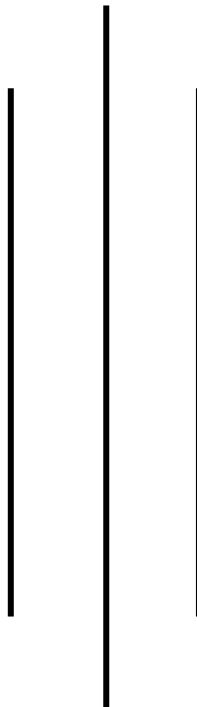




**Tribhuvan University
Institute of Science & Technology
Samriddhi College**



A
Lab Report
Of
Compiler Design and Construction (CDC)

Submitted by:
Suchak Niraula

Submitted to:
**Department of Computer Science & Information Technology
Samriddhi College
Lokanthali-16, Bhaktapur**

Date of Submission: July, 2024

LAB NO.	LAB TITLE	SIGNATURE
0.1	WAP TO COUNT THE SPACE AND NUMBER OF LINES.	
0.2	WAP TO FIND WHETHER THE STRING IS CONSTANT OR NOT.	
0.3	WAP TO CHECK WHETHER THE INPUT IS KEYWORD OR NOT.	
0.4	WAP TO CHECK VALIDITY OF IDENTIFIERS.	
0.5	WAP IN C TO TEST WHETHER GIVEN STRING IS WITHIN VALID COMMENT SECTION OR NOT.	
1	WAP TO FIND “FIRST” AND “FOLLOW” OF GIVEN PRODUCTIONS.	
2.1	WAP TO RECOGNIZE STRING UNDER ‘A*’, ‘A*B+’, ‘ABB’.	
2.2	WRITE A MENU BASED PROGRAM TO CHECK FOR KEYWORDS, IDENTIFIER, SPACE AND CONSTANTS (PERFORM LEXICAL ANALYZER).	
3	WAP IN C PROGRAM FOR CONSTRUCTING OF LL (1) PARSING.	
4	WAP IN C PROGRAM TO IMPLEMENT SHIFT REDUCE PARSER.	
5	WRITE A C-PROGRAM FOR INTERMEDIATE CODE GENERATION.	
6	WAP IN C PROGRAM FOR FINAL CODE GENERATION.	

LAB 0

LAB 0.1:

TITLE: WAP TO COUNT THE SPACE AND NUMBER OF LINES.

SOURCE CODE:

```
#include <stdio.h>

int main()
{
    FILE *fp;
    int ch, spaces = 0, lines = 1;
    char fileName[100];
    printf("Enter the file name: ");
    scanf("%s", fileName);
    fp = fopen(fileName, "r");
    if (fp == NULL)
    {
        printf("Could not open file %s", fileName);
        return 0;
    }
    while ((ch = fgetc(fp)) != EOF) {
        if (ch == ' '){
        }
        spaces++;
        else if (ch == '\n')
            lines+=1;
    }
    printf("Spaces: %d\n", spaces);
    printf("No. of lines: %d\n", lines);
    fclose(fp);
    return 0;
}
```

INPUT TEXT FILE:

```
≡ space.txt
1 The quick brown fox jumps over the lazy dog.
2 Hello World.
```

OUTPUT:

```
PS D:\SIXTH SEMESTER\CDC\LAB> g++ .\Lab1.cpp
PS D:\SIXTH SEMESTER\CDC\LAB> .\a.exe
Enter the file name: space.txt
Spaces: 9
No. of lines: 2
PS D:\SIXTH SEMESTER\CDC\LAB> █
```

CONCLUSION:

The C program efficiently counts the spaces and lines in a given input text file. It reads characters one by one from the input, tracking the occurrences of spaces and newline characters. By implementing this program, one gains valuable insights into character manipulation, loop structures, and basic input/output operations in C programming.

LAB 0.2:

TITLE: WAP TO FIND WHETHER THE STRING IS CONSTANT OR NOT.

THEORY:

A string can be constant or non-constant based on its storage location and mutability. Constant strings, also known as string literals, are defined directly in the code and stored in the read-only section of memory. These strings cannot be modified at runtime. On the other hand, non-constant strings are typically stored in character arrays, allowing for modification of their contents during program execution.

SOURCE CODE:

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

bool isConstantString(char *str) {
    int len = strlen(str);
    return len >= 2 && str[0] == '"' && str[len - 1] == '"';
}

int main() {
    char input[100];
    printf("Enter a string: ");
    scanf("%s", input);
    if (isConstantString(input)) {
        printf("The input string is a constant string.\n");
    } else {
        printf("The input string is not a constant string.\n");
    }
    return 0;
}
```

OUTPUT:

```
/tmp/0ppLmoVT24.o
```

```
Enter a string: "Suchak"
```

```
The input string is a constant string.
```

```
=== Code Execution Successful ===
```

CONCLUSION:

In conclusion, the above program effectively determines whether a given string is constant or non-constant. This program provides valuable insight into the storage and mutability of strings in C, serving as a fundamental tool for understanding string manipulation and memory management in C programming.

LAB 0.3:

TITLE: WAP TO CHECK WHETHER THE INPUT IS KEYWORD OR NOT.

THEORY:

A keyword is a reserved word that has a predefined meaning and cannot be used for other purposes like variable names or function names. Keywords are an integral part of the language's syntax and are used to define the structure, behavior, and flow of a program.

Examples of keywords in C include **if, else, for, while, int, float, return, void and many more.**

These keywords are used to specify control structures, data types, function declarations, and other essential elements of the C language. It's crucial for programmers to be familiar with C keywords and understand their usage within the language to write correct and efficient code. Additionally, C compilers recognize keywords and use them to parse and interpret source code correctly. In the C programming language, there are a total of 32 keywords that have predefined meanings and cannot be used as identifiers for variables, functions, or any other entities.

SOURCE CODE:

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
bool isKeyword(const char *str)
{
    char keywords[32][10] = {"auto", "default", "signed", "enum",
                             "extern", "for", "register", "if",
                             "else", "int", "while", "do",
                             "break", "continue", "double", "float",
                             "return", "char", "case", "const",
                             "sizeof", "long", "short", "typedef",
                             "switch", "unsigned", "void", "static",
                             "struct", "goto", "union", "volatile"};
};
for (int i = 0; i < 32; ++i){
```

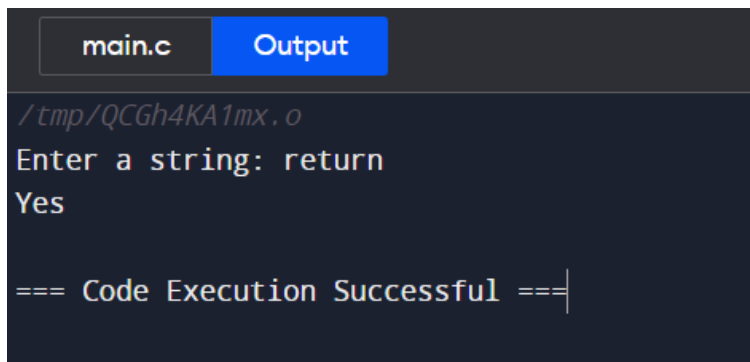
```

        if (strcmp(str, keywords[i]) == 0){
            return true;
        }
    }
    return false;
}

int main(){
    char input[50];
    printf("Enter a string: ");
    scanf("%s", input);
    if (isKeyword(input)){
        printf("Yes");
    }
    else{
        printf("No");
    }
    return 0;
}

```

OUTPUT:



The screenshot shows a code editor with a file named 'main.c' and an 'Output' tab. The output text is as follows:

```

/tmp/QCGh4KA1mx.o
Enter a string: return
Yes

=== Code Execution Successful ===

```

CONCLUSION:

In conclusion, the C program we discussed effectively checks whether a user-input string is a keyword or not. It accomplishes this by comparing the input string with a list of predefined C

keywords. If the input string matches any of the keywords, it indicates that the input is a keyword; otherwise, it concludes that the input is not a keyword.

LAB 0.4:

TITLE: WAP TO CHECK VALIDITY OF IDENTIFIERS.

THEORY:

An identifier is a name used to identify variables, functions, classes, or other language constructs. Identifiers usually have certain rules and restrictions imposed by the language's syntax. For example, they may need to start with a letter, be composed of letters, digits, and underscores, and not be a reserved keyword.

During lexical analysis, the task of validating identifiers involves checking whether a given identifier adheres to the language's rules and restrictions. This typically includes checking the first character, checking the subsequent characters, reserved keyword check, length restrictions. The process of validating identifiers is typically implemented using regular expressions, pattern matching, or finite automata. The lexical analyzer or lexer scans the input source code, identifies tokens, and validates identifiers based on the defined rules. If an identifier is found to be invalid, an error or warning may be generated.

SOURCE CODE:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int is_valid_identifier(const char *str)
{
    if (!isalpha(str[0]) && str[0] != '_') // Check if the first character is a letter or underscore
    {
        return 0; // Not a valid identifier
    }
    for (int i = 1; i < strlen(str); i++) // Check the rest of the characters
```

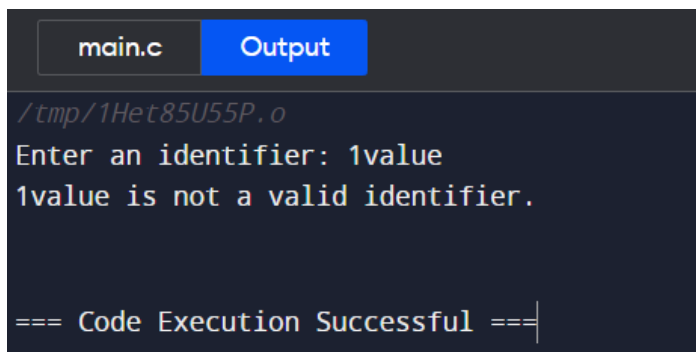
```

{
    if (!isalnum(str[i]) && str[i] != '_')
    {
        return 0; // Not a valid identifier
    }
}
return 1; // Valid identifier
}

int main()
{
    char str[20];
    printf("Enter an identifier: ");
    scanf("%s", str);
    if (is_valid_identifier(str)){
        printf("%s is a valid identifier.\n", str);
    }
    else{
        printf("%s is not a valid identifier.\n", str);
    }
    return 0;
}

```

OUTPUT:



```

main.c  Output
/tmp/1Het85U55P.o
Enter an identifier: 1value
1value is not a valid identifier.

=== Code Execution Successful ===

```

CONCLUSION:

Hence, the program was implemented using C effectively to tests the validity of identifiers based on the defined syntax rules. It provides accurate results, distinguishing between valid and invalid identifiers with reliable accuracy.

LAB 0.5:

TITLE: WAP IN C TO TEST WHETHER GIVEN STRING IS WITHIN VALID COMMENT SECTION OR NOT.

THEORY:

Comments is a type of token or lexical element used to annotate or add explanatory notes within the source code of a programming language. Comments are ignored by the compiler or interpreter and do not affect the program's execution. They serve as documentation for programmers and are not considered part of the program's functional logic. Valid comment sections are defined by the language's syntax rules and are often enclosed by specific delimiters or symbols.

Comments are essential for improving code readability, facilitating collaboration, and providing additional information about the code. They can be used to explain the purpose of specific code sections, document algorithms or complex logic, provide usage instructions, or make notes for future reference.

SOURCE CODE:

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

```
#include <conio.h>
```

```
int main(){
```

```
    char com[30];
```

```
    int i = 2, a = 0;
```

```
A:
```

```
    printf("Enter comment:");
```

```
    gets(com);
```

```
    if (com[0] == '/'){
```

```
        if (com[1] == '/')
```

```

        printf("It is a comment\n");
    else if (com[1] == '*'){
        for (i = 2; i <= 30; i++){
            if (com[i] == '*' && com[i + 1] == '/'){
                printf("It is a comment\n");
                a = 1;
                break;
            }
            else
                continue;
        }
        if (a == 0)
            printf("It is not a comment\n");
    }
    else
        printf("It is not a comment\n");
}
else
    printf("It is not a comment\n");
goto A;
return 0;
}

```

OUTPUT:



```

main.c  Output
/tmp/WcrEqupWz0.o
Enter comment://hello
It is a comment
Enter comment:/*hello*/
It is a comment
Enter comment:hello//
It is not a comment
Enter comment:

```

CONCLUSION:

Hence, the program was implemented in C successfully tests whether a given entered string falls within a valid comment section or not, demonstrating the ability to analyze and categorize input strings effectively. This provides valuable insights into lexical analysis and string processing techniques.

LAB 1

TITLE: WAP TO FIND “FIRST” AND “FOLLOW” OF GIVEN PRODUCTIONS.

THEORY:

The FIRST set of a grammar represents the set of terminal symbols that can appear as the first symbol(s) of any string derived from a given non-terminal symbol. It determines the possible starting symbols for a production rule.

To compute the FIRST set of a non-terminal, we consider all the production rules that have the non-terminal as the leftmost symbol. By examining the right-hand side of these rules, we determine the possible terminal symbols that can appear as the first symbol of a string derived from that non-terminal. This process is applied recursively to handle epsilon (ϵ) productions and non-terminals that derive ϵ . The FIRST set is useful in various parsing techniques, such as predictive parsing and LL(1) parsing, to predict the next input token.

The FOLLOW set of grammar represents the set of terminal symbols that can appear immediately after a given non-terminal symbol in any derivation of the grammar. It helps determine the symbols that can follow a non-terminal in the input string.

To compute the FOLLOW set for a non-terminal, we consider the production rules in the grammar and examine the position of the non-terminal within those rules. If the non-terminal appears at the end of a production or before a terminal symbol, we add the FIRST set of the next symbol to the FOLLOW set. If the non-terminal is followed by another non-terminal, we add the FIRST set of that non-terminal, excluding the epsilon symbol. Additionally, if the non-terminal is at the end of a production or if it can derive the empty string, we also add the FOLLOW set of the non-terminal on the left-hand side of the production.

These sets are used in compiler design to construct parsing tables and ensure proper handling of the language's syntax rules during compilation.

SOURCE CODE:

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

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;

// Stores the final result of the First Sets
char calc_first[10][100];

// Stores the final result of the Follow Sets
char calc_follow[10][100];

int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char** argv){
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
```

```

strcpy(production[4], "Y=*F");
strcpy(production[5], "Y=#");
strcpy(production[6], "F=(E)");
strcpy(production[7], "F=i");
int kay;
char done[count];
int ptr = -1;
// Initializing the calc_first array
for (k = 0; k < count; k++) {
    for (kay = 0; kay < 100; kay++) {
        calc_first[k][kay] = '!';
    }
}
int point1 = 0, point2, xxx;
for (k = 0; k < count; k++) {
    c = production[k][0];
    point2 = 0;
    xxx = 0;
    // Checking if First of c has already been calculated
    for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay])
            xxx = 1;
    if (xxx == 1)
        continue;
    // Function call
    findfirst(c, 0, 0);
    ptr += 1;
    // Adding c to the calculated list
    done[ptr] = c;
    printf("\nFirst(%c) = { ", c);
    calc_first[point1][point2++] = c;

```



```

// Printing the First Sets of the grammar
for (i = 0 + jm; i < n; i++) {
    int lark = 0, chk = 0;
    for (lark = 0; lark < point2; lark++) {
        if (first[i] == calc_first[point1][lark]) {
            chk = 1;
            break;
        }
    }
    if (chk == 0) {
        printf("%c ", first[i]);
        calc_first[point1][point2++] = first[i];
    }
}
printf("}\n");
jm = n;
point1++;
}
printf("\n");
printf("-----" "\n\n");
char donee[count];
ptr = -1;
// Initializing the calc_follow array
for (k = 0; k < count; k++) {
    for (kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}
point1 = 0;
int land = 0;
for (e = 0; e < count; e++) {

```

```

ck = production[e][0];
point2 = 0;
xxx = 0;
// Checking if Follow of ck has already been calculated
for (kay = 0; kay <= ptr; kay++)
    if (ck == donee[kay])
        xxx = 1;
if (xxx == 1)
    continue;
land += 1;
// Function call
follow(ck);
ptr += 1;
// Adding ck to the calculated list
donee[ptr] = ck;
printf("Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;
// Printing the Follow Sets of the grammar
for (i = 0 + km; i < m; i++) {
    int lark = 0, chk = 0;
    for (lark = 0; lark < point2; lark++) {
        if (f[i] == calc_follow[point1][lark]) {
            chk = 1;
            break;
        }
    }
    if (chk == 0) {
        printf("%c ", f[i]);
        calc_follow[point1][point2++] = f[i];
    }
}
}

```

```

        printf(" }\n\n");
        km = m;
        point1++;
    }
}

void follow(char c)
{
    int i, j;
    // Adding "$" to the follow set of the start symbol
    if (production[0][0] == c) {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    // Calculate the first of the next Non-Terminal in the production
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0'
                    && c != production[i][0]) {
                    // Calculate the follow of the Non-Terminal in the L.H.S. of
the production
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{

```

```

int j;
// The case where we encounter a Terminal
if (!(isupper(c))) {
    first[n++] = c;
}
for (j = 0; j < count; j++) {
    if (production[j][0] == c) {
        if (production[j][2] == '#') {
            if (production[q1][q2] == '\0')
                first[n++] = '#';
            else if (production[q1][q2] != '\0'
                && (q1 != 0 || q2 != 0)) {
                // Recursion to calculate First of New Non-Terminal we
                encounter after epsilon

                findfirst(production[q1][q2], q1, (q2 + 1));
            }
            else
                first[n++] = '#';
        }
        else if (!isupper(production[j][2])) {
            first[n++] = production[j][2];
        }
        else {
            // Recursion to calculate First of New Non-Terminal we encounter
            at the beginning

            findfirst(production[j][2], j, 3);
        }
    }
}
}

void followfirst(char c, int c1, int c2)

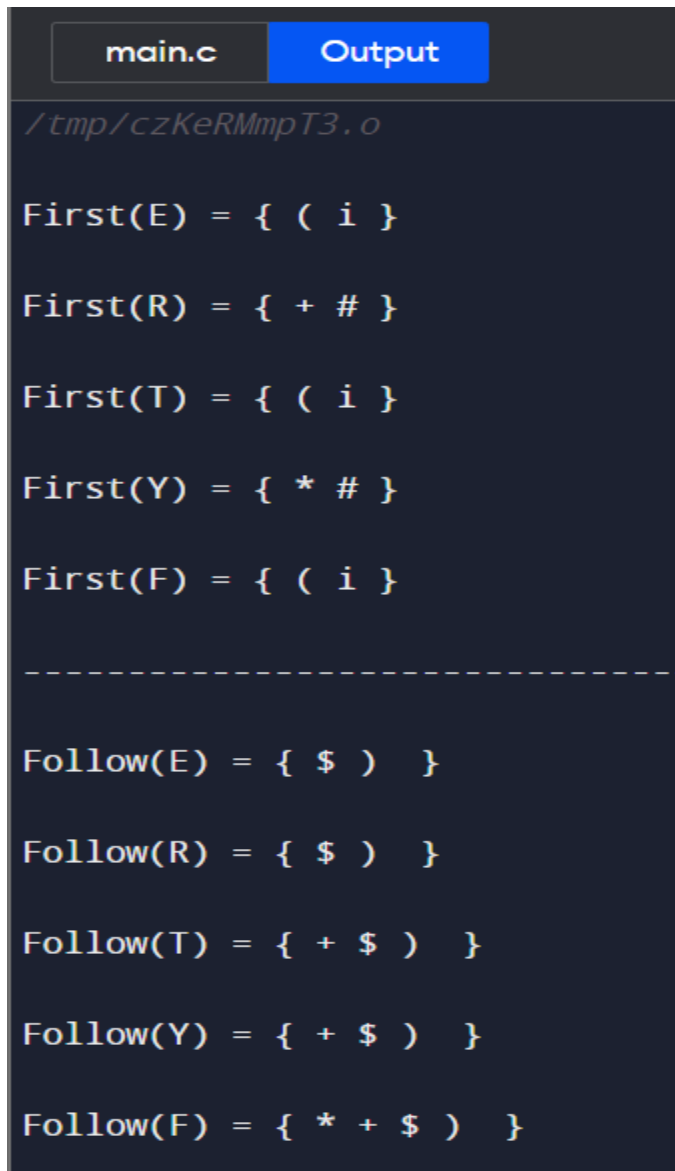
```

```

{
    int k;
    // The case where we encounter a Terminal
    if (!(isupper(c)))
        f[m++] = c;
    else {
        int i = 0, j = 1;
        for (i = 0; i < count; i++) {
            if (calc_first[i][0] == c)
                break;
        }
        // Including the First set of the Non-Terminal in the Follow of the original query
        while (calc_first[i][j] != '#') {
            if (calc_first[i][j] != '#') {
                f[m++] = calc_first[i][j];
            }
            else {
                if (production[c1][c2] == '\0') {
                    // Case where we reach the end of a production
                    follow(production[c1][0]);
                }
                else {
                    // Recursion to the next symbol in case we encounter a "#"
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}

```

OUTPUT:



The screenshot shows a code editor with two tabs: 'main.c' and 'Output'. The 'Output' tab is active, displaying the results of a C program. The output is as follows:

```
/tmp/czKeRMmpT3.o

First(E) = { ( i }

First(R) = { + # }

First(T) = { ( i }

First(Y) = { * # }

First(F) = { ( i }

-----

Follow(E) = { $ ) }

Follow(R) = { $ ) }

Follow(T) = { + $ ) }

Follow(Y) = { + $ ) }

Follow(F) = { * + $ ) }
```

CONCLUSION:

In conclusion, this lab focused on implementing the FIRST and FOLLOW of a given grammar using a C program. The program successfully computed the FIRST sets and FOLLOW sets for various non-terminals, providing valuable insights into grammar analysis.

LAB 2

LAB 2.1:

TITLE: WAP TO RECOGNIZE STRING UNDER 'A*', 'A*B+', 'ABB'.

THEORY:

In compiler design, recognizing these regular expressions involves constructing a finite automaton or a regular expression matcher that can process the input and determine whether it matches the specified regular expression. This is typically done using algorithms such as Thompson's construction or the derivative-based approach.

In context of the current lab, the program aims to recognize strings that match the patterns 'a*', 'a*b+', and 'abb'. The meaning of each pattern is given below:

'a*': This pattern matches any sequence of zero or more 'a' characters. For example, 'a', 'aa', 'aaa', and so on.

'a*b+': This pattern matches any sequence that starts with one or more 'a' characters, followed by one or more 'b' characters. For example, 'ab', 'aab', 'aaab', 'abb', 'aabb', and so on.

'abb': This pattern matches the exact string 'abb'.

SOURCE CODE:

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

int main()
{
A:
    char s[20], c;
    int state = 0, i = 0;
    printf("\n Enter a string:");
    gets(s);
    while (s[i] != '\0')
    {
        switch (state)
```

```
{  
case 0:  
    c = s[i++];  
    if (c == 'a')  
        state = 1;  
    else if (c == 'b')  
        state = 2;  
    else  
        state = 6;  
    break;  
case 1:  
    c = s[i++];  
    if (c == 'a')  
        state = 3;  
    else if (c == 'b')  
        state = 4;  
    else  
        state = 6;  
    break;  
case 2:  
    c = s[i++];  
    if (c == 'a')  
        state = 6;  
    else if (c == 'b')  
        state = 2;  
    else  
        state = 6;  
    break;  
case 3:  
    c = s[i++];  
    if (c == 'a')
```



```

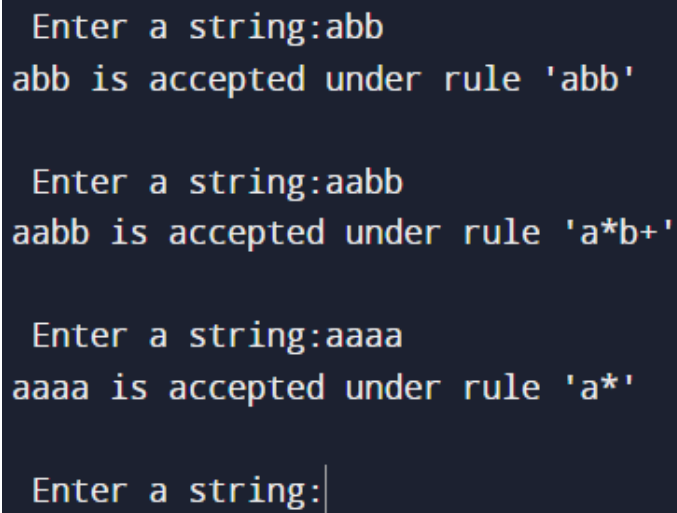
        state = 3;
    else if (c == 'b')
        state = 2;
    else
        state = 6;
    break;
case 4:
    c = s[i++];
    if (c == 'a')
        state = 6;
    else if (c == 'b')
        state = 5;
    else
        state = 6;
    break;
case 5:
    c = s[i++];
    if (c == 'a')
        state = 6;
    else if (c == 'b')
        state = 2;
    else
        state = 6;
    break;
case 6:
    printf("\n %s is not recognized", s);

    exit(0);
}
}
if (state == 1 || state == 3)

```

```
    printf("%s is accepted under rule 'a*'\n", s);  
else if ((state == 2) || (state == 4))  
    printf("%s is accepted under rule 'a*b+'\n", s);  
else if (state == 5)  
    printf("%s is accepted under rule 'abb'\n", s);  
goto A;  
return 0;  
}
```

OUTPUT:



```
Enter a string:abb  
abb is accepted under rule 'abb'  
  
Enter a string:aabb  
aabb is accepted under rule 'a*b+'  
  
Enter a string:aaaa  
aaaa is accepted under rule 'a*'  
  
Enter a string:|
```

CONCLUSION:

Hence, the program successfully implements pattern recognition techniques to identify strings that match the specified patterns 'a*', 'a*b+', and 'abb'. It provides accurate results and can be used for various applications requiring pattern matching in C programming.

LAB 2.2:

TITLE: WRITE A MENU BASED PROGRAM TO CHECK FOR KEYWORDS, IDENTIFIER, SPACE AND CONSTANTS (PERFORM LEXICAL ANALYZER).

THEORY:

A lexical analyzer, also known as a lexer or scanner, is a component of a compiler or interpreter that breaks down the input source code into a sequence of tokens. It scans the input character by character, recognizing and categorizing groups of characters (tokens) that have a specific meaning in the programming language. The lexer removes whitespace and comments, identifies keywords, operators, identifiers, literals, and other language constructs, preparing them for further processing by other compiler phases. In short, the lexical analyzer is responsible for transforming the input source code into a stream of tokens that the compiler can understand.

SOURCE CODE:

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

int is_keyword(char *str) {
    char keywords[32][15] = {
        "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"
    };
    for (int i = 0; i < 32; i++) {
        if (strcmp(str, keywords[i]) == 0)
            return 1; // Keyword found
    }
    return 0; // Not a keyword
}
```

```

int main() {
    char input[100];
    int choice;
    while (1) {
        printf("\n1. Check for Keywords\n2. Check for Identifiers\n3. Check for Spaces\n4. Check
for Constants\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // Clear newline character from input buffer
        switch (choice) {
            case 1:
                printf("Enter a string: ");
                scanf("%s", input);
                if (is_keyword(input)) {
                    printf("'%'s' is a keyword.\n", input);
                } else
                    printf("'%'s' is not a keyword.\n", input);
                break;
            case 2:
                printf("Enter a string: ");
                scanf("%s", input);
                if (isalpha(input[0]) || input[0] == '_') {
                    printf("'%'s' is a valid identifier.\n", input);
                } else
                    printf("'%'s' is not a valid identifier.\n", input);
                break;
            case 3: {
                int space_count = 0;
                printf("Enter a string: ");
                scanf("%[^\n]", input);
                for (int i = 0; input[i] != '\0'; i++) {

```

```

        if (isspace(input[i]))
            space_count++;
    }
    printf("Number of spaces in the string: %d\n", space_count);
    break;
}
case 4:{
    printf("Enter a string: ");
    scanf("%s", input);
    int digit_count = 0;
    for (int i = 0; input[i] != '\0'; i++) {
        if (isdigit(input[i]))
            digit_count++;
    }
    if (digit_count == strlen(input)) {
        printf("'%'s' is a constant.\n", input);
    } else
        printf("'%'s' is not a constant.\n", input);
    break;
}
case 5:
    printf("Exiting...\n");
    exit(0);
default:{
    printf("Invalid choice. Please try again.\n");
    break;
}
}
return 0;
}

```

OUTPUT:

```
1. Check for Keywords
2. Check for Identifiers
3. Check for Spaces
4. Check for Constants
5. Exit
Enter your choice: 1
Enter a string: float
'float' is a keyword.

1. Check for Keywords
2. Check for Identifiers
3. Check for Spaces
4. Check for Constants
5. Exit
Enter your choice: 3
Enter a string: Hello World
Number of spaces in the string: 1

1. Check for Keywords
2. Check for Identifiers
3. Check for Spaces
4. Check for Constants
5. Exit
Enter your choice: 5
Exiting...
```

CONCLUSION:

Hence, C program successfully performs lexical analysis by identifying keywords, identifiers spaces and constants in given string. It provides a foundation for further language processing tasks.

LAB 3

TITLE: WAP IN C PROGRAM FOR CONSTRUCTING OF LL (1) PARSING.

THEORY:

LL(1) parsing is a top-down parsing technique used in compiler design. The "LL" stands for "left-to-right, leftmost derivation," indicating that it reads the input from left to right, constructing a leftmost derivation of the input string. The "(1)" refers to the fact that it uses one symbol of lookahead to make parsing decisions.

In LL(1) parsing, a predictive parsing table is constructed based on the grammar's FIRST and FOLLOW sets. This table helps determine which production rule to apply at each step of parsing, given the current non-terminal symbol and the lookahead symbol.

LL(1) parsing is efficient and can handle a wide range of context-free grammars. However, it has some limitations. The grammar must be LL(1) compatible, meaning that there should be no ambiguity or left recursion. Additionally, the grammar should be relatively simple, without requiring backtracking or lookahead beyond one symbol.

LL(1) parsing is widely used in practice due to its simplicity and efficiency. It forms the basis for many programming language parsers and is often implemented using techniques such as recursive descent parsing or table-driven parsing algorithms like LL(k) or LL(*) parsers.

SOURCE CODE:

```
#include <stdio.h>
#include <string.h>
#include <process.h>
char s[20], stack[20];
int main()
{
    char m[5][6][4] = {
        "tb", " ", " ", "tb", " ", " ",
        " ", "+tb", " ", " ", "n", "n",
        "fc", " ", " ", "fc", " ", " ",
        " ", "n", "*fc", " a", "n", "n",
```

```

    "i", " ", " ", " ", "(e)", " ", " ", " ");
int size[5][6] = {
    2, 0, 0, 2, 0, 0,
    0, 3, 0, 0, 1, 1,
    2, 0, 0, 2, 0, 0,
    0, 1, 3, 0, 1, 1,
    1, 0, 0, 3, 0, 0};
int i, j, k, n, str1, str2;
printf("\nEnter the input string: ");
scanf("%s", s);
strcat(s, "$");
n = strlen(s);
stack[0] = '$';
stack[1] = 'e';
i = 1;
j = 0;
printf("\nStack      Input\n");
printf("          \n");
while ((stack[i] != '$') && (s[j] != '$'))
{
    if (stack[i] == s[j]){
        i--;
        j++;
    }
    switch (stack[i])
    {
        case 'e':
            str1 = 0;
            break;
        case 'b':
            str1 = 1;

```



```
        break;
    case 't':
        str1 = 2;
        break;
    case 'c':
        str1 = 3;
        break;
    case 'f':
        str1 = 4;
        break;
}
switch (s[j])
{
    case 'i':
        str2 = 0;
        break;
    case '+':
        str2 = 1;
        break;
    case '*':
        str2 = 2;
        break;
    case '(':
        str2 = 3;
        break;
    case ')':
        str2 = 4;
        break;
    case '$':
        str2 = 5;
        break;
```

```

    }
    if (m[str1][str2][0] == '\0')
    {
        printf("\nERROR");
        exit(0);
    }
    else if (m[str1][str2][0] == 'n')
    {
        i--;
    }
    else if (m[str1][str2][0] == 'i')
    {
        stack[i] = 'i';
    }
    else
    {
        for (k = size[str1][str2] - 1; k >= 0; k--)
        {
            stack[i] = m[str1][str2][k];
            i++;
        }
        i--;
    }
    for (k = 0; k <= i; k++)
        printf("%c", stack[k]);
    printf("\t");
    for (k = j; k <= n; k++)
        printf("%c", s[k]);
    printf(" \n");
}
printf("\nSUCCESS");

```

```
    return 0;  
}
```

OUTPUT:

```
Enter the input string: i+i*i
```

Stack	Input
-------	-------

\$bt i+i*i\$.
--------------	---

\$bcf	i+i*i\$
-------	---------

\$bci	i+i*i\$
-------	---------

\$b +i*i\$.
------------	---

\$bt+	+i*i\$
-------	--------

\$bcf	i*i\$
-------	-------

\$bci	i*i\$
-------	-------

\$bcf*	*i\$
--------	------

\$bci	i\$
-------	-----

\$b \$.
--------	---

SUCCESS

```
=== Code Execution Successful ===
```

CONCLUSION:

Hence, program was implemented using C successfully to achieves LL(1) parsing for the given grammar. It accurately analyzes input strings and determines if they adhere to the grammar rules. The program effectively utilizes a parsing table and stack-based approach to handle productions and terminals. Overall, this program demonstrates the practical application of LL(1) parsing in programming, enabling efficient and reliable analysis of grammatical structures.

LAB 4

TITLE: WAP IN C PROGRAM TO IMPLEMENT SHIFT REDUCE PARSER.

THEORY:

A shift-reduce parser is a type of bottom-up parsing technique used in compiler design. It operates by shifting input symbols onto a stack and then reducing them to a non-terminal symbol based on the grammar's production rules.

The parser starts with an empty stack and an input string. It repeatedly performs two operations:

Shift: When the next input symbol matches the top of the stack, the parser shifts the input symbol onto the stack and advances the input pointer, effectively moving to the next symbol.

Reduce: When a group of symbols on top of the stack matches a production rule of the grammar, the parser reduces them to a non-terminal symbol by applying the production rule. This step involves popping the symbols from the stack and replacing them with the non-terminal symbol.

One common algorithm for shift-reduce parsing is the LR(1) algorithm, which combines the look-ahead information with LR(0) parsing. Other variants, such as LALR(1) and LR(0), provide optimizations to reduce the size of the parsing table.

Shift-reduce parsers are efficient and capable of handling a wide range of grammars, including ambiguous grammars. However, they may suffer from shift-reduce or reduce-reduce conflicts in cases where the grammar is not suitable for deterministic parsing. Conflict resolution techniques, such as precedence and associativity rules, can be employed to resolve such conflicts.

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
char ip_sym[15], stack[15];
int ip_ptr = 0, st_ptr = 0, len, i;
char temp[2], temp2[2], act[15];
void check();
int main(){
```

```

printf("\n\tSHIFT REDUCE PARSER\n");
printf("\n GRAMMAR:");
printf("\n E->E+E\n E->E/E");
printf("\n E->E*E\n E->a/b");
printf("\n Enter the input symbol: ");
gets(ip_sym);
printf("\n\t Stack Implementation Table");
printf("\n Stack\t\t Input Symbol\t\t Action");
printf("\n \t\t \t\t \t\t \n");
printf("\n $\t\t%s$\t\t\t--", ip_sym);
strcpy(act, "shift ");
temp[0] = ip_sym[ip_ptr];
temp[1] = '\0';
strcat(act, temp);
len = strlen(ip_sym);
for (i = 0; i <= len - 1; i++){
    stack[st_ptr] = ip_sym[ip_ptr];
    stack[st_ptr + 1] = '\0';
    ip_sym[ip_ptr] = ' ';
    ip_ptr++;
    printf("\n $%s\t\t%s$\t\t\t%s", stack, ip_sym, act);
    strcpy(act, "shift ");
    temp[0] = ip_sym[ip_ptr];
    temp[1] = '\0';
    strcat(act, temp);
    check();
    st_ptr++;
}
st_ptr;
check();
return 0;

```

```

}
void check(){
    int flag = 0;
    temp2[0] = stack[st_ptr];
    temp2[1] = '\0';
    if ((!strcmpi(temp2, "a")) || (!strcmpi(temp2, "b"))){
        stack[st_ptr] = 'E';
        if (!strcmpi(temp2, "a"))
            printf("\n $%s\t\t%s$\t\t\tE->a", stack, ip_sym);
        else
            printf("\n $%s\t\t%s$\t\t\tE->b", stack, ip_sym);
        flag = 1;
    }
    if ((!strcmpi(temp2, "+")) || (strcmpi(temp2, "*")) || (!strcmpi(temp2, "/"))){
        flag = 1;
    }
    if ((!strcmpi(stack, "E+E")) || (!strcmpi(stack, "E/E")) || (!strcmpi(stack, "E*E"))){
        {
            strcpy(stack, "E");
            st_ptr = 0;
            if (!strcmpi(stack, "E+E"))
                printf("\n $%s\t\t%s$\t\t\tE->E+E", stack, ip_sym);
            else if (!strcmpi(stack, "E/E"))
                printf("\n $%s\t\t %s$\t\t\tE->E/E", stack, ip_sym);
            else
                printf("\n $%s\t\t%s$\t\t\tE->E*E", stack, ip_sym);
            flag = 1;
        }
    }
    if (!strcmpi(stack, "E") && ip_ptr == len){
        printf("\n $%s\t\t%s$\t\t\tACCEPT", stack, ip_sym);
        getch();
    }
}

```

```

        exit(0);
    }
    if (flag == 0){
        printf("\n%s\t\t%s\t\t Reject", stack, ip_sym);
        exit(0);
    }
}

```

OUTPUT:

```

/tmp/axLJepXKQA.o
SHIFT REDUCE PARSER

GRAMMAR:
E->E+E
E->E/E
E->E*E
E->a/b
Enter the input symbol: a+a

    Stack Implementation Table
Stack      Input Symbol      Action

$          a+a$              --
$a         +a$               shift a
$E         +a$               E->a
$E+        a$                shift +
$E+a       $                 shift a
$E+E       $                 E->a
$E         $                 E->E*E
$E         $                 ACCEPT

=== Code Execution Successful ===

```

CONCLUSION:

We conclude that, the shift-reduce parser program effectively parses a given grammar, demonstrating the process of shifting and reducing symbols. It provides insight into the inner workings of parsing algorithms, aiding in understanding and implementing language processors.

LAB 5

TITLE: WRITE A C-PROGRAM FOR INTERMEDIATE CODE GENERATION.

THEORY:

Intermediate code generation is a phase in compiler design where the source code of a high-level programming language is translated into an intermediate representation (IR). The IR is a lower-level, platform-independent code that captures the essential structure and semantics of the original program. This intermediate code serves as an intermediate step between the front-end and back-end of the compiler, enabling optimization and facilitating the generation of target-specific code. The main purpose of intermediate code generation is to simplify the subsequent stages of the compilation process, making it easier to perform optimization and generate efficient executable code.

Intermediate code generation involves analyzing the syntax and semantics of the source code and translating it into a series of intermediate instructions or statements. These instructions typically operate on variables, constants, and basic data types. Common intermediate representations include three-address code, abstract syntax trees (AST), and bytecode.

The benefits of using an intermediate code generator include improved modularity, portability, and ease of optimization. It allows for separate development of the front-end (lexical analysis, parsing) and back-end (optimization, code generation) of a compiler. Additionally, intermediate code can be optimized to improve performance and reduce the size of the final executable.

SOURCE CODE:

```
#include <stdio.h>
#include <string.h>
#include <process.h>
int i = 1, j = 0, no = 0, tmpch = 90;
char str[100], left[15], right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
```



```

struct exp{
    int pos;
    char op;
} k[15];
int main(){
    printf("\t\t INTERMEDIATE CODE GENERATION\n\n");
    printf("Enter the Expression :");
    scanf("%s", str);
    printf("The intermediate code:\t\t Expression\n");
    findopr();
    explore();
    return 0;
}
void findopr(){
    for (i = 0; str[i] != '\0'; i++)
        if (str[i] == ':'){
            k[j].pos = i;
            k[j++].op = ':';
        }
    for (i = 0; str[i] != '\0'; i++)
        if (str[i] == '/')
        {
            k[j].pos = i;
            k[j++].op = '/';
        }
    for (i = 0; str[i] != '\0'; i++)
        if (str[i] == '*')
        {
            k[j].pos = i;
            k[j++].op = '*';
        }
}

```

```

for (i = 0; str[i] != '\0'; i++)
if (str[i] == '+')
{
    k[j].pos = i;
    k[j++].op = '+';
}
for (i = 0; str[i] != '\0'; i++)
{
    if (str[i] == '-')
    {
        k[j].pos = i;
        k[j++].op = '-';
    }
}
}

void explore()
{
    i = 1;
    while (k[i].op != '\0')
    {
        fleft(k[i].pos);
        fright(k[i].pos);
        str[k[i].pos] = tmpch--;
        printf("\t%c := %s%c%s\t\t",
            str[k[i].pos], left, k[i].op, right);
        for (j = 0; j < strlen(str); j++)
            if (str[j] != '$')
                printf("%c", str[j]);
        printf("\n");
        i++;
    }
}

```

```

fright(-1);
if (no == 0)
{
    fleft(strlen(str));
    printf("\t%s := %s", right, left);
    exit(0);
}
printf("\t%s := %c", right, str[k--i].pos)];
}
void fleft(int x)
{
    int w = 0, flag = 0;
    x--;
    while (x != -1 && str[x] != '+' && str[x] != '*' && str[x] != '=' && str[x] != '\0' && str[x] != '
' && str[x] != '/' && str[x] != ':')
    {
        if (str[x] != '$' && flag == 0)
        {
            left[w++] = str[x];
            left[w] = '\0';
            str[x] = '$';
            flag = 1;
        }
        x--;
    }
}
void fright(int x)
{
    int w = 0, flag = 0;
    x++;
    while (x != -1 && str[x] != '+' && str[x] != '*' && str[x] != '\0' &&

```

```

        str[x] != '=' && str[x] != ':' && str[x] != '-' && str[x] != '/')
    {
        if (str[x] != '$' && flag == 0)
        {
            right[w++] = str[x];
            right[w] = '\0';
            str[x] = '$';
            flag = 1;
        }
        x++;
    }
}

```

OUTPUT:

```

                                INTERMEDIATE CODE GENERATION

Enter the Expression :a+b*c-d
The intermediate code:
    Z := a+b          Expression
    Y := c-d          Z*c-d
    Z := Y             Z*Y

```

CONCLUSION:

We conclude that, the implemented C program effectively generates intermediate code for a given expression. It extracts operators and operands to produce a sequence of instructions for further compilation and optimization.

LAB 6

TITLE: WAP IN C PROGRAM FOR FINAL CODE GENERATION.

THEORY:

A final code generator, also known as a back-end compiler phase, is responsible for generating the actual machine code or executable program from the intermediate code produced by earlier compiler phases. It involves translating the intermediate representation into the target machine's specific assembly language or binary instructions.

During code generation, the compiler analyzes the intermediate code, symbol tables, and other relevant information to make informed decisions about instruction selection and register allocation. It aims to optimize the code for performance, size, or a balance between the two, depending on the compiler's optimization goals.

The final code generator is a critical component of a compiler, as it transforms the high-level program into executable code that can be directly executed by the target machine. It plays a crucial role in bridging the gap between the abstract representations used during compilation and the concrete instructions executed by the hardware.

Overall, the final code generator is an essential phase in the compilation process, enabling the translation of intermediate code into executable code that can effectively run on the target machine architecture.

SOURCE CODE:

```
#include <stdio.h>
#include <string.h>
char op[2], arg1[5], arg2[5], result[5];
int main()
{
    FILE *fp1, *fp2;
    fp1 = fopen("input.txt", "r");
    fp2 = fopen("output.txt", "w");
    while (!feof(fp1)){
        fscanf(fp1, "%s%s%s%s", op, arg1, arg2, result);
```

```

    if (strcmp(op, "+") == 0){
        fprintf(fp2, "\n MOV R0, %s", arg1);
        fprintf(fp2, "\n ADD R0, %s", arg2);
        fprintf(fp2, "\n MOV %s, R0", result);
    }
    if (strcmp(op, "*") == 0){
        fprintf(fp2, "\n MOV R0, %s", arg1);
        fprintf(fp2, "\n MUL R0, %s", arg2);
        fprintf(fp2, "\n MOV %s, R0", result);
    }
    if (strcmp(op, "-") == 0){
        fprintf(fp2, "\n MOV R0, %s", arg1);
        fprintf(fp2, "\n SUB R0, %s", arg2);
        fprintf(fp2, "\n MOV %s, R0", result);
    }
    if (strcmp(op, "/") == 0){
        fprintf(fp2, "\n MOV R0, %s", arg1);
        fprintf(fp2, "\n DIV R0, %s", arg2);
        fprintf(fp2, "\n MOV %s, R0", result);
    }
    if (strcmp(op, "=") == 0){
        fprintf(fp2, "\n MOV R0, %s", arg1);
        fprintf(fp2, "\n MOV %s, R0", result);
    }
}
fclose(fp1);
fclose(fp2);
printf("Written into file successfully");
return 0;
}

```

INPUT TEXT FILE:

```
≡ input.txt
1  + a b t1
2  * c d t2
3  - t1 t2 t
4  = t ? x
```

OUTPUT:

```
Written into file successfully
```

```
≡ output.txt
1
2  MOV R0, a
3  ADD R0, b
4  MOV t1, R0
5  MOV R0, c
6  MUL R0, d
7  MOV t2, R0
8  MOV R0, t1
9  SUB R0, t2
10 MOV t, R0
11 MOV R0, t
12 MOV x, R0
```

CONCLUSION:

We concluded that, the implemented C program successfully generates the final code based on the input provided. It effectively translates operations and produces output in the desired format, aiding in code optimization and execution.