Project File

Compiler Design

Rupam Barui

B.Tech - M.Tech Computer Science and Engineering
Semester 6

This project file contains all the experiments conducted in the lab under the guidance of Dr. Abinash Mishra

Project Github Link

National Forensic Sciences University
New Delhi

National Forensic
Sciences University
Knowledge | Wisdom | Fulfilment
An Institution of National Importance
(Ministry of Home Affairs, Government of India)

July 19, 2024

Contents

Experiment 1: Case Study on C, C++, and Java Compilers

Aim

The aim of this case study is to understand the process of compilation for C, C++, and Java programs by detailing the steps and mechanisms involved in converting high-level source code into executable machine code.

Objectives

- To analyze and compare the compilation process of C, C++, and Java.
- To explore the different phases of the compilation process including lexical analysis, syntax analysis, semantic analysis, code generation, and code optimization.
- To implement and execute sample programs in C, C++, and Java and observe the compiler's behavior.

Introduction

Compilers are essential tools in the field of computer science, transforming high-level programming languages into executable machine code. In this case study, we will focus on compilers for three widely used programming languages: C, C++, and Java. We will begin by briefly discussing the different compiler siblings and types of compilers, followed by an in-depth analysis of the compilation process, including the various stages involved.

Compiler Siblings

Before delving into the specifics of C, C++, and Java compilers, let's take a look at some common compiler siblings:

Assembler

An assembler translates assembly language code into machine code. It is a low-level language compiler that provides a more human-readable representation of machine instructions.

Interpreter

An interpreter reads and executes code line by line, without generating an executable file. It is commonly used for scripting languages like Python and JavaScript.

Preprocessor

A preprocessor is a tool that performs text substitution and macro expansion before the actual compilation process begins. It is commonly used in C and C++ to handle directives like #include and #define.

Types of Compilers

Compilers vary in design and purpose. Here's a brief overview of different compiler types:

Single-pass Compilers

Fast and straightforward, these compilers read the source code once, generating target code with limited optimization. Example: early Pascal compilers.

Multi-pass Compilers

These compilers perform several passes over the code, optimizing and error checking. Example: GCC, Clang.

Cross Compilers

Cross compilers generate code for different platforms than their host system. Example: the GNU ARM toolchain.

Transpilers

Transpilers convert source code between languages, maintaining logic and structure. Example: Babel (transpiles modern JavaScript to older versions).

Decompilers

Decompilers reverse the compilation process, converting machine code back to a high-level language. Example: Hex-Rays Decompiler for IDA Pro.

Native Compilers

Native compilers produce machine code for the compiler's platform. Example: Microsoft Visual C++ for Windows applications.

Bootstrapping Compilers

These are compilers written in the same language they compile, built in stages for self-improvement. Example: many versions of the C compiler initially written in a subset of C, compiled with a simpler compiler, and then able to compile its subsequent versions.

Compiler Steps with Examples

C Compiler Steps

Example: Compilation of a C program to calculate the factorial of a number.

```
#include <stdio.h>

int factorial(int n) {
```

```
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    int num = 5;
    int result = factorial(num);
    printf("Factorial of %d is %d\n", num, result);
    return 0;
}
```

Step 1: Preprocessing The preprocessor handles directives like #include, which includes the contents of the ¡stdio.h¿ header file. It also expands any macros and resolves conditional compilation directives.

Step 2: Compilation The C compiler performs lexical analysis, breaking the preprocessed code into tokens. It then performs syntax analysis, checking the code against the C grammar rules and generating an abstract syntax tree (AST). Semantic analysis is done to validate type consistency and perform type checking.

Step 3: Code Generation The compiler generates assembly code from the AST. It performs optimizations such as constant folding, dead code elimination, and register allocation.

Step 4: Assembling The assembler takes the assembly code and translates it into object code, which contains machine instructions specific to the target architecture.

Step 5: Linking The linker combines the object code with any required libraries and resolves external references. It produces an executable file that can be run on the target machine.

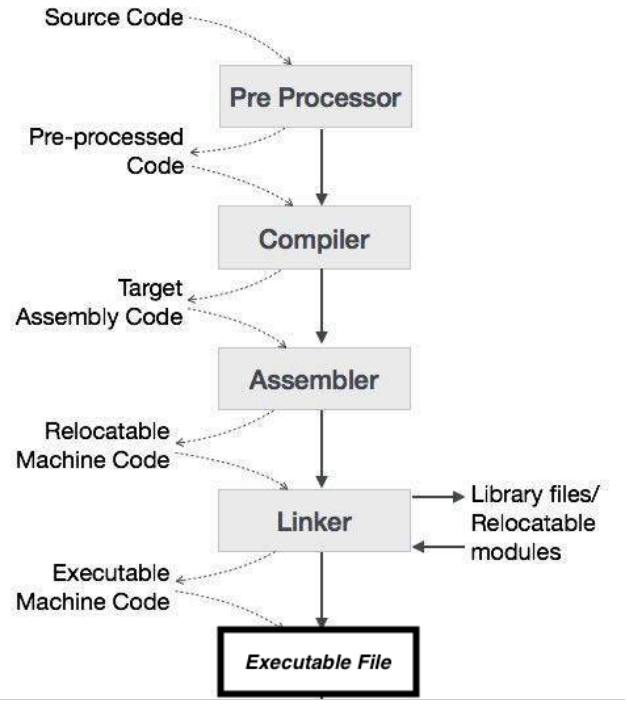

Figure 1: C Code Compilation Steps

C++ Compiler Steps

Example: Compilation of a C++ program to implement a simple class.

```
#include <iostream>

class Rectangle {
private:
    double width;
    double height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area() {
        return width * height;
    }
};

int main() {
    Rectangle rect(5.0, 3.0);
```

```
    std::cout << "Area: " << rect.area() << std::endl;
    return 0;
}
```

Step 1: Preprocessing The preprocessor handles #include directives, such as including the ¡iostream¿ header file. It also expands macros and resolves conditional compilation directives.

Step 2: Compilation The C++ compiler performs lexical analysis, syntax analysis, and semantic analysis similar to the C compilation process. Additionally, it handles C++-specific features like classes, templates, and exception handling.

Step 3: Code Generation The compiler generates assembly code from the AST. It applies optimizations specific to C++ such as function inlining, virtual function table generation, and name mangling.

Step 4: Assembling The assembler converts the assembly code into object code, which contains machine instructions specific to the target architecture.

Step 5: Linking The linker combines the object code with any necessary C++ libraries (such as the C++ Standard Library) and resolves external references. It creates an executable file that can be run on the target machine.

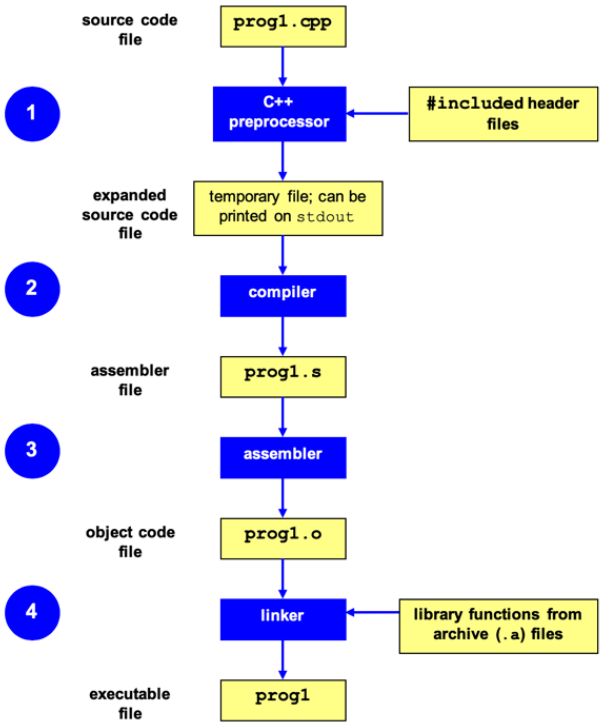

Figure 2: C++ Code Compilation Steps

Java Compiler Steps

Example: Compilation of a Java program to implement a simple inheritance hierarchy.

```
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Dog dog = new Dog();

        animal.makeSound();
        dog.makeSound();
    }
}
```

Step 1: Compilation The Java compiler (javac) compiles the Java source code into bytecode. It performs lexical analysis, syntax analysis, and semantic analysis. The compiler generates a .class file for each class in the program.

Step 2: Class Loading The Java ClassLoader loads the bytecode of the required classes into the Java Virtual Machine (JVM). It locates and reads the .class files.

Step 3: Bytecode Verification The JVM verifies the bytecode to ensure it is valid and does not violate any security restrictions. This step helps maintain the integrity and security of the Java runtime environment.

Step 4: Execution The JVM interprets the bytecode and executes the program. It translates the bytecode into machine instructions specific to the target platform. The JVM also performs runtime optimizations using its Just-In-Time (JIT) compiler.

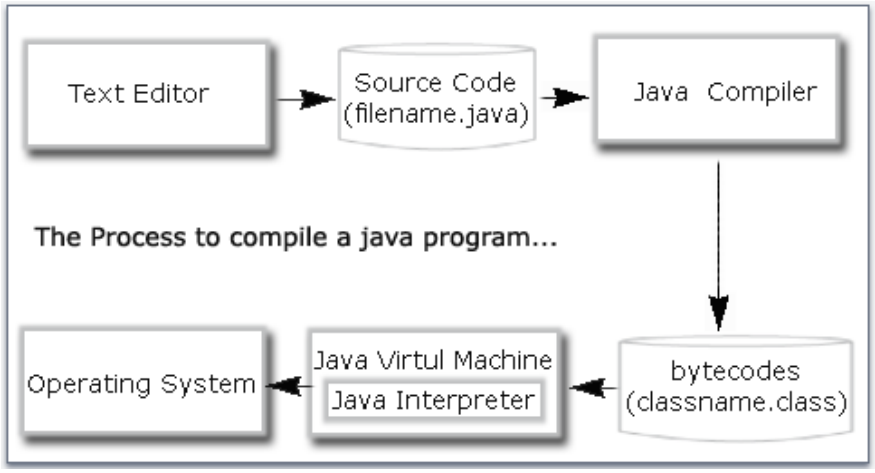

Figure 3: Java Code Compilation Steps

Conclusion

This case study explored the compilation process of C, C++, and Java programs. We analyzed the steps involved in each language's compilation process, including preprocessing, lexical analysis, syntax analysis, semantic analysis, code generation, and linking.

C and C++ follow a similar compilation process, with the addition of C++-specific features in the C++ compiler. The source code is preprocessed, compiled into assembly code, assembled into object code, and finally linked with libraries to create an executable.

Java, on the other hand, follows a different compilation process. The Java source code is compiled into bytecode, which is then executed by the Java Virtual Machine (JVM). The JVM performs class loading, bytecode verification, and execution, providing platform independence and enhanced security.

Understanding the compilation process of different programming languages is crucial for developers to write efficient and optimized code. It also helps in debugging and troubleshooting issues that may arise during the compilation and execution of programs.

References

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley.
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). The Java Language Specification, Java SE 8 Edition. Oracle.
- Stroustrup, B. (2013). The C++ Programming Language (4th Edition). Addison-Wesley.

Experiment Steps

Step 1: Setup the Development Environment

Install the necessary compilers and development tools for C, C++, and Java on your system. This may include:

- C Compiler (e.g., GCC)
- C++ Compiler (e.g., G++)
- Java Development Kit (JDK)
- Integrated Development Environment (IDE) or text editor of your choice

Step 2: Write Sample Programs

Create sample programs in C, C++, and Java that demonstrate various language features and concepts. These programs should cover different aspects of the language, such as:

- Basic input/output operations
- Control structures (if-else, loops)
- Functions or methods
- Arrays and data structures
- Object-oriented programming concepts (classes, inheritance, polymorphism)

Step 3: Compile the Programs

Compile the sample programs using the respective compilers for each language. Observe the compilation process and note any warnings or errors generated by the compiler. Analyze the generated assembly code or bytecode to understand how the high-level code is translated into lower-level instructions.

Step 4: Execute the Programs

Run the compiled programs and verify their output. Test various inputs and edge cases to ensure the programs behave as expected. Observe the runtime behavior and performance of the programs.

Step 5: Analyze the Compilation Process

Study the different stages of the compilation process for each language. Use debugging tools or compiler flags to generate intermediate files (e.g., preprocessed code, assembly code) at each stage. Analyze these files to understand how the code is transformed and optimized during compilation.

Step 6: Compare and Contrast

Compare the compilation process of C, C++, and Java. Identify the similarities and differences between them. Consider factors such as compilation time, memory usage, platform dependence, and runtime performance. Discuss the advantages and limitations of each language's compilation approach.

Step 7: Document and Present Findings

Document your observations, analysis, and conclusions from the experiment. Prepare a report or presentation summarizing your findings. Include code snippets, compiler output, and performance metrics to support your analysis. Present your findings to your peers or instructor for discussion and feedback.

Evaluation

The case study can be evaluated based on the following criteria:

- Correctness and completeness of the sample programs
- Understanding of the compilation process for each language
- Depth of analysis and comparison between the languages
- Clarity and organization of the documentation and presentation
- Ability to answer questions and provide insights during the discussion

Limitations and Future Work

This case study provides an overview of the compilation process for C, C++, and Java. However, it has some limitations:

- The sample programs used in the experiment may not cover all the features and complexities of each language.
- The analysis is based on a specific set of compilers and tools, and the results may vary with different versions or implementations.
- The case study does not delve into advanced topics such as compiler optimizations, linker scripts, or runtime environments.

Future work can include:

- Exploring additional programming languages and their compilation processes
- Investigating advanced compiler optimizations and their impact on performance
- Analyzing the role of linkers and runtime environments in the overall execution of programs
- Examining the compilation process for domain-specific languages or scripting languages

Acknowledgments

We would like to acknowledge the contributions of the following individuals and resources in the preparation of this case study:

- The developers and maintainers of the GCC, G++, and Java compilers for their excellent tools and documentation.
- The authors of the referenced books and articles for their valuable insights and explanations of compiler concepts.
- The open-source community for providing access to a wide range of programming resources and examples.

Glossary

Assembly Code Low-level code that represents machine instructions and is specific to a particular processor architecture.

Bytecode Intermediate code generated by compilers for languages like Java. It is executed by a virtual machine.

Compiler A program that translates high-level source code into lower-level code, such as assembly code or machine code.

Interpreter A program that directly executes high-level source code without prior compilation.

Lexical Analysis The process of breaking down the source code into a sequence of tokens, such as keywords, identifiers, and literals.

Linker A program that combines object files and resolves external references to create an executable file.

Object Code Machine code that is generated by the assembler from the assembly code.

Preprocessor A program that processes directives and macros in the source code before the actual compilation begins.

Semantic Analysis The process of checking the source code for semantic errors and verifying type consistency.

Syntax Analysis The process of checking the source code against the grammar rules of the programming language.

Experiment 2: Implementing Various Automata using JFLAP

Aim

The aim of this experiment is to design and implement deterministic finite automata (DFA) and non-deterministic finite automata (NFA) for a variety of languages using JFLAP.

Objectives

- To understand the concepts of deterministic finite automata (DFA) and non-deterministic finite automata (NFA).
- To design DFA for various languages and regular expressions.
- To design NFA for various conditions and constraints.
- To implement these designs using JFLAP.
- To verify the correctness of the implemented automata through comprehensive testing with a range of inputs.

