Sriyanshu Azad 07117702722 CSE-B

Experiment 1

Aim: Write a Lexical Analyzer program that identifies any 10 keywords from C language and identifiers following all the naming conventions of the C program

Theory:

A **lexical analyzer**, also known as a **lexer** or **scanner**, is a program that processes the input source code and converts it into a sequence of tokens. These tokens represent syntactic constructs such as keywords, identifiers, constants, and operators.

In C, a lexical analyzer performs:

Tokenization: It breaks down the source code into tokens.

- **Keywords**: Reserved words in C that have predefined meanings (e.g., `int`, `if`, `else`).
- **Identifiers**: Names used for variables, functions, or arrays. These must follow naming conventions.
    - Begin with a letter (A-Z or a-z) or an underscore (_).
    - The subsequent characters can be letters, digits (0-9), or underscores.
    - Identifiers are case-sensitive.
    - They must not match any C keywords.
    - Eg: main, variable1, foo_bar, _temp
- **Operators**: Symbols like `+`, `-`, `*`, and `=`.
- **Literals**: Constant values like numbers or strings.
- **Punctuation**: Characters like `;`, `,`, and `{}` that define structure.

Lexical Analysis Steps:

1. Read the input source code character by character.
2. Detect valid keywords and identifiers by comparing the input against known keywords and applying identifier rules.
3. Return the detected tokens for further processing (e.g., by a parser).

Structure of a Lex Program:

A Lex program has three sections, separated by `%%`:

1. **Declarations Section**:
    - Here, you define any global variables, header files (e.g., `#include`), and regular expressions for tokens.
2. **Rules Section**:
    - This section contains regular expression rules for token patterns and the actions to perform when a match is found.
    - Each line consists of a regular expression followed by C code, which gets executed when the pattern is matched.

3. **User Code Section**:
    - Any additional C code, such as the `main()` function, is defined here. You can write custom functions to handle specific tasks.

How Lex Works with Yacc:

- Lex is often used alongside **Yacc** (Yet Another Compiler Compiler), which is a parser generator. Lex identifies tokens, and Yacc performs the **syntax analysis** on these tokens.
- **Workflow**:
    1. Lex analyzes the input to produce tokens.
    2. Yacc processes these tokens to build a syntax tree or check for grammar rules.

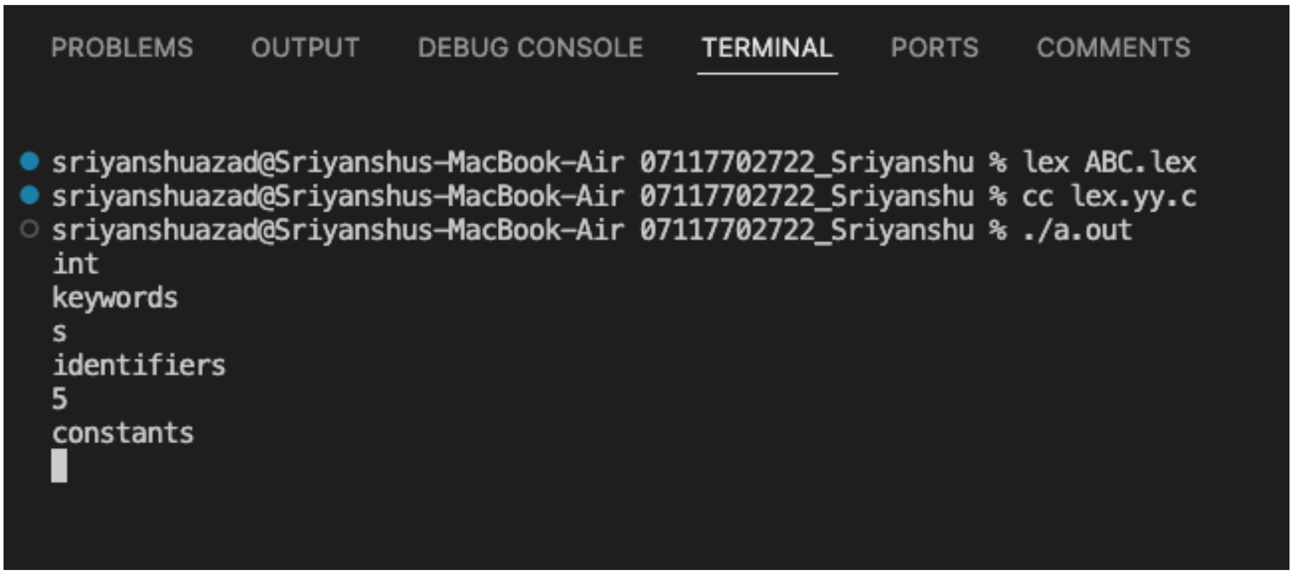
Advantages of Lex:

1. **Automation**: Lex generates efficient C code for lexical analysis, saving the programmer from writing a manual scanner.
2. **Pattern Matching**: It uses regular expressions, which are more concise and readable than traditional manual scanning code.
3. **Integration**: Works well with Yacc for building complete language processors (e.g., compilers, interpreters).

Sample Code:

```
%option noyywrap
%%
boolean|float|int|if|char printf("keywords");
[0-9][0-9]* printf("constants");
[a-zA-Z][a-zA-Z0-9]* printf("identifiers");
%%

int main()
{

     yylex();

}
```

Sample output:

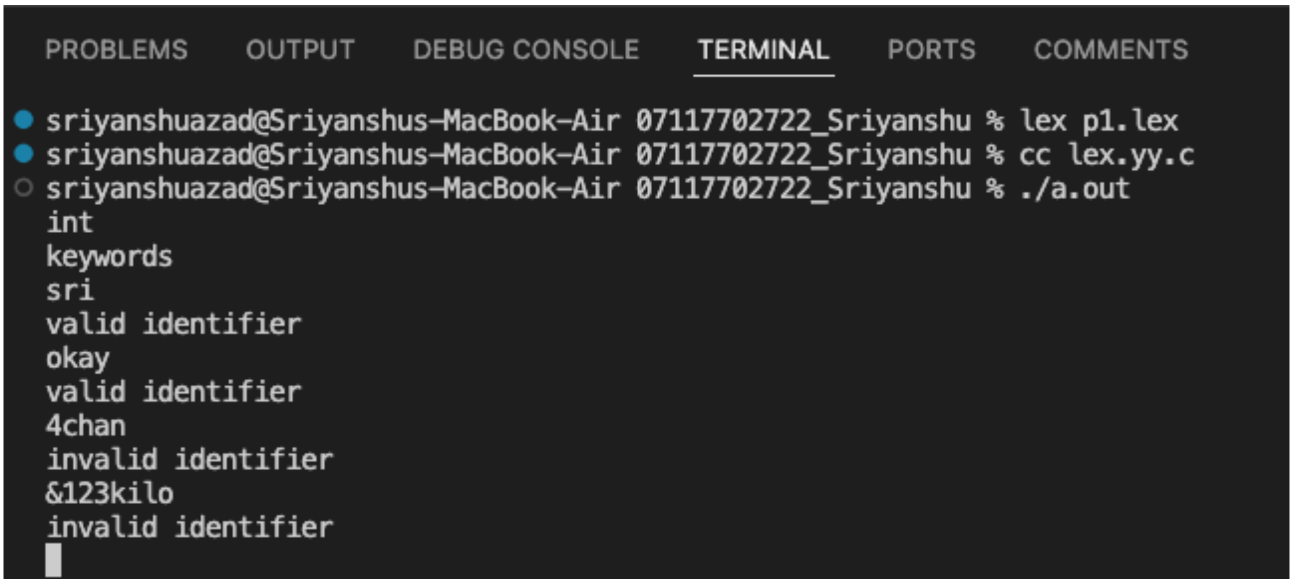


```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % lex ABC.lex
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % cc lex.yy.c
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./a.out
int
keywords
s
identifiers
5
constants
```

Code:

```
%option noyywrap
%%
break|float|int|if|char|for|return|void|struct|else
printf("keywords");
^[a-zA-Z_][a-zA-Z0-9_]* printf("valid identifier");
^[^a-zA-Z_]+[a-zA-Z0-9_]* printf("invalid identifier");
.;
%%

int main()
{

   yylex();

}
```

Output:



```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % lex p1.lex
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % cc lex.yy.c
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./a.out
int
keywords
sri
valid identifier
okay
valid identifier
4chan
invalid identifier
&123kilo
invalid identifier
```

Learning Outcomes:

Experiment 2

Aim: Write a C program that takes as input string from the text file (let's say input.txt), and identifies and counts the frequency of the keywords appearing in that string.

