Compiler Design LAB

(CIC-351)

Faculty Name : Ms. Sakshi Jha

Name : Amit Singhal

Enrollment No. : 11614802722

Semester : Vth

Group : C6

Branch : CSE (Shift - I)

Maharaja Agrasen Institute of Technology, PSP Area,

Sector – 22, Rohini, New Delhi – 110086

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

VISION

"To attain global excellence through **education, innovation, research,** and **work ethics** with the commitment to **serve humanity."**

MISSION

M1. To promote diversification by adopting advancement in science, technology, management, and allied discipline through continuous learning

M2. To foster **moral values** in students and equip them for developing sustainable solutions to serve both national and global needs in society and industry.

M3. To **digitize educational resources and process** for enhanced teaching and effective learning.

M4. To cultivate an **environment** supporting **incubation, product development, technology transfer, capacity building and entrepreneurship**.

M5. To encourage **faculty-student networking with alumni, industry, institutions,** and other **stakeholders** for collective engagement.

Department of Computer Science and Engineering

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

VISION

"To attain global excellence through education, innovation, research, and work ethics in the field of Computer Science and engineering with the commitment to serve humanity."

MISSION

M1. To lead in the advancement of computer science and engineering through internationally recognized research and education.

M2. To prepare students for full and ethical participation in a diverse society and encourage lifelong learning.

M3. To foster development of problem solving and communication skills as an integral component of the profession.

M4. To impart knowledge, skills and cultivate an environment supporting incubation, product development, technology transfer, capacity building and entrepreneurship in the field of computer science and engineering.

M5. To encourage faculty, student's networking with alumni, industry, institutions, and other stakeholders for collective engagement.

COMPILER DESIGN LAB

PRACTICAL RECORD

Name : **Amit Singhal** Paper Code: **CIC - 351**

Enrollment No. : **11614802722**

Branch : **CSE-I**

Semester/Group : **5C6**

S.No.	Experiment Name	Date of Perf.	M R1	A R2	R R3	K R4	S R5	Total Marks	Signature
1.	Basic program to practice flex, bison & lex.yy.c	02-08-24							
2.1	WAP to check whether a string includes a `keyword` or not.	02-08-24							
2.2	WAP to count no. of `tokens` in a string.	02-08-24							
3.	WAP to check whether a string belong to the grammer or not.	09-08-24							
4.	WAP to remove Left Recursion from a Grammer.	09-08-24							
5.	WAP to remove Left Factoring from a Grammer.	16-08-24							
6.	Write a YACC program that parses simple arithmetic expressions.	16-08-24							
7.	WAP to combine LEX and YACC to create a calculator that can evaluate arithmetic expressions with integers.	23-08-24							
8.	WAP to construct a recursive descent parser for an expression.	30-08-24							
9.	WAP to show all the operations of a stack.	30-08-24							
10.	WAP to find out the leading of the non-terminals in a grammar.	06-09-24							
11.	WAP to Implement Shift Reduce parsing for a String.	13-09-24							
12.	WAP to find out the FIRST of the Non-terminals in a grammar.	20-09-24							
13.	WAP to check whether a grammar is operator precedent.	27-09-24							
14.	WAP to implement Intermediate Code Generation (ICG) for simple expressions In The Compiler.	11-10-24							
15.	WAP to minimize any given DFA.	18-10-24							
16.	WAP to convert NFA to DFA.	25-10-24							

PROGRAM – 1

❖ AIM :: Basic program to practice flex, bison & lex.yy.c

Theory ::

Flex:

- **Purpose:** Tool for generating lexical analyzers.
- **Input:** .l files (define lexical rules and patterns).
- **Output:** C source file (lex.yy.c), which can be compiled into an executable.

Bison:

- **Purpose:** Parser generator for processing grammar and parsing tokenized input.
- **Works with Flex:** Bison creates a parser that processes tokens generated by the Flex lexer.

Dev C++:

- **Purpose:** IDE for compiling and debugging C programs.
- **Usage:** Compiles the lex.yy.c file produced by Flex into an executable (a.out).
- **Features:** Provides an integrated environment for editing, compiling, and debugging generated C code.

Key Concepts:

1. **.l File:** The input file for Flex, containing lexical rules and patterns.
2. **lex.yy.c:** C file generated by Flex, containing the lexical analyzer.
3. **a.out:** Default name of the compiled executable generated after compiling lex.yy.c.
4. **Basic Syntax in Lex:**
    - **Definitions Section:** %{ ... %} for including C code.
    - **Rules Section:** Pattern matching rules (regex) and actions.
    - **Main Section:** yylex() to process input, yywrap() to indicate end of input.

Lex Code ::

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

%%
```

```
[0-9]     { printf("Welcome\n"); exit(0); }
.         { printf("Wrong\n"); exit(0); }

%%

int yywrap() { return 1; }

int main() {
    printf("Enter: ");
    yylex();  // Start the lexer
    return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ls
prg1.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ flex prg1.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ls
lex.yy.c  prg1.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ gcc lex.yy.c
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ls
a.out  lex.yy.c  prg1.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter: 116
Welcome
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter: Amit
Wrong
```

PROGRAM – 2.1

❖ AIM :: WAP to check whether a string includes a `keyword` or not.

Theory ::

Keywords in C Programming

In C programming, **keywords** are predefined reserved words that are integral to the structure and syntax of the language. These words have specific meanings that define the operations and controls within a C program. Because of their predefined significance, keywords cannot be used for naming **variables**, **functions**, **arrays**, or **any other identifiers**. Doing so would conflict with their inherent roles in the language.

C was originally developed at **Bell Labs** by Dennis Ritchie between 1969 and 1973, and the keywords in C form a core part of the language's design, allowing for structured, efficient, and low-level programming. Each keyword is associated with specific functionality such as declaring data types, defining control flow, managing memory, and performing input/output operations.

Importance of Keywords

C's keywords are essential because they define the rules and structure of the language, guiding how programmers can build applications. They serve as the backbone for:

- **Control Flow**: Keywords like if, else, for, while, switch, and case are used to create conditional statements and loops, enabling developers to control the flow of execution based on various conditions.
- **Data Management**: Keywords such as int, float, char, double, struct, and union allow programmers to define variables and data structures to store and manipulate data efficiently.
- **Memory Allocation**: Keywords like static, auto, extern, register, and volatile play a key role in memory management, controlling how memory is allocated and accessed by the program.
- **Functionality**: Keywords such as return are used to indicate the output or result of a function. The keyword void specifies that a function does not return any value.
- **Error Handling**: Keywords like goto can be used for simple jumps in the program (though generally discouraged in modern C due to readability concerns).

Types of Keywords

1. **Data Type Keywords**: These define the type of data that can be stored in a variable. Examples include:
    - int: Represents integer data types.
    - float: Represents floating-point numbers.
    - char: Represents character data types.

    - double: Represents double-precision floating-point numbers.
2. **Control Flow Keywords**: These manage the flow of control within a program, allowing for conditional execution and looping. Examples include:
    - if, else: Conditional statements.
    - for, while, do: Looping statements.
    - switch, case: Used in switch-case statements for multi-way branching.
3. **Storage Class Keywords**: These control the scope, lifetime, and visibility of variables and functions. Examples include:
    - static: Limits the scope of a variable to the file or function where it's defined, retaining its value across function calls.
    - extern: Declares a global variable or function that is defined in another file.
    - auto: Defines automatic variables, typically the default storage class for local variables.
    - register: Requests the compiler to store the variable in a CPU register rather than RAM for faster access.
4. **Memory Management Keywords**: These are responsible for handling dynamic memory allocation and freeing memory. Examples include:
    - malloc(), free(): Functions for dynamic memory allocation, although not keywords, are integral to memory management.
5. **Return Keywords**: Keywords like return specify the value that a function should return after execution.
6. **Function Modifiers**: Keywords that modify the behavior of functions and blocks of code, such as inline for making a function inline and const for constant variables.

Conclusion

Keywords form the foundation of the C programming language, shaping how developers structure their programs, define data, control the flow of execution, and manage memory. The limited set of 32 keywords ensures simplicity and efficiency, both in terms of program writing and compilation. As reserved words, they embody the core logic and design philosophy of C, ensuring that C remains a powerful, flexible, and low-level language for systems programming, even decades after its creation. Understanding the proper use and function of keywords is essential for mastering C programming and writing efficient, maintainable code.

Lex Code ::

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

  // Array of C keywords
```

```
    char keyword[32][10] = {
        "auto", "double", "int", "struct", "break", "else", "long",
        "switch", "case", "enum", "register", "typedef", "char",
        "extern", "return", "union", "const", "float", "short",
        "unsigned", "continue", "for", "signed", "void", "default",
        "goto", "sizeof", "volatile", "do", "if", "static", "while"
    };

    int is_keyword(char *str) {
        for(int i = 0; i < 32; i++) {
            if(strcmp(str, keyword[i]) == 0)
                return 1;
        }
        return 0;
    }
%}

%%

[a-zA-Z]+ {
    if (is_keyword(yytext)) {
        printf("%s is a keyword\n", yytext);
    } else {
        printf("%s is not a keyword\n", yytext);
    }
}

[0-9]+ {
    printf("%s is a number\n", yytext);
}

\n { return 0; }    // Stops reading after a newline
```

```
.|\n { /* ignore any other characters */ }

%%

int yywrap() {
    return 1;
}

int main() {
    printf("Enter the string: ");
    yylex();
    return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ flex prg_2_checkKeywords.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ gcc lex.yy.c
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter the string: if Amit goto 116
if is a keyword
Amit is not a keyword
goto is a keyword
116 is a number
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter the string: do while return long double sizeof unsigned int
do is a keyword
while is a keyword
return is a keyword
long is a keyword
double is a keyword
sizeof is a keyword
unsigned is a keyword
int is a keyword
```

PROGRAM – 2.2

❖ AIM :: WAP to count no. of `tokens` in a string.