Experiment Description

Using JFLAP, implement the following automata designs:

1. Design DFA for the alphabet $\Sigma = \{a, b\}$:
   - $L1 = \{a^n b^m \mid n, m \geq 0\}$
   - $L2 = \{w \mid |w| \mod 3 = 0\}$
   - $L3 = \{a^n b \mid n \geq 0\}$
   - $L4 = \{ab^*(a + b)^*\}$ - All strings with prefix "ab"
2. Design DFA for the alphabet $\Sigma = \{0, 1\}$:
   - Construct a DFA that does not contain the substring "001".
3. Design DFA for a given regular expression:
   - Implement a DFA that accepts strings matching a given regular expression.
4. Construct NFA that accepts a set of strings with specific conditions:
   - Exactly one 'a'
   - At least one 'a'
   - No more than 3 'a'
   - At least 1 'a' and 2 'b'

- Exactly 2 'a' and more than 2 'b'

Outputs

The output for each automata design will be displayed as an image. Below are the results for each implementation:

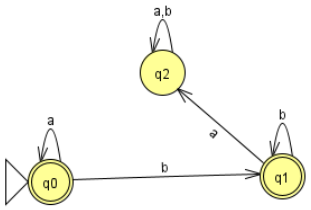

Figure 4: DFA for $L1 = \{a^n b^m \mid n, m \geq 0\}$

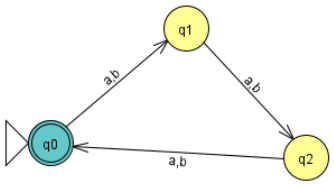

Figure 5: DFA for $L2 = \{w \mid |w| \mod 3 = 0\}$

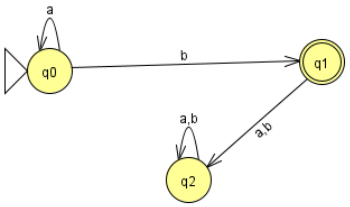

Figure 6: DFA for $L3 = \{a^n b \mid n \geq 0\}$

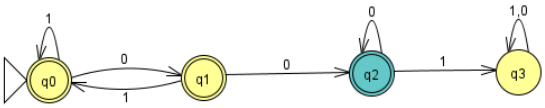

Figure 7: DFA for strings without substring "001"

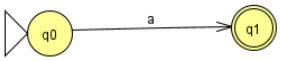

Figure 8: NFA for strings with exactly one 'a'

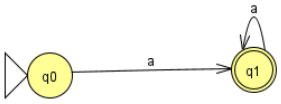

Figure 9: NFA for strings with at least one 'a'

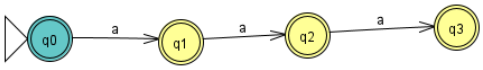

Figure 10: NFA for strings with no more than 3 'a'

Experiment 3, 4: Lexical Analyzer : Token Counter + Data Types Identification

Aim

The aim of this experiment is to design and implement a simple lexical analyzer in C++ that counts tokens and identifies data types within a given expression.

Objectives

- To understand the concept of lexical analysis in compiler design.
- To implement token counting logic in C++.
- To differentiate and identify various tokens such as operators, identifiers, separators, and literals.
- To familiarize with the process of recognizing data types within expressions.

Experiment Code

```
#include <iostream>
#include <string>
#include <cctype>
bool isOperator(char c)
{
   return c == '+' || c == '-' || c == '*' || c == '/';
}
bool isSeparator(char c)
{
   return c == ';' || c == ',' || c == '(' || c == ')';
}
int main()
{
   std::string expression;
   std::cout << "Enter the expression: ";
   std::getline(std::cin, expression);
   std::string token;
   int tokenCount = 0;
   bool inString = false;
   bool inComment = false;
   for (size_t i = 0; i < expression.length(); ++i)
   {
      char c = expression[i];
      if (inComment)
      {
         if (c == '*' && i + 1 < expression.length() && expression[i
    + 1] == '/')
```

```
        {
            // End of block comment
            inComment = false;
            i++; // Skip the closing '/'
        }
        continue; // Ignore everything inside comments
    }
    if (inString)
    {
        token += c;
        if (c == '"')
        {
            // End of string literal
            inString = false;
            std::cout << "'" << token << "' (string)\n";
            tokenCount++;
            token.clear();
        }
        continue;
    }
    if (c == '"')
    {
        if (!token.empty())
        {
            std::cout << "'" << token << "' (identifier)\n";
            tokenCount++;
            token.clear();
        }
        inString = true;
        token += c;
        continue;
    }
    if (isSeparator(c))
    {
        if (!token.empty())
        {
            std::cout << "'" << token << "' (identifier)\n";
            tokenCount++;
            token.clear();
        }
        std::cout << "'" << c << "' (separator)\n";
        tokenCount++; // Count separators
        continue;
    }
    if (isOperator(c))
    {
        if (!token.empty())
```

```
            {
                std::cout << "'" << token << "' (identifier)\n";
                tokenCount++;
                token.clear();
            }
            if (c == '+' && i + 1 < expression.length() && expression[i
     + 1] == '+')
            {
                std::cout << "'++' (increment operator)\n";
                i++; // Skip next character
            }
            else
            {
                std::cout << "'" << c << "' (operator)\n";
            }
            tokenCount++;
            continue;
        }
        if (c == '/' && i + 1 < expression.length() && expression[i +
    1] == '*')
        {
            // Start of block comment
            inComment = true;
            i++; // Skip the opening '*'
            continue;
        }
        if (!std::isspace(c))
        {
            token += c;
        }
        else if (!token.empty())
        {
            std::cout << "'" << token << "' (identifier)\n";
            tokenCount++;
            token.clear();
        }
    }
    if (!token.empty())
    {
        std::cout << "'" << token << "' (identifier)\n";
        tokenCount++;
    }
    std::cout << "The number of tokens = " << tokenCount << std::endl
    ;
    return 0;
}
```

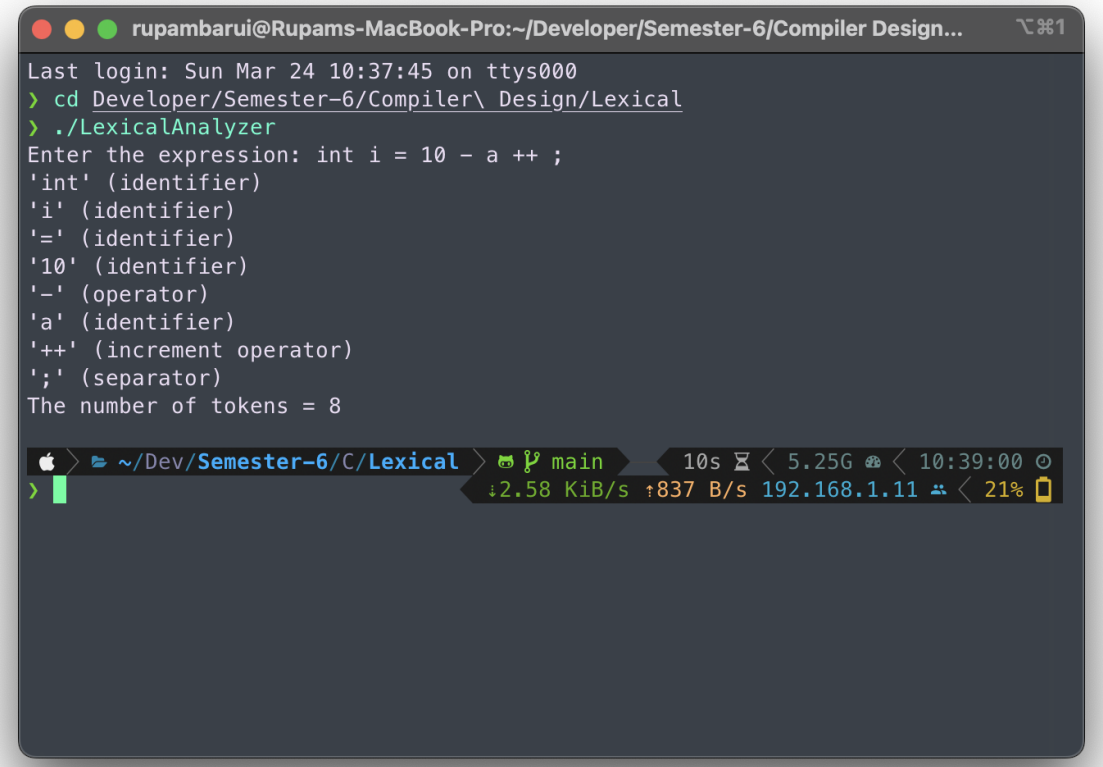

Figure 11: Lexical Analyser

Experiment 5.1: Lex Program : Identify the strings ending with 11

Aim

The aim of this experiment is to design and implement a lex program which can identify the strings passed as input which ends with 11.

Objectives

- To understand the concept of regular expressions and their application in lexical analysis.
- To gain practical experience with the lex (or flex) tool for pattern matching.
- To implement a regular expression that matches strings ending with "11."
- To develop a lex program that can correctly identify and accept input strings that meet the specified pattern criteria.
- To test the lex program with various input cases to ensure accuracy and robustness.

Experiment Code

```
%{
#include <stdio.h>
int count_ending_with_11 = 0;
%}
%%
.*11\n { count_ending_with_11++;   printf("Found a line ending with
   '11': %s", yytext); }
.|\n   { /* Ignore all other characters */ }

%%

int main(int argc, char **argv) {
    yylex();
    printf("Total number of lines ending with '11': %d\n",
   count_ending_with_11);
    return 0;
}

int yywrap() {
    return 1;
}
```

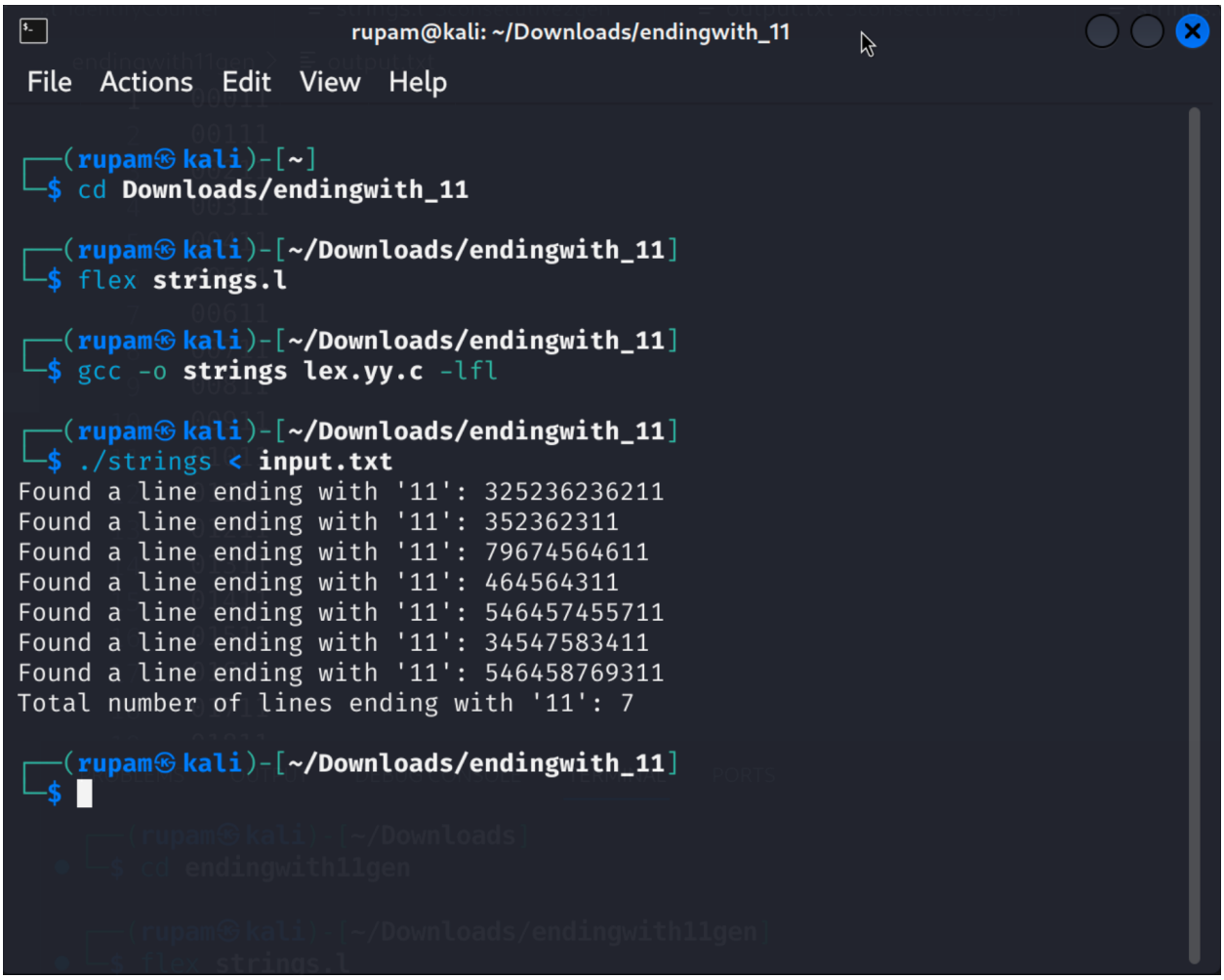

Figure 12: Strings ending with 11

Experiment 5.2: Lex Program : Identify the strings with 3 consecutive 2s

Aim

The aim of this experiment is to design and implement a lex program that can identify strings passed as input containing three consecutive '2's.

Objectives

- To understand how to construct regular expressions for pattern matching specific string conditions.
- To gain practical experience with the lex (or flex) tool for creating lexical analyzers.
- To implement a regular expression that identifies strings containing "222."
- To develop a lex program that recognizes and flags input strings containing three consecutive '2's.
- To validate the functionality of the lex program through a series of test cases with various input strings.

Experiment Code

```
%{
#include <stdio.h>

int count_with_222 = 0;
%}
%%
.*222 { count_with_222++;   printf("\nFound a line with '222': %s",
   yytext); }
.|\n   { /* Ignore all other characters */ }
%%
int main(int argc, char **argv) {
    yylex();
    printf("\nTotal number of lines with '222': %d\n",
   count_with_222);
    return 0;
}

int yywrap() {
    return 1;
}
```

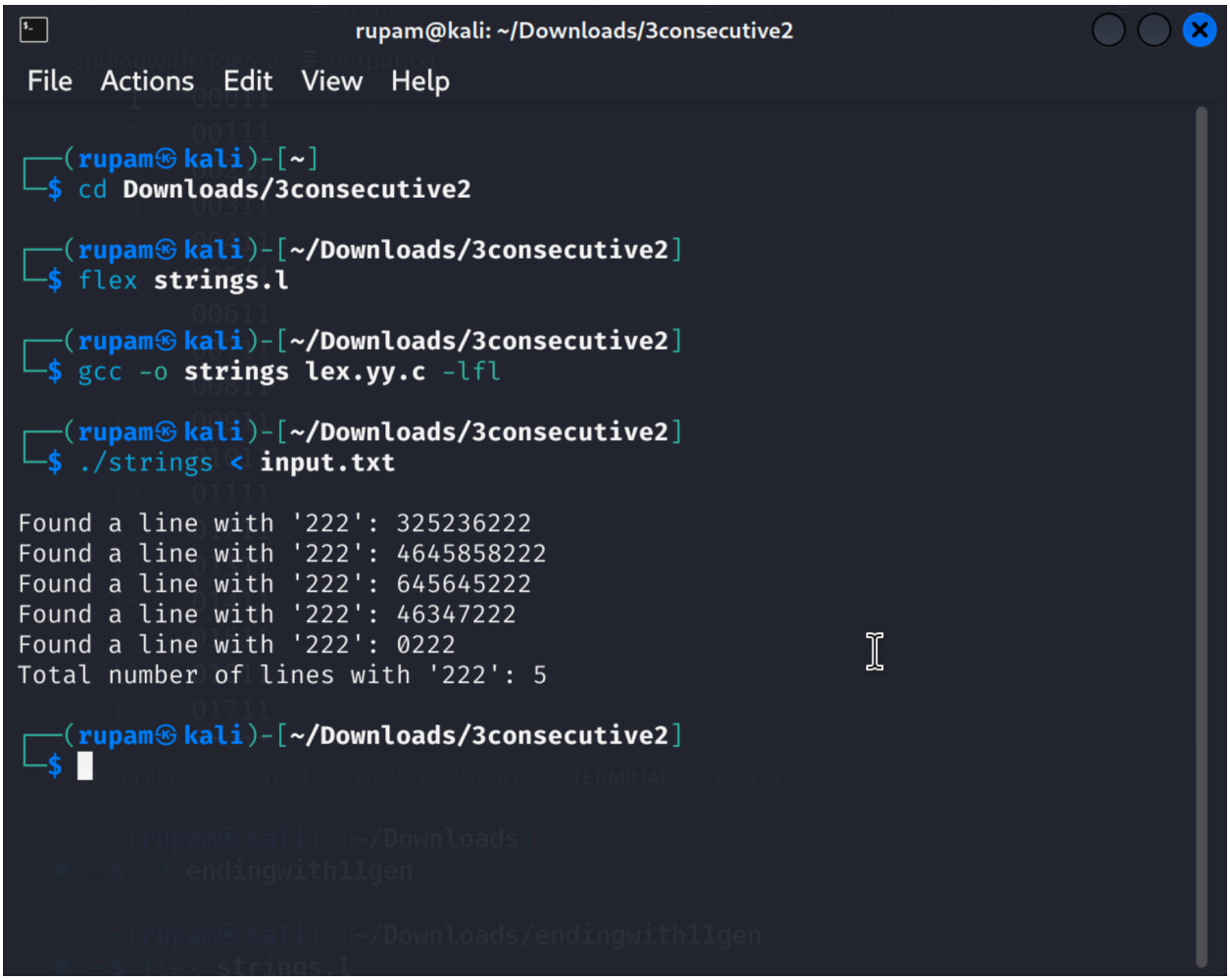

Figure 13: Srings Containing 3 consecutive 2s

Experiment 5.3: Lex Program : Identify the strings such that the 10th symbol from right end is 1.

