



MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

B.TECH. FIFTH SEMESTER

**COMPUTER SCIENCE AND
ENGINEERING
(CSE)**

COMPILER DESIGN

CSE_3161

LABORATORY MANUAL

CONTENTS

| LAB NO. | TITLE | PAGE NO. | REMARKS |
|----------------|--|-----------------|----------------|
| | COURSE OBJECTIVES AND OUTCOMES | iii | |
| | EVALUATION PLAN | iii | |
| | INSTRUCTIONS TO THE STUDENTS | iv | |
| 1 | BASIC FILE HANDLING OPERATION | 1 | |
| 2 | PRELIMINARY SCANNING APPLICATIONS | 5 | |
| 3 | CONSTRUCTION OF TOKEN GENERATOR | 8 | |
| 4 | CONSTRUCTION OF SYMBOL TABLE | 11 | |
| 5 | PROGRAMS ON FLEX | 15 | |
| 6 | RECURSIVE DESCENT(RD) PARSER FOR SIMPLE GRAMMARS | 21 | |
| 7 | RD PARSER FOR DECLARATIONS AND EXPRESSION STATEMENTS | 25 | |
| 8 | RD PARSER FOR DECISION MAKING AND LOOPING STATEMENTS | 27 | |
| 9 | RD PARSER FOR SIMPLE GRAMMARS USING YACC | 28 | |
| 10 | CODE GENERATION | 32 | |
| 11 | MINI PROJECT IMPLEMENTATION | 37 | |
| 12 | MINI PROJECT IMPLEMENTATION | 37 | |

Course Objectives

This laboratory course enables students to

- Understand implementation detail for a small subset of a language applying different techniques studied in this course.
- Implement various phases of compiler using LEX, YACC.

Course Outcomes

At the end of this course, students will have the

- Ability to create preliminary scanning applications and to classify different lexemes.
- Ability to create a lexical analyzer without/with using any lexical generation tools.
- Ability to select a suitable data structure to implement symbol table.
- Ability to implement a parser for a given grammar using any parser generation tools.
- Ability to design an intermediate code generator.
- Ability to design a code generator.

Evaluation Plan

- Internal Assessment Marks : 60 marks
 - ☐ Continuous evaluation: 24 Marks
 - The Continuous evaluation assessment will depend on performance in solving lab and additional Exercises, maintaining the lab report and answering the questions in viva voce or and/or execution the instructions told by the instructors.
 - ☐ Internal Exam: 24 Marks. Internal exam assessment will depend on Execution, writeup and viva.
 - ☐ Mini Project : 12 Marks. Mini project assessment will depend on demonstration of the project and the project report.
- End semester assessment of 2-hour duration : 40 marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session
2. Be in time and follow the institution dress code
3. Must Sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

- Follow the instructions on the allotted exercises
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
 - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
 - Use meaningful names for variables and procedures.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
 - Solved exercise
 - Lab exercises - to be completed during lab hours
 - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition lab with the permission of the faculty concerned.
- Questions for lab tests and examination are not necessarily limited to the questions in the manual but may involve some variations and / or combinations of the questions.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

BASIC FILE HANDLING OPERATIONS**I. Objectives:**

In this lab, student will be able to:

- Getting familiar with various file handling system calls.
- Ability to perform basic file operations.

II. Description: In any programming language it is vital to learn file handling techniques. Many applications will at some point involve accessing folders and files on the hard drive. In C, a stream is associated with a file. A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure. In C language, we use a structure pointer of file type to declare a file.