Theory ::

Tokens in C

In C programming, a **token** is the smallest unit of a program that has a meaning in the language. Tokens are categorized into several types, including:

1. **Keywords**: Reserved words with predefined meanings (e.g., int, return, if).
2. **Identifiers**: Names given to variables, functions, arrays, etc. (e.g., main, count).
3. **Constants**: Literal values like numbers and characters (e.g., 42, 'a').
4. **Operators**: Symbols representing operations (e.g., +, -, *, /).
5. **Punctuation**: Symbols that help structure the code (e.g., ;, ,, {, }).

Examples of Tokens:

- **Keywords**: if, while, for
- **Identifiers**: variable, sum, total
- **Constants**: 10, 3.14, 'x'
- **Operators**: +, -, =
- **Punctuation**: ;, {, }

Lex Code ::

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

int token_count = 0;
%}

%%
[a-zA-Z_][a-zA-Z0-9_]*     { token_count++; }  // Identifiers
[0-9]+                     { token_count++; }  // Numbers
"+"|"-"|"*"|"/"            { token_count++; }  // Operators
";"|"{"|"}"|"("|")"        { token_count++; }  // Punctuation
```

```
[ \t\n]+                    { /* Ignore whitespace */ }
.                           { /* Ignore other characters */ }
%%

int yywrap() {
    return 1;
}

int main() {
    printf("Enter a string: ");
    yylex();                                        // Start lexing
    printf("\nNumber of tokens: %d\n", token_count);
    return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ flex prg3.l
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ gcc lex.yy.c
singhal-amit@singhal-amit-ThinkPad-T430:~/Documents$ ./a.out
Enter a string: Amit Singhal 11614802722 Compliler Design LAB
Number of tokens: 6
```

PROGRAM – 3

❖ AIM :: WAP to check whether a string belong to the grammer or not.

Theory ::

Grammars:
A grammar is a formal system that defines the syntax of a language through a set of production rules. These rules determine how strings in the language can be formed from an initial symbol, often called the start symbol. The string generation process involves repeatedly replacing symbols with other symbols or terminal symbols (which represent actual characters in the language).

Grammars are essential in the field of compiler design as they define the syntactic structure of programming languages. They are used in various stages of compilation, such as parsing and syntax analysis, to ensure that the source code adheres to the correct syntax before further processing.

Types of Grammars:

Grammars are classified into the following types based on their complexity and generative power:

- **Regular Grammars:**
  Regular grammars define regular languages, which can be recognized by finite automata. These grammars have production rules where the left-hand side contains a single non-terminal, and the right-hand side contains a terminal, possibly followed by a non-terminal. Regular grammars are equivalent to regular expressions.
- **Context-Free Grammars (CFGs):**
  Context-Free Grammars define context-free languages, which can be recognized by pushdown automata. CFGs allow production rules where the left-hand side contains a single non-terminal and the right-hand side contains a string of terminals and/or non-terminals. CFGs are widely used to describe the syntax of programming languages, as they can model nested structures like parentheses or block statements.
- **Context-Sensitive Grammars (CSGs):**
  Context-Sensitive Grammars describe context-sensitive languages, which can be recognized by linear-bounded automata. In CSGs, production rules can have context-sensitive restrictions, meaning that the replacement of a non-terminal may depend on its surrounding symbols.
- **Unrestricted Grammars:**
  Unrestricted grammars describe recursively enumerable languages, which can be recognized by Turing machines. These grammars have the most general form of production rules, allowing for any combination of symbols on both sides of a rule.

Checking Whether a String Belongs to a Grammar:

To determine whether a string belongs to a grammar, follow these general steps:

1. **Identify the Production Rules.**
2. **Start with the Initial Symbol.**
3. **Apply Production Rules Recursively.**
4. **Compare with the Input String.**
5. **Recursive Descent Parsing.**

Example:

Consider the grammar S → aSb | ab and the string aabb:

1. **Initial Symbol:** Start with S.
2. **Apply Production Rules:**
    - Apply S → aSb, resulting in aSb.
    - Apply S → ab to S in the middle, resulting in aabb.
3. **Comparison:** The derived string aabb matches the input string exactly.
4. **Conclusion:** The string aabb belongs to the grammar.

If the input string were aaabb, no sequence of applying the rules S → aSb | ab would produce this string, so aaabb does not belong to the grammar.

Code ::

1. `gram.l` Lex Program (Lexer)

```
%{
        #include "y.tab.h"
%}

%%
        a       return A;
        b       return B;
        .|\n    return yytext[0];
%%

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

2. `**gram.y**` Yacc Program (Parser)

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

        extern int yylex();
        void yyerror(char *s);
        int valid = 1;
%}

%token A B

%%
    str: S '\n' { printf("Parsed with rule ab^n\n"); return 0; };

    S: A T;
    T: B T | /* empty */;
%%

void yyerror(char *msg) {
   valid = 0;
   fprintf(stderr, "%s\n", msg);
}

int main() {
   printf("Enter the string to check: \n");
   yyparse();
   if (valid) printf("\nValid\n");
   else printf("\nInvalid\n");
   return 0;
}
```

Output ::

1) Installing `flex` :

```
[mait@fedora exp4]$ flex gram.l
bash: flex: command not found...
Install package 'flex' to provide command 'flex'? [N/y] y


 * Waiting in queue...
 * Loading list of packages....
The following packages have to be installed:
 flex-2.6.4-12.fc38.x86_64      A tool for generating scanners (text pattern recognizers)
 m4-1.4.19-5.fc38.x86_64        GNU macro processor
Proceed with changes? [N/y] y


 * Waiting in queue...
 * Waiting for authentication...
 * Waiting in queue...
 * Downloading packages...
 * Requesting data...
 * Testing changes...
 * Installing packages...

[mait@fedora exp4]$ flex gram.l
[mait@fedora exp4]$
```

2) Installing `bison` :

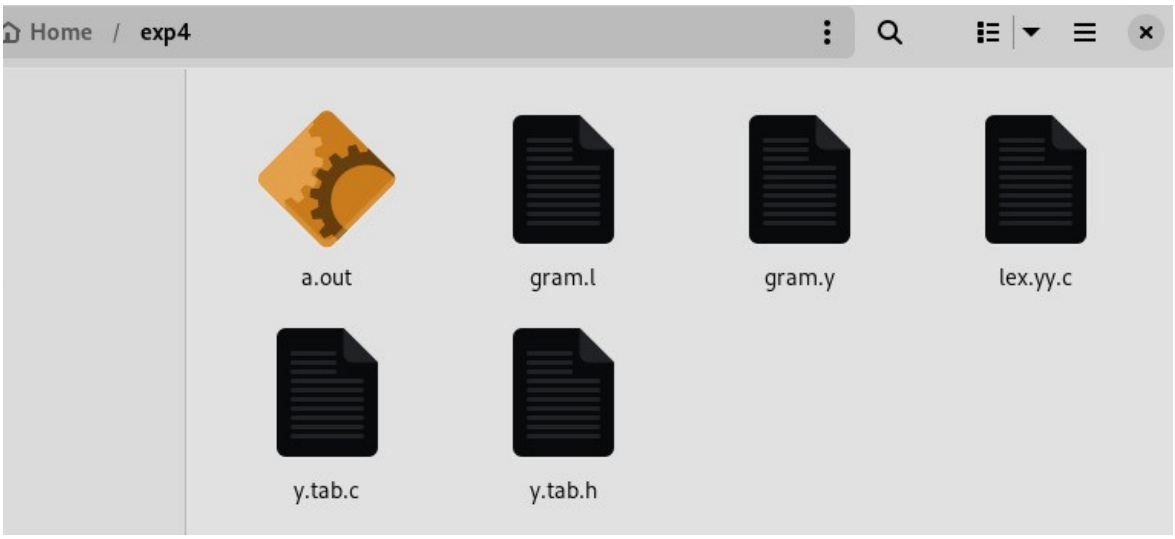
```
[mait@fedora exp4]$ bison gram.y
bash: bison: command not found...
Install package 'bison' to provide command 'bison'? [N/y] y


 * Waiting in queue...
 * Loading list of packages....
The following packages have to be installed:
 bison-3.8.2-4.fc38.x86_64      A GNU general-purpose parser generator
Proceed with changes? [N/y] y


 * Waiting in queue...
 * Waiting for authentication...
 * Waiting in queue...
 * Downloading packages...
 * Requesting data...
 * Testing changes...
 * Installing packages...