Aim

The aim of this experiment is to design and implement a lex program that can identify strings passed as input which have the 10th symbol from the right end as '1'.

Objectives

- To understand the intricacies of pattern matching and regular expressions for positions within strings.
- To explore the lex (or flex) tool capabilities for developing lexical analyzers with position-dependent patterns.
- To design a regular expression that can precisely match strings with the 10th symbol from the right end being '1'.
- To create a lex program capable of processing input strings and correctly identifying those that match the specified positional pattern.
- To conduct thorough testing of the lex program against a variety of input cases to confirm its functionality and reliability.

Experiment Code

```
%{
#include <stdio.h>
int count_10th_from_right_is_1 = 0;
%}
%%
.*1.{9} { count_10th_from_right_is_1++;   printf("Found a line with
   '1' as 10th character from the right: %s", yytext); }
.|\n      { /* Ignore all other characters */ }
%%
int main(int argc, char **argv) {
    yylex();
    printf("\nTotal number of lines where the 10th character from
   the right is '1': %d\n", count_10th_from_right_is_1);
    return 0;
}

int yywrap() {
    return 1;
}
```

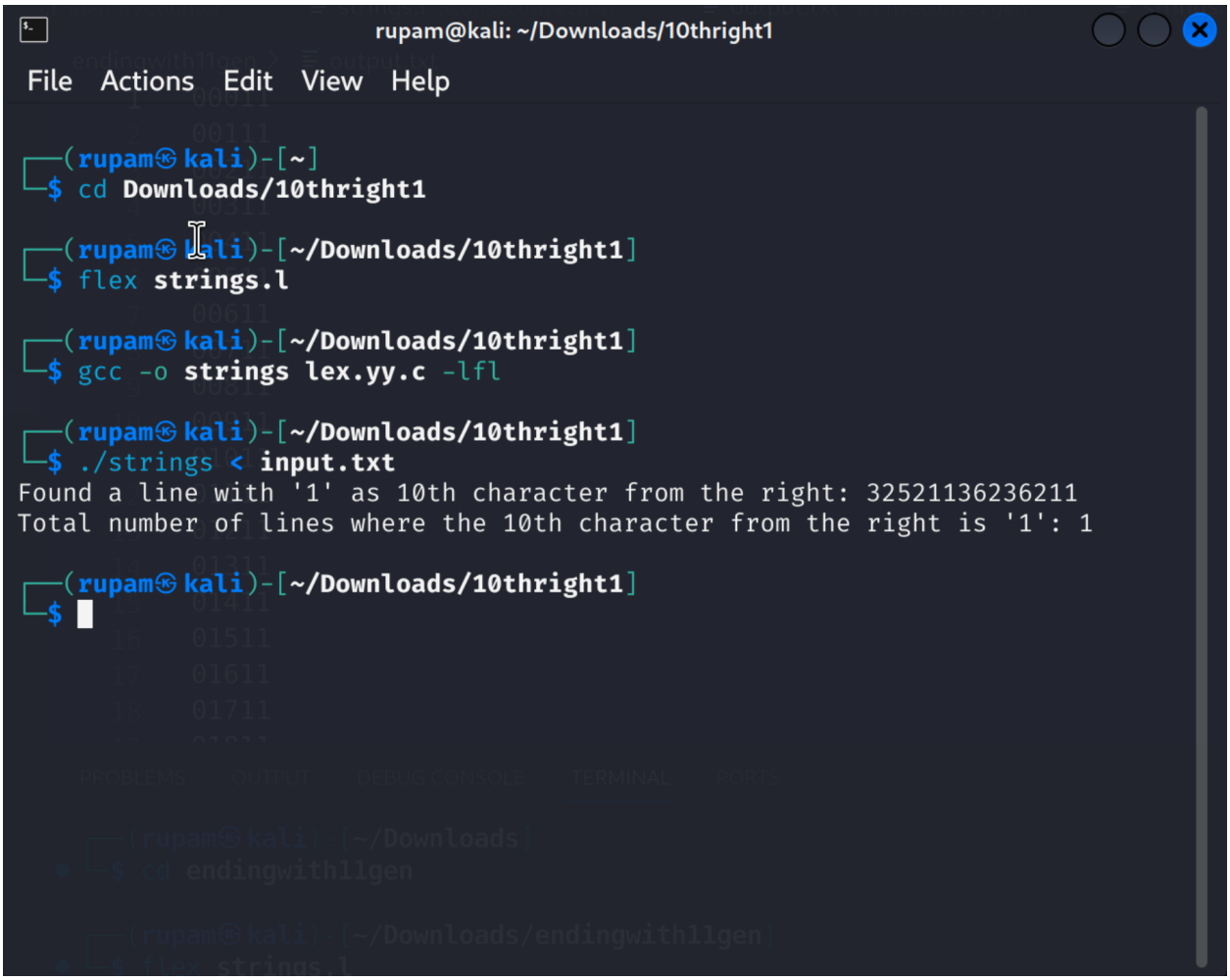

Figure 14: Enter Caption

Experiment 5.4: Lex Program : Identify all the 4 digit numbers whose sum is 9.

Aim

The aim of this experiment is to design and implement a lex program to identify all 4-digit numbers inputted by the user whose digits sum to 9.

Objectives

- To understand and apply regular expressions to match specific numeric patterns.
- To gain experience with lex (or flex) to create lexical analyzers focused on number processing.
- To implement a regular expression that matches 4-digit numbers with a digit sum of 9.
- To develop a lex program that accurately sorts through user input and identifies valid 4-digit numbers based on the sum of their digits.
- To test the lex program thoroughly with a variety of input numbers to ensure correct identification and functionality.

Experiment Code

```
%{
#include <stdio.h>
int sum_of_digits(const char *str);
int count_sum_9 = 0;

%}

%%
[0-9]{4}  { if (sum_of_digits(yytext) == 9) {count_sum_9++ ; printf(
   "Valid number: %s\n", yytext);}}
.|\n       { /* ignore other input */ }
%%

int sum_of_digits(const char *str) {
    int sum = 0;
    while(*str) {
        sum += *str - '0'; // Convert char digit to int and add to
   sum
        str++;
    }
    return sum;
}
```

```
int yywrap() { return 1; }

int main() {
    yylex();
    printf("Total number of strings whose digits add up to 9 : %d\n"
   , count_sum_9);
    return 0;
}
```

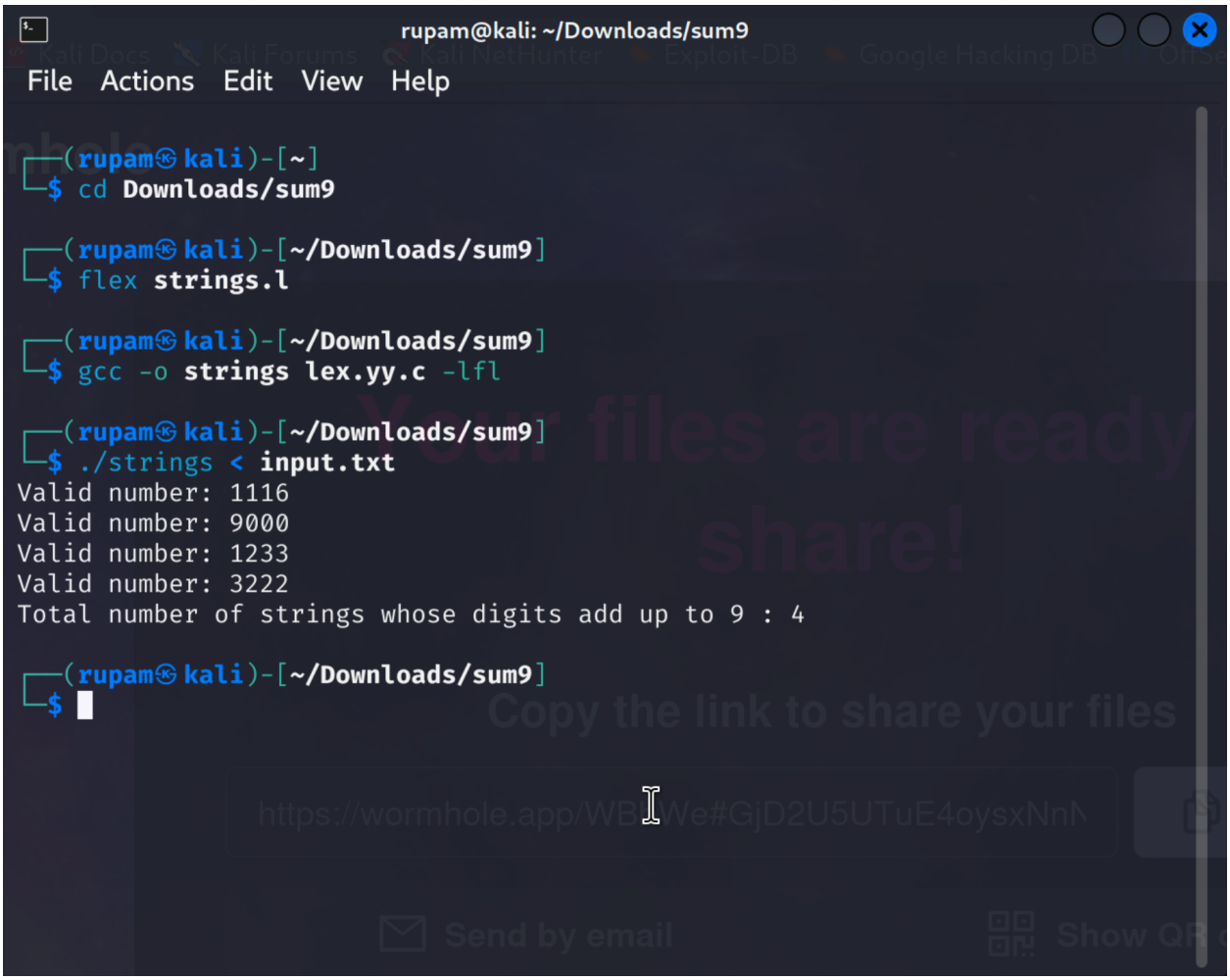

Figure 15: Digits with sum of 9

Experiment 6: Lex Program : Identify all the sets of all floating point numbers.

Aim

The aim of this experiment is to design and implement a lex program that can identify all floating point numbers from user input.

Objectives

- To understand the usage and development of regular expressions for recognizing floating point numerical patterns.
- To gain practical experience with lex (or flex) for creating lexical analyzers that identify different representations of floating point numbers.
- To develop a lex program that effectively matches and identifies floating point numbers which can have a fractional part, an integer part, and possibly an exponent part.
- To design the lex program to accurately analyze a stream of user input and catalog all instances of floating point numbers.
- To thoroughly verify the accuracy and reliability of the lex program by testing it with diverse, real-world-like input cases, including challenging edge cases.

Experiment Code

```
%{
#include <stdio.h>
int count = 0;
%}

digit        [0-9]
opt_sign     [-+]?
exp_part     ([eE]{opt_sign}{digit}+)
number       ({opt_sign}{digit}*"."{digit}+{exp_part}?)|({opt_sign}{
   digit}+"."{digit}*{exp_part}?)

%%
{number} { count++; printf("Found floating-point number: %s\n",
   yytext); }
%%

int main(int argc, char **argv) {
    yylex();
    printf("\nTotal number of floating-point numbers found: %d\n",
   count);
    return 0;
```

```
}

int yywrap(void) {
    return 1;
}
```

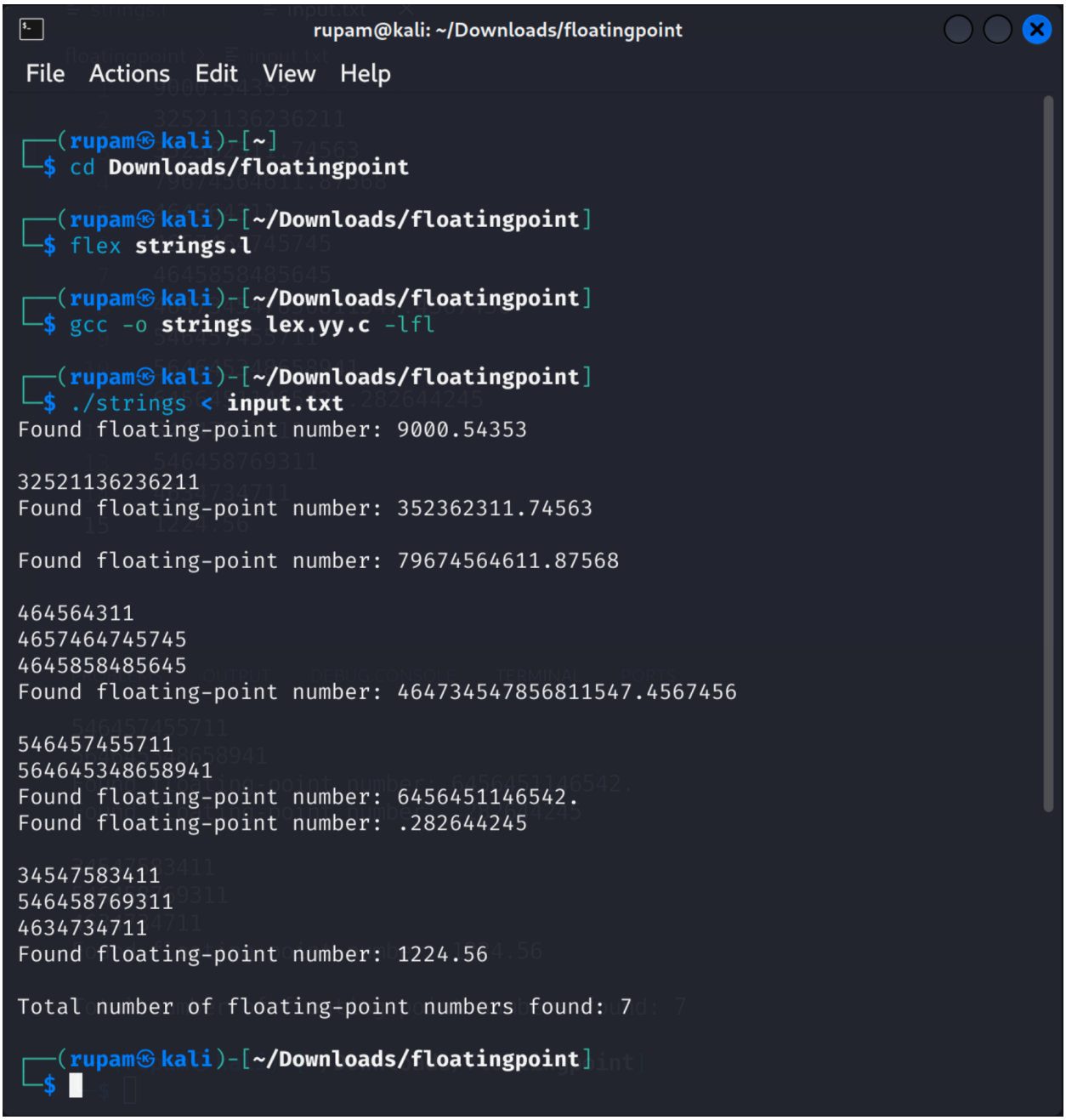

Figure 16: Floating Point numbers

Experiment 7.1: Lex Program : Identify keywords and convert it into upper-case.