FILE *fp

Table 1.1 shows some of the built-in functions for file handling

Table 1.1: File Handling functions

| Function | Description |
|------------------|--|
| fopen() | Create a new file or open an existing file |
| fclose() | Closes a file |
| getc() | Reads a character from a file |
| putc() | Writes a character to a file |
| fscanf() | Reads a set of data from a file |
| fprintf() | Writes a set of data to a file |
| getw() | Reads an integer from a file |
| putw() | Writes an integer to a file |
| fseek() | Set the position to desire point |
| ftell() | Gives current position in the file |
| rewind() | Set the position to the beginning point |

1.fopen(): This function accepts two arguments as strings. The first argument denotes the name of the file to be opened and the second signifies the mode in which the file is to be opened. The second argument can be any of the following Syntax: *fp = FILE *fopen(const char *filename, const char *mode); The various modes used in file handling is shown in Table 1.2.

2.fclose(): This function is used for closing opened files. The only argument it accepts is the file pointer. If a program terminates, it automatically closes all opened files. But it is a good programming habit to close any file once it is no longer needed. This helps in better utilization of system resources and is very useful when you are working on numerous files simultaneously. Some operating systems place a limit on the number of

files that can be open at any given point in time.

Syntax: `int fclose(FILE *fp);`

Table 1.2: Various modes in file handling

| File Mode | Description |
|-----------|--|
| r | Opens a text file for reading. |
| w | Creates a text file for writing, if exists, it is overwritten. |
| a | Opens a text file and append text to the end of the file. |
| rb | Opens a binary file for reading. |
| wb | Creates a binary file for writing, if exists, it is overwritten. |
| ab | Opens a binary file and append text to the end of the file. |

3.fscanf() and fprintf(): The functions `fprintf()` and `fscanf()` are similar to `printf()` and `scanf()` except that these functions operate on files and require one additional and first argument to be a file pointer.

Syntax: `fprintf(filepointer,"format specifier",v1,v2,...);`

`fscanf(filepointer,"format specifier",&v1,&v2,...);`

4.getc() and putc(): The functions `getc()` and `putc()` are equivalent to `getchar()` and `putchar()` functions except that these functions require an argument which is the file pointer. Function `getc()` reads a single character from the file which has previously been opened using a function like `fopen()`. Function `putc()` does the opposite, it writes a character to the file identified by its second argument.

Syntax: `getc(in_file);`

`putc(c, out_file);`

Note: The second argument in the `putc()` function must be a file opened in either write or append mode.

5.fseek(): This function positions the next I/O operation on an open stream to a new position relative to the current position.

Syntax: `int fseek(FILE *fp, long int offset, int origin);`

Here `fp` is the file pointer of the stream on which I/O operations are carried on, `offset` is the number of bytes to skip over. The offset can be either positive or negative, denoting forward or backward movement in the file. Origin is the position in the stream to which the offset is applied, this can be one of the following constants:

`SEEK_SET`: offset is relative to beginning of the file

`SEEK_CUR`: offset is relative to the current position in the file

`SEEK_END`: offset is relative to end of the file

Binary stream input and output The functions `fread()` and `fwrite()` are a somewhat complex file handling functions used for reading or writing chunks of data containing NULL characters ('\0') terminating strings.

The function prototype of `fread()` and `fwrite()` is as below :

```
size_t fread(void *ptr, size_t sz, size_t n, FILE *fp);
size_t fwrite(const void *ptr, size_t sz, size_t n, FILE *fp);
```

You may notice that the return type of `fread()` is `size_t` which is the number of items read. You will understand this once you understand how `fread()` works. It reads `n` items, each of size `sz` from a file pointed to by the pointer `fp` into a buffer pointed to by a void pointer `ptr` which is nothing but a generic pointer. Function `fread()` reads it as a stream of bytes and advances the file pointer by the number of bytes read. If it encounters an error or end of-file, it returns a zero, you have to use `feof()` or `ferror()` to distinguish between these two. Function `fwrite()` works similarly, it writes `n` objects of `sz` bytes long from a location pointed to by `ptr`, to a file pointed to by `fp`, and returns the number of items written to `fp`.

III. Solved Exercise:

Write a C program to copy the contents of source file to destination file

A. Algorithm: Copy File Contents

Step 1: Enter the source filename.

