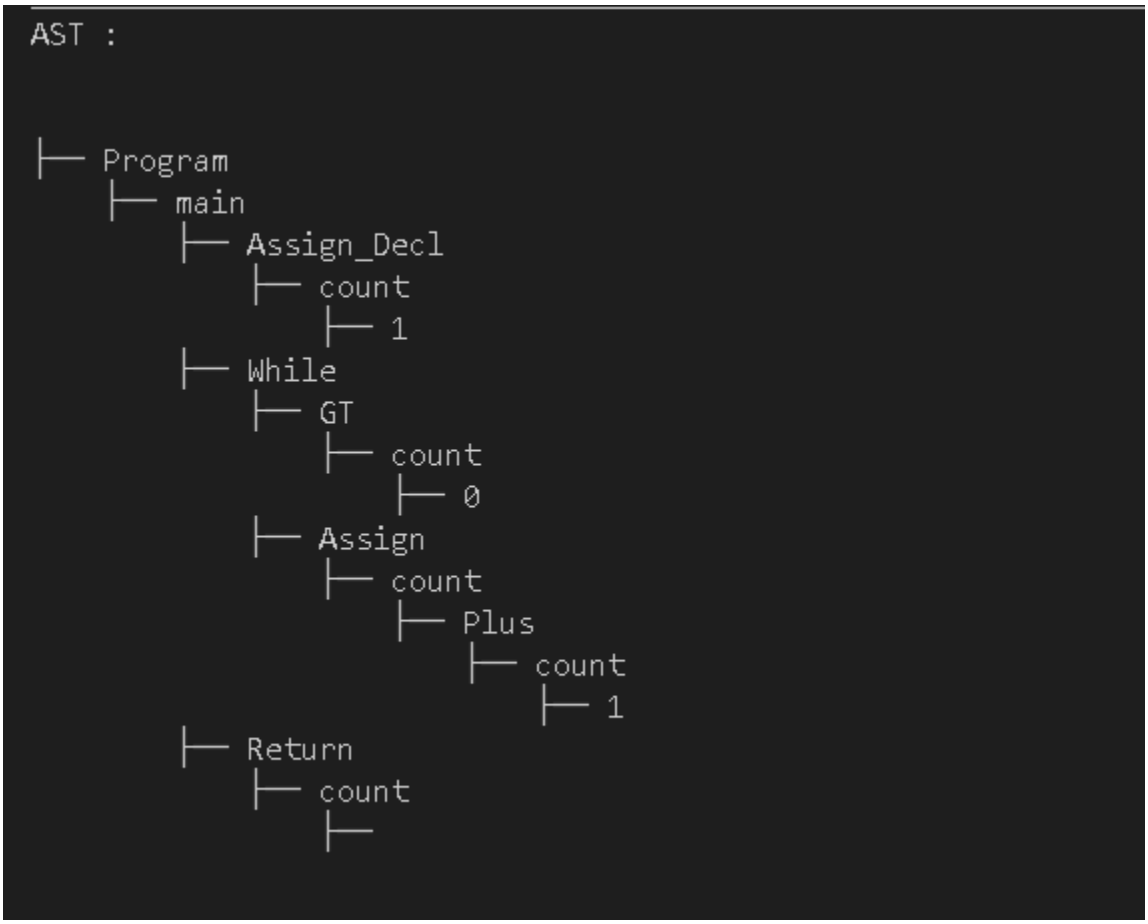

Fig 1.3 – AST

LALR PARSING TABLE (ACTION) :

ACTION Table:

State	$	ADD	ASSIGN	BEGIN	COMMA	END	FOR	ID	INTEGER	LPAREN	MAIN	MUL	NUM	PRINT	REAL	REAL_LIT	RETURN	RPAREN	SEMI	STRING	STRING_LIT	TO	W
0									s1														
1											s2												
2										s3													
3																	s4						
4				s5																			
5						r2	s22	s8	s21					s12	s17		s6			s13		s7	
6								s26		s25			s28			s23	s29						
7								s34		s33			s36			s31	s37						
8			s39																				
9						r5	r5	r5	r5					r5	r5		r5			r5		r5	
10						r6	r6	r6	r6					r6	r6		r6			r6		r6	
11						s40																	
12																	s41						
13								r13															
14								s42															
15						r4	r4	r4	r4					r4	r4		r4			r4		r4	
16						r3	r3	r3	r3					r3	r3		r3			r3		r3	
17								r12															
18						r8	r8	r8	r8					r8	r8		r8			r8		r8	
19						r7	r7	r7	r7					r7	r7		r7			r7		r7	
20						r2	s22	s8	s21					s12	s17		s6			s13		s7	
21								r11															
22								s45															
23		r26										r26					r26						
24		r22										s46					r22						
25								s50		s49			s52			s47	s53						
26		r28										r28					r28						
27		s55															s56						
28		r25										r25					r25						
29		r27										r27					r27						
30		r24										r24					r24						

31		r26			r26	r26	r26	r26		r26		r26	r26		r26	r26	r26
32		r22			r22	r22	r22	r22		s57		r22	r22		r22	r22	r22
33							s50		s49		s52			s47	s53		
34		r28			r28	r28	r28	r28		r28		r28	r28		r28	r28	r28
35		s60			r2	s22	s8	s21				s12	s17		s6	s13	s7
36		r25			r25	r25	r25	r25		r25		r25	r25		r25	r25	r25
37		r27			r27	r27	r27	r27		r27		r27	r27		r27	r27	r27
38		r24			r24	r24	r24	r24		r24		r24	r24		r24	r24	r24
39							s26		s25		s28			s23	s29		
40	acc														acc		
41															s62		
42				s63											r16		
43															s65		
44					r1												
45			s66														
46							s26		s25		s28			s23	s29		
47		r26								r26					r26		
48		r22								s68					r22		
49							s50		s49		s52			s47	s53		
50		r28								r28					r28		
51		s70													s71		
52		r25								r25					r25		
53		r27								r27					r27		
54		r24								r24					r24		
55							s26		s25		s28			s23	s29		
56					r20	r20	r20	r20				r20	r20		r20	r20	r20
57							s34		s33		s36			s31	s37		
58		s70													s74		
59					s75												
60							s34		s33		s36			s31	s37		
61		s55													s77		
62					r9	r9	r9	r9				r9	r9		r9	r9	r9
63							s78										
64															r14		
65					r10	r10	r10	r10				r10	r10		r10	r10	r10
66							s82		s81		s84			s79	s85		
67		r23								r23					r23		
68							s50		s49		s52			s47	s53		
69		s70													s88		
70							s50		s49		s52			s47	s53		
71		r29								r29					r29		
72		r21								s46					r21		
73		r23			r23	r23	r23	r23		r23		r23	r23		r23	r23	r23
74		r29			r29	r29	r29	r29		r29		r29	r29		r29	r29	r29
75															s90		
76		r21			r21	r21	r21	r21		s57		r21	r21		r21	r21	r21
77					r17	r17	r17	r17				r17	r17		r17	r17	r17
78				s63											r16		
79		r26								r26					r26		
80		r22								s92					r22		
81							s50		s49		s52			s47	s53		
82		r28								r28					r28		
83		s94													s95		
84		r25								r25					r25		
85		r27								r27					r27		
86		r24								r24					r24		
87		r23								r23					r23		
88		r29								r29					r29		
89		r21								s68					r21		
90					r19	r19	r19	r19				r19	r19		r19	r19	r19
91															r15		
92							s82		s81		s84			s79	s85		
93		s70													s97		
94							s82		s81		s84			s79	s85		
95							s34		s33		s36			s31	s37		
96		r23								r23					r23		
97		r29								r29					r29		
98		r21								s92					r21		
99		s60			r2	s22	s8	s21				s12	s17		s6	s13	s7
100					s101												
101															s102		
102					r18	r18	r18	r18				r18	r18		r18	r18	r18