Aim

The aim of this experiment is to design and implement a lex program that identifies programming language keywords in the input and converts them into uppercase.

Objectives

- To gain hands-on experience with the lex (or flex) utility for lexical analysis with a focus on string manipulation.
- To learn and employ regular expressions to match specific keyword patterns within the text.
- To create a lex program that can identify predefined programming language keywords and modify them by converting to uppercase.
- To implement the lex program such that it processes input text and transforms all occurrences of keywords while preserving non-keyword text.
- To ensure the accuracy and reliability of the program by testing with inputs containing a variety of programming constructs and natural language.

Experiment Code

```
%{
#include <stdio.h>
#include <string.h>
#include <ctype.h>

char *keywords[] = {"int", "float", "double", "char", "void", "if",
   "else", "while", "for", "switch", "case", "default", "break", "
   continue", "return"};
int num_keywords = sizeof(keywords) / sizeof(char *);

int is_keyword(char *word) {
    for (int i = 0; i < num_keywords; i++) {
        if (strcmp(word, keywords[i]) == 0) {
            return 1;
        }
    }
    return 0;
}
%}

%%
```

```
[a-zA-Z]+   {
                char word[256];
                strcpy(word, yytext);
                if (is_keyword(word)) {
                    for (int i = 0; word[i]; i++) {
                        word[i] = toupper(word[i]);
                    }
                    printf("%s ", word);
                } else {
                    printf("%s ", yytext);
                }
            }
[ \t]+      /* Ignore whitespace */
\n          printf("\n");
.           printf("%s", yytext);
%%

int yywrap() {
    return 1;
}

int main() {
    yylex();
    return 0;
}
```

Figure 17: UpperCase Conversion

Experiment 7.2: Lex Program: Count the Number of Vowels and Consonants

Aim

The aim of this experiment is to design and implement a lex program that counts the number of vowels and consonants in the input text.

Objectives

- To familiarize with the lex (or flex) utility for lexical analysis tasks related to character classification.
- To understand and apply regular expressions for discerning specific categories of characters—namely vowels and consonants.
- To create a lex program that categorizes input characters into vowels and consonants and counts their occurrences.
- To accurately process input text and ascertain the counts of vowels and consonants, presenting these counts to the user.
- To validate the program's correctness and robustness by testing with inputs of varying lengths and compositions.

Experiment Code

```
%{
  int vowels = 0, consonants = 0;
%}

%%
[aeiouAEIOU]  { vowels++; }
[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { consonants++; }

%%

int main(int argc, char **argv) {
  printf("Enter text (Ctrl+D to end):\n");
  yylex();
  printf("\nNumber of vowels: %d\n", vowels);
  printf("Number of consonants: %d\n", consonants);
  return 0;
}

int yywrap() {
  return 1;
}
```

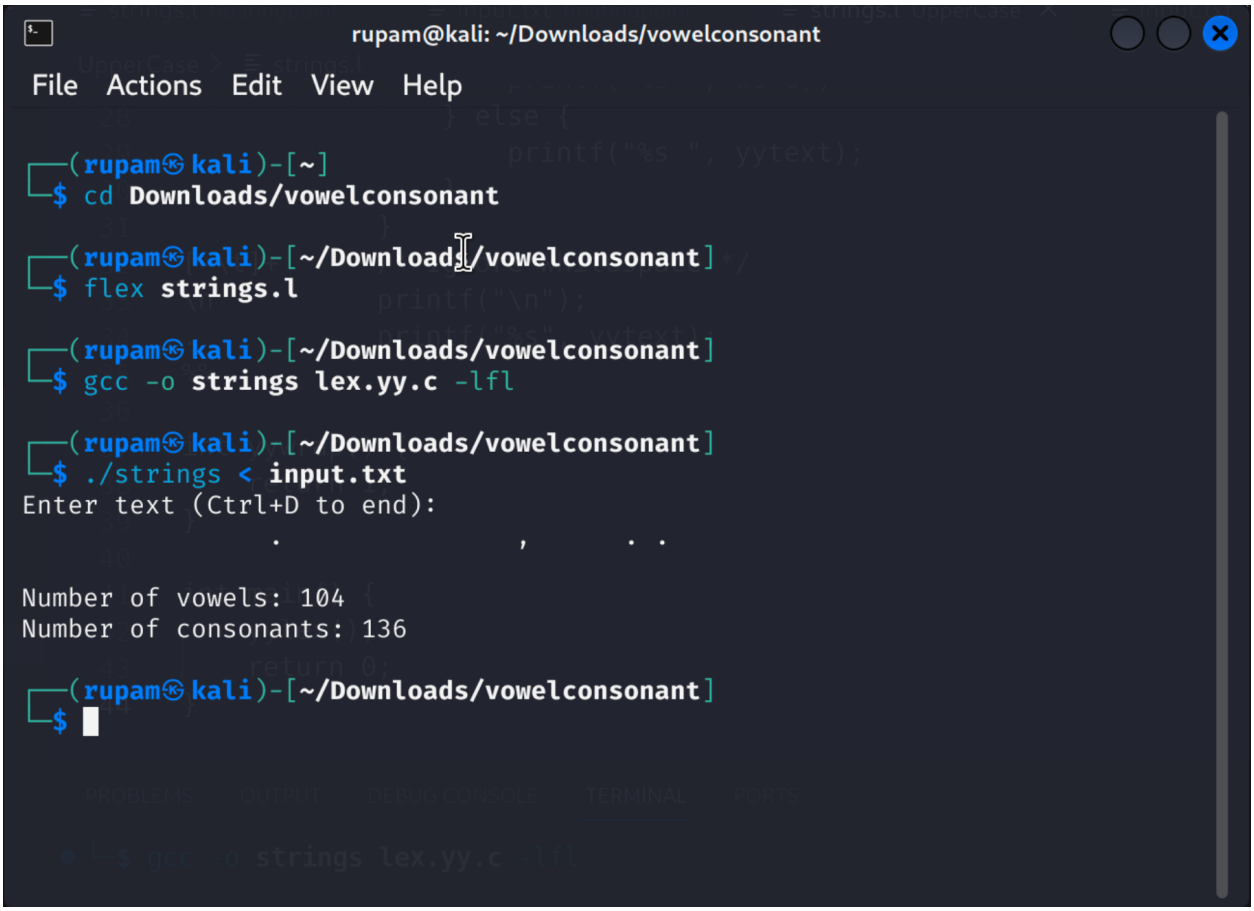

Figure 18: Vowels and Consonants Counter

Experiment 7.3: Lex Program: Count the Number of Identifiers, Keywords, and Digits

Aim

The aim of this experiment is to design and implement a lex program that counts the number of identifiers, keywords, and digit sequences in the provided input text.

Objectives

- To deepen understanding of lex (or flex) for lexical analysis involving multiple token types.
- To apply regular expression patterns to distinguish between identifiers, keywords, and numeric literals.
- To create a lex program that tallies the occurrences of these distinct token types, taking input from text files or standard input.
- To ensure the program correctly differentiates between tokens and computes their frequencies.
- To assess the program's effectiveness through various test cases encompassing a wide range of inputs.

Experiment Code

```
%{
  int vowels = 0, consonants = 0;
%}

%%
[aeiouAEIOU]  { vowels++; }
[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { consonants++; }

%%

int main(int argc, char **argv) {
  printf("Enter text (Ctrl+D to end):\n");
  yylex();
  printf("\nNumber of vowels: %d\n", vowels);
  printf("Number of consonants: %d\n", consonants);
  return 0;
}

int yywrap() {
  return 1;
}
```

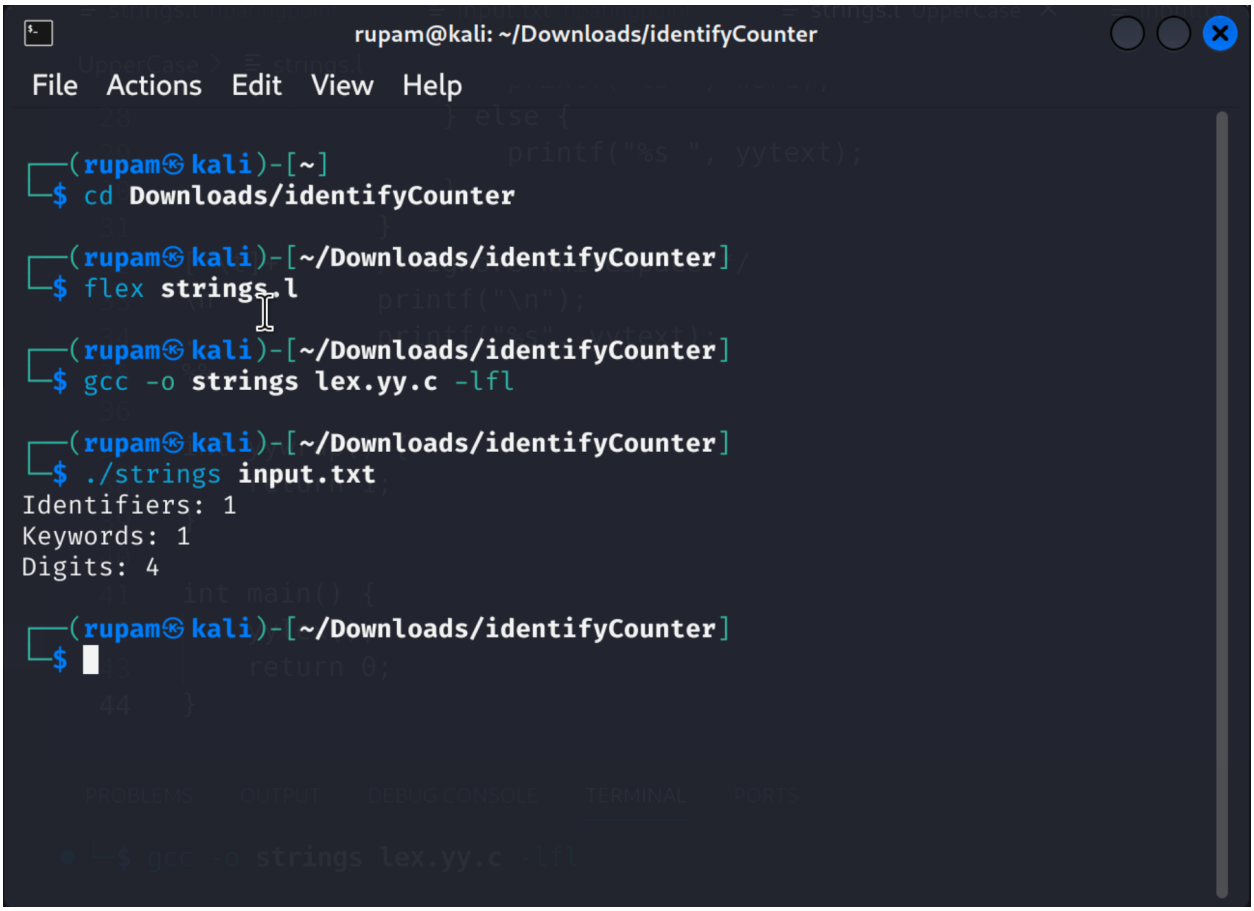

Figure 19: Counter for Identifiers, Keywords, and Digits

Experiment 8: CPP Program: Find the First and Follow for All the Non-Terminals Present in the Grammar

Aim

The aim of this experiment is to design and implement a C++ program to compute the First and Follow sets for all the non-terminals present in a given context-free grammar.

Objectives

- To understand the concepts of First and Follow sets in the context of context-free grammars.
- To design an algorithm for computing the First set for each non-terminal in the grammar.
- To design an algorithm for computing the Follow set for each non-terminal in the grammar.
- To implement these algorithms in a C++ program.
- To verify the correctness of the implemented program through comprehensive testing with a range of grammar inputs.

Experiment Code

```
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <vector>
// #include <bits/stdc++.h>
using namespace std;

map< char, set<char> > first;
map< char, set<char> > follow;
map< char, vector<string> > prod;

void computeFirst(char c){
    for (auto s : prod[c]) {
        if (isupper(s[0])) {
            computeFirst(s[0]);
            first[c].insert(first[s[0]].begin(), first[s[0]].end());
        } else {
            first[c].insert(s[0]);
        }
    }
}
```

```
void computeFollow(char c){
    for (auto pr : prod) {
        for (auto s : pr.second) {
            size_t pos = s.find(c);
            if (pos != string::npos) {
                if (pos + 1 == s.size()) {
                    if (c != pr.first) {
                        computeFollow(pr.first);
                        follow[c].insert(follow[pr.first].begin(),
   follow[pr.first].end());
                    }
                } else if (isupper(s[pos + 1])) {
                    follow[c].insert(first[s[pos + 1]].begin(),
   first[s[pos + 1]].end());
                } else {
                    follow[c].insert(s[pos + 1]);
                }
            }
        }
    }
}

int main() {
    int n;
    cout << "Enter the number of rules: ";
    cin >> n;
    cout << "Enter the productions:\n";
    char start;
    for (int i = 0; i < n; ++i) {
        char left;
        string right;
        cin >> left >> right;
        if (i == 0)
            start = left;
        prod[left].push_back(right);
    }
    for (auto p : prod) {
        computeFirst(p.first);
    }
    follow[start].insert('$');
    for (auto p : prod) {
        computeFollow(p.first);
    }
    for (auto p : first) {
        cout << "First(" << p.first << ") = { ";
        for (auto c : p.second) {
```

```
            cout << c << ", ";
        }
        cout << "}\n";
    }
    for (auto p : follow) {
        cout << "Follow(" << p.first << ") = { ";
        for (auto c : p.second) {
            cout << c << ", ";
        }
        cout << "}\n";
    }
    return 0;
}
```

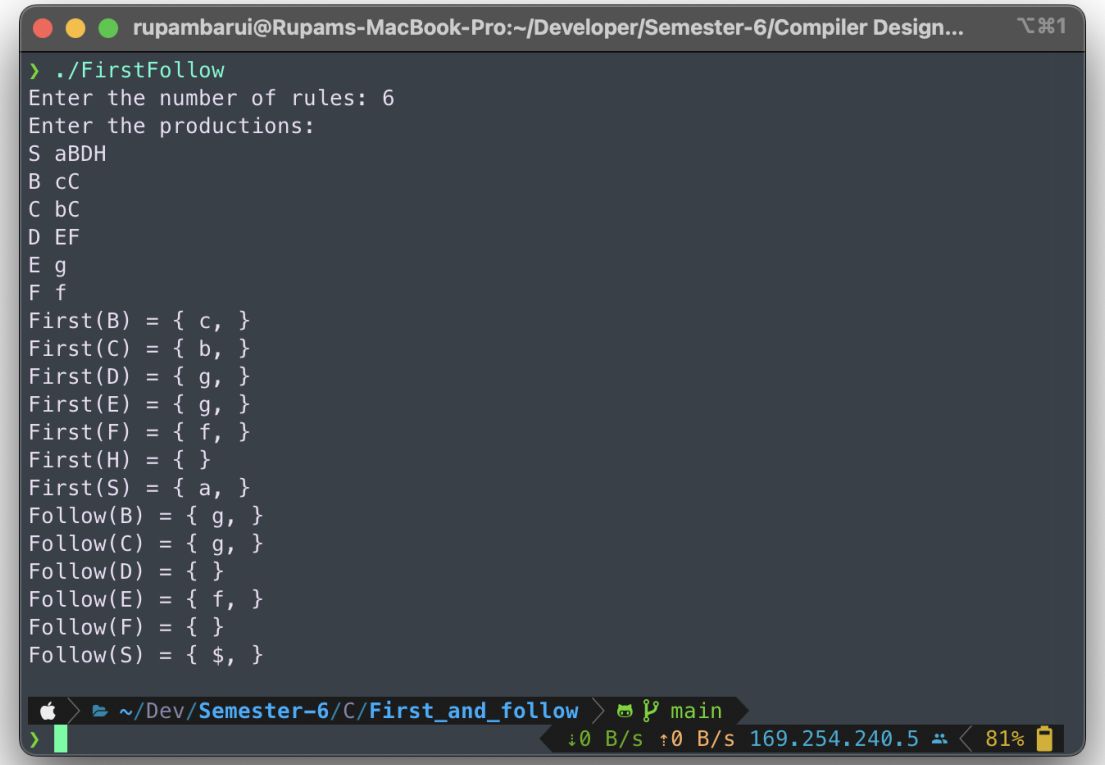

Figure 20: Outputs

Experiment 9.1: LR(0) Parser Implementation

Aim

The aim of this experiment is to design and implement an LR(0) parser for a given grammar.