Step 2: Check if the file exists. If NOT, display an error message and exit from the program.

Step 3: Enter the destination filename.

Step 4: Read each character from the source file and write into destination file using file pointers until

EOF character is encountered in the source file.

Step 5: Stop

B. Program:

```
// Program to copy contents of source file to destination file
#include <stdio.h>
#include <stdlib.h> // For exit()
int main()
{
    FILE *fptr1, *fptr2;
    char filename[100], c;
    printf("Enter the filename to open for reading: \n");
    scanf("%s", filename);
    fptr1 = fopen(filename, "r"); // Open one file for reading
    if (fptr1 == NULL)
    {
        printf("Cannot open file %s \n", filename);
        exit(0);
    }
    printf("Enter the filename to open for writing: \n");
    scanf("%s", filename);
    fptr2 = fopen(filename, "w+"); // Open another file for writing
    c = fgetc(fptr1); // Read contents from file
    while (c != EOF)
    {
        fputc(c, fptr2);
        c = fgetc(fptr1);
    }
}
```

```
}  
printf("\nContents copied to %s", filename);  
fclose(fp1);  
fclose(fp2);  
return 0;  
}
```

C. Sample Input and Output:

Enter the filename to open for reading: source.txt
Enter the filename to open for writing: destination.txt
Contents copied to destination.txt

IV. Lab Exercises:

Write a 'C' program

1. To count the number of lines and characters in a file.
2. To reverse the file contents and store in another file. Also display the size of file using file handling function.
3. To merge lines alternatively from 2 files and stores it in a resultant file.

V. Additional Exercises:

1. Write a C program to collect statistics of a source file and display total number of blank lines, total number of lines ending with semicolon, total number of blank spaces.
2. To print five lines of file at a time. The program prompts user to enter the suitable option. When the user presses 'Q' the program quits and continues when the user presses 'C'.

PRELIMINARY SCANNING APPLICATIONS**I. Objectives:**

In this lab, student will be able to:

- To understand basics of scanning process.
- Ability to preprocess the input file so that it becomes suitable for compilation.

II. Description: This lab deals with preprocessing the input file so that it becomes suitable for scanning process. Preprocessing aims at removal of blank spaces, tabs, preprocessor directives, comments from the given input file.

III. Solved Exercise:**A. Algorithm: Removal of single and multiline comments**

Step 1: Open the input C file in read mode.

Step 2: Check if the file exists. Display an error message if the file doesn't exist.

Step 3: Open the output file for writing.

Step 4: Read each character from the input file.

Step 5: If the character read is '/'

- a. If next character is '/' then
 - i. Continue reading until newline character is encountered.
- b. If the next character is '*' then
 - i. Continue reading until next '*' is encountered.
 - ii. Check if the next character is '/'

Step 6: Otherwise, write the characters into the output file.

Step 7: Repeat step 4, 5 and 6 until EOF is encountered.

Step 8: Stop

B. Program:

//Program to remove single and multiline comments from a given 'C' file.

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

```
int main()
```

```
{
```

```
FILE *fa, *fb;
```

```
int ca, cb;
```

```
fa = fopen("q4in.c", "r");
```

```
if (fa == NULL)
```

```
{
```

```
printf("Cannot open file \n");
```

```
exit(0);
```

```
}
```

```
fb = fopen("q4out.c", "w");
```

```
ca = getc(fa);
```

```
while (ca != EOF)
```

```
{
```

```

    if(ca == ' ')
    {
        putc(ca,fb);
        while(ca == ' ')
            ca = getc(fa);
    }
    if (ca == '/')
    {
        cb = getc(fa);
        if (cb == '/')
        {
            while(ca != '\n')
                ca = getc(fa);
        }
    }
    else if (cb == '*')
    {
        do
        {
            while(ca != '*')
                ca = getc(fa);
            ca = getc(fa);
        } while (ca != '/');
    }
    else
    {
        putc(ca,fb);
        putc(cb,fb);
    }
}
else putc(ca,fb);
ca = getc(fa);
}
fclose(fa);
fclose(fb);
return 0;
}
}