[mait@fedora exp4]$ bison gram.y
```

3) Directory & Files :



4) Program Output :

```
[mait@fedora ~]$ cd exp4/
[mait@fedora exp4]$ flex gram.l
[mait@fedora exp4]$ bison -dy gram.y
[mait@fedora exp4]$ gcc lex.yy.c y.tab.c
[mait@fedora exp4]$ ./a.out
Enter the string to check:
abb
Parsed with rule ab^n

Valid
[mait@fedora exp4]$ ./a.out
Enter the string to check:
amit
syntax error

Invalid
```

PROGRAM – 4

❖ AIM :: WAP to remove Left Recursion from a Grammer.

Theory ::

Left Recursion in Compilers

What is Left Recursion?

Left recursion in a grammar happens when a non-terminal symbol in a production rule appears as the first symbol on the right-hand side of its own definition. This creates a scenario where the non-terminal calls itself, potentially causing problems in certain parsers. For instance, in the production rule:

$A \rightarrow A\alpha \mid \beta$

The non-terminal A appears as the first symbol in its own production, which is considered left recursion.

Problem with Left Recursion

Left recursion is particularly troublesome for **top-down parsers** like **recursive descent parsers**, because it may lead to an **infinite loop**. In these parsers, the non-terminal is expanded repeatedly, without ever making progress in parsing, which results in endless recursion. Therefore, left recursion must be eliminated before parsing.

Solving Left Recursion

To eliminate left recursion, grammars are typically rewritten. Some common techniques include:

1. **Grammar Factoring**: Rearranging the grammar to remove recursion.
2. **Introducing New Non-terminals**: Defining new rules that break the cycle of recursion.

Lexical Analysis Files (.l Files)

Definition of .l File

A .l file is a source file used in **lexical analysis**, commonly with tools like **Flex (Fast Lexical Analyzer)**. These files contain **patterns and rules** for breaking down input text into tokens—basic units like keywords, identifiers, and symbols that the parser will use.

Role of Flex

The Flex tool reads the .l file and generates a **C source code** file, typically named lex.yy.c. This code implements the lexical analyzer, which processes the input text and converts it into a series of tokens based on the defined rules in the .l file.

lex.yy.c File

What is lex.yy.c?

The lex.yy.c file is a **C source file** generated by **Flex** from the corresponding .l file. It contains the actual implementation of the **lexical analyzer** as a C program. Once the C code is generated, it can be compiled into an executable program that performs tokenization.

How is lex.yy.c Used?

When the lexical analyzer is compiled and executed, it reads input text, identifies patterns based on the rules in the .l file, and outputs a sequence of **tokens** that can be used by the parser for further processing.

Code ::

```
#include <iostream>
using namespace std;
void removeLeftRecursion(string nonTerminal, vector<string> productions)
{
 vector<string> alpha, beta;
 string newNonTerminal = nonTerminal + "'";
 for (const auto &production : productions)
 {
  if (production[0] == nonTerminal[0])
  {
   alpha.push_back(production.substr(1));
  }
 }
 for (const auto &prod : beta)
 {
  cout << prod << " ";
 }
 cout << newNonTerminal << " -> ";
 for (const auto &prod : alpha)
 {
  cout << prod << " " << newNonTerminal << " | ";
 }
```

```
  cout << "\u03B5" << endl;
}
int main()
{
  vector<string> productionsE = {"E+T", "T"};
  vector<string> productionsT = {"T*F", "F"};
  cout << "Original Grammar:" << endl;
  cout << "E -> E + T | T" << endl;
  cout << "T -> T * F | F" << endl;
  cout << "F -> id" << endl
       << endl;
  cout << "Removing Left Recursion for E:" << endl;
  removeLeftRecursion("E", productionsE);
  cout << endl
       << "Removing Left Recursion for T:" << endl;
  removeLeftRecursion("T", productionsT);
  cout << endl
       << "F -> id" << endl;
  return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ left_recursion.cpp -o left_recursion
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./left_recursion

Original Grammar:
E -> E + T | T
T -> T * F | F
F -> id

Removing Left Recursion for E:
E -> T E'

E' -> + T E' | ε

Removing Left Recursion for T:
T -> F T'

T' -> * F T' | ε

F -> id
```

PROGRAM – 5

❖ AIM :: WAP to remove Left Factoring from a Grammer.

Theory ::

Left Factoring in Compilers

What is Left Factoring?

Left factoring is a technique used in compilers to simplify grammars for **top-down parsers**, especially when multiple production rules of a non-terminal share the same initial symbols (prefix). When two or more production rules begin similarly, the parser might struggle to determine which rule to apply. Left factoring helps by isolating the shared prefix and introducing a new non-terminal for the differing parts.

For example:

$$A \rightarrow \alpha\beta \mid \alpha\gamma$$

Can be rewritten using left factoring as:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta \mid \gamma$$

This transformation allows the parser to read only one symbol and decide the correct production rule to follow, making the grammar better suited for **predictive parsers**.

Why is Left Factoring Necessary?

In **top-down parsing**, especially **predictive parsing**, the parser requires a single lookahead symbol to make decisions. Left factoring ensures that no ambiguity arises by making it easier for the parser to choose a production rule based on the available input.

Challenges with Left Factoring

The main drawback of left factoring is that while it simplifies grammar for parsers, it can make the grammar more **complex** and harder to maintain. The introduction of new non-terminals and production rules can make the grammar **less intuitive** and difficult for developers to interpret.

Code ::

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
```

```
using namespace std;

void leftFactoring(const string &nonTerminal, const vector<string> &productions)
{
  unordered_map<string, vector<string>> prefixMap;
  for (const auto &production : productions)
  {
    string prefix = production.substr(0, 1);
    prefixMap[prefix].push_back(production.substr(1));
  }
  if (prefixMap.size() > 1)
  {
    cout << nonTerminal << " -> ";
    for (const auto &[prefix, suffixes] : prefixMap)
    {
      if (suffixes.size() > 1)
      {
        cout << prefix << nonTerminal + "' | ";
      }
    }
    Cout << endl;
    cout << nonTerminal + "' -> ";
    for (const auto &[prefix, suffixes] : prefixMap)
    {
      if (suffixes.size() > 1)
      {
        for (const auto &suffix : suffixes)
        {
          cout << suffix << " | ";
        }
      }
      else
      {
```

```
      cout << suffixes[0] << " | ";
    }
   }
   cout << "ε" << endl;
  }
  else
  {
   cout << nonTerminal << " -> ";
   for (const auto &production : productions)
   {
    cout << production << " | ";
   }
   cout << endl;
  }
}

int main()
{
  vector<string> productionsA = {"aB", "aC", "bD", "aE"};
  cout << "Original Grammar:" << endl;
  cout << "A -> aB | aC | bD | aE" << endl;
  cout << "After Left Factoring:" << endl;
  leftFactoring("A", productionsA);
  return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ left_factoring.cpp -o left_factoring
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./left_factoring

Original Grammar:
A -> aB | aC | bD | aE

After Left Factoring:
A -> aA' | bD |
A' -> B | C | E | ε
```

PROGRAM – 6

❖ AIM :: Write a YACC program that parses simple arithmetic expressions.

Theory ::

YACC (.y) File - Parsing and Evaluation

What is the .y File?

The .y file defines the **grammar** of the arithmetic expressions using **YACC (Yet Another Compiler Compiler)**. It specifies the rules for parsing the tokens generated by the lexer and constructs a parse tree, ensuring the correct evaluation of expressions based on precedence and associativity rules.

Grammar and Precedence Rules

The .y file contains the **BNF (Backus-Naur Form)** grammar for arithmetic expressions. It defines:

- **Operators** and their **precedence levels**.
- **Associativity** rules (left or right) to resolve ambiguity in expression evaluation.

For example:

- Multiplication and division have higher precedence than addition and subtraction.
- Operators like + and - associate from left to right.

Handling Parentheses

Parentheses are explicitly supported, allowing users to group sub-expressions and control the evaluation order. For example, (2 + 3) * 4 is evaluated as 20 rather than 14.

Error Handling: Division by Zero

To prevent runtime errors, division by zero is specifically checked in the YACC code. If a division by zero is attempted, an error message is generated, and the program halts further evaluation.

Workflow of the Parser-Interpreter

1. **Lexical Analysis (Lex)**: The input string (e.g., 3 + 4 * 5) is scanned by the lexer, which identifies tokens such as numbers (3, 4, 5), operators (+, *), and parentheses (if present).
2. **Parsing (YACC)**: The tokens are passed to the parser, which uses the defined grammar to construct a parse tree.
3. **Evaluation**: Once the parse tree is built, the parser evaluates the expression based on precedence and associativity, and the result is printed to the console.

Summary of Key Files

1. .l File (Lex)

- **Purpose**: Defines rules for lexical analysis (tokenization).
- **Recognizes**: Numbers, operators (+, -, *, /), parentheses.

2. .y File (YACC)

- **Purpose**: Defines grammar and parsing rules for arithmetic expressions.
- **Handles**: Operator precedence, associativity, parentheses, and error handling (e.g., division by zero).

Code ::

1) arith.y (Yacc/Bison)

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(const char *s);
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%%
expr:
     expr '+' expr  { printf("%d + %d = %d\n", $1, $3, $1 + $3); }
   | expr '-' expr  { printf("%d - %d = %d\n", $1, $3, $1 - $3); }
   | expr '*' expr  { printf("%d * %d = %d\n", $1, $3, $1 * $3); }
   | expr '/' expr  {
                  if ($3 == 0) {
                     yyerror("Division by zero!");
                  } else {
                     printf("%d / %d = %d\n", $1, $3, $1 / $3);
                  }
               }
```

```
    | '(' expr ')'   { $$ = $2; }
    | NUMBER        { $$ = $1; }
    ;
%%
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main(void) {
    printf("Enter the expression: ");
    return yyparse();
}
```

2) arith.l (Lex/Flex)

```
%{
#include "y.tab.h"
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[\t ]+      /* Ignore whitespace */
\n          { return 0; }  /* End of input */
.           { return yytext[0]; }
%%

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

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ flex arith.l
singhal-amit@singhal-amit-ThinkPad-T430:~$ bison -d arith.y
singhal-amit@singhal-amit-ThinkPad-T430:~$ gcc lex.yy.c arith.tab.c -o arith -l
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./arith