Theory:

1. Keywords in C:

- **Keywords** are reserved words in the C language that have special meanings and cannot be used as identifiers (e.g., `int`, `return`, `for`, etc.).
- The keywords in C are: `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, `while`

2. Reading Input from a File:

- To analyze the text, the program must read a string from a file. This can be done using functions like `fopen()` to open the file, `fgets()` or `fscanf()` to read from it, and `fclose()` to close the file.

3. Tokenization:

- To identify words in the input string, the program will use a **tokenizer**. A common way to tokenize strings is by using `strtok()` which breaks the input string into tokens (words) based on a set of delimiters (e.g., spaces, punctuation, etc.).

4. Keyword Matching:

- Once the input is tokenized, the program will compare each word with a list of predefined C keywords to determine if the word is a keyword.

5. Counting Frequency:

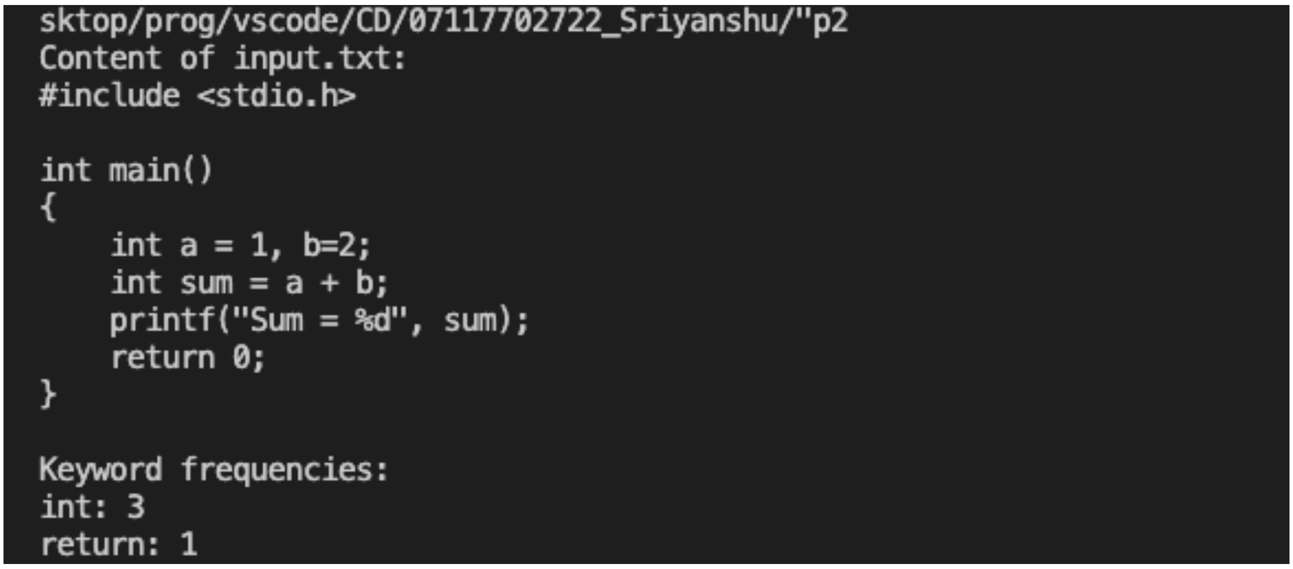
- The program will maintain a count of how many times each keyword appears in the input. This can be done using an array that tracks the frequency of each keyword.

Code:

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

```
// List of 32 C keywords
const char *keywords[] = {
    "auto", "break", "case", "char", "const", "continue",
"default", "do",
    "double", "else", "enum", "extern", "float", "for", "goto",
"if",
    "int", "long", "register", "return", "short", "signed",
"sizeof",
    "static", "struct", "switch", "typedef", "union", "unsigned",
"void",
    "volatile", "while"
};
#define NUM_KEYWORDS (sizeof(keywords) / sizeof(keywords[0]))

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

int main() {
    FILE *file;
    char filename[] = "input.txt";
    char buffer[1000];
    int keyword_count[NUM_KEYWORDS] = {0};

    file = fopen(filename, "r");
    if (file == NULL) {
        printf("Could not open file %s\n", filename);
        return 1;
    }

    printf("Content of %s:\n", filename);
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);

        // Process the current line for keywords
        char *token = strtok(buffer, " \t\n.,;(){}[]<>+-/*&|^%=!
~");
        while (token != NULL) {
            // Check if the token is a keyword
            if (is_keyword(token)) {
                for (int i = 0; i < NUM_KEYWORDS; i++) {
                    if (strcmp(token, keywords[i]) == 0) {
                        keyword_count[i]++;
                        break;
                    }
                }
            }
            token = strtok(NULL, " \t\n.,;(){}[]<>+-/*&|^%=!~");
        }
    }

    fclose(file);

    printf("\n\nKeyword frequencies:\n");
    for (int i = 0; i < NUM_KEYWORDS; i++) {
        if (keyword_count[i] > 0) {
            printf("%s: %d\n", keywords[i], keyword_count[i]);
        }
    }

    return 0;
}
```

Output:



```
sktop/prog/vscode/CD/07117702722_Sriyanshu/"p2
Content of input.txt:
#include <stdio.h>

int main()
{
    int a = 1, b=2;
    int sum = a + b;
    printf("Sum = %d", sum);
    return 0;
}

Keyword frequencies:
int: 3
return: 1
```

Learning Outcomes:

Experiment 3

Aim: Write a Syntax Analyzer program using Yacc tool that will have grammar rules for the operators : *,/,%.

Theory:

A **syntax analyzer** or **parser** is a component of a compiler that takes the output of the lexical analyzer (i.e., a stream of tokens) and builds a syntax tree based on the grammar rules of the programming language. For this task, we are going to create a syntax analyzer using **Yacc (Yet Another Compiler Compiler)**, which will analyze expressions involving the operators `*`, `/`, and `%`.

Key Concepts:

1. **Yacc**:
    - **Yacc** is a parser generator that works with **LALR(1)** (Look-Ahead LR) parsing. It takes a set of **grammar rules** and generates a C program that can parse tokens from the lexical analyzer.
    - Yacc grammar consists of **non-terminals**, **terminals (tokens)**, and **production rules** that define how valid expressions are formed in the language.
    - The Yacc-generated parser interacts with a **Lex scanner** (for lexical analysis) to receive tokens.
2. **Grammar Rules**:
    - The grammar defines how operators `*`, `/`, and `%` are used in expressions. These operators are typically associated with **multiplicative precedence** in programming languages, meaning they are evaluated before addition and subtraction.
    - We define grammar rules in **Backus-Naur Form (BNF)** to specify valid expressions, with each rule representing part of the syntax structure.
3. **Lex**:
    - **Lex** is used to generate the lexical analyzer (scanner). It tokenizes the input and recognizes keywords, operators, and identifiers.
    - Lex interacts with Yacc by passing **tokens**. For example, `*`, `/`, and `%` would be passed as tokens from Lex to Yacc.
4. **Action and Semantic Values**:
    - The rules in Yacc are associated with **actions** that are executed when a rule is recognized. These actions generally involve operations on the semantic values of tokens, such as evaluating expressions or storing values.
    - **Semantic values** (in Yacc) allow us to perform computations during parsing (e.g., evaluating the result of `3 * 4`).