```

C. Sample Input and Output:

```

/ This is a single line comment
/* *****This is a
*****Multiline Comment
**** */

#include <stdio.h>
void main()
{
FILE *fopen(), *fp;
int c ;

```

```

fp = fopen( "prog.c", "r" ); //Comment
c = getc( fp );
while ( c != EOF )
{
    putchar( c );
    c = getc ( fp );
} /*multiline
comment */
fclose( fp );
}

```

D. Output file after the removal of comments:

```

#include <stdio.h>
void main(){
    FILE *fopen(), *fp;
    int c ;
    fp = fopen( "prog.c", "r" );
    c = getc( fp );
    while ( c != EOF )
    {
        putchar( c );
        c = getc ( fp ); }
    fclose( fp );
}

```

IV. Lab Exercises:

Write a 'C' program

1. That takes a file as input and replaces blank spaces and tabs by single space and writes the output to a file.
2. To discard preprocessor directives from the given input 'C' file.
3. That takes C program as input, recognizes all the keywords and prints them in upper case.

V. Additional Exercises:

1. Write a program to display the function names present in the given input 'C' file along with its return type and number of arguments.

CONSTRUCTION OF TOKEN GENERATOR

I. Objectives:

In this lab, student will be able to:

- To Design a token generator
- To recognize the various lexemes and generate its corresponding tokens

II. Description:

The process of compilation includes different phases. It starts with the first phase called lexical analysis. The overview of this phase is displayed in Fig. 2.1. A lexical analyzer or scanner is a program that groups sequences of characters in the given input file into lexemes, and outputs (to the syntax analyzer) a sequence of tokens. Here:

- *Tokens* are symbolic names for the entities that make up the text of the program, e.g., “if” for the keyword *if*, and “id” for any identifier. These make up the output of the lexical analyzer.
- A *pattern* is a rule that specifies when a sequence of characters from the input constitutes a token, e.g., the sequence i, f for the token if, and any sequence of alphanumeric starting with a letter for the token id.
- A *lexeme* is a sequence of characters from the input that match a pattern (and hence constitute an instance of a token); for example, “if” matches the pattern for if.

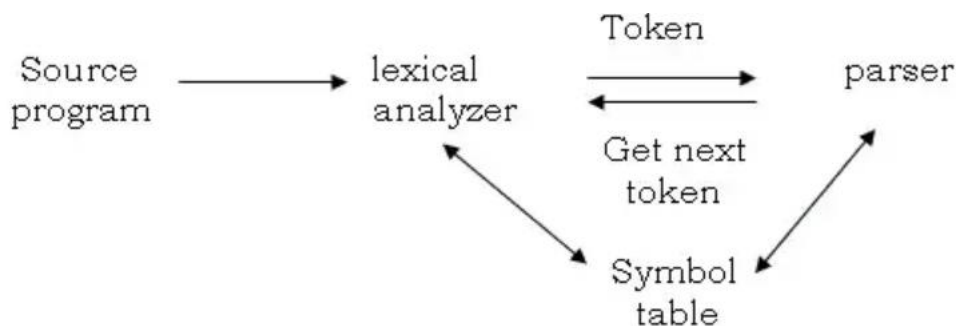


Figure 2.1: Overview of Scanning Process

The main task of Lexical Analyser is to generate tokens. Lexical Analyzer uses getNextToken() to extract each lexeme from the given source file and generate corresponding token one at a time. For each identified token an entry is made in the symbol table. If entry is already found in the table, then it returns the pointer to the symbol table entry for that token. The getNextToken () returns a structure of the following format.

```

struct token
{
    char token_name [ ];
    int index;
    unsigned int row,col; //Line numbers.
    char type[ ];
}
  
```

III. Solved Exercise:

Write a program in 'C' to identify the arithmetic and relational operators from the given input 'C' file.

