SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

| Laboratory Manual |
| --- |

**TABLE OF CONTENT**

| **Sr. No** | **Experiment Title** | | | |
| --- | --- | --- | --- | --- |
|  | | | |
| 1 |  | 1. Write a program to recognize strings starts with ‘a’ over {a, b}. 2. Write a program to recognize strings end with ‘a’. 3. Write a program to recognize strings end with ‘ab’. Take the input from text file. 4. Write a program to recognize strings contains ‘ab’. Take the input from text file. | | |
| 2 |  | 1. Write a program to recognize the valid identifiers and keywords. 2. Write a program to recognize the valid operators. 3. Write a program to recognize the valid number. 4. Write a program to recognize the valid comments. 5. Program to implement Lexical Analyzer. | | |
| 3 |  | To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer) | | |
| 4 |  | Implement following programs using Lex.  a. Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.  b. Write a Lex program to take input from text file and count number of vowels and consonants.  c. Write a Lex program to print out all numbers from the given file.  d. Write a Lex program which adds line numbers to the given file and display the same into different file.  e. Write a Lex program to printout all markup tags and HTML comments in file. | | |
| 5 |  | a. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.  b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program. | | |
| 6 |  | Program to implement Recursive Descent Parsing in C. | | |
| 7 |  | a. To Study about Yet Another Compiler-Compiler(YACC).  b. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, \* and / .  c. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.  d. Create Yacc and Lex specification files are used to convert infix expression to postfix expression. | | |

**Q1.**

**Aim: a) Write a program to recognize strings starts with ‘a’ over {a, b}.**

**Code:**

#include<stdio.h>

int main(){

char input[10];

int i = 0;

printf("Enter input string to check in the automata: ");

scanf("%s", input);

int state = 0;

while(input[i]!= '\0'){

switch(state){

case 0:

if (input[i]=='a')

{

/\* code \*/

state = 1;

}

else if (input[i] == 'b')

{

/\* code \*/

state = 2;

}

else

{

state = 3;

}

break;

case 1:

if (input[i] == 'a' || input[i] == 'b')

{

/\* code \*/

state = 1;

}

else

{

state = 3;

}

break;

case 2:

if (input[i] == 'a' || input[i] == 'b')

{

/\* code \*/

state = 2;

}

else

{

state = 3;

}

break;

case 3:

state = 3;

}

i++;

}

if(state == 1) printf("Input string is valid");

else if(state == 2 || state == 0) printf("Input string is not valid");

else if(state == 3) printf("String is not recogized");

return 0;

}

**Output:**

****

**Aim: b) Write a program to recognize strings end with ‘a’.**

**Code:**

#include<stdio.h>

int main() {

char input[10];

int state=0, i=0;

printf("Enter the input string: ");

scanf("%s",input);

while(input[i]!='\0') {

switch(state) {

case 0:

if(input[i]=='a') state=1;

else state=0;

break;

case 1:

if(input[i]=='a') state=1;

else state=0;

break;

}

i++;

}

if(state==0) printf("String is invalid!");

else printf("String is valid!");

return 0;

}

**Output:**

****

**Aim: c) Write a program to recognize strings end with ‘ab’. Take the input from text file.**

**Code:**

#include <stdio.h>

#include <string.h>

int main() {

FILE \*file = fopen("input.txt", "r");

if (!file) {

printf("Failed to open input.txt\n");

return 1;

}

char input[100];

while (fgets(input, sizeof(input), file)) {

input[strcspn(input, "\n")] = '\0'; // Remove newline

int state = 0, i = 0;

while (input[i] != '\0') {

switch (state) {

case 0:

if (input[i] == 'a') state = 1;

else state = 0;

break;

case 1:

if (input[i] == 'b') state = 2;

else if (input[i] == 'a') state = 1;

else state = 0;

break;

case 2:

if (input[i] == 'a') state = 1;

else state = 0;

break;

}

i++;

}

if (state == 2 && input[strlen(input) - 2] == 'a' && input[strlen(input) - 1] == 'b') {

printf("Accepted: %s ends with 'ab'\n", input);

} else {

printf("Rejected: %s does not end with 'ab'\n", input);

}

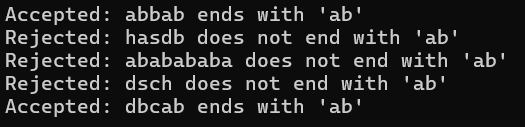
}

fclose(file);

return 0;

}

**Output:**

****

**Aim: d) Write a program to recognize strings contains ‘ab’. Take the input from text file.**

**Code:**

#include <stdio.h>

#include <string.h>

int main() {

FILE \*file = fopen("input.txt", "r");

if (!file) {

printf("Failed to open input.txt\n");

return 1;

}

char input[100];

while (fgets(input, sizeof(input), file)) {

// Remove newline character if present

input[strcspn(input, "\n")] = '\0';

int state = 0, i = 0, accepted = 0;

while (input[i] != '\0') {

switch (state) {

case 0:

if (input[i] == 'a') state = 1;

else state = 0;

break;

case 1:

if (input[i] == 'b') {

state = 2;

accepted = 1;

} else if (input[i] == 'a') {

state = 1;

} else {

state = 0;

}

break;

case 2:

// Already accepted

break;

}

if (accepted) break; // Stop if "ab" is found

i++;

}

if (accepted) {

printf("Accepted: %s contains 'ab'\n", input);

} else {

printf("Rejected: %s does not contain 'ab'\n", input);

}

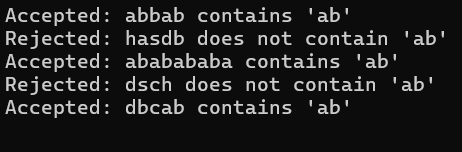
}

fclose(file);

return 0;

}

**Output:**

****

**Q2.**

**Aim: a) Write a program to recognize the valid identifiers and keywords.**

**Code:**

#include <stdio.h>

#include <ctype.h>

int main() {

FILE \*file;

char filename[] = "input.txt";

char ch;

int state = 0, i = 0;

file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file");

return 1;

}

while ((ch = fgetc(file)) != EOF) {

switch (state) {

case 0:

if (ch == 'i')

state = 1;

else if (isalpha(ch))

state = 4;

else if (ch == ' ')

state = 5;

break;

case 1:

if (ch == 'n')

state = 3;

else if (isalpha(ch) || isdigit(ch))

state = 4;

else if (ch == ' ')

state = 5;

break;

case 3:

if (ch == 't')

state = 6;

else if (isalpha(ch) || isdigit(ch))

state = 4;

break;

case 4:

if (isalpha(ch) || isdigit(ch))

state = 4;

else

state = 5;

break;

}

i++;

}

if (state == 6)

printf("Found 'int' keyword\n");

else if (state == 4)

printf("String is a valid identifier\n");

else

printf("Invalid identifier\n");

fclose(file);

return 0;

}

**Output:**

****

**Aim: b) Write a program to recognize the valid operators.**

**Code:**

#include <stdio.h>

#include <string.h>

int main() {

char input[10];

int state = 1, i = 0;

printf("Enter the input string: ");

scanf("%s", input);

while (input[i] != '\0') {

switch (state) {

case 1:

if (input[i] == '+') state = 2;

else if (input[i] == '-') state = 5;

else if (input[i] == '\*') state = 9;

else if (input[i] == '/') state = 12;

else if (input[i] == '%') state = 15;

else if (input[i] == '=') state = 18;

else state = -1;

break;

case 2:

if (input[i] == '+') state = 3; // Unary ++

else if (input[i] == '=') state = 4; // Assignment +=

else state = 17; // Arithmetic +

break;

case 5:

if (input[i] == '-') state = 6; // Unary --

else if (input[i] == '=') state = 7; // Assignment -=

else state = 8; // Arithmetic -

break;

case 9:

if (input[i] == '=') state = 10; // Assignment \*=

else state = 11; // Arithmetic \*

break;

case 12:

if (input[i] == '=') state = 13; // Assignment /=

else state = 14; // Arithmetic /

break;

case 15:

if (input[i] == '=') state = 16; // Assignment %=

else state = 17; // Arithmetic %

break;

case 18:

if (input[i] == '=') state = 19; //Relational ==

else state = 20; //Assignment =

break;

}

i++;

}

// Final states classification

if (state == 3) printf("It is a unary operator: ++\n");

else if (state == 6) printf("It is a unary operator: --\n");

else if (state == 4) printf("It is an assignment operator: +=\n");

else if (state == 7) printf("It is an assignment operator: -=\n");

else if (state == 10) printf("It is an assignment operator: \*=\n");

else if (state == 13) printf("It is an assignment operator: /=\n");

else if (state == 16) printf("It is an assignment operator: %=\n");

else if (state == 17) printf("It is an arithmetic operator: +, -, \*, /, %\n");

else if (state == 8) printf("It is an arithmetic operator: -\n");

else if (state == 11) printf("It is an arithmetic operator: \*\n");

else if (state == 14) printf("It is an arithmetic operator: /\n");

else if (state == 17) printf("It is an arithmetic operator: %\n");

else if (state == 19) printf("It is a relational operator: ==\n");

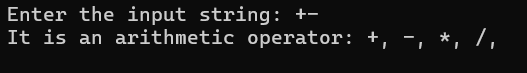
else if (state == 20) printf("It is an assignment operator: =\n");

else printf("Invalid input!\n");

return 0;

}

**Output:**

****

**Aim: c) Write a program to recognize the valid number.**

**Code:**

#include <stdio.h>

#include <ctype.h>

int main() {

char input[100];

int i = 0, state = 0;

printf("Enter number: ");

scanf("%s", input);

char ch;

while ((ch = input[i++]) != '\0') {

switch (state) {

case 0:

if (isdigit(ch)) state = 1;

else state = -1;

break;

case 1:

if (isdigit(ch)) state = 1;

else if (ch == '.') state = 2;

else state = -1;

break;

case 2:

if (isdigit(ch)) state = 3;

else state = -1;

break;

case 3:

if (isdigit(ch)) state = 3;

else state = -1;

break;

}

if (state == -1) break;

}

if (state == 1)

printf("'%s' is a valid integer.\n", input);

else if (state == 3)

printf("'%s' is a valid floating-point number.\n", input);

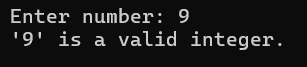
else

printf("'%s' is not a valid number.\n", input);

return 0;

}

**Output:**

****

**Aim: d) Write a program to recognize the valid comments.**

**Code:**

#include<stdio.h>

#include<string.h>

int main() {

char input[100];

int state = 0, i = 0;

printf("Enter the input string: ");

fgets(input, sizeof(input), stdin);

while (input[i] != '\0' && input[i] != '\n') { // Process until end of string or newline

switch (state) {

case 0:

if (input[i] == '/') state = 1; // Possible start of a comment

else state = 0; // Stay in state 0 for other characters

break;

case 1:

if (input[i] == '/') state = 2; // Single-line comment detected

else if (input[i] == '\*') state = 3; // Multi-line comment start detected

else state = 0; // Reset to initial state if not a comment

break;

case 2:

// Single-line comment: Remain in state 2 for the rest of the string

state = 2;

break;

case 3:

if (input[i] == '\*') state = 4; // Check for potential end of multi-line comment

else state = 3; // Stay in multi-line comment

break;

case 4:

if (input[i] == '/') state = 5; // Multi-line comment end detected

else if (input[i] == '\*') state = 4; // Stay in end-checking state

else state = 3; // Go back to multi-line comment state

break;

}

i++;

}

if (state == 2) {

printf("It is a single-line comment!\n");

} else if (state == 5) {

printf("It is a multi-line comment!\n");

} else {

printf("It is not a comment!\n");

}

return 0;

}

**Output:**

****

**Aim: e) Program to implement Lexical Analyzer.**

**Code:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

int isKeyword(char \*word) {

const char \*keywords[] = { "int", "float", "if", "else", "return", "while" };

for (int i = 0; i < 6; i++) {

if (strcmp(word, keywords[i]) == 0) return 1;

}

return 0;

}

int main() {

char input[100];

printf("Enter input: ");

fgets(input, sizeof(input), stdin);

int i = 0;

while (input[i] != '\0') {

if (isspace(input[i])) {

i++;

continue;

}

// Identifiers or keywords

if (isalpha(input[i]) || input[i] == '\_') {

char buffer[20];

int j = 0;

while (isalnum(input[i]) || input[i] == '\_') {

buffer[j++] = input[i++];

}

buffer[j] = '\0';

if (isKeyword(buffer))

printf("Keyword: %s\n", buffer);

else

printf("Identifier: %s\n", buffer);

}

// Numbers

else if (isdigit(input[i])) {

char num[20];

int j = 0;

while (isdigit(input[i]) || input[i] == '.') {

num[j++] = input[i++];

}

num[j] = '\0';

printf("Number: %s\n", num);

}

// Operators

else if (strchr("+-\*/=<>!", input[i])) {

char op[3] = { input[i], '\0', '\0' };

if ((input[i+1] == '=')) {

op[1] = '=';

i++;

}

printf("Operator: %s\n", op);

i++;

}

else {

printf("Unknown token: %c\n", input[i]);

i++;

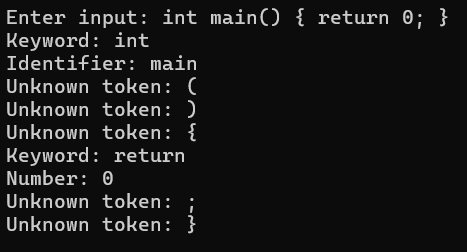
}

}

return 0;

}

**Output:**

****

**Q3. To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)**

**Procedure:**

What is a Lexical Analyzer?

A Lexical Analyzer (also called a scanner or lexer) is the first phase of a compiler. It processes the source code to break it into tokens, which are meaningful character sequences such as identifiers, keywords, numbers, operators, and symbols. These tokens are passed to the parser for syntax analysis. Functions of a Lexical Analyzer:

* Removes whitespace and comments
* Recognizes tokensProvides token information to the parser
* Reports lexical errors
* Lexical Analyzer Generator:
* A Lexical Analyzer Generator is a tool that automatically generates source code (usually in C or C++) to perform lexical analysis. Instead of manually writing a lexical analyzer, developers write regular expressions and actions in a specific format, and the generator creates the scanner code.
* Definition: LEX is a tool for generating lexical analyzers. It was developed in the early 1970s by Mike Lesk and Eric Schmidt at AT&T Bell Labs.
* Purpose: LEX helps in automatically generating C code for lexical analysis from regular expressions.
* Structure: LEX programs are divided into three sections:

1. Definition Section (%{ ... %}) – Contains header files or macros.
2. Rules Section (%%) – Each line defines a pattern (regular expression) and an action (C code).
3. Code Section (%%) – Optional user-defined C functions (e.g., main()).

* Output: The LEX tool generates a file called lex.yy.c containing the C source code of the lexer.
* Flex: Fast Lexical Analyzer Generator
* Definition: Flex (Fast Lex) is an improved version of LEX, created to be faster, more powerful, and more portable. It is open-source and commonly used in Unix/Linux environments.
* Features:

1. Compatible with LEX
2. Faster and more efficient
3. Open-source and actively maintained
4. Generates C code from .l files
5. Provides better error handling and debugging tools

* Output: Like LEX, Flex generates a lex.yy.c file, which is then compiled using a C compiler (e.g., gcc) to create the lexical analyzer.
* Working Process of LEX/Flex

1. Write a Lex/Flex file (e.g., scanner.l) with token patterns.
2. Run: lex scanner.l or flex scanner.l
3. Compile: gcc lex.yy.c -o scanner -lfl
4. Execute: ./scanner
5. The analyzer reads input, matches patterns, and executes actions.

* Regular Expressions in LEX/Flex
* [a-z] – any lowercase letter
* [0-9]+ – one or more digits
* int|float – matches int or float
* [ \t\n]+ – matches whitespace
* Advantages of Using LEX/Flex
* Simplifies scanner development
* Reduces manual errors
* Supports complex pattern matching
* Easily integrates with YACC/Bison (parser generators)
* Applications
* Compilers (tokenizing source code)
* Interpreters for custom scripting languages
* Syntax highlighting tools
* Static code analyzers
* Log file analyzers
* Conclusion: LEX and Flex are powerful tools in the domain of compiler design and language processing. They automate the process of building lexical analyzers using regular expressions and actions, drastically reducing development time and errors. While LEX is the historical tool, Flex is the modern, efficient alternative preferred in most practical applications today.

**Q4.**

**Aim: a) Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.**

**Code:**

%{

#include <stdio.h>

int char\_count = 0, word\_count = 0, line\_count = 0;

%}

%%

\n { line\_count++; }

[a-zA-Z]+ { word\_count++; }

. { char\_count++; }

%%

int main() {

FILE \*file = fopen("input.txt", "r");

if (!file) {

printf("Failed to open file.\n");

return 1;

}

yyin = file;

yylex();

fclose(file);

printf("Characters: %d\n", char\_count);

printf("Words: %d\n", word\_count);

printf("Lines: %d\n", line\_count);

return 0;

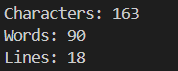
}

int yywrap() {

return 1;

}

**Output:**

****

**Aim: b) Write a Lex program to take input from text file and count number of vowels and consonants.**

**Code:**

%{

#include <stdio.h>

int vowel\_count = 0, consonant\_count = 0;

%}

%%

[aAeEiIoOuU] { vowel\_count++; }

[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { consonant\_count++; }

. ;

%%

int main() {

FILE \*file = fopen("input.txt", "r");

if (!file) {

printf("Failed to open file.\n");

return 1;

}

yyin = file;

yylex();

fclose(file);

printf("Vowels: %d\n", vowel\_count);

printf("Consonants: %d\n", consonant\_count);

return 0;

}

int yywrap() {

return 1;

}

**Output:**

****

**Aim: c) Write a Lex program to print out all numbers from the given file.**

**Code:**

%{

#include <stdio.h>

%}

%%

[0-9]+ { printf("Number: %s\n", yytext); }

. ;

%%

int main() {

FILE \*file = fopen("input.txt", "r");

if (!file) {

printf("Failed to open file.\n");

return 1;

}

yyin = file;

yylex();

fclose(file);

return 0;

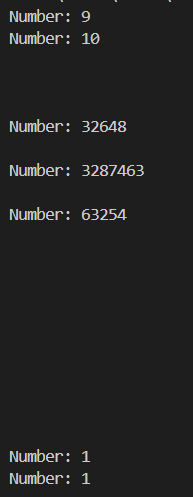
}

int yywrap() {

return 1;

}

**Output:**

****

**Aim: d) Write a Lex program which adds line numbers to the given file and display the same into different files.**

**Code:**

%{

#include <stdio.h>

int line\_num = 1;

%}

%%

\n { printf("%d: ", line\_num++); }

. { putchar(yytext[0]); }

%%

int main() {

FILE \*input\_file = fopen("input.txt", "r");

FILE \*output\_file = fopen("output.txt", "w");

if (!input\_file || !output\_file) {

printf("Error opening file(s).\n");

return 1;

}

yyin = input\_file;

yyout = output\_file;

yylex();

fclose(input\_file);

fclose(output\_file);

printf("Line numbers added to 'output.txt'.\n");

return 0;

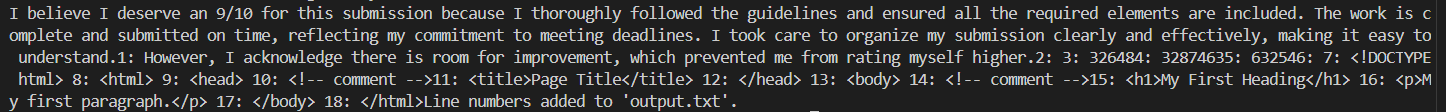
}

int yywrap() {

return 1;

}

**Output:**

****

**Aim: e) Write a Lex program to printout all markup tags and HTML comments in file.**

**Code:**

%{

#include <stdio.h>

extern FILE \*yyin;

%}

%%

"<!--"([^-\n]|"-"[^-])\*"-->" { printf("HTML Comment: %s\n", yytext); }

"<[^>]+>" { printf("HTML Tag: %s\n", yytext); }

. ; // Ignore all other characters

%%

int main() {

yyin = fopen("trial.html", "r");

if (!yyin) {

perror("Error opening input.html");

return 1;

}

yylex();

fclose(yyin);

return 0;

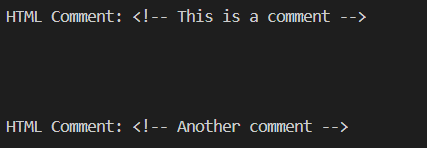
}

int yywrap() {

return 1;

}

**Output:**

****

**Q5.**

**Aim: a) Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.**

**Code:**

%{

#include <stdio.h>

int comment\_count = 0;

FILE \*out;

%}

%%

"//".\* { comment\_count++; /\* Skip single-line comment \*/ }

"/\*"([^\*]|\\*[^/])\*"\*"+"/" { comment\_count++; /\* Skip multi-line comment \*/ }

.|\n { fputc(yytext[0], out); } // Copy everything else

%%

int main() {

FILE \*in = fopen("input.c", "r");

out = fopen("output.c", "w");

if (!in || !out) {

perror("Error opening file");

return 1;

}

yyin = in;

yylex();

fclose(in);

fclose(out);

printf("Total comment blocks removed: %d\n", comment\_count);

return 0;

}

int yywrap() {

return 1;

}

**Output:**

****

**Aim: b) Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.**

**Code:**

%{

#include <stdio.h>

%}

KEYWORD int|float|char|double|if|else|while|for|return|void

IDENTIFIER [a-zA-Z\_][a-zA-Z0-9\_]\*

NUMBER [0-9]+(\.[0-9]+)?

OPERATOR (\+|\-|\\*|\/|\=|\==|\!=|\<|\>|\<=|\>=)

LITERAL \"([^\"\n]|\\\")\*\"

SPECIAL [\(\)\{\}\[\]\;\,\&]

%%

{KEYWORD} { printf("Keyword: %s\n", yytext); }

{IDENTIFIER} { printf("Identifier: %s\n", yytext); }

{NUMBER} { printf("Number: %s\n", yytext); }

{LITERAL} { printf("String Literal: %s\n", yytext); }

{OPERATOR} { printf("Operator: %s\n", yytext); }

{SPECIAL} { printf("Special Symbol: %s\n", yytext); }

[ \t\n]+ ; // ignore whitespace

. { printf("Unknown: %s\n", yytext); }

%%

int main() {

FILE \*in = fopen("input.c", "r");

if (!in) {

perror("Failed to open input.c");

return 1;

}

yyin = in;

yylex();

fclose(in);

return 0;

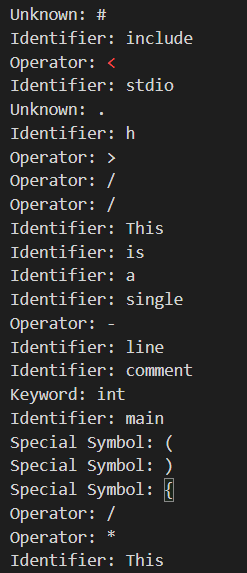
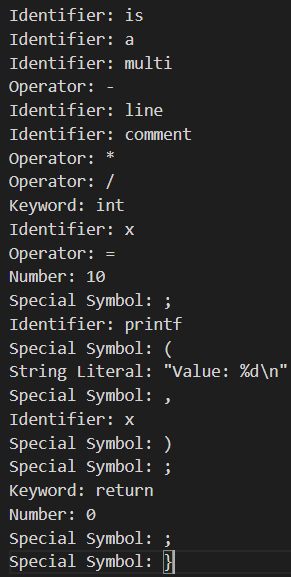
}

int yywrap() {

return 1;

}

**Output:**

** **

**Q6. Program to implement Recursive Descent Parsing in C.**

**Code:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#include <stdlib.h>

#define SUCCESS 1

#define FAILED 0

const char \*input;

int pos = 0;

// Function prototypes

int E(), Edash(), T(), Tdash(), F();

void skipWhitespace() {

while (isspace(input[pos])) {

pos++;

}

}

char lookahead() {

skipWhitespace();

return input[pos];

}

void match(char expected) {

if (lookahead() == expected) {

pos++;

} else {

printf("Syntax Error: expected '%c' at position %d\n", expected, pos);

exit(1);

}

}

int E() {

printf("%-16s E -> T E'\n", input + pos);

if (T()) {

if (Edash()) return SUCCESS;

}

return FAILED;

}

int Edash() {

if (lookahead() == '+') {

printf("%-16s E' -> + T E'\n", input + pos);

match('+');

if (T()) {

if (Edash()) return SUCCESS;

}

return FAILED;

} else {

printf("%-16s E' -> $\n", input + pos);

return SUCCESS;

}

}

int T() {

printf("%-16s T -> F T'\n", input + pos);

if (F()) {

if (Tdash()) return SUCCESS;

}

return FAILED;

}

int Tdash() {

if (lookahead() == '\*') {

printf("%-16s T' -> \* F T'\n", input + pos);

match('\*');

if (F()) {

if (Tdash()) return SUCCESS;

}

return FAILED;

} else {

printf("%-16s T' -> $\n", input + pos);

return SUCCESS;

}

}

int F() {

if (lookahead() == '(') {

printf("%-16s F -> ( E )\n", input + pos);

match('(');

if (E()) {

if (lookahead() == ')') {

match(')');

return SUCCESS;

}

}

return FAILED;

} else if (lookahead() == 'i') {

printf("%-16s F -> i\n", input + pos);

match('i');

return SUCCESS;

} else {

printf("Syntax Error: unexpected character '%c' at position %d\n", lookahead(), pos);

return FAILED;

}

}

int main() {

char buffer[100];

printf("Enter the string\n");

fgets(buffer, sizeof(buffer), stdin); // use fgets to accept spaces

input = buffer;

printf("\nInput Action\n");

printf("--------------------------------\n");

if (E() && lookahead() == '$') {

printf("--------------------------------\n");

printf("String is successfully parsed\n");

return 0;

} else {

printf("--------------------------------\n");

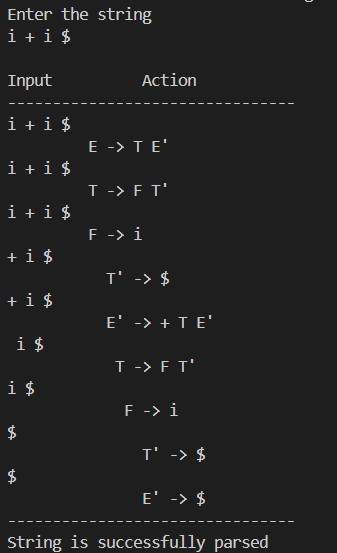
printf("Error in parsing String\n");

return 1;

}

}

**Output:**

****

**Q7.**

**Aim: a) Study of Yet Another Compiler-Compiler (YACC)**

YACC (Yet Another Compiler-Compiler) is a tool that generates parsers, specifically for context-free grammars, which are commonly used in the design and implementation of compilers and interpreters. It is a parser generator that takes a formal grammar description and produces a parser that can process input strings according to that grammar.

YACC is primarily used for creating syntax analyzers (parsers) in the context of compilers and other systems that require the interpretation or validation of structured text (e.g., programming languages, configuration files, etc.).

* Introduction to YACC: YACC is a tool that provides a high-level method for defining parsers. It is a LALR(1) parser generator, meaning it constructs parsers that follow Look-Ahead, Left-to-Right parsing strategy with a 1-symbol lookahead. The tool is often used with Lex, a tool that generates lexical analyzers (scanners), to build compilers and interpreters.
* A typical YACC input is a file that contains:

1. A grammar specification (productions).
2. Action code (usually C code) that is executed when a rule is matched.
3. Optionally, semantic actions that are linked to the grammar rules.
4. YACC is widely used in the development of compilers, interpreters, static analysis tools, and other applications that require parsing of structured text.

* Components of YACC

YACC operates in three main parts:

1. Input Grammar: Defines the syntax of the language being parsed (often described using Backus-Naur Form or Context-Free Grammar).
2. Semantic Actions: The code to be executed when a grammar rule is matched. These are typically written in C and can perform tasks such as creating a parse tree or checking for semantic errors.
3. Parser: The parser generated by YACC based on the grammar and semantic actions.

* YACC Syntax and Structure

A YACC file typically has three sections:

1. Declarations Section (Optional): Contains C code for including libraries, defining tokens, or defining external variables.
2. Grammar Rules Section: Contains the grammar rules, with C code embedded to perform actions when rules are matched.
3. Code Section: Contains any C code to be used for the parser’s operations, such as helper functions and additional declarations.

* Here is a simple YACC example that parses basic arithmetic expressions.

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token NUM

%%

expr:

term { printf("Single term\n"); }

| expr '+' term { printf("Add operation\n"); }

;

term:

NUM { printf("Term: %d\n", $1); }

;

%%

int main(void) {

printf("Enter an expression:\n");

yyparse();

return 0;

}

int yyerror(char \*s) {

fprintf(stderr, "%s\n", s);

return 0;

}

In this example: The grammar rules define an expression (expr) that can either be a term or a term followed by a + operator and another term. The NUM token represents a number and is matched in the term rule. The C code embedded inside {} gets executed whenever a rule is matched, such as printing messages when specific operations occur.

* Steps to Use YACC

The general steps for using YACC to create a parser are:

1. Write the YACC file: Create a .y file that contains the grammar rules and C code for actions.
2. Run YACC: Process the .y file using the YACC tool, which generates a C file (usually y.tab.c).
3. Example: yacc -d example.y This command generates y.tab.c (the parser code) and y.tab.h (the header file with token definitions).
4. Write a Lex file (optional): If your parser depends on a lexer (such as to handle tokens like NUM in the example), write a .l file using the Lex tool.
5. Compile the generated C code: Compile the generated parser code along with any Lex- generated code (usually lex.yy.c) to create the executable.
6. Example: gcc -o parser y.tab.c lex.yy.c -ll Run the parser: Execute the compiled parser to analyze input according to the grammar. Example:./parser

* Advantages of YACC

1. High-Level Specification: YACC allows defining complex grammars in a concise, readable manner, avoiding manual management of parsing tables or other low-level operations.
2. Integration with Lex: YACC works seamlessly with Lex to generate both the lexical analyzer and parser for a compiler.
3. Automation: YACC automates much of the process of parsing, allowing developers to focus on grammar rules and semantic actions.
4. Flexibility: YACC can be customized with C code to perform various tasks such as building abstract syntax trees, symbol tables, or handling errors.

* Applications of YACC

YACC is commonly used in various domains, including:

1. Compiler Design: For building parsers for programming languages.
2. Interpreter Design: Used for building interpreters for custom languages or domain- specific languages (DSLs).
3. Static Analysis Tools: To analyze source code for patterns, errors, or optimizations.
4. Data Validation: For validating structured data formats (e.g., XML, JSON).
5. Protocol Parsing: In network communication protocols where the data structure follows a well-defined grammar.

* Common Errors in YACC

1. Syntax Errors: Errors in the grammar rules, such as mismatched parentheses or incorrect rule definitions.
2. Shift/Reduce Conflicts: Occurs when YACC cannot decide whether to shift a symbol or reduce it by a rule. This often happens in ambiguous grammars.
3. Reduce/Reduce Conflicts: When there are multiple rules that could match the same input, YACC may encounter ambiguity in deciding which rule to reduce.
4. To resolve conflicts, the user can manually adjust the grammar, use precedence and associativity rules, or employ techniques like rewriting ambiguous rules.

* Conclusion: YACC is an essential tool for compiler construction, offering an efficient and automated way to generate parsers from formal grammar definitions. While YACC handles the parsing process, it is typically used in conjunction with Lex (a lexical analyzer generator) to handle tokenization of input strings. YACC simplifies complex grammar specifications, making it easier to develop compilers, interpreters, and other applications that require structured text parsing.

**Aim: b) Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, \* and / .**

**Code:**

* Lex file:

%{

#include <stdlib.h>

#include "q1.tab.h"

void yyerror(const char \*); // declare yyerror to avoid compiler warning

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUM; }

[+\-\*/()] { return yytext[0]; }

[ \t] ; // skip whitespace

\n { return '\n'; }

. { yyerror("invalid character"); }

%%

int yywrap() {

return 1;

}

* Yacc File:

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

void yyerror(const char \*);

%}

%token NUM

%%

S : E '\n' { printf("Valid expression\n"); }

;

E : E '+' T

| E '-' T

| T

;

T : T '\*' F

| T '/' F

| F

;

F : '(' E ')'

| NUM

;

%%

void yyerror(const char \*s) {

printf("Error: %s\n", s);

}

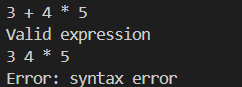
int main() {

yyparse();

return 0;

}

**Output:**

****

**Aim: c) Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.**

**Code:**

* Lex File:

%{

#include "q2.tab.h"

%}

%%

[0-9]+ { yylval.num = atoi(yytext); return NUMBER; }

[+\-\*/()\n] { return yytext[0]; }

[ \t] ; // skip whitespace

. { printf("Unknown character: %s\n", yytext); }

%%

int yywrap() {

return 1;

}

* Yacc File:

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

void yyerror(char \*);

%}

%union {

int num;

}

%token <num> NUMBER

%type <num> E T F

%%

S : E '\n' { printf("Result = %d\n", $1); }

;

E : E '+' T { $$ = $1 + $3; }

| E '-' T { $$ = $1 - $3; }

| T

;

T : T '\*' F { $$ = $1 \* $3; }

| T '/' F { $$ = $1 / $3; }

| F

;

F : '(' E ')' { $$ = $2; }

| NUMBER

;

%%

void yyerror(char \*s) {

printf("Error: %s\n", s);

}

int main() {

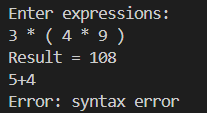
printf("Enter expressions:\n");

yyparse();

return 0;

}

**Output:**

****

**Aim: 7d) Create Yacc and Lex specification files are used to convert infix expression to postfix expression.**

**Code:**

* Lex File:

%{

#include "q3.tab.h"

#include <stdlib.h>

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

[+\-\*/()] { return yytext[0]; }

\n { return '\n'; }

[ \t] ; // skip whitespace

. { printf("Unknown character: %s\n", yytext); }

%%

int yywrap() {

return 1;

}

* Yacc File:

%{

#include <stdio.h>

#include <stdlib.h>

int yylex(void);

void yyerror(const char \*);

%}

%token NUMBER

%left '+' '-'

%left '\*' '/'

%right UMINUS

%%

input:

/\* empty \*/

| input line

;

line:

expr '\n' { printf("\n"); }

;

expr:

expr '+' expr { printf("+ "); }

| expr '-' expr { printf("- "); }

| expr '\*' expr { printf("\* "); }

| expr '/' expr { printf("/ "); }

| '-' expr %prec UMINUS { printf("~ "); } // Unary minus, optional

| '(' expr ')'

| NUMBER { printf("%d ", $1); }

;

%%

void yyerror(const char \*s) {

fprintf(stderr, "Error: %s\n", s);

}

int main() {

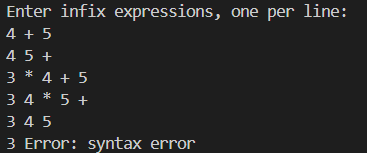
printf("Enter infix expressions, one per line:\n");

yyparse();

return 0;

}

**Output:**

****

***~ End of File ~***