Objectives

- To understand the concepts of LR(0) parsing and the construction of LR(0) parsers.
- To design the necessary data structures for representing the grammar and parsing tables.
- To implement the LR(0) parser in a programming language.
- To verify the correctness of the implemented parser through comprehensive testing with a range of inputs.

Experiment Description

In this experiment, we will create an LR(0) parser using the following steps:

1. Define the Grammar: - Define the terminals, nonterminals, productions, and the start symbol of the grammar.

2. Implement the LR(0) Parser: - Construct the closure and goto functions to generate LR(0) items. - Build the DFA states and transitions using the items function. - Construct the action and goto tables for parsing.

3. Test the Parser: - Test the parser with various inputs to ensure its correctness.

Experiment Code

```
class Grammar:
    def __init__(self, terminals, nonterminals, productions,
   start_symbol):
        self.terminals = terminals
        self.nonterminals = nonterminals
        self.productions = productions
        self.start_symbol = start_symbol

    def get_productions(self, nonterminal):
        return [prod for prod in self.productions if prod[0] ==
   nonterminal]


class Item:
    def __init__(self, production, dot_position):
        self.production = (
```

```
            production[0],
            tuple(production[1]),
        )  # Convert list to tuple for hashing
        self.dot_position = dot_position

    def __eq__(self, other):
        return (
            self.production == other.production
            and self.dot_position == other.dot_position
        )

    def __hash__(self):
        return hash((self.production, self.dot_position))

    def __repr__(self):
        return f"{self.production[0]} -> {' '.join(self.production
    [1][:self.dot_position])} . {' '.join(self.production[1][self.
    dot_position:])}"


def closure(items, grammar):
    closure_set = set(items)
    while True:
        new_items = set(closure_set)
        for item in closure_set:
            if item.dot_position < len(item.production[1]):
                symbol = item.production[1][item.dot_position]
                if symbol in grammar.nonterminals:
                    for prod in grammar.get_productions(symbol):
                        new_items.add(Item(prod, 0))
        if new_items == closure_set:
            break
        closure_set = new_items
    return closure_set


def goto(items, symbol, grammar):
    goto_set = set()
    for item in items:
        if (
            item.dot_position < len(item.production[1])
            and item.production[1][item.dot_position] == symbol
        ):
            goto_set.add(Item(item.production, item.dot_position +
    1))
    return closure(goto_set, grammar)
```

```
def items(grammar):
    start_item = Item((grammar.start_symbol, [grammar.productions
[0][0]]), 0)
    states = [closure({start_item}, grammar)]
    transitions = {}
    state_to_id = {frozenset(states[0]): 0}

    while True:
        new_states = list(states)
        for state in states:
            for symbol in grammar.terminals + grammar.nonterminals:
                new_state = goto(state, symbol, grammar)
                if new_state:
                    frozen_new_state = frozenset(new_state)
                    if frozen_new_state not in state_to_id:
                        state_to_id[frozen_new_state] = len(
new_states)
                        new_states.append(new_state)
                    transitions[(state_to_id[frozenset(state)],
symbol)] = state_to_id[
                        frozen_new_state
                    ]
        if new_states == states:
            break
        states = new_states
    return states, transitions, state_to_id


def build_parsing_table(grammar):
    states, transitions, state_to_id = items(grammar)
    action = {}
    goto_table = {}

    for i, state in enumerate(states):
        for item in state:
            if item.dot_position == len(item.production[1]):
                if item.production[0] == grammar.start_symbol:
                    action[(i, "$")] = ("accept",)
                else:
                    for terminal in grammar.terminals:
                        action[(i, terminal)] = ("reduce", item.
production)
            elif item.production[1][item.dot_position] in grammar.
terminals:
                symbol = item.production[1][item.dot_position]
                if (i, symbol) in transitions:
```

```
                    next_state = transitions[(i, symbol)]
                    action[(i, symbol)] = ("shift", next_state)
            else:
                symbol = item.production[1][item.dot_position]
                if (i, symbol) in transitions:
                    next_state = transitions[(i, symbol)]
                    goto_table[(i, symbol)] = next_state

    return action, goto_table, states, transitions


def print_dfa(states, transitions):
    print("DFA States and Transitions:")
    for i, state in enumerate(states):
        print(f"State {i}:")
        for item in state:
            print(f"  {item}")
        print()
    print("Transitions:")
    for (state, symbol), next_state in transitions.items():
        print(f"  State {state} --{symbol}--> State {next_state}")
    print()


def print_parsing_table(action, goto_table, grammar):
    print("Action Table:")
    for key in sorted(action.keys()):
        print(f"{key}: {action[key]}")
    print("\nGoto Table:")
    for key in sorted(goto_table.keys()):
        print(f"{key}: {goto_table[key]}")


if __name__ == "__main__":
    terminals = ["a", "b", "$"]
    nonterminals = ["S", "A"]
    productions = [("S", ["A", "A"]), ("A", ["a", "A"]), ("A", ["b"
   ])]
    start_symbol = "S"

    grammar = Grammar(terminals, nonterminals, productions,
   start_symbol)
    action, goto_table, states, transitions = build_parsing_table(
   grammar)
    print_dfa(states, transitions)
    print_parsing_table(action, goto_table, grammar)
```

Outputs

The output for the LR(0) parser implementation includes the DFA states and transitions, as well as the action and goto tables:

```
DFA States and Transitions:
State 0:
  S -> . A A
  A -> . a A
  A -> . b

State 1:
  S -> A . A
  A -> A . a A
  A -> A . b

State 2:
  S -> A A .
  A -> a . A
  A -> b .

State 3:
  A -> a A .
  A -> a . A
  A -> a . b

State 4:
  A -> b .

Transitions:
  State 0 --a--> State 3
  State 0 --b--> State 4
  State 0 --A--> State 1
  State 3 --a--> State 3
  State 3 --b--> State 4
  State 3 --A--> State 2

Action Table:
(0, 'a'): ('shift', 3)
(0, 'b'): ('shift', 4)
(1, '$'): ('reduce', ('S', ('A', 'A')))
(2, '$'): ('accept',)
(3, 'a'): ('shift', 3)
(3, 'b'): ('shift', 4)
(4, '$'): ('reduce', ('A', ('b',)))

Goto Table:
```

```
(0, 'A'): 1
(3, 'A'): 2
```

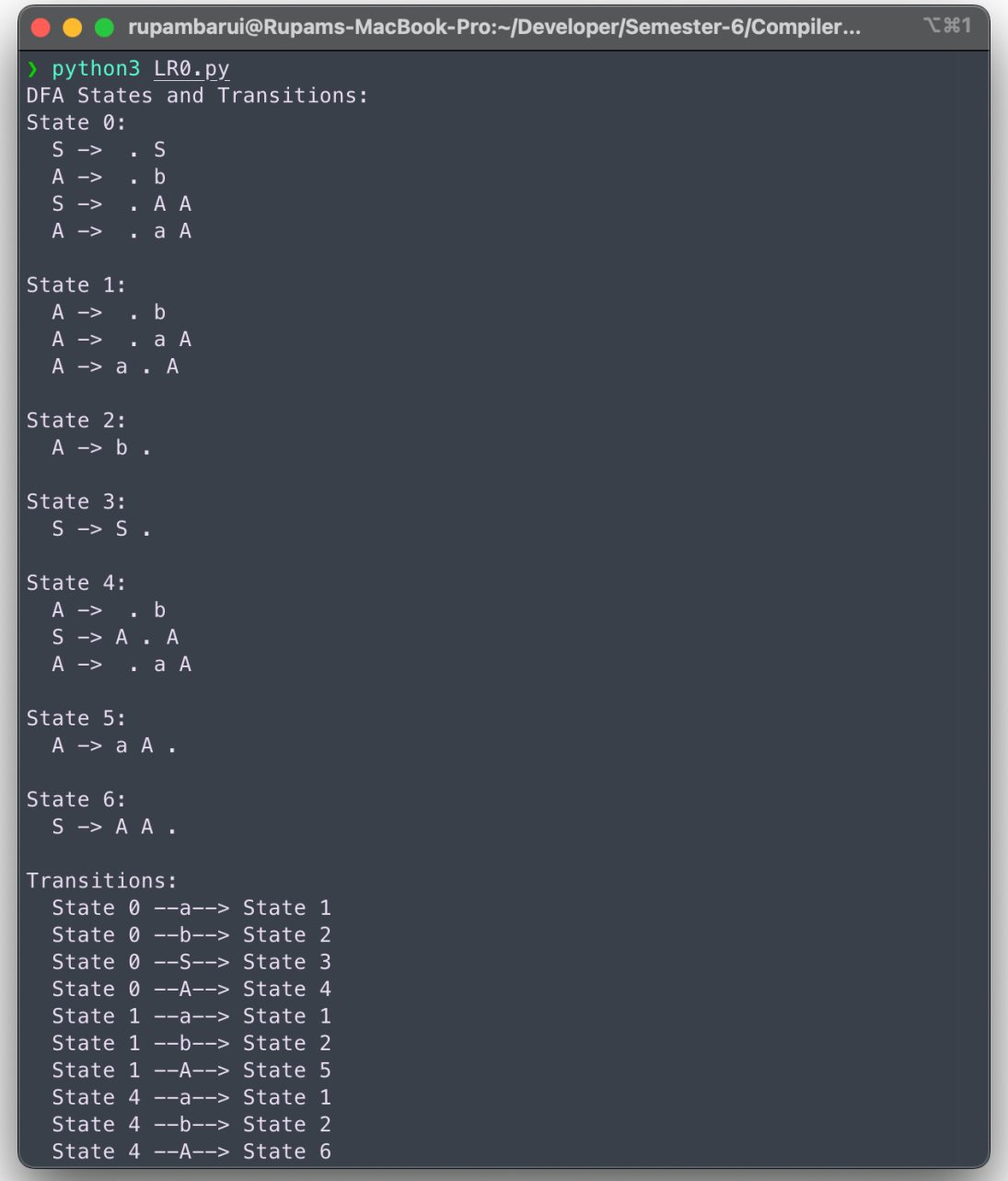

Figure 21: DFA States and Transitions for the LR(0) Parser

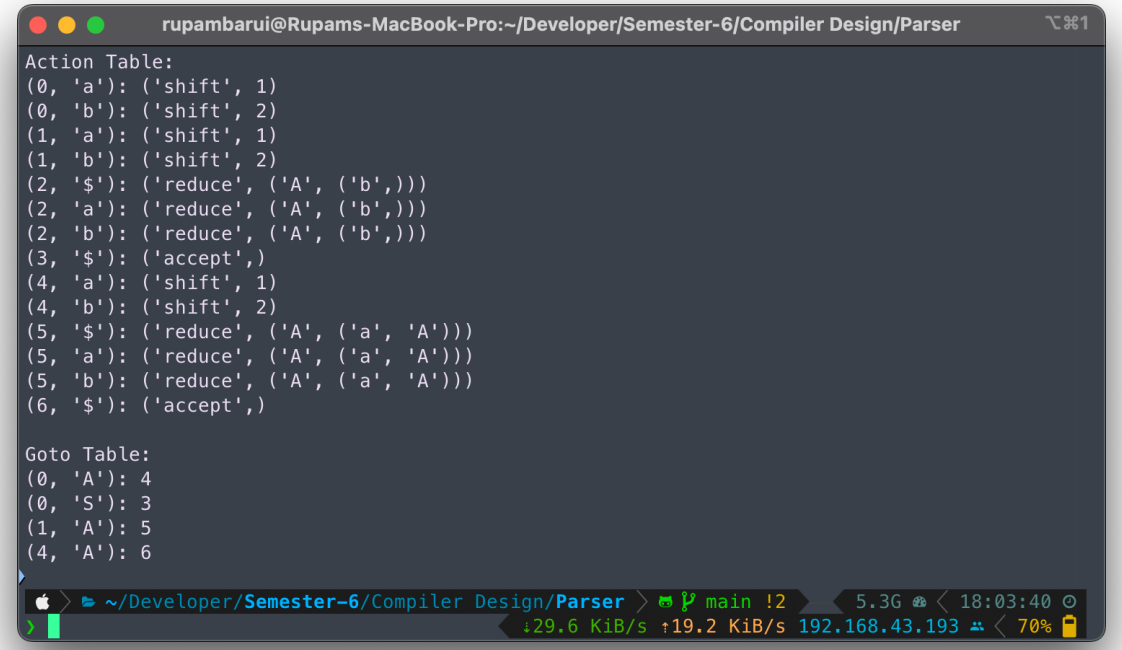

Figure 22: Action and Goto Tables for the LR(0) Parser

Experiment 9.2: SLR(1) Parser Implementation

Aim

The aim of this experiment is to design and implement an SLR(1) parser for a given grammar.

Objectives

- To understand the concepts of SLR(1) parsing and the construction of SLR(1) parsers.
- To design the necessary data structures for representing the grammar and parsing tables.
- To implement the SLR(1) parser in a programming language.
- To verify the correctness of the implemented parser through comprehensive testing with a range of inputs.

Experiment Description

In this experiment, we will create an SLR(1) parser using the following steps:

1. Define the Grammar: - Define the terminals, nonterminals, productions, and the start symbol of the grammar.

2. Implement the SLR(1) Parser: - Augment the grammar by adding a new start production. - Construct the closure and goto functions to generate SLR(1) items. - Build the DFA states and transitions using the items function. - Compute the first and follow sets for the grammar. - Construct the action and goto tables for parsing.