A. Algorithm: Write a program in 'C' to identify the arithmetic and relational operators from the given input 'C' file.

Step 1: Open the input 'C' file.

Step 2: Check if the file exists. Display an error message if the file doesn't exist.

Step 3: Read each character from the input file.

Step 4: If character read is '=' add it to the buffer.

a. If the next character is '=' display it as relational operator.

b. Otherwise, display it as assignment operator.

Step 5: Otherwise, check if the next character is '<' or '>' or '!'.

a. Add it to the buffer.

b. If next character is '=' display It as Less Than Equal (LTE), Greater Than Equal (GTE) or NotEqualsTo (NE).

c. Otherwise, display it as Less Than (LT), Greater Than (GT).

Else

Step 6: Repeat step 3, 4 and 5 until EOF is encountered.

Step 7: Stop

B. Program:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
    char c,buf[10];
    FILE *fp=fopen("digit.c","r");
    c = fgetc(fp);
    if (fp == NULL)
    {
        printf("Cannot open file \n");
        exit(0);
    }
    while(c!=EOF)
    {
        int i=0;
        buf[0]='\0';
        if(c=='=')
        {
            buf[i++]=c;
            c = fgetc(fp);
            if(c=='=')
            {
                buf[i++]=c;
                buf[i]='\0';
                printf("\n Relational operator : %s",buf);
            }
        }
    }
}
```

```

else
{
buf[i]='\0';
printf("\n Assignment operator: %s",buf);
}
}
else
{
if(c=='<'||c=='>'||c=='!')
{
buf[i++]=c;
c = fgetc(fp);
if(c=='=')
{
buf[i++]=c;
}
buf[i]='\0';
printf("\n Relational operator : %s",buf);
}
else
{ buf[i]='\0';
}
}
c = fgetc(fp);
}
}

```

IV. Lab Exercises:

1. Write functions to identify the following tokens.
 - a. Arithmetic, relational and logical operators.
 - b. Special symbols, keywords, numerical constants, string literals and identifiers.

Hints: Design a lexical analyzer which contains getNextToken() for a simple C program to create a structure of token, which includes row number, column number and token type of the identified tokens. The getNextToken() should ignore all the tokens when encountered inside single line or multiline comment inside string literal. Preprocessor directives should also be stripped.

V. Additional Exercises:

1. Design a Lexical Analyzer to generate tokens for a simple arithmetic calculator.
2. Design a lexical Analyzer to generate tokens for functions and structures in 'C'.

CONSTRUCTION OF SYMBOL TABLE**I. Objectives:**

In this lab, student will be able to:

- To Design a Lexical analyzer.
- To recognize the various lexemes and generate its corresponding tokens.
- To store the tokens in the symbol table.

II. Description:**A. Symbol Table Management**

A symbol table is a data structure containing all the identifiers (i.e. names of variables, procedures etc.) of a source program together with all the attributes of each identifier.

For variables, typical attributes include:

- Variable name
- Variable type
- Size of memory it occupies
- Its scope.
- Arguments
- Number of arguments
- Return type

An entry is made in the symbol table during lexical analysis phase, and it is updated during syntax and semantic analysis phases. Hash table is used for the implementation of symbol table due to fast look up capability.

B. Structure of symbol table:

There are two types of symbol table

- Global Symbol table: Contains entry for each function.
- Local symbol table: Created for each function. It stores identifier details inside the function.

Structure of symbol table is as follows:

```
struct node
{
char lexeme[20];
int size;
char type[ ];
char scope;
} symbol;
```

III. Solved Exercise:

Write a program in 'C' that implements the Symbol Table using Linked Lists.

A. Program