Sample Code:

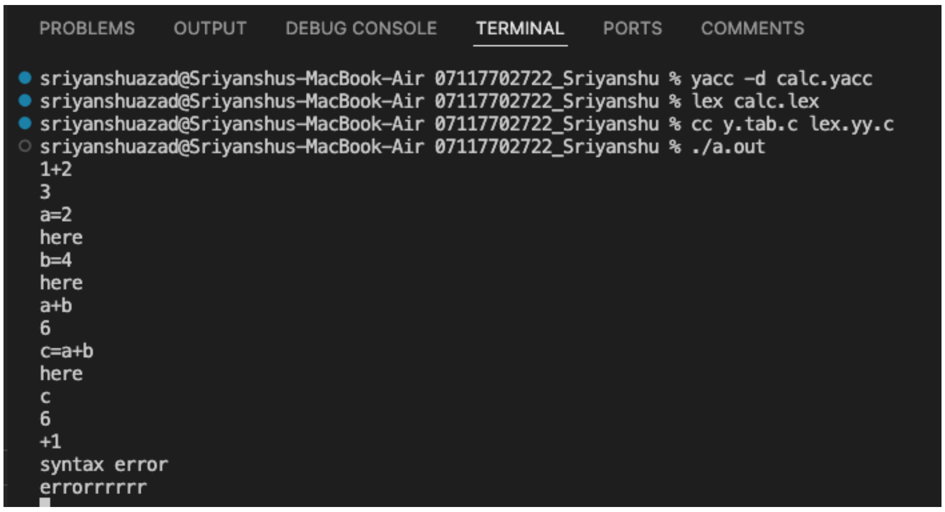
calc.yacc

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

int yylex(void);
void yyerror(const char *s);

int regs[26];
int base;
%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%%                /* beginning of rules section */
list:                    /*empty */
        |
        list stat '\n'
        |
        list error '\n'
         {
          printf("errorrrrrr\n");
          yyerrok;
         }
         ;
stat:    expr
         {
          printf("%d\n",$1);
         }
         |
         LETTER '=' expr
         {
          printf("here\n");
          regs[$1] = $3;
         }
         ;
expr:    '(' expr ')'
         {
          $$ = $2;
         }
         |
         expr '+' expr
         {
          $$ = $1 + $3;
         }
          |
         expr '-' expr
         {
          $$ = $1 - $3;
         }
         |
         LETTER
         {
          $$ = regs[$1];
         }
         |
         number
         ;
number:  DIGIT
         {
          //printf("first here!!");
          $$ = $1;
          base = ($1==0) ? 8 : 10;
         }       |
         number DIGIT
         {
          printf("%d %d \n", $1, $2);
          $$ = base * $1 + $2;
         }
         ;
%%
int main()
{
 return(yyparse());
}
void yyerror(const char *s)
{
  fprintf(stderr, "%s\n",s);
}
int yywrap()
{
  return(1);
}
```

calc.lex

```
%{

#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "       ;
[a-z]     {
           c = yytext[0];
           yylval = c - 'a';
           return(LETTER);
          }
[0-9]     {
           c = yytext[0];
           yylval = c - '0';
           return(DIGIT);
          }
[^a-z0-9\b]    {
                c = yytext[0];
                return(c);
               }
```

Sample Output:



```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % yacc -d calc.yacc
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % lex calc.lex
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % cc y.tab.c lex.yy.c
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./a.out
1+2
3
a=2
here
b=4
here
a+b
6
c=a+b
here
c
6
+1
syntax error
errorrrrrr
```

Code:

calc.yacc

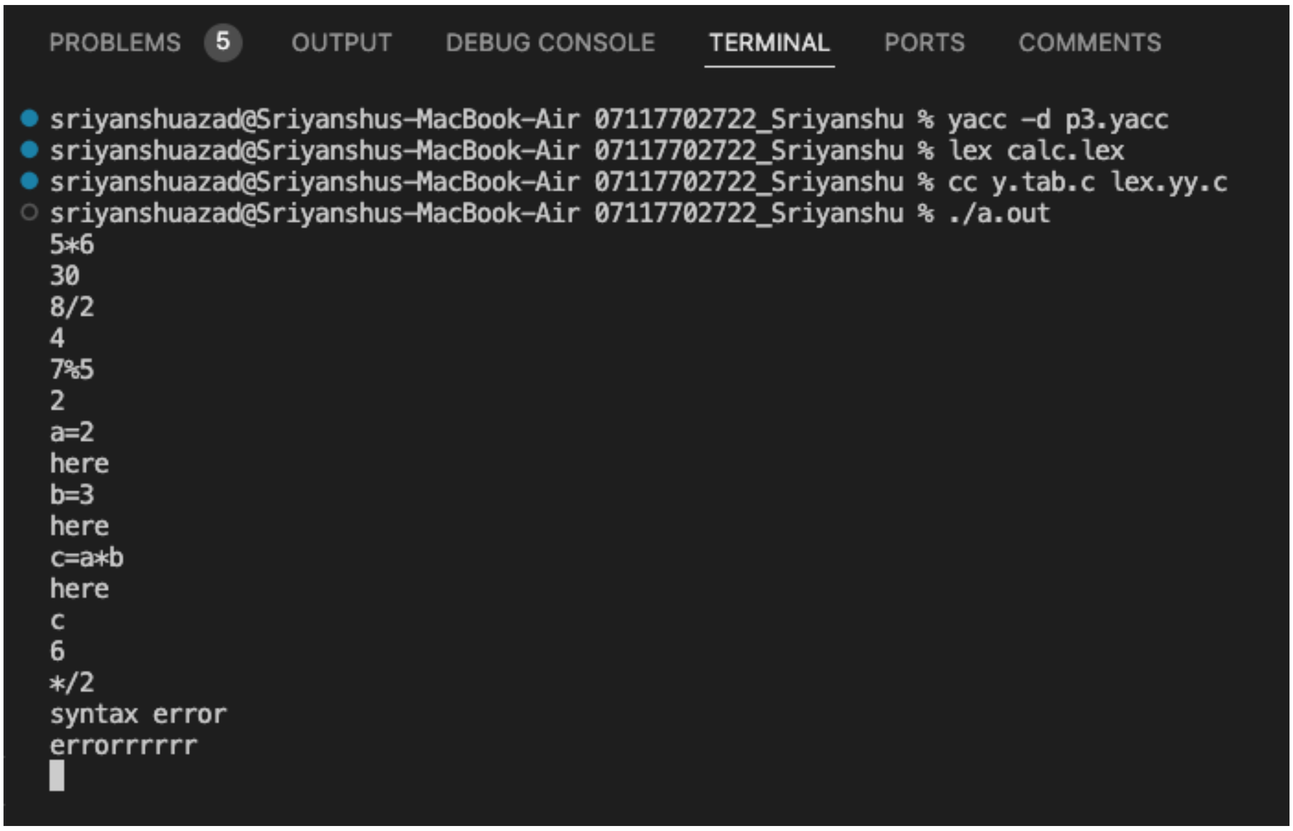
```
%{
#include <stdio.h>

int yylex(void);
void yyerror(const char *s);