Enter the expression: 3 + 5 * ( 10 - 2 )
10 - 2 = 8
5 * 8 = 40
3 + 40 = 43
```

PROGRAM – 7

❖ AIM :: WAP To combine LEX and YACC to create a calculator that can evaluate arithmetic expressions with integers.

Theory ::

Lex and YACC for a Simple Arithmetic Calculator

Overview of the Calculator

The simple arithmetic calculator, built using **Lex** and **YACC**, can evaluate basic expressions involving:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

Lex and YACC work together to tokenize and parse arithmetic expressions, following operator precedence and associativity rules. **Parentheses** are supported to allow more complex grouping of expressions. The calculator ensures correct evaluation of the input, and division by zero is handled to avoid runtime errors.

Lex and YACC: Key Tools in Compiler Construction

Lex: The Lexical Analyzer

Lex (Lexical Analyzer Generator) is responsible for scanning the input text, identifying **tokens** like numbers, operators, and parentheses. It uses **regular expressions** to recognize patterns in the input. For example, Lex will recognize sequences like 123 as numbers, and symbols like + or * as operators. These tokens are then passed to YACC for further processing.

YACC: The Parser and Evaluator

YACC (Yet Another Compiler Compiler) defines the **grammar** of the arithmetic expressions and creates a syntax tree based on these rules. It uses **context-free grammar** rules to parse the tokens provided by Lex and then evaluates the expression following operator precedence and associativity.

Structure of the Calculator

- Lex (.l) File - Tokenizing Arithmetic Expressions

What is the .l File?

The .l file contains **rules and patterns** for identifying tokens in the input. It describes how numbers and operators should be broken down into **tokens** using **regular expressions**.

Key Tokens in the Lex File:

- **Numbers**: Integers, e.g., 1, 42, 99.
- **Operators**: +, -, *, /.
- **Parentheses**: (,) for grouping expressions.

Lex reads the input and produces a sequence of these tokens, which is passed to YACC for parsing.

- YACC (.y) File - Defining Grammar and Evaluation

What is the .y File?

The .y file defines the **grammar** of arithmetic expressions using **BNF (Backus-Naur Form)** notation. It outlines how operators and numbers combine to form valid expressions, and it also specifies the **evaluation rules**. These rules ensure that operations are performed in the correct order according to **precedence** and **associativity**.

Grammar and Precedence Rules:

The YACC file ensures that:

- Multiplication (*) and division (/) have higher precedence than addition (+) and subtraction (-).
- Parentheses override default precedence to allow grouped expressions.
- Left associativity is applied to operators, meaning expressions like 1 - 2 - 3 are evaluated from left to right.

Error Handling: Division by Zero

Division by zero is handled in the YACC file by checking for this condition during parsing. If division by zero is attempted, the program prints an error message and halts further evaluation.

How the Calculator Works?

1. **Lexical Analysis**: Lex scans the input (e.g., 3 + 4 * (5 - 2)) and breaks it down into tokens like numbers (3, 4, 5, 2), operators (+, *, -), and parentheses.
2. **Parsing and Evaluation**: YACC parses these tokens according to the grammar and builds a **parse tree**. It then evaluates the expression based on operator precedence, associativity, and parentheses.
3. **Result Output**: Once parsed and evaluated, the result is printed to the console. For instance, 3 + 4 * (5 - 2) would output 15.

Summary of Key Files

1. .l File (Lex)

- **Purpose**: Defines rules for tokenizing arithmetic expressions.
- **Recognizes**: Numbers, operators (+, -, *, /), and parentheses.

2. .y File (YACC)

- **Purpose**: Defines grammar and parsing rules, handles operator precedence and associativity, evaluates expressions.
- **Handles**: Arithmetic operators, parentheses, and division-by-zero errors.

Code ::

1) calc.y (Yacc/Bison)

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

%token NUMBER
```

```
%left PLUS MINUS
%left MULT DIV
%%
expr:
     expr PLUS term  { printf("%d\n", $1 + $3); }
   | expr MINUS term { printf("%d\n", $1 - $3); }
   | term;

term:
     term MULT factor { printf("%d\n", $1 * $3); }
   | term DIV factor  { printf("%d\n", $1 / $3); }
   | factor;

factor:
     NUMBER;
%%
int main() {
        printf("\nEnter any Arithmetic Expression which can have operations - Addition, Subtraction, Multiplication, Division:\n");
   yyparse();
   return 0;
}

int yyerror() {
   printf("Error\n");
   return 1;
}
```

2) calc.l (Lex/Flex)

```
%{
#include "y.tab.h"
%}
```

```
%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[\n]        { return 0; }
"+"         { return PLUS; }
"-"         { return MINUS; }
"*"         { return MULT; }
"/"         { return DIV; }
[\t ]+      { /* Ignore whitespace */ }
.           { printf("Unexpected character: %s\n", yytext); return 1; }
%%
int yywrap() {
    return 1;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ flex calc.l
singhal-amit@singhal-amit-ThinkPad-T430:~$ bison -d calc.y
singhal-amit@singhal-amit-ThinkPad-T430:~$ gcc lex.yy.c calc.tab.c -o calc -lm
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./calc

Enter any Arithmetic Expression which can have operations - Addition, Subtraction, Multiplication, Division:
10 + 20 * 3 - 5 / 5

20 * 3 = 60
10 + 60 = 70
5 / 5 = 1
70 - 1 = 69
```

PROGRAM – 8

❖ AIM :: WAP To construct a recursive descent parser for an expression.

Theory ::

Recursive Descent Parser

Overview

A **recursive descent parser** is a type of **top-down parser** that interprets input based on a collection of recursive functions, each corresponding to a non-terminal in the grammar. This method allows the parser to process the input by matching it against the grammar's production rules through function calls.

How It Works

1. **Recursive Procedures**: Each non-terminal in the grammar is represented by a separate function. The parser begins processing by invoking the appropriate function for the starting non-terminal.
2. **Matching Input**: The parser reads the input character by character, checking if the current input matches the expected terminals as defined in the grammar.
3. **Consumption of Terminals**: When a terminal matches, it is consumed (removed from the input). If a non-terminal is encountered, the corresponding function is called recursively to handle it.
4. **Backtracking**: If a match fails, the parser can backtrack to explore alternative parsing paths, although this may lead to inefficiencies.

Advantages and Disadvantages

- **Advantages**:
  - Simplicity and ease of implementation for many grammars.
  - Directly reflects the grammar structure in the code, making it intuitive.
- **Disadvantages**:
  - Cannot handle left-recursive grammars directly without transformation.
  - May require significant backtracking in ambiguous grammars, affecting performance.

Lexical Analysis Files

.l File - Lexical Analysis

Definition

A .l file is a source file utilized in **lexical analysis**, commonly associated with the **Flex (Fast Lexical Analyzer)** tool. This file specifies rules and patterns for tokenizing the input text.

Key Features

- **Tokenization Rules**: Defines how to identify different types of tokens such as keywords, operators, identifiers, and literals using **regular expressions**.

- **Token Generation**: The lexer reads the input and converts it into a sequence of tokens, which are passed to the parser for further processing.

.y File - Grammar Definition

Definition

A .y file is a source file used by **YACC (Yet Another Compiler Compiler)** that defines the grammar of a programming language. It employs **BNF (Backus-Naur Form)** notation to specify the structure of valid input sequences.

Key Features

- **Grammar Rules**: Contains the production rules that describe how tokens generated by the lexer can be combined to form valid expressions or statements.
- **Parsing Actions**: Specifies actions to be performed during parsing, such as building parse trees and evaluating expressions.
- **Error Handling**: Implements mechanisms for error detection and recovery during the parsing process.

Summary

The combination of a **recursive descent parser**, along with the **.l** and **.y** files, enables the efficient interpretation and processing of programming languages or structured input.

- **.l File**:
  - Handles **tokenization** of input into meaningful symbols.
- **.y File**:
  - Defines the **grammar** and parsing rules, guiding the parser's behavior.

Code ::

```
#include <iostream>
#include <string>
using namespace std;

#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();
const char *cursor;
string input;

int main()
{
```

```
  cout << "Enter the string: ";
  cin >> input;           // Read input from the user
  cursor = input.c_str(); // Initialize cursor to point to the input string

  cout << "\nInput\t\t Action\n";
  cout << "--------------------------------\n";
  if (E() && *cursor == '\0')
  {
    cout << "--------------------------------\n";
    cout << "String is successfully parsed\n";
    return 0;
  }
  else
  {
    cout << "--------------------------------\n";
    cout << "Error in parsing String\n";
    return 1;
  }
}

int E()
{
  cout << cursor << "\t\t E -> T E'\n";
  if (T())
  {
    if (Edash())
      return SUCCESS;
    else
      return FAILED;
  }
  else
    return FAILED;
}

int Edash()
{
  if (*cursor == '+')
  {
    cout << cursor << "\t\t E' -> + T E'\n";
```

```
    cursor++;
    if (T())
    {
      if (Edash())
        return SUCCESS;
      else
        return FAILED;
    }
    else
      return FAILED;
  }
  else
  {
    cout << cursor << "\t\t E' -> $\n";
    return SUCCESS; // E' -> ε (empty string)
  }
}

int T()
{
  cout << cursor << "\t\t T -> F T'\n";
  if (F())
  {
    if (Tdash())
      return SUCCESS;
    else
      return FAILED;
  }
  else
    return FAILED;
}

int Tdash()
{
  if (*cursor == '*')
  {
    cout << cursor << "\t\t T' -> * F T'\n";
    cursor++;
    if (F())
```

```
    {
      if (Tdash())
        return SUCCESS;
      else
        return FAILED;
    }
    else
      return FAILED;
  }
  else
  {
    cout << cursor << "\t\t T' -> $\n";
    return SUCCESS; // T' -> ε (empty string)
  }
}

int F()
{
  if (*cursor == '(')
  {
    cout << cursor << "\t\t F -> ( E )\n";
    cursor++;
    if (E())
    {
      if (*cursor == ')')
      {
        cursor++;
        return SUCCESS;
      }
      else
        return FAILED;
    }
    else
      return FAILED;
  }
  else if (*cursor == 'i')
  {
    cout << cursor << "\t\t F -> i\n";
    cursor++;
```

```
    return SUCCESS;
  }
  else
    return FAILED;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o parser parser.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./parser

Enter the string: i+i*i

Input             Action
--------------------------------
i+i*i             F -> i
                  E' -> + T E'
                  T -> F T'
i*i               F -> i
                  T' -> * F T'
i                 F -> i
                  T' -> $
--------------------------------

String is successfully parsed

singhal-amit@singhal-amit-ThinkPad-T430:~$
```

PROGRAM – 9

❖ AIM :: WAP to show all the operations of a stack.

Theory :: **Stack Data Structure**

Overview

A **stack** is a linear data structure that adheres to the **Last In First Out (LIFO)** principle. This means that the most recently added element is the first one to be removed. Stacks are commonly used in various programming applications, including expression evaluation, backtracking algorithms, and managing function calls.