```
/*
```

This a program that implements the Symbol Table using Linked Lists.

It uses Open Hashing...

The entire implementation done with the functions Search, Insert, Hash

Display function displays the whole symbol table.

*/

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TableLength 30
enum tokenType { EOFILE=-1, LESS_THAN,
LESS_THAN_OR_EQUAL, GREATER_THAN, GREATER_THAN_OR_EQUAL,
EQUAL, NOT_EQUAL };
struct token
{
char *lexeme;
int index;
unsigned int rowno,colno; //row number, column number.
enum tokenType type;
};

struct ListElement{
struct token tok;
struct ListElement *next;
};

struct ListElement *TABLE[TableLength];
void Initialize(){
for(int i=0;i<TableLength;i++){
TABLE[i] = NULL;
}
}
void Display(){
//iterate through the linked list and display
}
int HASH(char *str){
//Develop an OpenHash function on a string.
}
int SEARCH(char *str){
//Write a search routine to check whether a lexeme exists in the Symbol table.
//Returns 1, if lexeme is found
//else returns 0
}
void INSERT(struct token tk){
if(SEARCH(tk.lexeme)==1){
return; // Before inserting we check if the element is present already.
}
int val = HASH(tk.lexeme);
struct ListElement* cur = (struct ListElement*)malloc(sizeof(struct
ListElement));
cur->tok = tk;
cur->next = NULL;
if(TABLE[val]==NULL){
TABLE[val] = cur; // No collision.
}
else{
struct ListElement * ele= TABLE[val];
```



```

while(ele->next!=NULL){
ele = ele->next; // Add the element at the End in the case of a collision.
}
ele->next = cur;
}}

```

B. Sample input program:

```

int sum(int a, int b)
{ int s=a+b;
  return s;
}
bool search(int *arr,int key)
{
int i;
for(i=0;i<10;i++){
if(arr[i]==key)
return true;
else return false;
}
}
void main()
{
int a[20],i,sum;
bool status;
printf("Enter array elements:");
for(i=0;i<10;++i)
scanf("%d",&a[i]);
sum=a[0]+a[4];
status=search(a,sum);
printf("%d",status);
}

```

C. Output:

Global Symbol table:

| SINo | LexemeName | TokenType | Ptr to SymTab entry |
|------|------------|-----------|---------------------|
| 1 | main | Func | 24088 |
| 2 | search | Func | 34972 |
| 3 | sum | Func | 18644 |

Local symbol Table

Main

| | Lex_Name | Type | Size |
|---|----------|------|------|
| 1 | a | int | 4 |
| 2 | i | int | 4 |
| 3 | sum | int | 4 |
| 4 | status | bool | 1 |

Search

| | LexemeName | Type | Size |
|---|------------|------|------|
| 1 | i | int | 4 |

Sum

| | LexemeName | Type | Size |
|---|------------|------|------|
| 1 | s | int | 4 |

IV. Lab Exercises:

Using getNextToken() implemented in Lab No 3, design a Lexical Analyser to implement the following symbol tables.

- local symbol table
- global symbol table.

V. Additional Exercises:

- Design a lexical analyser to generate tokens for the C program with “structure” construct.
- Write a getNextToken () to generate tokens for the perl script given below.

```
#!/usr/bin/perl
#get total number of arguments passed.
$n = scalar (@_);
$sum = 0;
foreach $item(@_) {
    $sum += $item;
}
$average = $sum + $n;
```

Description of the Perl Script:

#! Represents path which has to be ignored by getNextToken().
followed by any character other than ! represents comments.
\$n followed by any identifier should be treated as a single token.
Scalar, foreach are considered as keywords.
@_, += are treated as single tokens.
Remaining symbols are tokenized accordingly.

PROGRAMS ON FLEX

I. Objectives:

In this lab, student will be able to:

- To implement programs using a Lexical Analyzer tool called FLEX.
- To apply regular expressions in pattern matching under FLEX.