int regs[26];
int base;
%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%%                  /* beginning of rules section */
list:                      /*empty */
        |
        list stat '\n'
        |
        list error '\n'
         {
          printf("errorrrrrr\n");
          yyerrok;
         }
         ;
stat:    expr
         {
          printf("%d\n",$1);
         }
         |
         LETTER '=' expr
         {
          printf("here\n");
          regs[$1] = $3;
         }
         ;
expr:    '(' expr ')'
         {
          $$ = $2;
         }
         |
         expr '+' expr
         {
          $$ = $1 + $3;
         }
          |
         expr '-' expr
         {
          $$ = $1 - $3;
         }
         |
         expr '*' expr
         {
          $$ = $1 * $3;
         }
          |
         expr '/' expr
         {
          $$ = $1 / $3;
         }
         |
         expr '%' expr
         {
          $$ = $1 % $3;
         }
         |
         LETTER
         {
          $$ = regs[$1];
         }
         |
         number
         ;
number:  DIGIT
         {
          //printf("first here!!");
          $$ = $1;
          base = ($1==0) ? 8 : 10;
         }       |
         number DIGIT
         {
          printf("%d %d \n", $1, $2);
          $$ = base * $1 + $2;
         }
         ;
%%
int main()
{
 return(yyparse());
}
void yyerror(const char *s)
{
  fprintf(stderr, "%s\n",s);
}
int yywrap()
{
  return(1);
}
```

calc.lex

```
%{

#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "       ;
[a-z]     {
            c = yytext[0];
            yylval = c - 'a';
            return(LETTER);
           }
[0-9]     {
```

```
                c = yytext[0];
                yylval = c - '0';
                return(DIGIT);
              }
    [^a-z0-9\b]    {
                     c = yytext[0];
                     return(c);
                   }
```

Output:



```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % yacc -d p3.yacc
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % lex calc.lex
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % cc y.tab.c lex.yy.c
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./a.out
5*6
30
8/2
4
7%5
2
a=2
here
b=3
here
c=a*b
here
c
6
*/2
syntax error
errorrrrrr
```

Learning Outcomes:

Experiment 4

Aim: To write a C program that takes the single line production rule in a string as input and checks if it has Left-Recursion or not and give the unambiguous grammar, in case, if it has Left- Recursion.

Theory:

In context-free grammars, **left recursion** occurs when a non-terminal in a production rule refers to itself as the first symbol on the right-hand side. Left recursion can lead to non-termination or inefficiencies when parsing, especially with **top-down parsers** like **recursive descent parsers**. Hence, it is necessary to detect and eliminate left recursion in grammars to make them **unambiguous** and suitable for efficient parsing.

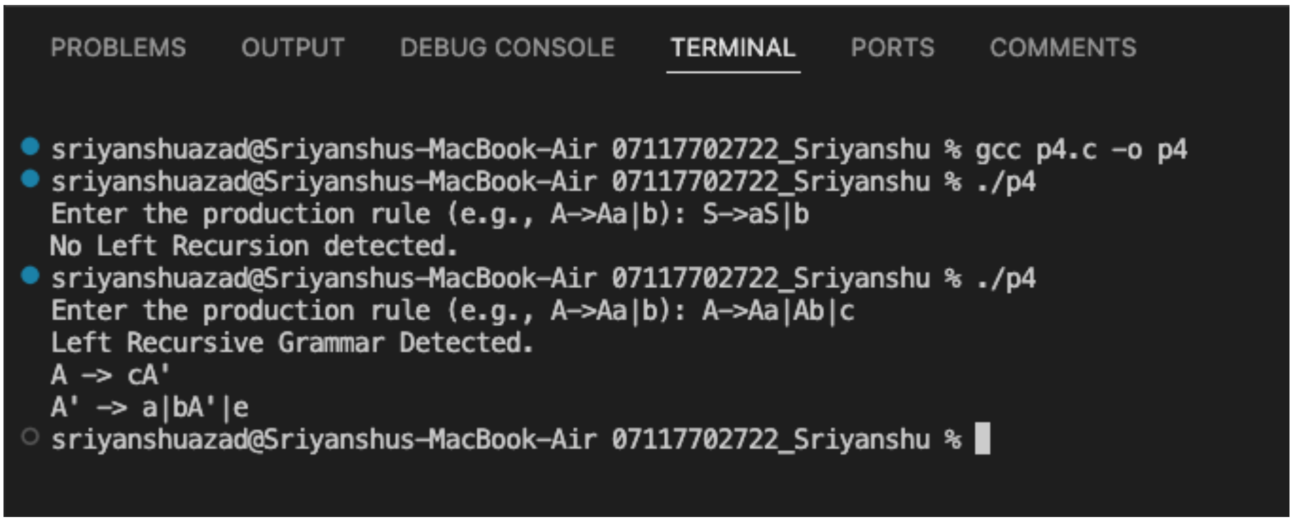
Key Concepts:

1. **Context-Free Grammar**:
   - A **context-free grammar** (CFG) is a set of rules that define the structure of valid strings for a language. Each rule consists of a **non-terminal symbol** on the left-hand side and a combination of terminal and non-terminal symbols on the right-hand side.
   - A production rule in CFG is typically written as: `A → Aα | β`, where `A` is a non-terminal, `α` and `β` are strings of terminal and/or non-terminal symbols.
2. **Left Recursion**:
   - **Left recursion** occurs when a non-terminal symbol on the left-hand side of a production rule appears as the first symbol on the right-hand side. For example, in the rule `A → Aα`, the non-terminal `A` refers to itself.
   - Example:
     - `A → Aα | β` (left recursive)
     - Here, `A` on the left appears first in the right-hand side of the production.
3. **Right Recursion**:
   - **Right recursion** occurs when a non-terminal symbol appears on the right-hand side but not in the first position. It doesn't cause issues for top-down parsers.
   - Example:
     - `A → βA | α` (right recursive)
4. **Elimination of Left Recursion**:
   - Left recursion can be eliminated by converting a left-recursive grammar into a **right-recursive** form. The general technique involves transforming:
     - `A → Aα | β`
     - into an equivalent **non-recursive form**:
       - `A → βA'`
       - `A' → αA' | ε`
     - Here, `A'` is a new non-terminal symbol, `α` is the recursive part, `β` is the non-recursive part, and `ε` represents the **empty string**.

Code:

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

#define MAX 100

void checkLeftRecursion(char nonTerminal, char *productions) {
    char alpha[MAX], beta[MAX];
    int isLeftRecursive = 0;

    char *token = strtok(productions, "|");
    int alphaIndex = 0, betaIndex = 0;

    while (token != NULL) {
        if (token[0] == nonTerminal) {
            isLeftRecursive = 1;
            strcpy(&alpha[alphaIndex], token + 1);
            alphaIndex += strlen(token + 1);
            alpha[alphaIndex++] = '|';
        } else {
            strcpy(&beta[betaIndex], token);
            betaIndex += strlen(token);
            beta[betaIndex++] = '|';
        }
        token = strtok(NULL, "|");
    }

    if (alphaIndex > 0) alpha[alphaIndex - 1] = '\0';
    if (betaIndex > 0) beta[betaIndex - 1] = '\0';

    if (isLeftRecursive) {
        printf("Left Recursive Grammar Detected.\n");
        printf("%c -> %s%c'\n", nonTerminal, beta, nonTerminal);
        printf("%c' -> %s%c'|e\n", nonTerminal, alpha,
nonTerminal);
    } else {
        printf("No Left Recursion detected.\n");
    }
}

int validateProductionRule(char *input) {
    if (strstr(input, "-->") != NULL) {
        printf("Error: '->' should not be greater than 3
characters long.\n");
        return 0;
    }

    if (!isupper(input[0]) || strncmp(input + 1, "->", 2) != 0) {
        printf("Error: Invalid production rule format.\n");
        return 0;
    }

    return 1;
}

int main() {
    char input[MAX], nonTerminal, productions[MAX];

    printf("Enter the production rule (e.g., A->Aa|b): ");
    fgets(input, MAX, stdin);
    input[strcspn(input, "\n")] = '\0';

    if (!validateProductionRule(input)) {
        return 0;
    }

    nonTerminal = input[0];
    strcpy(productions, input + 3);

    checkLeftRecursion(nonTerminal, productions);

    return 0;
}
```

Output:



```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % gcc p4.c -o p4
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p4
Enter the production rule (e.g., A->Aa|b): S->aS|b
No Left Recursion detected.
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p4
Enter the production rule (e.g., A->Aa|b): A->Aa|Ab|c
Left Recursive Grammar Detected.
A -> cA'
A' -> a|bA'|e
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu %
```

Learning Outcomes:

Experiment 5

Aim: To write a C program that takes the single line production rule in a string as input and checks if it has Left-Factoring or not and give the unambiguous grammar, in case, if it has Left- Factoring.