Key Operations

1. **Push**:
    - **Definition**: The push operation adds an element to the top of the stack.
    - **Functionality**: When an element is pushed onto the stack, it becomes the new top element, and any previously top elements are now below it.
2. **Pop**:
    - **Definition**: The pop operation removes the element from the top of the stack.
    - **Functionality**: The element that is removed is the most recently added element. After a pop operation, the next element down becomes the new top.
3. **Peek**:
    - **Definition**: The peek operation allows the user to view the top element of the stack without removing it.
    - **Functionality**: This is useful for accessing the top element while preserving the state of the stack.

Implementation

Stacks can be implemented using two common methods:

1. **Arrays**:
    - An array can be used to hold the stack elements.
    - An index or pointer keeps track of the position of the top element.
    - When pushing, the index is incremented, and when popping, the index is decremented.
2. **Linked Lists**:
    - A linked list can also be used to implement a stack, where each node contains a data element and a reference to the next node.
    - The head of the list can represent the top of the stack, allowing push and pop operations to occur at the head efficiently.

Use Cases

- **Function Call Management**: Stacks are used to keep track of active functions in programming, allowing for backtracking and handling of local variables.
- **Expression Evaluation**: Stacks facilitate the evaluation of arithmetic expressions, particularly in converting from infix to postfix notation.
- **Backtracking Algorithms**: They are instrumental in algorithms that require exploring multiple potential solutions, such as maze-solving.

Code ::

```
#include <iostream>
#define MAX 5

using namespace std;

class Stack
{
private:
  int stack[MAX];
  int top;

public:
  Stack() : top(-1) {}

  void push(int val)
  {
    if (top == MAX - 1)
    {
      cout << "Stack Overflow\n";
      return;
    }
    stack[++top] = val;
    cout << "Pushed " << val << " onto the stack" << endl;
  }
```

```
  int pop()
  {
    if (top == -1)
    {
      cout << "Stack Underflow\n";
      return -1;
    }
    cout << "Popped " << stack[top] << " from the stack" << endl;
    return stack[top--];
  }

  void display()
  {
    if (top == -1)
    {
      cout << "Stack is empty\n";
      return;
    }
    cout << "Stack elements: ";
    for (int i = 0; i <= top; i++)
    {
      cout << stack[i] << " ";
    }
    cout << endl; // Added newline for better formatting
  }
};

int main()
{
  Stack s;
  s.push(10);
  s.push(20);
  s.display();
  s.pop();
```

```
    s.display();
    return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o stack_example stack_example.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./stack_example

Pushed 10 onto the stack
Pushed 20 onto the stack

Stack elements: 10 20

Popped 20 from the stack

Stack elements: 10

singhal-amit@singhal-amit-ThinkPad-T430:~$
```

PROGRAM – 10

❖ AIM :: WAP to find out the leading of the non-terminals in a grammar.

Theory ::

Leading of a Non-Terminal in Context-Free Grammar

Definition

In a **context-free grammar**, the **leading** of a non-terminal is defined as the set of **terminals** that can appear as the first symbols of strings derived from that non-terminal during a **leftmost derivation**. Understanding the leading is vital for implementing **predictive parsing**, where the parser selects the appropriate production rule based on the next input symbol.

Importance in Predictive Parsing

- **Predictive Parsing**: This technique allows parsers to make decisions about which production to use based on the next symbol in the input stream. The leading information helps the parser to anticipate the symbols that can follow a given non-terminal, facilitating efficient parsing without ambiguity.
- **LL(1) Parsing**: The leading set is crucial for constructing **LL(1) parsing tables**, which are used to guide the parsing process in top-down parsers.

How to Determine the Leading of a Non-Terminal

1. **Examine Production Rules**:
    - Review all production rules that include the non-terminal in question. For instance, if you have a non-terminal A with production rules like:
        - $A \rightarrow aB$
        - $A \rightarrow bC$
        - $A \rightarrow D$
2. **Identify Direct Terminals**:
    - If a production rule starts with a **terminal**, add that terminal to the leading set. For example, from the rules above, the terminals a and b would be included in the leading set.
3. **Follow Non-Terminals**:
    - If a production rule starts with another **non-terminal**, recursively determine the leading of that non-terminal and include its terminals in the leading set. For instance, if D has productions $D \rightarrow c$ or $D \rightarrow E$ (where E leads to terminals), then the leading set of D influences A's leading set.
4. **Account for Nullable Non-Terminals**:
    - If a non-terminal can derive an empty string (i.e., it is **nullable**), include the leading of subsequent non-terminals in the production as well. For example, if D can derive an empty string, you must also consider the leading of C in the derivation.

Example

For a non-terminal A with the following productions:

- $A \rightarrow aB$
- $A \rightarrow bC$
- $A \rightarrow D$

Assuming D can derive c and is nullable:

- The leading of A would include a, b, and c based on the rules above.

Summary

The leading of a non-terminal plays a crucial role in the process of predictive parsing in context-free grammars. By identifying which terminals can appear as the first symbols in derivations, parsers can efficiently decide which production rules to apply, making the parsing process smoother and unambiguous. The construction of **LL(1) parsing tables** relies heavily on this leading information, ensuring effective parsing strategies in top-down approaches.

Code ::

```
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <string>
#include <sstream>

using namespace std;

void findLeading(const string& nonTerminal, const map<string, vector<string>>& productions,
                 set<char>& leading, set<string>& visited) {
    if (visited.count(nonTerminal)) return;
    visited.insert(nonTerminal);

    for (const auto& production : productions.at(nonTerminal)) {
        for (char symbol : production) {
            if (isalpha(symbol)) {
                leading.insert(symbol);
                break; // No need to check further symbols in this production
```

```
            } else if (isupper(symbol) && productions.count(string(1, symbol))) {
                findLeading(string(1, symbol), productions, leading, visited);
                break; // Break after checking the first non-terminal
            }
        }
    }
}

int main() {
    map<string, vector<string>> productions;
    productions["A"] = {"BC"};
    productions["B"] = {"a"};
    productions["C"] = {"d"};

    map<string, set<char>> leadingMap;

    for (const auto& production : productions) {
        const string& nonTerminal = production.first;
        set<char> leading;
        set<string> visited;
        findLeading(nonTerminal, productions, leading, visited);
        leadingMap[nonTerminal] = leading;
    }

    cout << "Leading of non-terminals:\n";
    for (const auto& entry : leadingMap) {
        cout << entry.first << " -> { ";
        for (char terminal : entry.second) {
            cout << terminal << " ";
        }
        cout << "}\n";
    }
    return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o leading_example leading_example.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./leading_example

Leading of non-terminals:
A -> { a d }
B -> { a }
C -> { d }

singhal-amit@singhal-amit-ThinkPad-T430:~$
```

PROGRAM – 11

❖ AIM :: WAP to Implement Shift Reduce parsing for a String.

Theory :: **Shift-Reduce Parsing in Compilers**

Overview

A **shift-reduce parser** is a type of **bottom-up parser** utilized in compiler design to verify the syntax of input strings based on **context-free grammars**. This parsing technique involves two primary operations: **shift** and **reduce**.

- **Shift**: Moves the next input symbol onto a stack.
- **Reduce**: Replaces a sequence of symbols on the stack with a non-terminal symbol according to the grammar's production rules.

Benefits

Shift-reduce parsing is particularly effective for handling ambiguous grammars and is often implemented using **LR parsing algorithms**. These algorithms allow for efficient parsing of programming languages by determining the next action based on the current state and the input symbol.

Steps in Shift-Reduce Parsing

1. **Shift**:
    - **Action**: Move the next input symbol onto the stack.
    - **Purpose**: This step expands the stack, preparing it for potential reductions.
2. **Reduce**:
    - **Action**: Replace the top sequence of symbols on the stack with a non-terminal according to the production rules.
    - **Purpose**: This step simplifies the stack, gradually constructing higher-level structures based on the grammar.
3. **Accept**:
    - **Action**: If the entire input has been consumed, and the stack contains only the start symbol of the grammar, the parsing is considered successful.
    - **Purpose**: This signifies that the input string is valid according to the grammar rules.
4. **Error**:
    - **Action**: If neither the shift nor the reduce action can be applied, report an error.
    - **Purpose**: This step handles invalid input or cases where the parser cannot proceed.

Example

Consider a simple grammar:

- $S \rightarrow AB$

- $A \rightarrow a$
- $B \rightarrow b$

Parsing Process

1. **Input**: ab
2. **Stack**: Initially empty.

- **Step 1 (Shift)**: Push a onto the stack.
  - Stack: a
  - Input: b
- **Step 2 (Shift)**: Push b onto the stack.
  - Stack: ab
  - Input: (empty)
- **Step 3 (Reduce)**: Apply the rule $B \rightarrow b$.
  - Stack: aB
- **Step 4 (Reduce)**: Apply the rule $A \rightarrow a$.
  - Stack: S
- **Step 5 (Accept)**: The input is consumed, and the stack contains the start symbol S. Parsing is successful.

Conclusion

Shift-reduce parsing is a powerful method in compiler design that efficiently verifies the syntax of input strings. By employing shift and reduce operations, it constructs parse trees from the bottom up. The approach can effectively manage complex and ambiguous grammars, making it a fundamental technique in the field of syntax analysis.