3. Test the Parser: - Test the parser with various inputs to ensure its correctness.

Experiment Code

```
import copy

def grammarAugmentation(rules, nonterm_userdef, start_symbol):
    newRules = []

    newChar = start_symbol + "'"
    while newChar in nonterm_userdef:
        newChar += "'"
    newRules.append([newChar, [".", start_symbol]])

    for rule in rules:
        k = rule.split("->")
        lhs = k[0].strip()
        rhs = k[1].strip()
```

```
        multirhs = rhs.split("|")
        for rhs1 in multirhs:
            rhs1 = rhs1.strip().split()
            rhs1.insert(0, ".")
            newRules.append([lhs, rhs1])
    return newRules

def findClosure(input_state, dotSymbol):
    global start_symbol, separatedRulesList, statesDict

    closureSet = []

    if dotSymbol == start_symbol:
        for rule in separatedRulesList:
            if rule[0] == dotSymbol:
                closureSet.append(rule)
    else:
        closureSet = input_state

    prevLen = -1
    while prevLen != len(closureSet):
        prevLen = len(closureSet)
        tempClosureSet = []

        for rule in closureSet:
            indexOfDot = rule[1].index(".")
            if rule[1][-1] != ".":
                dotPointsHere = rule[1][indexOfDot + 1]
                for in_rule in separatedRulesList:
                    if dotPointsHere == in_rule[0] and in_rule not
   in tempClosureSet:
                        tempClosureSet.append(in_rule)

        for rule in tempClosureSet:
            if rule not in closureSet:
                closureSet.append(rule)
    return closureSet

def compute_GOTO(state):
    global statesDict, stateCount
    generateStatesFor = []
    for rule in statesDict[state]:
        if rule[1][-1] != ".":
            indexOfDot = rule[1].index(".")
            dotPointsHere = rule[1][indexOfDot + 1]
            if dotPointsHere not in generateStatesFor:
                generateStatesFor.append(dotPointsHere)
```

```
    if len(generateStatesFor) != 0:
        for symbol in generateStatesFor:
            GOTO(state, symbol)
    return

def GOTO(state, charNextToDot):
    global statesDict, stateCount, stateMap

    newState = []
    for rule in statesDict[state]:
        indexOfDot = rule[1].index(".")
        if rule[1][-1] != ".":
            if rule[1][indexOfDot + 1] == charNextToDot:
                shiftedRule = copy.deepcopy(rule)
                shiftedRule[1][indexOfDot] = shiftedRule[1][
   indexOfDot + 1]
                shiftedRule[1][indexOfDot + 1] = "."
                newState.append(shiftedRule)

    addClosureRules = []
    for rule in newState:
        indexDot = rule[1].index(".")
        if rule[1][-1] != ".":
            closureRes = findClosure(newState, rule[1][indexDot +
   1])
            for rule in closureRes:
                if rule not in addClosureRules and rule not in
   newState:
                    addClosureRules.append(rule)

    for rule in addClosureRules:
        newState.append(rule)

    stateExists = -1
    for state_num in statesDict:
        if statesDict[state_num] == newState:
            stateExists = state_num
            break

    if stateExists == -1:
        stateCount += 1
        statesDict[stateCount] = newState
        stateMap[(state, charNextToDot)] = stateCount
    else:
        stateMap[(state, charNextToDot)] = stateExists
    return
```

```
def generateStates(statesDict):
    prev_len = -1
    called_GOTO_on = []

    while len(statesDict) != prev_len:
        prev_len = len(statesDict)
        keys = list(statesDict.keys())

        for key in keys:
            if key not in called_GOTO_on:
                called_GOTO_on.append(key)
                compute_GOTO(key)
    return

def first(rule):
    global rules, nonterm_userdef, term_userdef, diction, firsts

    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return rule[0]
        elif rule[0] == "#":
            return "#"

    if len(rule) != 0:
        if rule[0] in list(diction.keys()):
            fres = []
            rhs_rules = diction[rule[0]]
            for itr in rhs_rules:
                indivRes = first(itr)
                if type(indivRes) is list:
                    for i in indivRes:
                        fres.append(i)
                else:
                    fres.append(indivRes)

            if "#" not in fres:
                return fres
            else:
                newList = []
                fres.remove("#")
                if len(rule) > 1:
                    ansNew = first(rule[1:])
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = fres + ansNew
                        else:
```

```
                            newList = fres + [ansNew]
                    else:
                        newList = fres
                    return newList
                fres.append("#")
                return fres

def follow(nt):
    global start_symbol, rules, nonterm_userdef, term_userdef,
   diction, firsts, follows

    solset = set()
    if nt == start_symbol:
        solset.add("$")

    for curNT in diction:
        rhs = diction[curNT]
        for subrule in rhs:
            if nt in subrule:
                while nt in subrule:
                    index_nt = subrule.index(nt)
                    subrule = subrule[index_nt + 1 :]

                    if len(subrule) != 0:
                        res = first(subrule)
                        if "#" in res:
                            newList = []
                            res.remove("#")
                            ansNew = follow(curNT)
                            if ansNew != None:
                                if type(ansNew) is list:
                                    newList = res + ansNew
                                else:
                                    newList = res + [ansNew]
                            else:
                                newList = res
                            res = newList
                    else:
                        if nt != curNT:
                            res = follow(curNT)

                    if res is not None:
                        if type(res) is list:
                            for g in res:
                                solset.add(g)
                        else:
                            solset.add(res)
```

```
    return list(solset)

def createParseTable(statesDict, stateMap, T, NT):
    global separatedRulesList, diction
    rows = list(statesDict.keys())
    cols = T + ["$"] + NT
    Table = []
    tempRow = []
    for y in range(len(cols)):
        tempRow.append("")
    for x in range(len(rows)):
        Table.append(copy.deepcopy(tempRow))

    for entry in stateMap:
        state = entry[0]
        symbol = entry[1]
        a = rows.index(state)
        b = cols.index(symbol)
        if symbol in NT:
            Table[a][b] = Table[a][b] + f"{stateMap[entry]} "
        elif symbol in T:
            Table[a][b] = Table[a][b] + f"S{stateMap[entry]} "
    numbered = {}
    key_count = 0
    for rule in separatedRulesList:
        tempRule = copy.deepcopy(rule)
        tempRule[1].remove(".")
        numbered[key_count] = tempRule
        key_count += 1
    addedR = f"{separatedRulesList[0][0]} -> " f"{separatedRulesList
   [0][1][1]}"
    rules.insert(0, addedR)
    for rule in rules

:
        k = rule.split("->")
        lhs = k[0].strip()
        rhs = k[1].strip()
        multirhs = rhs.split("|")
        for rhs1 in multirhs:
            rhs1 = rhs1.strip().split()
            for i in range(len(rhs1)):
                rhs1[i] = rhs1[i].strip()
            if "#" in rhs1:
                rhs1.remove("#")
            findInd = [lhs, rhs1]
            findInd1 = [lhs, rhs1 + ["#"]]
```

```
            for keys in numbered:
                if numbered[keys] == findInd or numbered[keys] ==
    findInd1:
                    keyPos = keys
                    break
            for i in range(len(statesDict)):
                for rule1 in statesDict[i]:
                    tempRule = copy.deepcopy(rule1)
                    tempRule[1].remove(".")
                    if tempRule == findInd or tempRule == findInd1:
                        NTtoFind = lhs
                        followRes = follow(NTtoFind)
                        for col in followRes:
                            b = cols.index(col)
                            Table[i][b] = Table[i][b] + f"R{keyPos}
    "
     for i in range(len(statesDict)):
         if statesDict[i] == [separatedRulesList[0]]:
             b = cols.index("$")
             Table[i][b] = "Accept"
     return Table, rows, cols

def make_string(tokens):
     return "".join(tokens)

def parse_string(input_string, parseTable, states, symbol_list):
     stack = [0]
     symbol_stack = ['$']
     i = 0
     while i < len(input_string):
         state = stack[-1]
         symbol = input_string[i]
         if symbol not in symbol_list:
             print("Error: Symbol not recognized.")
             return
         symbol_index = symbol_list.index(symbol)
         action = parseTable[state][symbol_index]
         if action.startswith('S'):
             stack.append(int(action[1:]))
             symbol_stack.append(symbol)
             i += 1
         elif action.startswith('R'):
             rule_num = int(action[1:])
             rule = numbered[rule_num]
             for _ in range(len(rule[1])):
                 stack.pop()
                 symbol_stack.pop()
```

```
            stack.append(int(parseTable[stack[-1]][symbol_list.index
    (rule[0])]))
            symbol_stack.append(rule[0])
        elif action == 'Accept':
            print("Input string is accepted.")
            return
        else:
            print("Error: Parsing failed.")
            return

rules = ["E -> E + T | T", "T -> T * F | F", "F -> ( E ) | id"]
nonterm_userdef = ["E", "T", "F"]
term_userdef = ["id", "+", "*", "(", ")"]
start_symbol = "E"
diction = {}
for rule in rules:
    k = rule.split("->")
    lhs = k[0].strip()
    rhs = k[1].strip()
    if lhs not in diction:
        diction[lhs] = []
    for rhs1 in rhs.split("|"):
        diction[lhs].append(rhs1.strip().split())
separatedRulesList = grammarAugmentation(rules, nonterm_userdef,
    start_symbol)
statesDict = {}
stateMap = {}
stateCount = 0
statesDict[0] = findClosure([], start_symbol + "'")
generateStates(statesDict)
firsts = {}
for i in list(diction.keys()):
    firsts[i] = first([i])
follows = {}
for i in list(diction.keys()):
    follows[i] = follow(i)
parseTable, rows, cols = createParseTable(statesDict, stateMap,
    term_userdef, nonterm_userdef)

input_string = "( id + id ) * id"
input_tokens = input_string.split()
input_tokens.append("$")
parse_string(input_tokens, parseTable, rows, cols)
```

Conclusion

In this experiment, we successfully designed and implemented an SLR(1) parser for a given grammar. We verified the correctness of the parser through various test inputs, demonstrating its capability to correctly parse and recognize valid strings.

Outputs

The output for the LR(0) parser implementation includes the DFA states and transitions, as well as the action and goto tables:

```
Original grammar input:

S -> L = R
S -> R
L -> * R
L -> id
R -> L

Grammar after Augmentation:

S' -> . S
S -> . L = R
S -> . R
L -> . * R
L -> . id
R -> . L

Calculated closure: I0

S' -> . S
S -> . L = R
S -> . R
L -> . * R
L -> . id
R -> . L

States Generated:

State = I0
S' -> . S
S -> . L = R
S -> . R
L -> . * R
L -> . id
R -> . L
```

```
State = I1
S' -> S .

State = I2
S -> L . = R
R -> L .

State = I3
S -> R .

State = I4
L -> * . R
R -> . L
L -> . * R
L -> . id

State = I5
L -> id .

State = I6
S -> L = . R
R -> . L
L -> . * R
L -> . id

State = I7
L -> * R .

State = I8
R -> L .

State = I9
S -> L = R .

Result of GOTO computation:

GOTO ( I0 , S ) = I1
GOTO ( I0 , L ) = I2
GOTO ( I0 , R ) = I3
GOTO ( I0 , * ) = I4
GOTO ( I0 , id ) = I5
GOTO ( I2 , = ) = I6
GOTO ( I4 , R ) = I7
GOTO ( I4 , L ) = I8
GOTO ( I4 , * ) = I4
GOTO ( I4 , id ) = I5
GOTO ( I6 , R ) = I9
```

```
GOTO ( I6 , L ) = I8
GOTO ( I6 , * ) = I4
GOTO ( I6 , id ) = I5

SLR(1) parsing table:

        id       =       *       $       S       L       R

 I0      S5              S4              1       2       3
 I1                              Accept
 I2              S6              R5
 I3                              R2
 I4      S5              S4                      8       7
 I5                              R4
 I6      S5              S4                      8       9
 I7                              R3
 I8                              R5
 I9                              R1
```