Theory:

In the design of parsers for context-free grammars, **left factoring** is a process used to transform a grammar to remove ambiguity and make it suitable for **top-down parsing**. Left factoring is necessary when two or more production rules for the same non-terminal begin with the same symbols, leading to **non-determinism**. This makes it difficult for parsers to decide which production to apply. The solution is to rewrite the grammar by factoring out the common prefix, resulting in an unambiguous grammar.

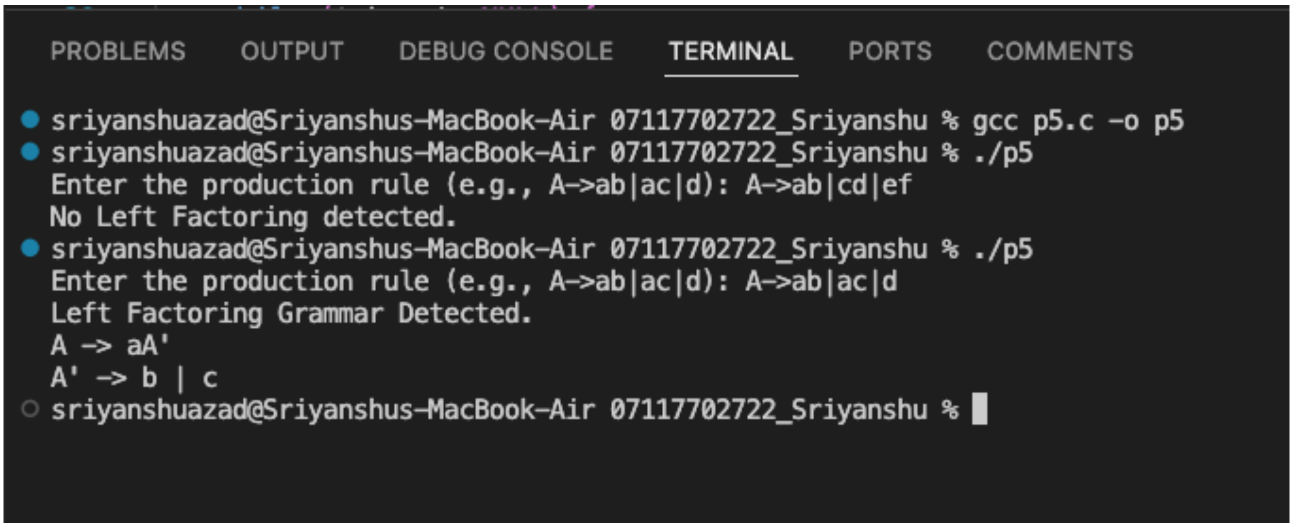
Key Concepts:

1. **Context-Free Grammar (CFG)**:
    - A **context-free grammar** consists of a set of production rules where each rule describes how a **non-terminal** can be expanded into a sequence of terminal and non-terminal symbols.
    - A production rule is typically written as `A → α | β`, where `A` is a non-terminal and α and β are sequences of terminal and/or non-terminal symbols.
2. **Left Factoring**:
    - **Left factoring** is a situation in grammar when two or more productions for the same non-terminal begin with the same prefix. For example:
        - `A → αβ1 | αβ2`
    - In this case, the prefix α is common to both productions, leading to **non-determinism**, as a top-down parser cannot decide which production to apply by looking only at α.
    - **Eliminating Left Factoring**:
        - To eliminate left factoring, we rewrite the grammar to factor out the common prefix, introducing a new non-terminal symbol. The grammar is transformed as follows:
            - Original: `A → αβ1 | αβ2`
            - Transformed: `A → αA'`
                - `A' → β1 | β2`
3. **Ambiguity in Grammar**:
    - A grammar is **ambiguous** if a string can have more than one valid parse tree. Left factoring is one of the causes of ambiguity and non-determinism in parsers, making it difficult to decide the correct production to apply.
4. **Why Left Factoring is Important**:
    - **Top-down parsers**, especially **LL(1) parsers**, require grammars to be left-factored to ensure they can select the appropriate production based on one lookahead token. Left factoring resolves the ambiguity by restructuring the grammar, making it **deterministic** and easy to parse.

Code:

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

#define MAX 100

void longestCommonPrefix(char* result, char* str1, char* str2) {
    int i = 0;
    while (str1[i] != '\0' && str2[i] != '\0' && str1[i] ==
str2[i]) {
        result[i] = str1[i];
        i++;
    }
    result[i] = '\0';
}

void checkLeftFactoring(char nonTerminal, char *productions) {
    char commonPrefix[MAX] = {0}, tempPrefix[MAX] = {0};
    char factoredPart[MAX] = {0};
    char remainingParts[MAX][MAX];
    int hasLeftFactoring = 0;
    int partCount = 0;

    char *token = strtok(productions, "|");

    strcpy(factoredPart, token);
    token = strtok(NULL, "|");

    while (token != NULL) {
        longestCommonPrefix(tempPrefix, factoredPart, token);

        if (strlen(tempPrefix) > 0) {
            hasLeftFactoring = 1;
            strcpy(commonPrefix, tempPrefix);
            strcpy(remainingParts[partCount++], factoredPart +
strlen(commonPrefix));
            strcpy(remainingParts[partCount++], token +
strlen(commonPrefix));
        }
        factoredPart[0] = '\0';
        token = strtok(NULL, "|");
    }


    if (hasLeftFactoring) {
        printf("Left Factoring Grammar Detected.\n");
        printf("%c -> %s%c'\n", nonTerminal, commonPrefix, 
nonTerminal);
        printf("%c' -> ", nonTerminal);
        for (int i = 0; i < partCount; i++) {
            if (strlen(remainingParts[i]) == 0) {
                printf("e");
            } else {
                printf("%s", remainingParts[i]);
            }
            if (i < partCount - 1) {
                printf(" | ");
            }
        }
        printf("\n");
    } else {
        printf("No Left Factoring detected.\n");
    }
}


int validateProductionRule(char *input) {
    if (strstr(input, "-->") != NULL) {
        printf("Error: '->' should not be greater than 3 
characters long.\n");
        return 0;
    }
    if (!isupper(input[0]) || strncmp(input + 1, "->", 2) != 0) {
        printf("Error: Invalid production rule format.\n");
        return 0;
    }

    return 1;
}

int main() {
    char input[MAX], nonTerminal, productions[MAX];

    printf("Enter the production rule (e.g., A->ab|ac|d): ");
    fgets(input, MAX, stdin);
    input[strcspn(input, "\n")] = '\0';

    if (!validateProductionRule(input)) {
        return 0;
    }

    nonTerminal = input[0];
    strcpy(productions, input + 3);

    checkLeftFactoring(nonTerminal, productions);

    return 0;
}
```

Output:



```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % gcc p5.c -o p5
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p5
Enter the production rule (e.g., A->ab|ac|d): A->ab|cd|ef
No Left Factoring detected.
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p5
Enter the production rule (e.g., A->ab|ac|d): A->ab|ac|d
Left Factoring Grammar Detected.
A -> aA'
A' -> b | c
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu %
```

Learning Outcomes:

Experiment 6

Aim: Write a program to find out the FIRST of the Non-terminals in a grammar.

Theory:

The FIRST set of a non-terminal symbol in a grammar is defined as the set of terminals (or tokens) that begin the strings derivable from that non-terminal. In simpler terms, it tells us which terminals can appear at the start of any string generated by that non-terminal.

Formal Definition

For a non-terminal A:

- FIRST(A) is the set of terminals a such that A can derive aα for some string α (where the symbol "→" denotes derivation).

Rules for Computing FIRST Sets

To compute the FIRST set for a grammar, follow these rules:

1. **For Terminal Symbols**:
   - If X is a terminal symbol, then: FIRST(X) = { X }
2. **For Epsilon**:
   - If A can derive the empty string (denoted as "e"), then: e is included in FIRST(A).
3. **For Non-terminals**:
   - For a non-terminal A:
     - If A can derive B1 B2 ... Bk, compute FIRST(B1):
       - If FIRST(B1) contains terminals, add those terminals to FIRST(A).
       - If FIRST(B1) contains e, continue to B2, and so forth, until you find a non-terminal that doesn't derive e or you exhaust all Bi.
     - If none of the Bi lead to terminals and A can derive e, add e to FIRST(A).
4. **Multiple Productions**:
   - For multiple productions of the same non-terminal, combine the FIRST sets of all productions.

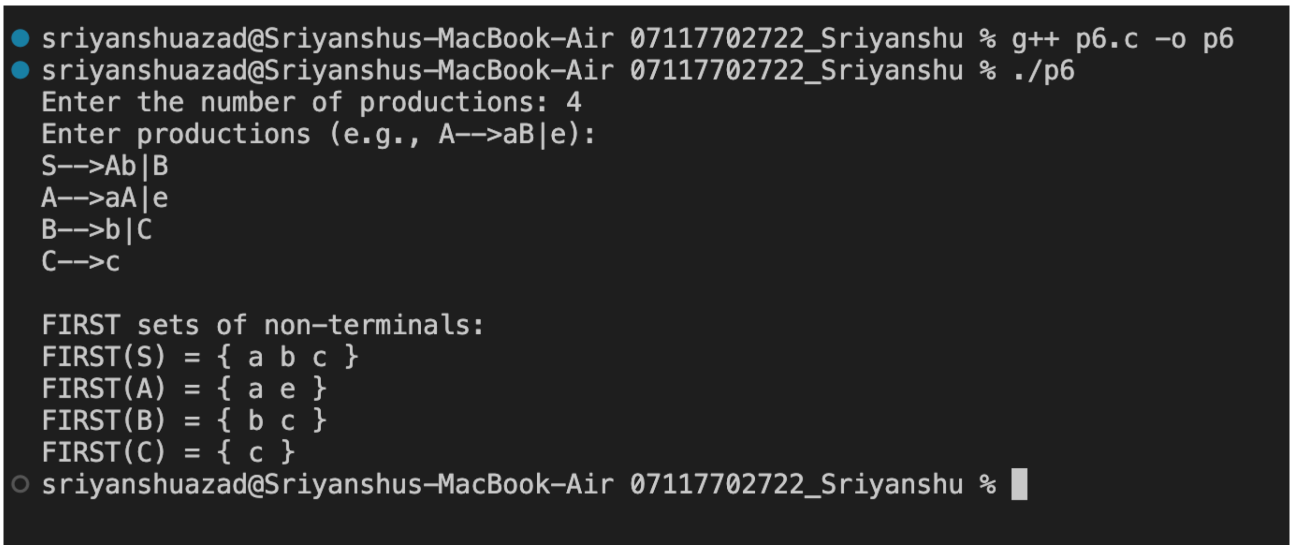
Importance of FIRST Sets

- **Parsing Decisions**: FIRST sets allow parsers to decide which production to apply based on the current input symbol.
- **Conflict Resolution**: In LL parsers, FIRST sets help identify potential conflicts in the grammar.
- **Efficiency**: FIRST sets can optimize the parsing process by reducing unnecessary computations.

Code:

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

#define MAX_NON_TERMINALS 10
#define MAX_PRODUCTIONS 10
#define MAX_SYMBOLS 10

typedef struct {
    char non_terminal;
    char production[MAX_SYMBOLS][MAX_SYMBOLS];
    int count;
} Production;

Production productions[MAX_PRODUCTIONS];
char first[MAX_NON_TERMINALS][MAX_SYMBOLS];
int production_count = 0, non_terminal_count = 0;

void addProduction(char non_terminal, char *prod) {
    int i;
    for (i = 0; i < production_count; i++) {
        if (productions[i].non_terminal == non_terminal) {
            strcpy(productions[i].production[productions[i].count+
+], prod);
            return;
        }
    }
    productions[production_count].non_terminal = non_terminal;

strcpy(productions[production_count].production[productions[produc
tion_count].count++], prod);
    production_count++;
}

int findNonTerminalIndex(char non_terminal) {
    for (int i = 0; i < non_terminal_count; i++) {
        if (first[i][0] == non_terminal)
            return i;
    }
    first[non_terminal_count++][0] = non_terminal;
    return non_terminal_count - 1;
}

int addToFirst(int index, char symbol) {
    for (int i = 1; first[index][i] != '\0'; i++) {
        if (first[index][i] == symbol)
            return 0;  // Symbol already exists, no change
    }
    int i = 1;
    while (first[index][i] != '\0') i++;
    first[index][i] = symbol;
    first[index][i + 1] = '\0';
    return 1;  // New symbol added
}

int computeFirstForProduction(int index, char *prod) {
    int addedEpsilon = 0;
    for (int k = 0; prod[k] != '\0'; k++) {
        if (islower(prod[k]) || prod[k] == 'e') {
            return addToFirst(index, prod[k]);
        } else {
            int nextIndex = findNonTerminalIndex(prod[k]);
            int containsEpsilon = 0;
            for (int l = 1; first[nextIndex][l] != '\0'; l++) {
                if (first[nextIndex][l] == 'e')
                    containsEpsilon = 1;
                else
                    addedEpsilon |= addToFirst(index, 
first[nextIndex][l]);
            }
            if (!containsEpsilon)
                return addedEpsilon;
            addedEpsilon = 1;
        }
    }
    return addedEpsilon;
}

void computeFirst() {
    int changes;
    do {
        changes = 0;
        for (int i = 0; i < production_count; i++) {
            int index = 
findNonTerminalIndex(productions[i].non_terminal);
            for (int j = 0; j < productions[i].count; j++) {
                changes |= computeFirstForProduction(index, 
productions[i].production[j]);
            }
        }
    } while (changes);
}

void displayFirstSets() {
    printf("\nFIRST sets of non-terminals:\n");
    for (int i = 0; i < non_terminal_count; i++) {
        printf("FIRST(%c) = { ", first[i][0]);
        for (int j = 1; first[i][j] != '\0'; j++) {
            printf("%c ", first[i][j]);
        }
        printf("}\n");
    }
}

int main() {
    int n;
    printf("Enter the number of productions: ");
    scanf("%d", &n);
    getchar();

    printf("Enter productions (e.g., A-->aB|e):\n");
    for (int i = 0; i < n; i++) {
        char line[100], *token;
        fgets(line, sizeof(line), stdin);
        line[strcspn(line, "\n")] = '\0';

        token = strtok(line, "-->");
        char non_terminal = token[0];

        token = strtok(NULL, "-->");
        char *production = strtok(token, "|");

        while (production) {
            addProduction(non_terminal, production);
            production = strtok(NULL, "|");
        }
    }

    computeFirst();
    displayFirstSets();

    return 0;
}
```

Output:



```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % g++ p6.c -o p6
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p6
Enter the number of productions: 4
Enter productions (e.g., A-->aB|e):
S-->Ab|B
A-->aA|e
B-->b|C
C-->c

FIRST sets of non-terminals:
FIRST(S) = { a b c }
FIRST(A) = { a e }
FIRST(B) = { b c }
FIRST(C) = { c }
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu %
```

Learning Outcomes:

Experiment 7

Aim: Write a program to Implement Shift Reduce parsing for a String.