Code ::

```
#include <iostream>

#include <vector>

#include <string>

using namespace std;


vector<char> st, a;


void StackAndInput()

{

  for (auto x : st)

  {

    cout << x << "\t";

  }

  for (auto x : a)
```

```
  {
    cout << x;
  }
}

void check()
{
  for (int i = 0; i < st.size(); i++)
  {
    if (st[i] == 'a')
    {
      st[i] = 'E';
      StackAndInput();
      cout << "$\t" << "REDUCE E->a" << endl;
      check();
    }
    if (i + 2 < st.size() && st[i] == 'E' && (st[i + 1] == '+' || st[i + 1] == '*') && st[i + 2] == 'E')
    {
      st.pop_back(); // pop the last 'E'
      st.pop_back(); // pop the operator ('+' or '*')
      st.pop_back(); // pop the first 'E'
      StackAndInput();
      if (st.empty() || st.back() != 'E')
      {
        st.push_back('E'); // push back 'E' after reduction
      }
      cout << "$\t" << "REDUCE E->E+E or E->E*E" << endl;
    }
  }
}

int main()
{
  cout << "GRAMMAR is: \nE -> E + E \nE -> E * E \nE -> a\n";
```

```
  string input;
  cout << "Enter input string (consisting of 'a', '+', '*'): ";
  cin >> input;

  for (char ch : input)
  {
    a.push_back(ch);
  }

  cout << "\nStack\tInput\tAction\n";
  for (int i = 0; i < a.size(); i++)
  {
    st.push_back(a[i]);
    a[i] = ' ';
    StackAndInput();
    cout << "$\tSHIFT -> " << st.back() << endl;
    check();
  }

  if (st.size() == 1 && st[0] == 'E')
  {
    cout << "\n\nString accepted\n";
  }
  else
  {
    cout << "\n\nString rejected\n";
  }
  return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o parser_example parser_example.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./parser_example

GRAMMAR is:
E -> E + E
E -> E * E
E -> a

Enter input string (consisting of 'a', '+', '*'): a+a*a

Stack   Input   Action
a       a+a*a   $   SHIFT -> a
E       a+a*a   $   REDUCE E->a
E       a+a*a   $   SHIFT -> +
E       a+a*a   $   SHIFT -> a
E       a+a*a   $   REDUCE E->a
E       a+a*a   $   SHIFT -> *
E       a+a*a   $   SHIFT -> a
E       a+a*a   $   REDUCE E->a
E       a+a*a   $   REDUCE E->E+E
E       a+a*a   $   REDUCE E->E*E

String accepted

singhal-amit@singhal-amit-ThinkPad-T430:~$
```

PROGRAM – 12

❖ AIM :: WAP to find out the FIRST of the Non-terminals in a grammar.

Theory :: **FIRST Set in Compiler Design**

Definition

In compiler design, the **FIRST** of a non-terminal is defined as the set of **terminal symbols** that can appear at the beginning of any string derived from that non-terminal. The FIRST set is crucial for constructing parsing tables, particularly for **LL** and **LR parsers**, as it helps determine which terminal symbol is expected first during parsing.

Importance

The FIRST set plays a significant role in:

- **Parsing Table Construction**: It helps parsers make decisions based on the expected input symbols.
- **Syntax Analysis**: It allows for efficient error detection and handling in the parsing process.

Rules for Computing the FIRST Set

1. **Terminal Rule**:
    - If x is a terminal, then: FIRST(x)={′x′}
    - **Example**: If x=a, then FIRST(a)={a}.
2. **Empty Production Rule**:
    - If $x \rightarrow \epsilon$ is a production rule, then: $\epsilon \in$ FIRST(x)
    - **Example**: If x can derive an empty string, then FIRST(x) includes ϵ.
3. **Production Rules with Non-Terminals**:
    - For a production $X \rightarrow Y1Y2Y3 \ldots Yn$:
        - Start with: FIRST(X)=FIRST(Y1)
        - If FIRST(Y1) contains ϵ: FIRST(X)=FIRST(Y1)$-\{\epsilon\} \cup$ FIRST(Y2)
        - If FIRST(Yi) contains ϵ for all i=1 to n:
            - Add ϵ to FIRST(X):

        $\epsilon \in$ FIRST(X)

Example Calculation

Consider a grammar with the following productions:

- $S \rightarrow AB$
- $A \rightarrow a$
- $A \rightarrow \epsilon$
- $B \rightarrow b$

Computing FIRST Sets:

1. **For Terminal a**:

   FIRST(a)={a}
2. **For Non-Terminal A**:
   - $A \rightarrow a$ gives: FIRST(A)={a}
   - $A \rightarrow \epsilon$ adds ϵ: FIRST(A)={a,ϵ}
3. **For Non-Terminal B**:
   - $B \rightarrow b$ gives: FIRST(B)={b}
4. **For Non-Terminal S**:
   - From $S \rightarrow AB$: FIRST(S)=FIRST(A)={a,ϵ}
   - Since FIRST(A) includes ϵ, we check B: FIRST(S)={a,b} ∪ {ϵ}={a,b,ϵ}

Code ::

```
#include <bits/stdc++.h>
using namespace std;

char productionSet[10][10];
int n; // Declare n globally

void FIRST(set<char> &result, char c)
{
  if (!(isupper(c)))
  {
    result.insert(c);
    return;
  }
  for (int i = 0; i < n; i++)
  {
    if (productionSet[i][0] == c)
    {
      FIRST(result, productionSet[i][2]); // Recursively find FIRST of the production
    }
  }
}
```

```
int main()
{
  cout << "Enter number of productions: ";
  cin >> n;

  cout << "Enter productions (example: A -> aB): \n";
  for (int i = 0; i < n; i++)
  {
    cin >> productionSet[i];
  }

  set<char> nonTerminals;
  for (int i = 0; i < n; i++)
  {
    nonTerminals.insert(productionSet[i][0]);
  }

  for (auto c : nonTerminals)
  {
    set<char> result;
    FIRST(result, c);
    cout << "First (" << c << ") : { ";
    for (auto x : result)
    {
      cout << x << " ";
    }
    cout << "}\n";
  }

  return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o first_set first_set.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./first_set

Enter number of productions: 3

Enter productions (example: A -> aB):
A -> aB
B -> b
C -> cD
D -> d

First (A) : { a }
First (B) : { b }
First (C) : { c }
First (D) : { d }

singhal-amit@singhal-amit-ThinkPad-T430:~$
```

PROGRAM – 13

❖ AIM :: WAP to check whether a grammar is operator precedent.

Theory ::

Operator Precedence Parsing in Compiler Design

Overview

Operator precedence parsing is a specific type of **shift-reduce parsing** used in compiler design to analyze the syntax of expressions based on operator precedence. It takes an **operator grammar** and an **input string**, attempting to generate a parse tree that represents the structure of the input expression.

Key Concepts

- **Parse Tree Generation**: A parse tree is produced only if the input string is accepted by the parser. This tree illustrates how the input can be derived from the grammar.
- **Shift and Reduce Operations**: The parsing process involves shifting and reducing based on the precedence of operators in relation to the current input symbol and the symbol at the top of the stack.

How Operator Precedence Parsing Works

1. **Input and Stack Management**:
    - The operator precedence parser maintains a stack to hold symbols and an input buffer for the current expression.
2. **Precedence Rules**:
    - The parser determines whether to **shift** (add the next input symbol to the stack) or **reduce** (replace symbols in the stack with a non-terminal) based on the precedence of the symbols involved.
3. **Operator Grammar**:
    - The grammar used in operator precedence parsing must adhere to specific properties to ensure it can be parsed effectively.

Properties of Operator Grammar

A grammar is classified as an **operator grammar** if it meets the following conditions:

1. **No Epsilon Productions**:
    - There should be no epsilon (ε) on the right-hand side of any production rule. This means that each production must derive at least one terminal symbol, ensuring that every derivation contributes to the formation of the string.
2. **Non-Terminals Separation**:

- There should be no two non-terminals adjacent to each other in the production rules. This restriction simplifies the parsing process by avoiding ambiguity and ensuring that operators can be clearly distinguished.

Examples of Operator Grammar

Common examples of operator grammar include expressions involving binary operators, such as:

- A+B
- A−B
- A×B

These expressions are structured in a way that allows the parser to apply operator precedence rules efficiently.

Parsing Process

1. **Initialization**: Begin with an empty stack and the input string to be parsed.
2. **Shifting and Reducing**:
    - **Shift**: If the symbol at the top of the stack has lower precedence than the current input symbol, shift the input symbol onto the stack.
    - **Reduce**: If the symbol at the top of the stack has higher precedence than the current input symbol, reduce the symbols on the stack according to the production rules.
3. **Final Acceptance**: The parsing is successful if, after processing the entire input, the stack contains only the start symbol of the grammar.

Summary

Operator precedence parsing is an effective method for handling expressions in compiler design, particularly when dealing with operators. By adhering to the properties of operator grammar and leveraging the concepts of shift and reduce, this parsing technique facilitates accurate syntax analysis and parse tree generation for expressions. This approach ensures that operator precedence is respected, leading to correct interpretation of the input strings.