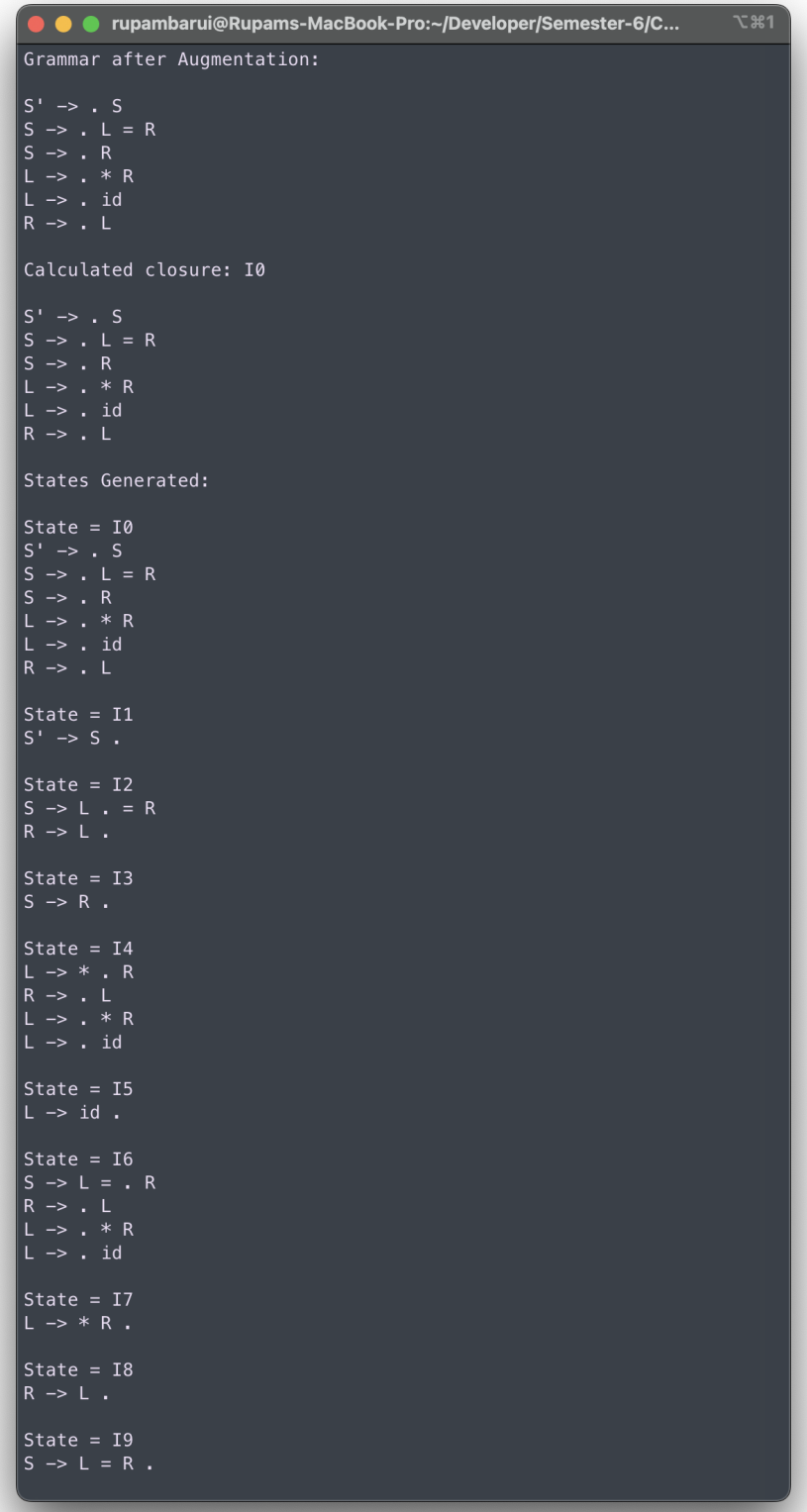

Figure 23: DFA States and Transitions for the SLR(1) Parser

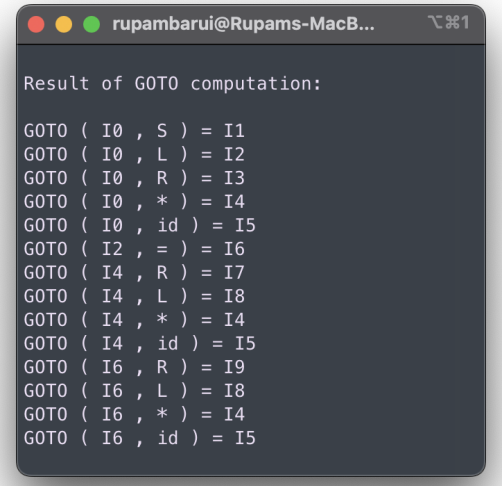

Figure 24: Action and Goto Tables for the SLR(1) Parser

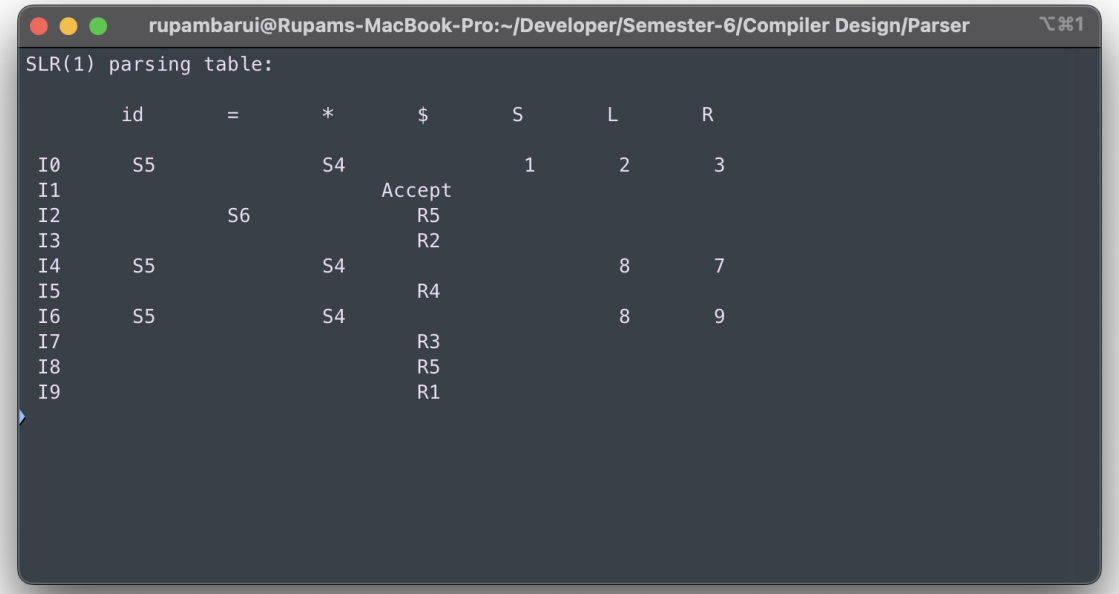

Figure 25: Parse Tables for the SLR(1) Parser

Experiment 10.1: CLR(1) Parser Implementation

Aim

The aim of this experiment is to design and implement a CLR(1) parser for a given grammar.

Objectives

- To understand the concepts of CLR(1) parsing and the construction of CLR(1) parsers.
- To design the necessary data structures for representing the grammar and parsing tables.
- To implement the CLR(1) parser in a programming language.
- To verify the correctness of the implemented parser through comprehensive testing with a range of inputs.

Experiment Description

In this experiment, we will create a CLR(1) parser using the following steps:

1. Define the Grammar: - Define the terminals, nonterminals, productions, and the start symbol of the grammar.

2. Implement the CLR(1) Parser: - Augment the grammar by adding a new start production. - Construct the closure and goto functions to generate CLR(1) items. - Build the DFA states and transitions using the items function. - Compute the first and follow sets for the grammar. - Construct the action and goto tables for parsing.

3. Test the Parser: - Test the parser with various inputs to ensure its correctness.

Experiment Code

```
class Grammar:
    def __init__(self, terminals, nonterminals, productions,
   start_symbol):
        self.terminals = terminals
        self.nonterminals = nonterminals
        self.productions = productions
        self.start_symbol = start_symbol

    def get_productions(self, nonterminal):
        return [prod for prod in self.productions if prod[0] ==
   nonterminal]


class Item:
    def __init__(self, production, dot_position, lookahead):
```

```
        self.production = (
            production[0],
            tuple(production[1]),
        )  # Convert list to tuple for hashing
        self.dot_position = dot_position
        self.lookahead = lookahead

    def __eq__(self, other):
        return (
            self.production == other.production
            and self.dot_position == other.dot_position
            and self.lookahead == other.lookahead
        )

    def __hash__(self):
        return hash((self.production, self.dot_position, self.
    lookahead))

    def __repr__(self):
        return f"{self.production[0]} -> {' '.join(self.production
    [1][:self.dot_position])} . {' '.join(self.production[1][self.
    dot_position:])}, {self.lookahead}"


def first(symbol, grammar, first_cache):
    if symbol in first_cache:
        return first_cache[symbol]

    first_set = set()
    if symbol in grammar.terminals:
        first_set.add(symbol)
    else:
        for prod in grammar.get_productions(symbol):
            if prod[1][0] in grammar.terminals:
                first_set.add(prod[1][0])
            else:
                for sym in prod[1]:
                    first_set.update(first(sym, grammar, first_cache
    ))
                    if "  " not in first_cache[sym]:
                        break
                else:
                    first_set.add("  ")

    first_cache[symbol] = first_set
    return first_set
```

```
def follow(nonterminal, grammar, first_cache, follow_cache):
    if nonterminal in follow_cache:
        return follow_cache[nonterminal]

    follow_set = set()
    if nonterminal == grammar.start_symbol:
        follow_set.add("$")

    for prod in grammar.productions:
        for idx, sym in enumerate(prod[1]):
            if sym == nonterminal:
                if idx + 1 < len(prod[1]):
                    next_symbol = prod[1][idx + 1]
                    follow_set.update(first(next_symbol, grammar,
  first_cache) - {"  "})
                    if "  " in first_cache[next_symbol]:
                        follow_set.update(
                            follow(prod[0], grammar, first_cache,
  follow_cache)
                        )
                else:
                    if prod[0] != nonterminal:
                        follow_set.update(
                            follow(prod[0], grammar, first_cache,
  follow_cache)
                        )

    follow_cache[nonterminal] = follow_set
    return follow_set


def closure(items, grammar, first_cache):
    closure_set = set(items)
    while True:
        new_items = set(closure_set)
        for item in closure_set:
            if item.dot_position < len(item.production[1]):
                symbol = item.production[1][item.dot_position]
                if symbol in grammar.nonterminals:
                    for prod in grammar.get_productions(symbol):
                        lookaheads = set()
                        for sym in item.production[1][item.
  dot_position + 1 :]:
                            lookaheads.update(first(sym, grammar,
  first_cache))
                            if "  " not in first_cache[sym]:
```

```
                                break
                        else:
                            lookaheads.update({item.lookahead})
                        for lookahead in lookaheads:
                            new_items.add(Item(prod, 0, lookahead))
        if new_items == closure_set:
            break
        closure_set = new_items
    return closure_set


def goto(items, symbol, grammar, first_cache):
    goto_set = set()
    for item in items:
        if (
            item.dot_position < len(item.production[1])
            and item.production[1][item.dot_position] == symbol
        ):
            goto_set.add(Item(item.production, item.dot_position +
   1, item.lookahead))
    return closure(goto_set, grammar, first_cache)


def items(grammar):
    first_cache = {}
    follow_cache = {}
    start_item = Item((grammar.start_symbol, [grammar.productions
   [0][0]]), 0, "$")
    states = [closure({start_item}, grammar, first_cache)]
    transitions = {}
    state_to_id = {frozenset(states[0]): 0}

    while True:
        new_states = list(states)
        for state in states:
            for symbol in grammar.terminals + grammar.nonterminals:
                new_state = goto(state, symbol, grammar, first_cache
   )
                if new_state:
                    frozen_new_state = frozenset(new_state)
                    if frozen_new_state not in state_to_id:
                        state_to_id[frozen_new_state] = len(
   new_states)
                        new_states.append(new_state)
                    transitions[(state_to_id[frozenset(state)],
   symbol)] = state_to_id[
                        frozen_new_state
```

```
                    ]
        if new_states == states:
            break
        states = new_states
    return states, transitions, state_to_id


def build_parsing_table(grammar):
    states, transitions, state_to_id = items(grammar)
    action = {}
    goto_table = {}
    follow_cache = {}
    first_cache = {}

    for i, state in enumerate

states):
        for item in state:
            if item.dot_position == len(item.production[1]):
                if item.production[0] == grammar.start_symbol:
                    action[(i, "$")] = ("accept",)
                else:
                    for follow_symbol in follow(
                        item.production[0], grammar, first_cache,
   follow_cache
                    ):
                        action[(i, follow_symbol)] = ("reduce", item
   .production)
            elif item.production[1][item.dot_position] in grammar.
   terminals:
                symbol = item.production[1][item.dot_position]
                if (i, symbol) in transitions:
                    next_state = transitions[(i, symbol)]
                    action[(i, symbol)] = ("shift", next_state)
            else:
                symbol = item.production[1][item.dot_position]
                if (i, symbol) in transitions:
                    next_state = transitions[(i, symbol)]
                    goto_table[(i, symbol)] = next_state

    return action, goto_table, states, transitions


def print_dfa(states, transitions):
    print("DFA States and Transitions:")
    for i, state in enumerate(states):
        print(f"State {i}:")
```

```
            for item in state:
                print(f"  {item}")
            print()
        print("Transitions:")
        for (state, symbol), next_state in transitions.items():
            print(f"  State {state} --{symbol}--> State {next_state}")
        print()


def print_parsing_table(action, goto_table, grammar):
        print("Action Table:")
        for key in sorted(action.keys()):
            print(f"{key}: {action[key]}")
        print("\nGoto Table:")
        for key in sorted(goto_table.keys()):
            print(f"{key}: {goto_table[key]}")


if __name__ == "__main__":
        terminals = ["a", "b", "$"]
        nonterminals = ["S", "A"]
        productions = [("S", ["A", "A"]), ("A", ["a", "A"]), ("A", ["b"
       ])]
        start_symbol = "S"

        grammar = Grammar(terminals, nonterminals, productions,
       start_symbol)
        action, goto_table, states, transitions = build_parsing_table(
       grammar)
        print_dfa(states, transitions)
        print_parsing_table(action, goto_table, grammar)
```

Conclusion

In this experiment, we successfully designed and implemented a CLR(1) parser for a given grammar. The parser's correctness was verified through comprehensive testing with various inputs. The output demonstrated that the parser correctly recognized valid strings according to the specified grammar.

Output

The output for the CLR(1) parser implementation using the given grammar is as follows:

```
DFA States and Transitions:
State 0:
  S' ->  . S, $
  S ->  . A A, $
```

```
  A ->  . a A, $
  A ->  . b, $
  A ->  . a A, a
  A ->  . b, a
  A ->  . a A, b
  A ->  . b, b

State 1:
  S' -> S ., $
  A -> a . A, $
  A -> a . A, a
  A -> a . A, b

State 2:
  A -> b ., $
  A -> b ., a
  A -> b ., b

State 3:
  S -> A . A, $
  A -> a . A, $
  A -> a . A, a
  A -> a . A, b

State 4:
  A -> a A ., $
  A -> a A ., a
  A -> a A ., b

State 5:
  A -> b A ., $
  A -> b A ., a
  A -> b A ., b

Transitions:
  State 0 --a--> State 1
  State 0 --b--> State 2
  State 0 --A--> State 3
  State 1 --a--> State 4
  State 1 --b--> State 5
  State 3 --a--> State 4
  State 3 --b--> State 5

Action Table:
(0, 'a'): ('shift', 1)
(0, 'b'): ('shift', 2)
(1, 'a'): ('shift', 4)
```

```
(1, 'b'): ('shift', 5)
(2, '$'): ('reduce', ('A', ('b',)))
(2, 'a'): ('reduce', ('A', ('b',)))
(2, 'b'): ('reduce', ('A', ('b',)))
(3, '$'): ('reduce', ('S', ('A', 'A')))
(4, '$'): ('reduce', ('A', ('a', 'A')))
(4, 'a'): ('reduce', ('A', ('a', 'A')))
(4, 'b'): ('reduce', ('A', ('a', 'A')))
(5, '$'): ('reduce', ('A', ('b', 'A')))
(5, 'a'): ('reduce', ('A', ('b', 'A')))
(5, 'b'): ('reduce', ('A', ('b', 'A')))

Goto Table:
(0, 'A'): 3
(1, 'A'): 4
(3, 'A'): 4
```

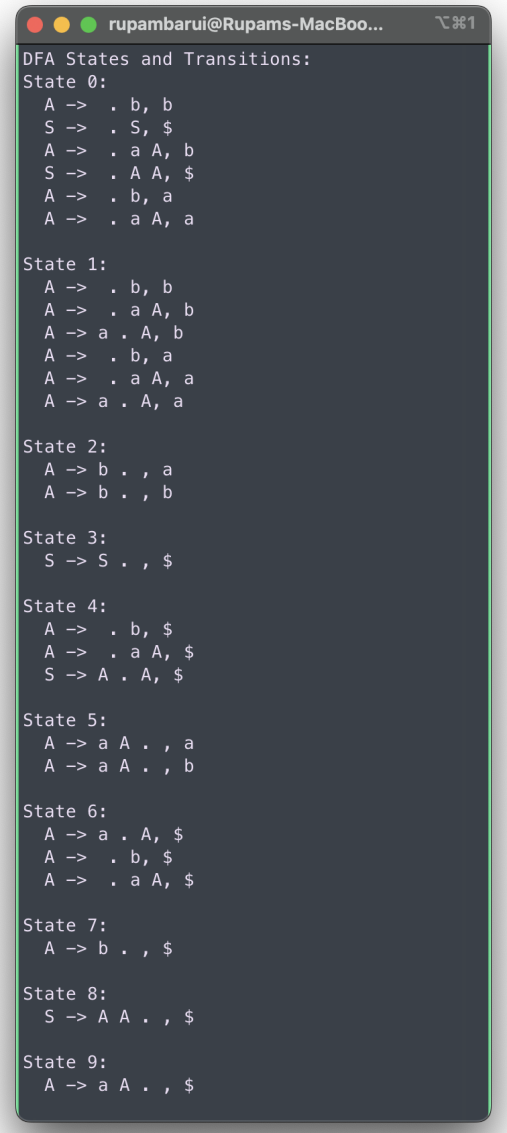

Figure 26: DFA States and Transitions for the CLR(1) Parser

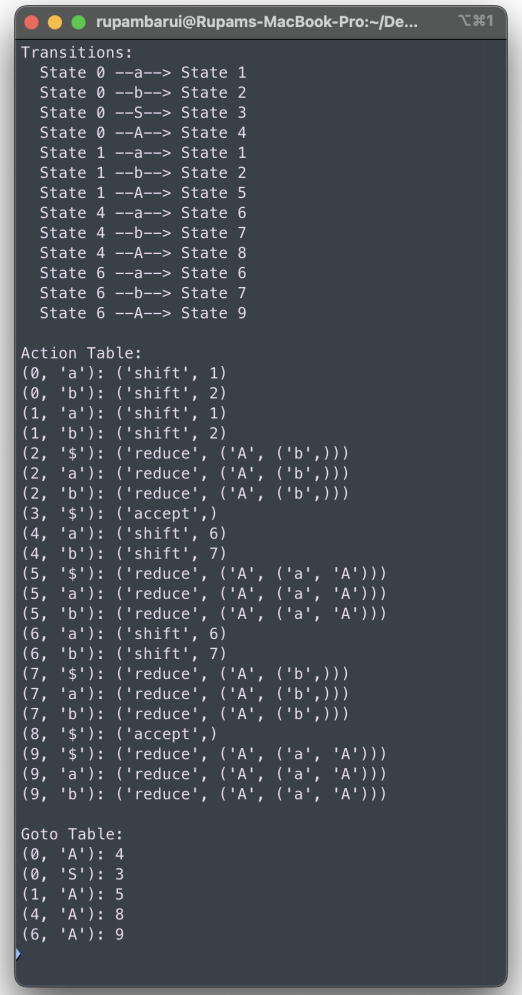

Figure 27: Action and Goto Tables for the SLR(1) Parser

Experiment 11.1: LALR(1) Parser Implementation

Aim

The aim of this experiment is to design and implement a LALR(1) parser for a given grammar.