Theory:

Shift-Reduce Parsing is a bottom-up parsing technique used in syntax analysis of programming languages. It constructs a parse tree for a given input string based on the production rules of a context-free grammar. This parsing method is particularly useful for implementing parsers in compilers and interpreters.

Basic Concepts

1. **Shift Operation**:
   - The **shift** operation moves a symbol from the input buffer to the stack. This means that the parser reads the next input symbol and adds it to the stack. This operation continues until a production rule can be applied.
2. **Reduce Operation**:
   - The **reduce** operation replaces a sequence of symbols on the stack that matches the right-hand side of a production rule with the corresponding non-terminal on the left-hand side of that rule. This effectively reduces the symbols on the stack to a single non-terminal, representing a higher-level structure in the parse tree.

Process of Shift-Reduce Parsing

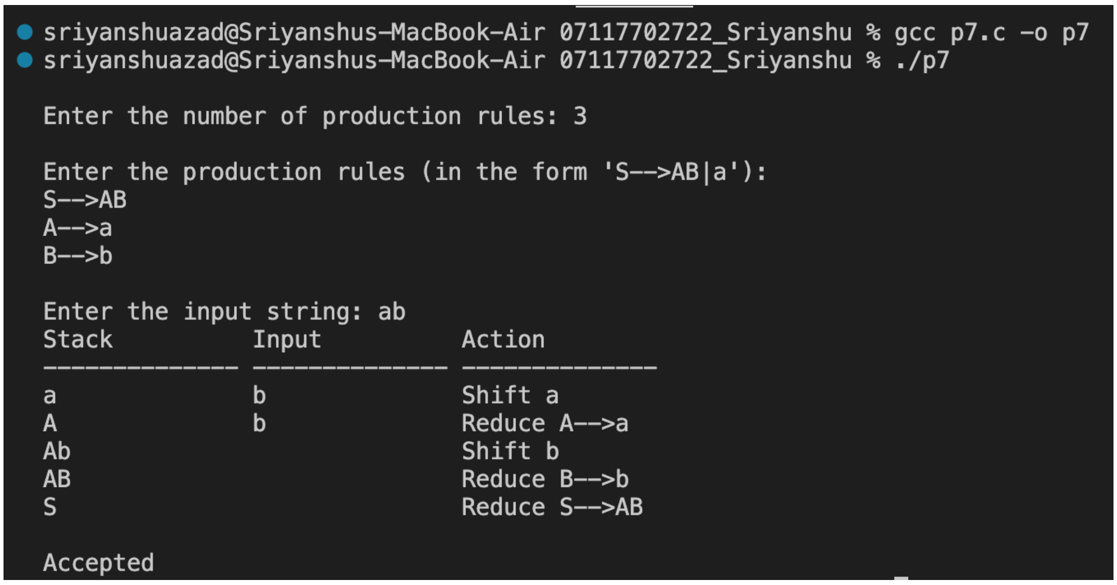
The parsing process involves the following steps:

1. **Initialization**:
   - Start with an empty stack and an input string that needs to be parsed.
2. **Shift Operation**:
   - If the stack does not contain a complete parse tree and there are still symbols in the input string, perform a shift operation by moving the next input symbol onto the stack.
3. **Reduce Operation**:
   - After each shift, check if the symbols on the stack match the right-hand side of any production rule. If there is a match, perform the reduce operation, replacing the matched symbols with the corresponding non-terminal.
4. **Repeat**:
   - Repeat the shift and reduce operations until the input is fully consumed and the stack contains only the start symbol of the grammar.
5. **Acceptance or Rejection**:
   - If the stack contains only the start symbol and the input string has been completely parsed, the input is accepted. Otherwise, it is rejected.

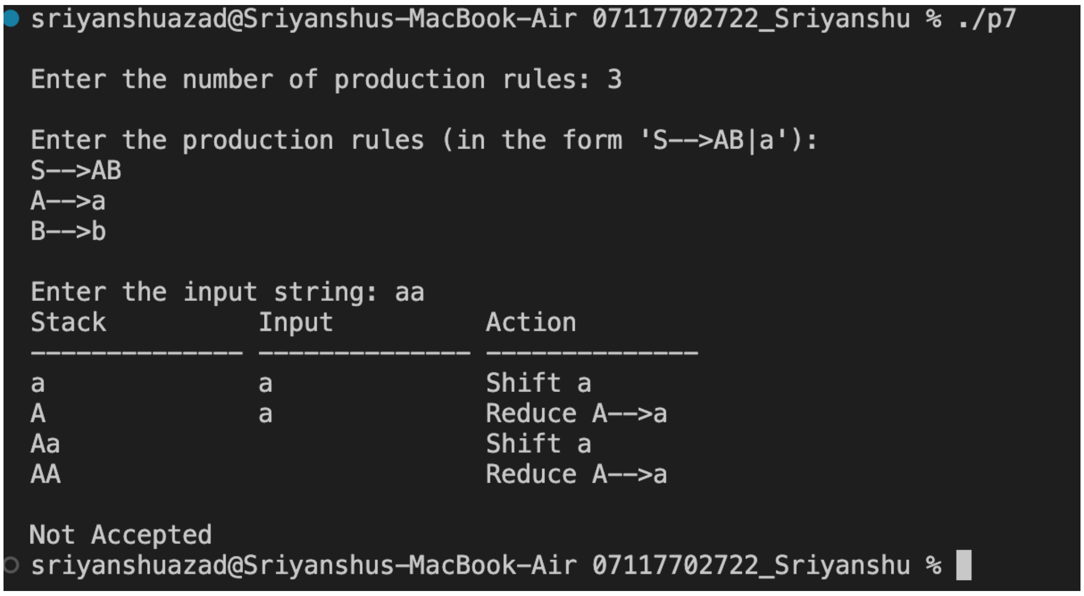
Code:

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

struct ProductionRule {
    char left;              // Single character for the non-
terminal
    char right[10];         // Right side of the production rule
};

int main() {
    char input[20], stack[50] = "", temp[50], ch[2], *substring;
    int i = 0, j, rule_count = 0, reduced;
    struct ProductionRule rules[10];

    // Input for the number of production rules
    printf("\nEnter the number of production rules: ");
    scanf("%d", &rule_count);
    getchar();  // Consume the newline

    // Input for each production rule in the specified form 'S--
>AB|a'
    printf("\nEnter the production rules (in the form 'S-->AB|a'):
\n");
    for (i = 0; i < rule_count; i++) {
        fgets(temp, sizeof(temp), stdin);
        temp[strcspn(temp, "\n")] = '\0'; // Remove newline at end
of input

        // Parse the left and right parts of the production
        char *token1 = strtok(temp, "-->");
        char *token2 = strtok(NULL, "-->");
        rules[i].left = token1[0];  // Store left side as a single
non-terminal character

        // Right side of the rule (e.g., "AB" or "a")
        if (token2 != NULL) {
            strcpy(rules[i].right, token2);
        }
    }