Code ::

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void exitWithError()
{
  cout << "Not operator grammar" << endl;
```

```
  exit(0);
}

int main()
{
  int n;
  cout << "Enter the number of productions: ";
  cin >> n;
  vector<string> grammar(n);

  cout << "Enter the productions (e.g., A=B+C):\n";
  for (int i = 0; i < n; i++)
  {
    cin >> grammar[i];
  }

  for (const auto &prod : grammar)
  {
    int j = 2; // Start after the left-hand side (assuming format A=B+C)
    char c;

    while (prod[j] != '\0')
    {
      c = prod[j];
      // Check for operators and ensure they are not at the end of the production
      if (c == '+' || c == '-' || c == '*' || c == '/')
      {
        if (prod[j + 1] == '\0' || prod[j + 1] == '$')
        {
          exitWithError();
        }
      }
```

```
    else if (c == '$')
    {
      exitWithError();
    }
    else if (!(isupper(c)) && c != 'A')
    { // Allow only uppercase letters and 'A'
      exitWithError();
    }
    j++;
   }
  }

  cout << "Operator grammar" << endl;
  return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o operator_grammar operator_grammar.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./operator_grammar

Enter the number of productions: 3

Enter the productions (e.g., A=B+C):
A=B+C
B=C-D
C=A*B

Operator grammar

singhal-amit@singhal-amit-ThinkPad-T430:~$ ./operator_grammar

Enter the number of productions: 2

Enter the productions (e.g., A=B+C):
A=B+
C=A*

Not operator grammar
```

PROGRAM – 14

❖ AIM :: WAP to implement Intermediate Code Generation (ICG) for simple expressions In The Compiler.

Theory ::

Intermediate Code Generation in Compiler Design

Overview

Intermediate Code Generation is a critical phase in the compilation process where the compiler translates high-level source code into an **intermediate representation (IR)**. This IR is neither machine code nor the original high-level code but serves as a simplified version that facilitates further processing.

Purpose of Intermediate Code

The use of intermediate code provides several advantages:

- **Abstraction**: It acts as a bridge between high-level programming languages and machine code, allowing for better optimization and analysis.
- **Portability**: The IR is often architecture-independent, enabling it to run on various hardware platforms without extensive modifications.
- **Optimization**: It allows the compiler to optimize the code before final generation of machine code, enhancing performance and resource utilization.

Key Components of Intermediate Code Generation

1. **Translation**:
    - The compiler takes high-level code (e.g., C, Java) and converts it into an intermediate form that is easier to analyze and manipulate. This translation often involves breaking down complex constructs into simpler operations.
    - **Example**: A high-level statement like x = a + b * c; may be translated into a series of intermediate instructions that clearly define the order of operations.
2. **Portability**:
    - The intermediate representation can often be executed on different machine architectures with minimal changes, making the code more versatile.
    - By decoupling the source code from the target machine architecture, the same intermediate code can be used across different platforms, which simplifies cross-compilation.
3. **Optimization**:
    - Before generating machine code, the compiler can perform optimizations on the intermediate code. This can include techniques such as:
        - **Constant Folding**: Evaluating constant expressions at compile time.
        - **Dead Code Elimination**: Removing code that does not affect program output.
        - **Loop Optimization**: Improving loop performance through various techniques like unrolling or invariant code motion.

- These optimizations help in producing more efficient machine code, leading to faster execution and reduced memory usage.

Example Process

1. **High-Level Code**:

   c
   Copy code
   int x = a + b * c;

2. **Intermediate Representation**:

   plaintext
   Copy code
   t1 = b * c
   x = a + t1

3. **Optimization**:
   - If b and c are constants, the compiler could evaluate b * c at compile time.
4. **Machine Code Generation**:
   - Finally, the optimized intermediate code is translated into machine code specific to the target architecture.

Summary

Intermediate Code Generation is an essential stage in the compilation process that translates high-level source code into a more manageable and architecture-independent representation. By enabling easier analysis, optimization, and portability, intermediate code serves as a vital link between high-level programming languages and machine code, contributing to the efficiency and effectiveness of modern compilers.

Code ::

```
#include <iostream>
#include <sstream>
#include <vector>
#include <string>
#include <map>
using namespace std;

void generateIntermediateCode(const string &expression)
{
  stringstream ss(expression);
  string token;
```

```
vector<string> tokens;

// Tokenize the input expression
while (ss >> token)
{
  tokens.push_back(token);
}

int tempCount = 1;
vector<string> intermediateCode;
vector<string> operands;
vector<string> operators;

// Generate intermediate code
for (const auto &t : tokens)
{
  if (t == "=")
  {
    continue; // Skip the assignment operator
  }
  else if (t == "+" || t == "-" || t == "*" || t == "/")
  {
    operators.push_back(t);
  }
  else
  {
    operands.push_back(t);
    if (!operators.empty())
    {
      string op = operators.back();
      operators.pop_back();
      string rhs2 = operands.back();
      operands.pop_back();
      string rhs1 = operands.back();
      operands.pop_back();
      string tempVar = "t" + to_string(tempCount++);
      intermediateCode.push_back(tempVar + " = " + rhs1 + " " + op + " " + rhs2);
      operands.push_back(tempVar); // Push the temporary variable back to operands
    }
```

```
    }
  }

  // Final assignment
  if (!operands.empty())
  {
    string lhs = tokens[0];               // Left-hand side variable
    string finalResult = operands.back(); // The result of the last operation
    intermediateCode.push_back(lhs + " = " + finalResult);
  }

  // Output the intermediate code
  cout << "Intermediate Code:\n";
  for (const auto &code : intermediateCode)
  {
    cout << code << endl;
  }
}

int main()
{
  string expression;
  cout << "Enter an expression (e.g., a = b + c * d): ";
  getline(cin, expression);
  generateIntermediateCode(expression);
  return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o intermediate_code intermediate_code.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./intermediate_code

Enter an expression (e.g., a = b + c * d):
a = b + c * d

Intermediate Code:
t1 = c * d
t2 = b + t1
a = t2

singhal-amit@singhal-amit-ThinkPad-T430:~$
```

PROGRAM – 15

❖ AIM :: WAP to minimize any given DFA.

Theory :: **Deterministic Finite Automaton (DFA) in Compiler Design**

Overview

A **Deterministic Finite Automaton (DFA)** is a computational model used in computer science and compiler design to recognize patterns within input strings. The DFA processes a string of symbols, transitioning between different states based on the input it reads, ultimately determining whether to accept or reject the string.

Key Characteristics of DFA

- **Deterministic Transitions**: For each state and input symbol, there is exactly one transition to a next state, ensuring a unique path through the states for a given input string. This means that the DFA behaves predictably and does not have any ambiguity in its state transitions.
- **State Acceptance**: The DFA has designated final states that determine whether the input string is accepted. If the input leads the DFA to a final state upon completion of reading the string, the input is accepted; otherwise, it is rejected.

Steps to Minimize a DFA

Minimizing a DFA involves reducing the number of states while preserving its language recognition capability. The following steps outline the minimization process:

1. **Remove Unreachable States**:
    - Identify and eliminate states that cannot be reached from the initial state. These states are irrelevant for the DFA's operation as they do not contribute to processing any accepted strings.
2. **Distinguish Final and Non-Final States**:
    - Partition the states into two distinct groups: final states (accepting states) and non-final states (rejecting states). This initial grouping forms the basis for further refinement.
3. **Refine State Partitions**:
    - Iteratively refine the state partitions based on their transitions. Two states are considered equivalent if, for every input symbol, they transition to the same partition of states. This process helps in identifying states that can be merged.
4. **Merge Equivalent States**:
    - Combine all equivalent states into a single state. This step reduces the overall number of states in the DFA and creates a more efficient automaton.
5. **Construct the Minimized DFA**:

- Define the new minimized DFA by specifying the new states and updating the transitions accordingly. The minimized DFA should have the same language acceptance properties as the original DFA but with fewer states.

Summary

The DFA is a foundational concept in compiler design, crucial for lexical analysis and pattern recognition. Its deterministic nature ensures predictable behavior when processing input strings. The process of minimizing a DFA enhances efficiency by reducing unnecessary complexity while preserving its language recognition capabilities. Through systematic partitioning and merging of states, compilers can create optimized DFAs that perform efficiently during the parsing and analysis of programming languages.

Code ::

```
#include <bits/stdc++.h>
using namespace std;

const int N = 109;
int t[N][N];
bool fs[N];
bool printed[N];
vector<vector<int>> pi[2];
int n, m, nfs;

int idx(int x)
{
  for (int i = 0; i < pi[0].size(); i++)
  {
    for (int j = 0; j < pi[0][i].size(); j++)
    {
      if (pi[0][i][j] == x)
      {
        return i;
      }
    }
  }
  return -1;
}
```

```
void print_min()
{
  cout << "\nMinimized DFA Table:\n";
  cout << "----------------------\n";
  cout << "Q\t";
  for (int j = 0; j < m; j++)
  {
    cout << j << "\t";
  }
  cout << endl
       << endl;

  for (int i = 0; i < n; i++)
  {
    if (printed[i])
      continue;
    int ind = idx(i);
    cout << "[";
    for (int k = 0; k < pi[0][ind].size(); k++)
    {
      cout << pi[0][ind][k];
      printed[pi[0][ind][k]] = true;
      if (k != pi[0][ind].size() - 1)
        cout << ",";
    }
    cout << "]\t";
    for (int j = 0; j < m; j++)
    {
      ind = idx(t[i][j]);
      cout << "[";
      for (int k = 0; k < pi[0][ind].size(); k++)
      {
        cout << pi[0][ind][k];
        if (k != pi[0][ind].size() - 1)
          cout << ",";
      }
      cout << "]\t";
    }
    cout << endl;
```

```
  }
  cout << endl;
}

int main()
{
  cout << "Enter number of states, input symbols, and final states: ";
  cin >> n >> m >> nfs;

  cout << "Enter the final states: ";
  for (int i = 0; i < nfs; i++)
  {
    int x;
    cin >> x;
    fs[x] = true;
  }

  cout << "Enter the transition table (state x input -> next state):\n";
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < m; j++)
    {
      cin >> t[i][j];
    }
  }

  vector<int> v[2];
  for (int i = 0; i < n; i++)
  {
    v[fs[i]].push_back(i);
  }
  pi[0].push_back(v[0]);
  pi[0].push_back(v[1]);
  pi[1] = pi[0];

  int iters = 100;
  while (iters--)
  {
    pi[0] = pi[1];
```

```
for (int i = 0; i < pi[1].size(); i++)
{
  for (int j = 1; j < pi[1][i].size(); j++)
  {
    bool placed = false;
    for (int k = 0; k < j; k++)
    {
      bool compatible = true;
      for (int l = 0; l < m; l++)
      {
        if (idx(t[pi[1][i][j]][l]) != idx(t[pi[1][i][k]][l]))
        {
          compatible = false;
          break;
        }
      }
      if (compatible)
      {
        placed = true;
        break;
      }
    }
    if (!placed)
    {
      bool found = false;
      for (int x = pi[0].size(); x < pi[1].size(); x++)
      {
        for (int k = 0; k < pi[1][x].size(); k++)
        {
          bool compatible = true;
          for (int l = 0; l < m; l++)
          {
            if (idx(t[pi[1][i][j]][l]) != idx(t[pi[1][x][k]][l]))
            {
              compatible = false;
              break;
            }
          }
          if (compatible)
```

```
            {
              pi[1][x].push_back(pi[1][i][j]);
              pi[1][i].erase(pi[1][i].begin() + j);
              placed = true;
              found = true;
              break;
            }
          }
          if (found)
            break;
        }
        if (!found)
        {
          vector<int> newClass = {pi[1][i][j]};
          pi[1].push_back(newClass);
          pi[1][i].erase(pi[1][i].begin() + j);
        }
      }
    }
  }

  for (int i = 0; i < pi[0].size(); i++)
  {
    sort(pi[0][i].begin(), pi[0][i].end());
  }
  sort(pi[0].begin(), pi[0].end());
  for (int i = 0; i < pi[1].size(); i++)
  {
    sort(pi[1][i].begin(), pi[1][i].end());
  }
  sort(pi[1].begin(), pi[1].end());
  if (pi[0] == pi[1])
    break;
 }

 assert(pi[0] == pi[1]);
 print_min();
 return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o minimize_dfa minimize_dfa.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./minimize_dfa

Enter number of states, input symbols, and final states: 4 2 2
Enter the final states: 2 3

Enter the transition table (state x input -> next state):
0 1
1 1
2 3
3 0
0 0
1 1
2 2
3 3

Minimized DFA Table:
---------------------
Q       Q0      Q1
[0]     [0]     [1]
[1]     [1]     [1]
[2]     [2]     [3]
[3]     [0]     [0]

singhal-amit@singhal-amit-ThinkPad-T430:~$
```

PROGRAM – 16

❖ AIM :: WAP to convert NFA to DFA.

Theory :: **Non-Deterministic Finite Automaton (NFA) in Compiler Design**

Overview

A **Non-Deterministic Finite Automaton (NFA)** is a type of finite state machine that allows for multiple possible transitions for a given state and input symbol. Unlike a **Deterministic Finite Automaton (DFA)**, which has a single unique transition for each state and input, an NFA can have:

- **Multiple Next States**: For a particular state and input, the NFA may transition to several different states or none at all.
- **Epsilon (ε) Transitions**: These are transitions that occur without consuming any input symbol, allowing the automaton to change states freely.

Key Characteristics of NFA

- **Multiple Paths**: An NFA can explore multiple paths for a given input string, leading to a more flexible recognition of patterns.
- **Acceptance Criteria**: An NFA accepts a string if at least one of the possible paths leads to a final state after consuming the input.

Steps to Convert an NFA to a DFA

The conversion from an NFA to a DFA involves creating a deterministic model that can recognize the same language as the original NFA. The following steps outline this conversion process:

1. **Start State**:
    - Begin with the **ε-closure** of the NFA's start state. The ε-closure includes all states reachable from the start state through ε transitions.
2. **Transitions**:
    - For each state in the newly created DFA, compute its transitions by considering all possible NFA states reachable via each input symbol. This means evaluating which states can be reached for every input from the current set of NFA states.
3. **New States**:
    - Add new DFA states for any sets of NFA states that are not yet represented in the DFA. Each unique set of NFA states corresponds to a single DFA state.
4. **Final States**:
    - Designate any DFA state that includes an NFA final state as a final state in the DFA. This ensures that the DFA recognizes the same strings as the NFA.
5. **Repeat**:
    - Continue the process until no new DFA states can be generated. This iterative approach ensures that all reachable states are accounted for in the DFA.

Summary

The NFA is a versatile model in compiler design that enables efficient pattern recognition and processing through its non-deterministic nature. The conversion from an NFA to a DFA results in a deterministic automaton that can operate without ambiguity while maintaining the same language acceptance characteristics. This conversion is essential for practical implementations of automata in compilers, particularly in lexical analysis, where DFAs are commonly used to parse input efficiently.

Code ::

```
#include <iostream>
#include <vector>
#include <string>
#include <queue>
#include <algorithm>
#include <set>

using namespace std;

void printNFA(const vector<vector<vector<int>>> &nfa)
{
  cout << "##################### TRANSITION TABLE of NFA ####################\n";
  cout << "    INPUT  |";
  for (char c = 'a'; c < 'a' + nfa[0].size(); ++c)
  {
    cout << " " << c << " |";
  }
  cout << "\nSTATE\n";
  for (int i = 0; i < nfa.size(); ++i)
  {
    cout << " q" << i << "        |";
    for (const auto &transitions : nfa[i])
    {
      if (transitions.empty())
      {
        cout << " ∅ ";
      }
```

```
      else
      {
        for (int j = 0; j < transitions.size(); ++j)
        {
          cout << " q" << transitions[j];
          if (j < transitions.size() - 1)
            cout << " ";
        }
      }
      cout << " |";
    }
    cout << "\n";
  }
}

void printDFA(const vector<vector<int>> &dfa, const vector<vector<int>> &states)
{
  cout << "##################### TRANSITION TABLE of DFA ####################\n";
  cout << "    INPUT  |";
  for (char c = 'a'; c < 'a' + dfa[0].size(); ++c)
  {
    cout << " " << c << " |";
  }
  cout << "\nSTATE\n";
  for (int i = 0; i < dfa.size(); ++i)
  {
    cout << " ";
    for (int state : states[i])
    {
      cout << "q" << state << " ";
    }
    cout << "|";
    for (const auto &transitions : dfa[i])
    {
```

```
      if (transitions.empty())
      {
        cout << " ∅ ";
      }
      else
      {
        for (int j = 0; j < transitions.size(); ++j)
        {
          cout << " q" << transitions[j];
          if (j < transitions.size() - 1)
            cout << " ";
        }
      }
      cout << " |";
    }
    cout << "\n";
  }
}

vector<int> closure(int state, const vector<vector<vector<int>>> &nfa)
{
  vector<int> result = {state};
  queue<int> q;
  q.push(state);

  while (!q.empty())
  {
    int current = q.front();
    q.pop();

    for (int nextState : nfa[current][nfa[current].size() - 1])
    {
      if (find(result.begin(), result.end(), nextState) == result.end())
      {
```

```
        result.push_back(nextState);
        q.push(nextState);
      }
    }
  }
  return result;
}

int main()
{
  int n, alpha;
  cout << "###################### NFA to DFA Conversion ########################\n";
  cout << "Enter total number of states in NFA :: ";
  cin >> n;
  cout << "Enter number of elements in Alphabet (a,b,c,...) :: ";
  cin >> alpha;

  vector<vector<vector<int>>> nfa(n);
  for (int i = 0; i < n; ++i)
  {
    cout << "---------------------- For state :: q" << i << " ----------------------\n";
    for (int j = 0; j < alpha; ++j)
    {
      char inputChar = 'a' + j;
      int outputStates;
      cout << "δ(q" << i << "," << inputChar << ") goes to no. of output states :: ";
      cin >> outputStates;

      cout << "Enter output states :: ";
      for (int k = 0; k < outputStates; ++k)
      {
        int state;
        cin >> state;
        nfa[i][j].push_back(state);
```

```
    }
  }

  // Handle ε-transitions
  int epsilonOutputStates;
  cout << "δ(q" << i << ",ε) goes to no. of output states :: ";
  cin >> epsilonOutputStates;

  cout << "Enter output states :: ";
  for (int k = 0; k < epsilonOutputStates; ++k)
  {
    int state;
    cin >> state;
    nfa[i].emplace_back(vector<int>{state});
  }
}

printNFA(nfa);

// Construct the DFA
vector<vector<int>> dfa;
vector<vector<int>> states;

states.push_back(closure(0, nfa));
dfa.push_back(vector<int>(alpha, -1));

queue<vector<int>> q;
q.push(states[0]);

while (!q.empty())
{
  vector<int> currentStates = q.front();
  q.pop();

  for (int i = 0; i < alpha; ++i)
```

```
    {
      vector<int> nextStates;
      for (int state : currentStates)
      {
        for (int nextState : nfa[state][i])
        {
          for (int closureState : closure(nextState, nfa))
          {
            if (find(nextStates.begin(), nextStates.end(), closureState) == nextStates.end())
            {
              nextStates.push_back(closureState);
            }
          }
        }
      }

      // Check if this combination of states already exists
      auto it = find(states.begin(), states.end(), nextStates);
      if (it != states.end())
      {
        dfa.push_back({it - states.begin()});
      }
      else if (!nextStates.empty())
      {
        states.push_back(nextStates);
        dfa.push_back({});
        q.push(nextStates);
      }
      dfa[states.size() - 1].push_back(nextStates);
    }
  }

  printDFA(dfa, states);
  return 0;
}
```

Output ::

```
singhal-amit@singhal-amit-ThinkPad-T430:~$ g++ -o a nfa_to_dfa.cpp
singhal-amit@singhal-amit-ThinkPad-T430:~$ ./a

###################### NFA to DFA Conversion ########################

Enter total number of states in NFA :: 2
Enter number of elements in Alphabet (a,b,c,...) :: 2

---------------------- For state :: q0 ---------------------

δ(q0,a) goes to no. of output states :: 1
Enter output states :: 0

δ(q0,b) goes to no. of output states :: 2
Enter output states :: 0 1

---------------------- For state :: q1 ---------------------

δ(q1,a) goes to no. of output states :: 0

δ(q1,b) goes to no. of output states :: 0

#################### TRANSITION TABLE of NFA ###################

    INPUT  | a  |  b    |
STATE
 q0        | q0 | q0 q1 |
 q1        | ∅  | ∅     |

#################### TRANSITION TABLE of DFA ###################

    INPUT  | a     |  b |
STATE
 q0        | q0 | q0 q1 |
 q0 q1     | q0 | q0 q1 |

singhal-amit@singhal-amit-ThinkPad-T430:~$
```