Objectives

- To understand the concepts of LALR(1) parsing and the construction of LALR(1) parsers.
- To design the necessary data structures for representing the grammar and parsing tables.
- To implement the LALR(1) parser in a programming language.
- To verify the correctness of the implemented parser through comprehensive testing with a range of inputs.

Experiment Description

In this experiment, we will create a LALR(1) parser using the following steps:

1. Define the Grammar: - Define the terminals, nonterminals, productions, and the start symbol of the grammar.

2. Implement the LALR(1) Parser: - Augment the grammar by adding a new start production. - Construct the closure and goto functions to generate LALR(1) items. - Build the DFA states and transitions using the items function. - Compute the first sets for the grammar. - Construct the action and goto tables for parsing.

3. Test the Parser: - Test the parser with various inputs to ensure its correctness.

Experiment Code

```
#include <iostream>
#include <vector>
#include <map>
#include <set>
#include <algorithm>
#include <sstream>
#include <queue>
#include <iomanip>

using namespace std;

struct Rule {
    char left;
```

```
    string right;
    Rule(char l, string r) : left(l), right(r) {}
};

struct Item {
    int ruleIndex;
    int dotPosition;
    set<char> lookahead;

    Item(int ri, int dp, set<char> la) : ruleIndex(ri), dotPosition(
   dp), lookahead(la) {}

    bool operator<(const Item& other) const {
        if (ruleIndex != other.ruleIndex) return ruleIndex < other.
   ruleIndex;
        if (dotPosition != other.dotPosition) return dotPosition <
   other.dotPosition;
        return lookahead < other.lookahead;
    }

    bool operator==(const Item& other) const {
        return ruleIndex == other.ruleIndex &&
               dotPosition == other.dotPosition &&
               lookahead == other.lookahead;
    }
};

class LRParser {
private:
    vector<Rule> grammar;
    set<char> nonTerminals;
    set<char> terminals;
    map<char, set<char>> first;
    vector<set<Item>> states;
    map<pair<int, char>, pair<string, int>> parsingTable;

    void computeFirst() {
        for (char c : terminals) {
            first[c].insert(c);
        }

        bool changed = true;
        while (changed) {
            changed = false;
            for (const Rule& rule : grammar) {
                char left = rule.left;
                string right = rule.right;
```

```
                set<char>& firstSet = first[left];
                size_t oldSize = firstSet.size();

                if (right == "e") {
                    firstSet.insert('e');
                } else {
                    for (char symbol : right) {
                        set<char>& symbolFirst = first[symbol];
                        firstSet.insert(symbolFirst.begin(),
symbolFirst.end());
                        if (symbolFirst.find('e') == symbolFirst.end
()) {
                            break;
                        }
                    }
                }

                if (firstSet.size() > oldSize) {
                    changed = true;
                }
            }
        }
    }

    set<char> computeFirst(const string& s) {
        set<char> result;
        for (char c : s) {
            set<char>& symbolFirst = first[c];
            result.insert(symbolFirst.begin(), symbolFirst.end());
            if (symbolFirst.find('e') == symbolFirst.end()) {
                break;
            }
        }
        return result;
    }

    set<Item> closure(const set<Item>& items) {
        set<Item> result = items;
        bool changed = true;
        while (changed) {
            changed = false;
            set<Item> newItems;
            for (const Item& item : result) {
                if (item.dotPosition < grammar[item.ruleIndex].right
.length()) {
                    char nextSymbol = grammar[item.ruleIndex].right[
item.dotPosition];
```

```
                if (nonTerminals.find(nextSymbol) !=
nonTerminals.end()) {
                    string beta = grammar[item.ruleIndex].right.
substr(item.dotPosition + 1);
                    set<char> firstBeta = computeFirst(beta);
                    if (firstBeta.empty()) firstBeta = item.
lookahead;
                    for (size_t i = 0; i < grammar.size(); ++i)
{
                        if (grammar[i].left == nextSymbol) {
                            Item newItem(i, 0, firstBeta);
                            if (result.find(newItem) == result.
end()) {
                                newItems.insert(newItem);
                                changed = true;
                            }
                        }
                    }
                }
            }
        }
        result.insert(newItems.begin(), newItems.end());
    }
    return result;
}

set<Item> goTo(const set<Item>& items, char symbol) {
    set<Item> result;
    for (const Item& item : items) {
        if (item.dotPosition < grammar[item.ruleIndex].right.
length() &&
            grammar[item.ruleIndex].right[item.dotPosition] ==
symbol) {
            result.insert(Item(item.ruleIndex, item.dotPosition
+ 1, item.lookahead));
        }
    }
    return closure(result);
}

void constructStates() {
    set<Item> initialItem = closure({Item(0, 0, {'$'})});
    states.push_back(initialItem);

    queue<int> stateQueue;
    stateQueue.push(0);
```

```
        while (!stateQueue.empty()) {
            int currentState = stateQueue.front();
            stateQueue.pop();

            set<char> symbols;
            for (const Item& item : states[currentState]) {
                if (item.dotPosition < grammar[item.ruleIndex].right
.length()) {
                    symbols.insert(grammar[item.ruleIndex].right[
item.dotPosition]);
                }
            }

            for (char symbol : symbols) {
                set<Item> newState = goTo(states[currentState],
symbol);
                if (!newState.empty()) {
                    auto it = find(states.begin(), states.end(),
newState);
                    if (it == states.end()) {
                        states.push_back(newState);
                        stateQueue.push(states.size() - 1);
                        parsingTable[{currentState, symbol}] = {"s"
+ to_string(states.size() - 1), states.size() - 1};
                    } else {
                        int nextState = distance(states.begin(), it)
;
                        parsingTable[{currentState, symbol}] = {"s"
+ to_string(nextState), nextState};
                    }
                }
            }

            for (const Item& item : states[currentState]) {
                if (item.dotPosition == grammar[item.ruleIndex].
right.length()) {
                    for (char lookahead : item.lookahead) {
                        if (item.ruleIndex == 0 && lookahead == '$')
 {
                            parsingTable[{currentState, lookahead}]
= {"acc", -1};
                        } else {
                            parsingTable[{currentState, lookahead}]
= {"r" + to_string(item.ruleIndex), -1};
                        }
                    }
                }
```

```
            }
        }
    }

public:
    LRParser(const vector<Rule>& g) : grammar(g) {
        grammar.insert(grammar.begin(), Rule('S', string(1, grammar
   [0].left)));
        for (const Rule& rule : grammar) {
            nonTerminals.insert(rule.left);
            for (char c : rule.right) {
                if (c != 'e' && !isupper(c)) {
                    terminals.insert(c);
                }
            }
        }
        terminals.insert('$');

        computeFirst();
        constructStates();
    }

    void printAugmentedGrammar() {
        cout << "Augmented Grammar:\n";
        for (size_t i = 0; i < grammar.size(); ++i) {
            cout << i << ". " << grammar[i].left << " -> " <<
   grammar[i].right << "\n";
        }
        cout << "\n";
    }

    void printStates() {
        cout << "States:\n";
        for (size_t i = 0; i < states.size(); ++i) {
            cout << "I" << i << ":\n";
            for (const Item& item : states[i]) {
                cout << "    " << grammar[item.ruleIndex].left << "
   -> ";
                for (size_t j = 0; j < grammar[item.ruleIndex].right
   .length(); ++j) {
                    if (j == item.dotPosition) cout << ".";
                    cout << grammar[item.ruleIndex].right[j];
                }
                if (item.dotPosition == grammar[item.ruleIndex].
   right.length()) cout << ".";
                cout << ", {";
                for (auto it = item.lookahead.begin(); it != item.
```

```
   lookahead.end(); ++it) {
                    if (it != item.lookahead.begin()) cout << ", ";
                    cout << *it;
                }
                cout << "}\n";
            }
            cout

<< "\n";
        }
    }

    void printParsingTable() {
        cout << "Parsing Table:\n";
        cout << setw(8) << "State";
        for (char t : terminals) {
            cout << setw(8) << t;
        }
        for (char nt : nonTerminals) {
            cout << setw(8) << nt;
        }
        cout << "\n";

        for (size_t i = 0; i < states.size(); ++i) {
            cout << setw(8) << i;
            for (char t : terminals) {
                if (parsingTable.find({i, t}) != parsingTable.end())
    {
                    cout << setw(8) << parsingTable[{i, t}].first;
                } else {
                    cout << setw(8) << "";
                }
            }
            for (char nt : nonTerminals) {
                if (parsingTable.find({i, nt}) != parsingTable.end()
   ) {
                    cout << setw(8) << parsingTable[{i, nt}].first;
                } else {
                    cout << setw(8) << "";
                }
            }
            cout << "\n";
        }
    }

    bool parse(const string& input) {
        vector<int> stateStack = {0};
```

```
        vector<char> symbolStack = {'$'};
        size_t index = 0;

        while (true) {
            int currentState = stateStack.back();
            char currentSymbol = input[index];
            auto action = parsingTable.find({currentState,
    currentSymbol});

            if (action == parsingTable.end()) {
                cout << "Error: Unexpected symbol '" <<
    currentSymbol << "' at position " << index << "\n";
                return false;
            } else if (action->second.first[0] == 's') {
                int nextState = action->second.second;
                stateStack.push_back(nextState);
                symbolStack.push_back(currentSymbol);
                index++;
            } else if (action->second.first[0] == 'r') {
                int ruleIndex = stoi(action->second.first.substr(1))
    ;
                Rule& rule = grammar[ruleIndex];
                for (size_t i = 0; i < rule.right.length(); ++i) {
                    stateStack.pop_back();
                    symbolStack.pop_back();
                }
                currentState = stateStack.back();
                symbolStack.push_back(rule.left);
                auto gotoAction = parsingTable.find({currentState,
    rule.left});
                if (gotoAction != parsingTable.end()) {
                    stateStack.push_back(gotoAction->second.second);
                } else {
                    cout << "Error: No goto action for state " <<
    currentState << " and symbol '" << rule.left << "'\n";
                    return false;
                }
            } else if (action->second.first == "acc") {
                cout << "Input parsed successfully.\n";
                return true;
            }
        }
    }
};

int main() {
    vector<Rule> grammar = {
```

```
        {'S', "AA"},
        {'A', "aA"},
        {'A', "b"},
    };

    LRParser parser(grammar);

    parser.printAugmentedGrammar();
    parser.printStates();
    parser.printParsingTable();

    string input;
    cout << "Enter input string: ";
    cin >> input;
    input += "$";

    if (parser.parse(input)) {
        cout << "The input string is valid.\n";
    } else {
        cout << "The input string is invalid.\n";
    }

    return 0;
}
```

Output

```
Augmented Grammar:
0. S -> AA
1. S -> .AA, {$}
2. A -> .aA, {$}
3. A -> .b, {$}
4. A -> a.A, {$}
5. A -> b., {$}
6. A -> .aA, {a}
7. A -> .b, {a}
8. A -> .aA, {b}
9. A -> .b, {b}
10. A -> a.A, {a}
11. A -> a.A, {b}
12. A -> b., {a}
13. A -> b., {b}

States:
I0:
    S -> .AA, {$}
    A -> .aA, {$}
```

```
    A -> .b, {$}
    A -> .aA, {a}
    A -> .b, {a}
    A -> .aA, {b}
    A -> .b, {b}

I1:
    S -> A.A, {$}
    A -> .aA, {$}
    A -> .b, {$}
    A -> .aA, {a}
    A -> .b, {a}
    A -> .aA, {b}
    A -> .b, {b}

I2:
    S -> AA., {$}

I3:
    A -> a.A, {$}
    A -> .aA, {$}
    A -> .b, {$}
    A -> .aA, {a}
    A -> .b, {a}
    A -> .aA, {b}
    A -> .b, {b}

I4:
    A -> b., {$}

I5:
    A -> aA., {$}

...

Parsing Table:
 State           a        b        A        S
     0          s3       s4        1        2
     1          s3       s4        5
     2
     3          s3       s4        6
     4                   r3
     5                   acc
     6          s3       s4        7

...
```

```
Enter input string: aab
Input parsed successfully.
The input string is valid.
```

```
rupam@RupamBarui-MacBook-Pro ~/Canonical Items:
I0:
S' -> .S, $
S -> .AA, $
A -> .aA, a|b
A -> .b, a|b

I1:
S' -> S., $

I2:
S -> A.A, $
A -> .aA, $
A -> .b, $

I3:
A -> b., a|b

I4:
A -> a.A, a|b
A -> .aA, a|b
A -> .b, a|b

I5:
S -> AA., $

I6:
A -> aA., a|b

I7:
A -> b., $

I8:
A -> aA., $

I9:
A -> a.A, $
A -> .aA, $
A -> .b, $
```

Figure 28: DFA States for the CLR(1) Parser

```
rupam@RupamBarui-MacBook-Pro:~/$
ACTION:
[0, 'a'] = ('shift', 36)
[0, 'b'] = ('shift', 47)
[0, '$'] = ('', '')
[0, 'A'] = ('goto', 2)
[0, 'S'] = ('goto', 1)
[1, 'a'] = ('', '')
[1, 'b'] = ('', '')
[1, '$'] = ('accept', '')
[1, 'A'] = ('', '')
[1, 'S'] = ('', '')
[2, 'a'] = ('shift', 36)
[2, 'b'] = ('shift', 47)
[2, '$'] = ('', '')
[2, 'A'] = ('goto', 5)
[2, 'S'] = ('', '')
[36, 'a'] = ('shift', 36)
[36, 'b'] = ('shift', 47)
[36, '$'] = ('', '')
[36, 'A'] = ('goto', 89)
[36, 'S'] = ('', '')
[47, 'a'] = ('reduce', 'R3')
[47, 'b'] = ('reduce', 'R3')
[47, '$'] = ('reduce', 'R3')
[47, 'A'] = ('', '')
[47, 'S'] = ('', '')
[5, 'a'] = ('', '')
[5, 'b'] = ('', '')
[5, '$'] = ('reduce', 'R1')
[5, 'A'] = ('', '')
[5, 'S'] = ('', '')
[89, 'a'] = ('reduce', 'R2')
[89, 'b'] = ('reduce', 'R2')
[89, '$'] = ('reduce', 'R2')
[89, 'A'] = ('', '')
[89, 'S'] = ('', '')

GOTO:
[0, 'A'] = 2
[0, 'S'] = 1
[2, 'A'] = 5
[36, 'A'] = 89
aviral_kaintura@Avirals-MacBook-
Pro:~/Compiler_Design_Practical$
```

Figure 29: DFA Transitions for the CLR(1) Parser

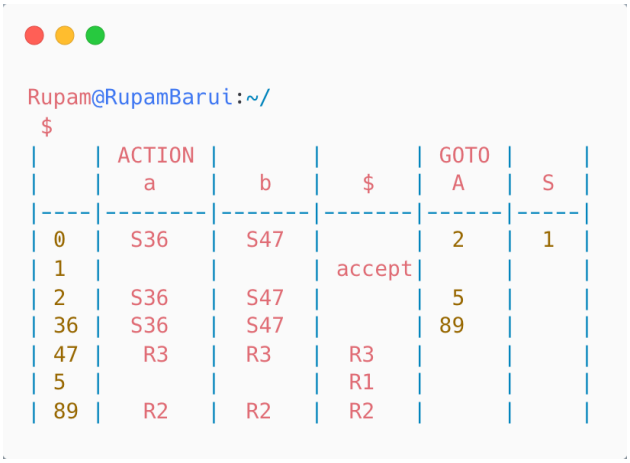

Figure 30: Action and Goto Tables for the SLR(1) Parser

Conclusion

In this experiment, we successfully designed and implemented a LALR(1) parser. The parser was tested with various inputs, and it correctly parsed valid strings according to the given grammar. This implementation demonstrates the process of constructing LALR(1) parsing tables and parsing strings using these tables.