    // Input for the string to parse
    printf("\nEnter the input string: ");
    scanf("%s", input);

    printf("%-14s %-14s %-14s\n", "Stack", "Input", "Action");
    printf("-------------- -------------- --------------\n");

    i = 0;
    while (1) {
        // Shift: If there are more characters in the input, shift
the next character to the stack
        if (i < strlen(input)) {
            ch[0] = input[i++];
            ch[1] = '\0';
            strcat(stack, ch);  // Shift operation
            printf("%-15s%-15sShift %s\n", stack, &input[i], ch);
        }

        // Attempt to apply reductions
        reduced = 0;
        for (j = 0; j < rule_count; j++) {
            int right_len = strlen(rules[j].right);
            // Check if the stack ends with the right side of the
production
            if (strlen(stack) >= right_len &&
                strcmp(stack + strlen(stack) - right_len,
rules[j].right) == 0) {
                // Replace the right side of the rule in the stack
with the left side
                stack[strlen(stack) - right_len] = '\0';  //
Remove the matched substring
                strncat(stack, &rules[j].left, 1);  // Add the
left side of the rule
                printf("%-15s%-15sReduce %c-->%s\n", stack,
&input[i], rules[j].left, rules[j].right);
                reduced = 1;
                break;  // After a reduction, restart the loop
            }
        }

        // Final acceptance condition
        char start_symbol[2] = { rules[0].left, '\0' };  //
Convert start symbol to a string
        if (!reduced && i == strlen(input) && strcmp(stack,
start_symbol) == 0) {
            printf("\nAccepted\n");
            return 0;
        }

        // If no reduction was possible and input is fully
consumed, reject
        if (!reduced && i == strlen(input)) {
            printf("\nNot Accepted\n");
            return 0;
        }
    }
}
```

Output:



```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % gcc p7.c -o p7
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p7

Enter the number of production rules: 3

Enter the production rules (in the form 'S-->AB|a'):
S-->AB
A-->a
B-->b

Enter the input string: ab
Stack          Input          Action
-------------- -------------- --------------
a              b              Shift a
A              b              Reduce A-->a
Ab                            Shift b
AB                            Reduce B-->b
S                             Reduce S-->AB

Accepted
```



```
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p7

Enter the number of production rules: 3

Enter the production rules (in the form 'S-->AB|a'):
S-->AB
A-->a
B-->b

Enter the input string: aa
Stack          Input          Action
-------------- -------------- --------------
a              a              Shift a
A              a              Reduce A-->a
Aa                            Shift a
AA                            Reduce A-->a

Not Accepted
sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu %
```

Learning Outcomes:

Experiment 8

Aim: Write a program to check whether a grammar is operator precedent.

Theory:

Operator Precedence refers to the rules that define the order in which operators in expressions are evaluated. In programming languages, the precedence of operators affects how expressions are parsed and evaluated. Understanding operator precedence is crucial for designing grammars that accurately represent expressions involving operators.

Basic Concepts

1. **Operators**: Symbols that represent computations or operations, such as arithmetic operators (`+`, `-`, `*`, `/`) and relational operators (`<`, `>`, `==`).
2. **Non-Terminals**: Symbols in a grammar that can be replaced by groups of terminals and/or non-terminals. They represent syntactic categories, such as expressions or terms.
3. **Terminal Symbols**: Actual symbols in the input that represent the smallest units of the grammar, such as identifiers, numbers, or specific operator symbols.

Operator Precedence in Grammar

When designing a grammar that incorporates operators, it is important to establish rules for operator precedence and associativity:

1. **Precedence Levels**: Different operators may have different levels of precedence. For example, multiplication and division have higher precedence than addition and subtraction. This means that multiplication and division should be evaluated before addition and subtraction.
2. **Associativity**: This defines how operators of the same precedence level are grouped in the absence of parentheses. For example, the addition operator (`+`) is left-associative, meaning that in an expression like `a + b + c`, the addition is performed left to right.

Rules for Determining Operator Precedence

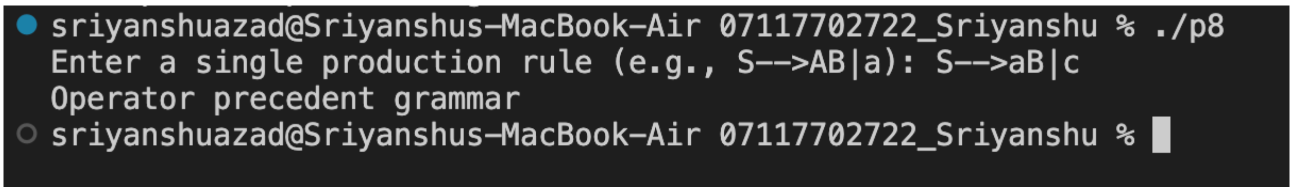
To check whether a grammar follows operator precedence, the following rules can be applied:

1. **No Two Non-terminals Together**: In a valid operator-precedent grammar, no two non-terminal symbols should appear directly adjacent to each other. This helps maintain a clear separation of operations and ensures that the precedence rules can be applied unambiguously.
2. **No Epsilon Productions**: Epsilon (empty string) productions should not appear in the right-hand side of any production rules. The presence of epsilon can lead to ambiguity in determining which operators take precedence and can complicate the parsing process.

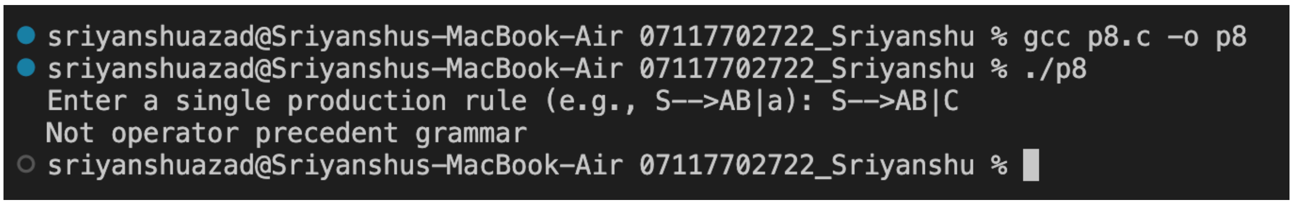
Code:

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#define MAX_RULE_LENGTH 20

// Function to check if the grammar is operator precedent
bool is_operator_precedent(char *rule) {
    bool has_non_terminal = false;

    // Iterate over each character in the production rule
    for (int j = 0; j < strlen(rule); j++) {
        if (rule[j] >= 'A' && rule[j] <= 'Z') { // Check for non-
terminals
            if (has_non_terminal) {
                // Found two consecutive non-terminals
                return false;
            }
            has_non_terminal = true; // Set flag for non-terminal
found
        } else if (rule[j] >= 'a' && rule[j] <= 'z') { // Check
for terminals
            has_non_terminal = false; // Reset the flag for the
next character
        } else if (rule[j] == 'e') { // Epsilon check
            return false; // Epsilon is not allowed
        }
    }
    return true; // The grammar is operator precedent
}

int main() {
    char production[MAX_RULE_LENGTH];

    // Input for the production rule
    printf("Enter a single production rule (e.g., S-->AB|a): ");
    fgets(production, sizeof(production), stdin);
    production[strcspn(production, "\n")] = '\0'; // Remove
newline character

    // Check if the grammar is operator precedent
    if (is_operator_precedent(production)) {
        printf("Operator precedent grammar\n");
    } else {
        printf("Not operator precedent grammar\n");
    }

    return 0;
}
```

Output:



```
● sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p8
  Enter a single production rule (e.g., S-->AB|a): S-->aB|c
  Operator precedent grammar
○ sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu %
```



```
● sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % gcc p8.c -o p8
● sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu % ./p8
  Enter a single production rule (e.g., S-->AB|a): S-->AB|C
  Not operator precedent grammar
○ sriyanshuazad@Sriyanshus-MacBook-Air 07117702722_Sriyanshu %
```

Learning Outcome: