Compiler Design Lab Manual

Bachelor of Technology (CSE)

By

Roma Rajbhar (22000921)

Third Year, Semester 6, Div- B2

Course In-charge: **prof. Vaibhavi Patel**

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Autumn Semester (2024-2025)

TABLE OF CONTENT

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:

```
Enter input string to check in the automata: abab
Input string is valid
```

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:

```
Enter the input string: ababa
String is valid!
```

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:

```
Accepted: abbab ends with 'ab'
Rejected: hasdb does not end with 'ab'
Rejected: abababa does not end with 'ab'
Rejected: dsch does not end with 'ab'
Accepted: dbcab ends with 'ab'
```

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:

```
Accepted: abbab contains 'ab'
Rejected: hasdb does not contain 'ab'
Accepted: abababab a contains 'ab'
Rejected: dsch does not contain 'ab'
Accepted: dbcab contains 'ab'
```

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:

```
Found 'int' keyword
```

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:

```
Enter the input string: +-
It is an arithmetic operator: +, -, *, /,
```

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:

```
Enter number: 9
'9' is a valid integer.
```

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:

```
Enter the input string: /*dhgsv*/
It is a multi-line comment!
```

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:

```
Enter input: int main() { return 0; }
Keyword: int
Identifier: main
Unknown token: (
Unknown token: )
Unknown token: {
Keyword: return
Number: 0
Unknown token: ;
Unknown token: }
```

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:

```
Characters: 163
Words: 90
Lines: 18
```

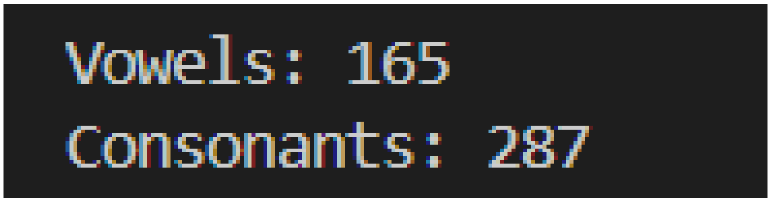
Aim: b) Write a Lex program to take input from text file and count number of vowels and consonants.

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:

```
Number: 9
Number: 10



Number: 32648

Number: 3287463

Number: 63254









Number: 1
Number: 1
```

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:

```
I believe I deserve an 9/10 for this submission because I thoroughly followed the guidelines and ensured all the required elements are included. The work is c
omplete and submitted on time, reflecting my commitment to meeting deadlines. I took care to organize my submission clearly and effectively, making it easy to
 understand.1: However, I acknowledge there is room for improvement, which prevented me from rating myself higher.2: 3: 326484: 32874635: 632546: 7: <!DOCTYPE
 html> 8: <html> 9: <head> 10: <!-- comment -->11: <title>Page Title</title> 12: </head> 13: <body> 14: <!-- comment -->15: <h1>My First Heading</h1> 16: <p>M
y first paragraph.</p> 17: </body> 18: </html>Line numbers added to 'output.txt'.
```

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:

```
HTML Comment: <!-- This is a comment -->




HTML Comment: <!-- Another comment -->
```

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:

```
Total comment blocks removed: 2
```

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:

```
Unknown: #
Identifier: include
Operator: <
Identifier: stdio
Unknown: .
Identifier: h
Operator: >
Operator: /
Operator: /
Identifier: This
Identifier: is
Identifier: a
Identifier: single
Operator: -
Identifier: line
Identifier: comment
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Operator: /
Operator: *
Identifier: This
```

```
Identifier: is
Identifier: a
Identifier: multi
Operator: -
Identifier: line
Identifier: comment
Operator: *
Operator: /
Keyword: int
Identifier: x
Operator: =
Number: 10
Special Symbol: ;
Identifier: printf
Special Symbol: (
String Literal: "Value: %d\n"
Special Symbol: ,
Identifier: x
Special Symbol: )
Special Symbol: ;
Keyword: return
Number: 0
Special Symbol: ;
Special Symbol: }
```

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;
    }
}
```

```c
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:

```
Enter the string
i + i $

Input          Action
--------------------------------
i + i $
         E -> T E'
i + i $
         T -> F T'
i + i $
         F -> i
+ i $
           T' -> $
+ i $
           E' -> + T E'
 i $
            T -> F T'
i $
             F -> i
$
               T' -> $
$
               E' -> $
--------------------------------
String is successfully parsed
```

Q7.

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

Introduction

YACC stands for **Yet Another Compiler-Compiler**. It is a tool used in **compiler design** to generate **parsers**. A parser checks if a given input follows the syntax (structure) of a language. YACC was developed by **Stephen C. Johnson** at Bell Labs in the 1970s and is still widely used in academic and professional environments to learn and build language processors.

Purpose of YACC

The main purpose of YACC is to simplify the creation of the **syntax analysis** phase in a compiler. Writing a parser by hand can be complex and error-prone. YACC solves this by letting the user define grammar rules and automatically generating the corresponding parser code in C.

YACC is used for:

- Automating parser generation
- Simplifying the process of compiler construction
- Creating interpreters and language processors
- Making custom tools that understand structured input

Working of YACC

YACC works hand-in-hand with **LEX**, which performs **lexical analysis** (splitting input into tokens). YACC takes these tokens and checks if they match the rules of the grammar.

Steps involved:

1. **Write grammar rules** in a YACC file.
2. **Generate parser code** using YACC (output is usually `y.tab.c`).
3. **Compile and run** this parser with input.

4. When used with **LEX**, the process becomes a complete front-end system for language processing.

Structure of a YACC File

A YACC program is divided into **three sections**:

➔ `%{`

➔ `   C declarations (like #include, variable declarations)`

➔ `%}`

➔

➔ `YACC Declarations (like token definitions)`

➔

➔ `%%`

➔ `Grammar rules and associated actions`

➔

➔ `%%`

➔ `User-defined functions (like main(), yyerror())`

Features of YACC

- **Supports LALR(1) parsing** (a type of bottom-up parsing)
- Generates **efficient C code** for the parser
- Allows adding custom **actions in C** for each rule
- Works well with **LEX** to create full compilers or interpreters

---

Advantages of YACC

- Reduces manual effort in parser writing

- Easier to modify and test grammar
- Reusable and maintainable code
- Ideal for prototyping new languages or tools

Applications of YACC

- Developing **programming language compilers**
- Creating **interpreters** for small languages
- Parsing **config files, expression evaluators**
- Building **domain-specific languages (DSLs)**

Simple Example: Arithmetic Expression Parser

This is a basic example of a YACC file that parses arithmetic expressions like `2 + 3`.

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

%token NUMBER

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

term: NUMBER          { printf("Number: %d\n", yylval); };
%%

```

```
➔ int main() {
➔     return yyparse();
➔ }
➔
➔ int yyerror(char *s) {
➔     printf("Error: %s\n", s);
➔     return 0;
➔ }
```

In this example:

- `NUMBER` is a token (usually defined by LEX).
- When a rule like `expr + term` is matched, it prints `"Add"`.
- When a number is found, it prints the value.

Conclusion

YACC is a foundational tool in compiler development that makes parser generation easier and more reliable. It helps students understand how programming languages are built and allows developers to create custom parsers for various applications. Even though newer tools exist (like Bison and ANTLR), YACC remains popular for learning and building compact, efficient language tools.

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:

```
3 + 4 * 5
Valid expression
3 4 * 5
Error: syntax error
```

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:

```
Enter expressions:
3 * ( 4 * 9 )
Result = 108
5+4
Error: syntax error
```

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:

```
Enter infix expressions, one per line:
4 + 5
4 5 +
3 * 4 + 5
3 4 * 5 +
3 4 5
3 Error: syntax error
```