Fig 1.4 – action table

LALR PARSING TABLE (GOTO) :

GOTO Table:

State	Assign	Decl	Expr	Factor	ForStmt	IdList	IdListRest		PrintStmt		Program	ReturnStmt		Stmt	StmtList		Term	Type	WhileStmt
0																			
1																			
2																			
3																			
4																			
5	9	15			10			16		18	20	11		14	19				
6			27	30									24						
7			35	38									32						
8																			
9																			
10																			
11																			
12																			
13																			
14						43													
15																			
16																			
17																			
18																			
19																			
20	9	15			10			16		18	20	44		14	19				
21																			
22																			
23																			
24																			
25			51	54									48						

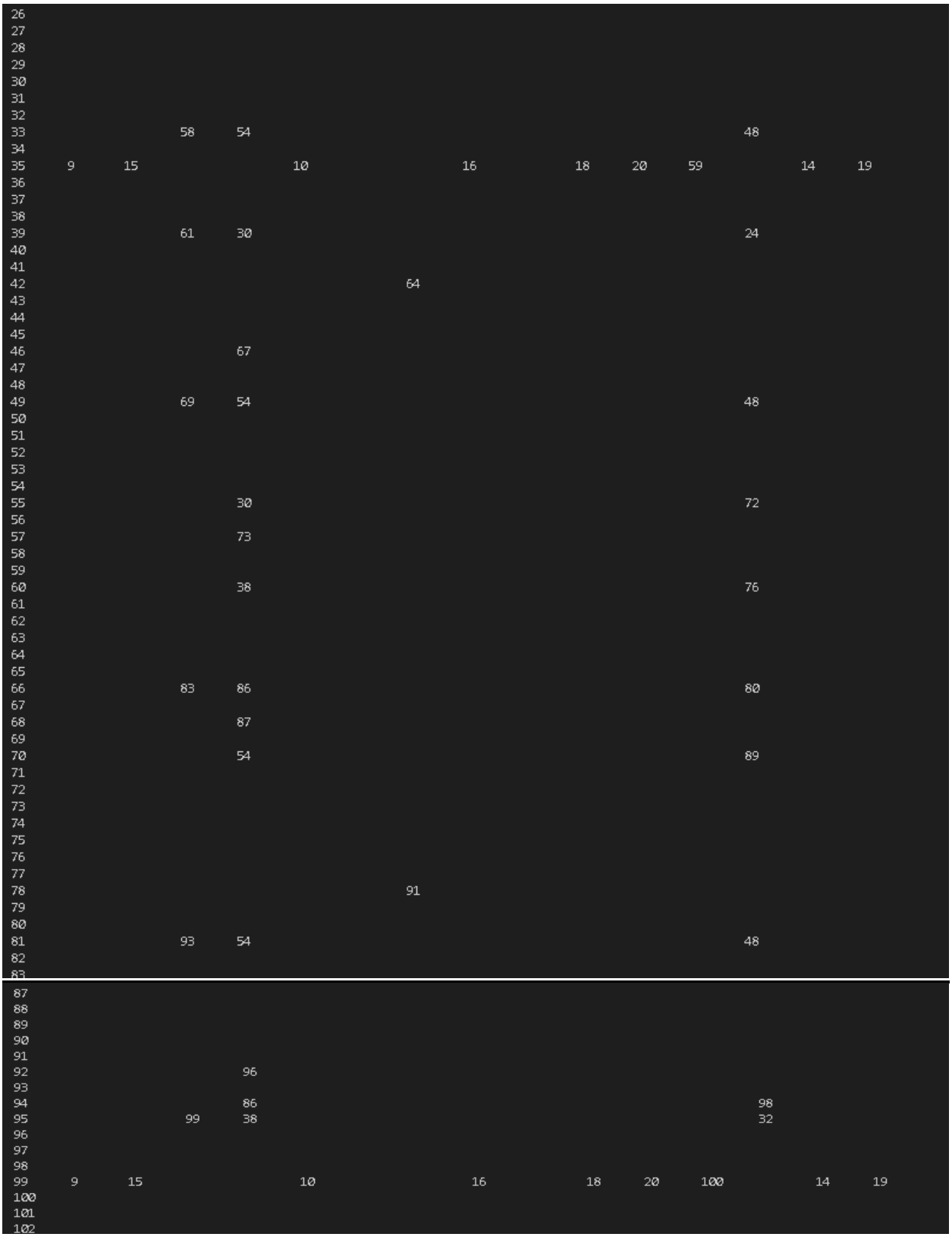

Fig 1.5 – goto table

PARCING STEPS : for valid input

```
Parsing Steps:


001: Shift to state 1
002: Shift to state 2
003: Shift to state 3
004: Shift to state 4
005: Shift to state 5
006: Shift to state 21
007: Reduce by Type -> INTEGER
008: Shift to state 42
009: Reduce by IdListRest -> ε
010: Reduce by IdList -> ID IdListRest
011: Shift to state 65
012: Reduce by Decl -> Type IdList SEMI
013: Reduce by Stmt -> Decl
014: Shift to state 8
015: Shift to state 39
016: Shift to state 28
017: Reduce by Factor -> NUM
018: Reduce by Term -> Factor
019: Reduce by Expr -> Term
020: Shift to state 77
021: Reduce by Assign -> ID ASSIGN Expr SEMI
022: Reduce by Stmt -> Assign
023: Shift to state 7
024: Shift to state 34
025: Reduce by Factor -> ID
026: Reduce by Term -> Factor
027: Reduce by Expr -> Term
028: Shift to state 8
029: Shift to state 39
030: Shift to state 26
031: Reduce by Factor -> ID
032: Reduce by Term -> Factor
033: Reduce by Expr -> Term
034: Shift to state 55
035: Shift to state 28
036: Reduce by Factor -> NUM
037: Reduce by Term -> Factor
038: Reduce by Expr -> Expr ADD Term
```

```
039: Shift to state 77
040: Reduce by Assign -> ID ASSIGN Expr SEMI
041: Reduce by Stmt -> Assign
042: Reduce by StmtList -> ε
043: Reduce by StmtList -> Stmt StmtList
044: Shift to state 75
045: Shift to state 90
046: Reduce by WhileStmt -> WHILE Expr StmtList END WHILE
047: Reduce by Stmt -> WhileStmt
048: Shift to state 6
049: Shift to state 26
050: Reduce by Factor -> ID
051: Reduce by Term -> Factor
052: Reduce by Expr -> Term
053: Shift to state 56
054: Reduce by ReturnStmt -> RETURN Expr SEMI
055: Reduce by Stmt -> ReturnStmt
056: Reduce by StmtList -> ε
057: Reduce by StmtList -> Stmt StmtList
058: Reduce by StmtList -> Stmt StmtList
059: Reduce by StmtList -> Stmt StmtList
060: Reduce by StmtList -> Stmt StmtList
061: Shift to state 40
062: Accepted

Final Result: SUCCESS
```

Fig 1.6 – successful parsing

PARCING STEPS : for invalid input

```
Parsing Steps:


001: Shift to state 1
002: Shift to state 2
003: Shift to state 3
004: Shift to state 4
005: Shift to state 5
006: Shift to state 12
007: Reduce by Type -> INTEGER
008: Shift to state 26
009: Reduce by IdListRest -> ϵ
010: Reduce by IdList -> ID IdListRest
011: Shift to state 47
012: Reduce by Decl -> Type IdList SEMI
013: Reduce by Stmt -> Decl
014: Shift to state 21
015: Shift to state 44
016: Shift to state 40
017: Error: No action for state 40 and token WHILE

Final Result: FAILURE
```

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).