II. Description:

A. Introduction to FLEX

FLEX (Fast LEXical analyzer generator) is a tool for generating tokens. Instead of writing a lexical analyzer from scratch, you only need to identify the vocabulary of a certain language, write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a lexical analyzer for you. FLEX is generally used in the manner depicted in Fig. 4.1.

Firstly, FLEX reads a specification of a scanner either from an input file *.flex, or from standard input, and it generates as output a C source file lex.yy.c. Next, lex.yy.c is compiled and linked with the flex library (using -lfl) to produce an executable a.out. Finally, a.out analyzes its input stream and transforms it into a sequence of tokens.

- *.l is in the form of pairs of regular expressions and C code.
- lex.yy.c defines a routine yylex() that uses the specification to recognize tokens.
- a.out is actually the scanner.

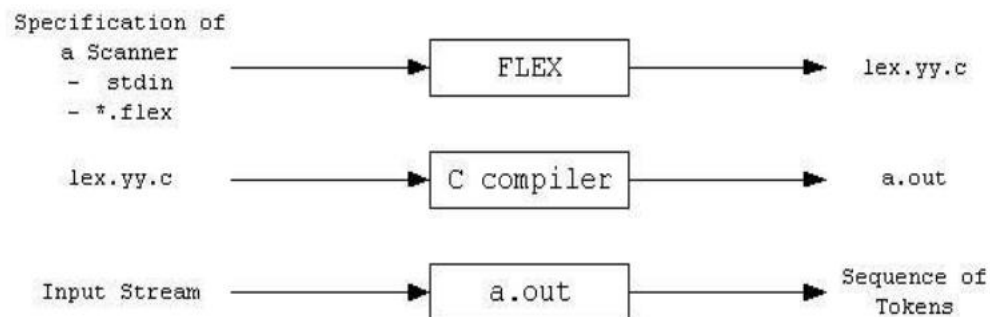


Figure 4.1: Steps involved in generating Lexical Analyzer using Flex

B. Regular Expressions and Scanning

Scanners generally work by looking for patterns of characters in the input. For example, in a C program, an integer constant is a string of one or more digits, a variable name is a letter or an underscore followed by zero or more letters, underscores or digits, and the various operators are single characters or pairs of characters. A straightforward way to describe these patterns is regular expressions, often shortened to regex or regexp. A flex program basically consists of a list of regexps with instructions about what to do when the input matches any of them, known as actions. A flex-generated scanner reads through its input, matching the input against all of the regexps and doing the appropriate action on each

match. Flex translates all the regexps into an efficient internal form that lets it match the input against all the patterns simultaneously.

C. The general format of Flex source program

The structure of Flex program has three sections as follows:

```
%{ definitions% }
%%
rules
%%
user subroutines
```

Definition section: Declaration of variables and constants can be done in this section. This section introduces any initial C program code we want to get copied into the final program. This is especially important if, for example, we have header files that must be included for code later in the file to work. We surround the C code with the special delimiters "%{" and "%}." Lex copies the material between "{" and "}" directly to the generated C file, so we may write any valid C code here. The %% marks the end of this section.

Rule section: Each rule is made up of two parts: a pattern and an action, separated by whitespace. The lexer that lex generates will execute the action when it recognizes the pattern. These patterns are UNIX style regular expressions. Each pattern is at the beginning of a line (since flex considers any line that starts with whitespace to be code to be copied into the generated C program.), followed by the C code to execute when the pattern matches. The C code can be one statement or possibly a multiline block in braces, { }. If more than one rule matches the input, the longer match is taken. If two matches are the same length, the earlier one in the list is taken.

User Subroutines section: This is the final section which consists of any legal C code. This section has functions namely main() and yywrap().

- The function yylex() is defined in lex.yy.c file and is called from main(). Unless the actions contain explicit return statements, yylex() won't return until it has processed the entire input. The function yywrap() is called when EOF is encountered. If this function returns 1, the parsing stops. If the function returns 0, then the scanner continues scanning.

Sample Flex program

```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}

%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n { chars++; lines++; }
. { chars++; }
%%

main(int argc, char **argv)
{
yylex();
printf("%d%d%d\n", lines, words, chars);
}
```

```

int yywrap()
{
return 1;
}

```

In this program, the definition section contains the declaration for character, word and line counts. The rule section consists of only three patterns. The first one, `[a-zA-Z]+`, matches a word. The characters in brackets, known as a character class, match any single upper- or lowercase letter, and the `+` sign means to match one or more of the preceding things, which here means a string of letters or a word. The action code updates the number of words and characters seen. In any flex action, the variable `yytext` is set to point to the input text that the pattern just matched. The second pattern, `\n`, just matches a new line. The action updates the number of lines and characters. The final pattern is a dot, which is regex that matches any character. The action updates the number of characters. The end of the rules section is delimited by another `%%`.

D. Handling ambiguous patterns

Most flex programs are quite ambiguous, with multiple patterns that can match the same input. Flex resolves the ambiguity with two simple rules:

- Match the longest possible string every time the scanner matches input.
- In the case of a tie, use the pattern that appears first in the program.

These turn out to do the right thing in most cases. Consider this snippet from a scanner for C source code:

```

"+" { return ADD; }
"=" { return ASSIGN; }
"+=" { return ASSIGNADD; }
"if" { return KEYWORDIF; }
"else" { return KEYWORDELSE; }
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }

```

For the first three patterns, the string `+=` is matched as one token, since `+=` is longer than `+`. For the last three patterns, as long as the patterns for keywords precede the pattern that matches an identifier, the scanner will match keywords correctly. A list of various variables and functions available in Flex are given in Table 4.1. The regular definitions are given in Table 4.2.

The basic operators to make more complex regular expressions are, with `r` and `s` being two regular expressions:

- `(r)` : Match an `r`; parentheses are used to override precedence.
- `rs` : Match the regular expression `r` followed by the regular expression `s`. This is called concatenation.
- `r|s` : Match either an `r` or an `s`. This is called alternation.
- `{abbreviation}`: Match the expansion of the abbreviation definition. Instead of writing regular expression.

Table 4.1: Variables and functions available by default in Flex

| | |
|--------|---|
| yytext | When the lexical analyzer matches or recognizes the token from the input, then the lexeme is stored in a null terminated string called <code>yytext</code> . It is an array of pointer to char where lex places the current token's lexeme. The string is automatically null terminated |
|--------|---|

| | |
|----------|---|
| yylen | Stores the length or number of characters in the input string. The value of yylen is same as strlen() functions. In other words, it is a integer that holds strlen(yytext) |
| yyval | This variable returns the value associated with token. |
| yyin | Points to the input file. |
| yyout | Points to the output file. |
| yylex() | The function that starts the analysis process. It is automatically generated by Lex |
| yywrap() | This function is called when EOF is encountered. If this function returns 1, the parsing stops. If the function returns 0, then the scanner continues scanning. |

Table 4.2: Regular Definitions in Flex

| | |
|-----------|--|
| x | Matches the character x. |
| [xyz] | Any characters amongst x, y or z. You may use a dash for character intervals: [a-z] denotes any letter from a through z. You may use a leading hat to negate the class: [0-9] stands for any character which is not a decimal digit, including new-line. |
| "string" | "..." Anything within the quotation marks is treated literally |
| <<EOF>> | Match the end-of-file. |
| [a,b,c] | matches a, b or c |
| [a-f] | matches either a,b,c,d,e, or f in the range a to f |
| [0-9] | matches any digit |
| X+ | matches one or more occurrences of X |
| X* | matches zero or more occurrences of X |
| [0-9]+ | matches any integer |
| () | grouping an expression into a single unit |
| | alternation (or) |
| (a b c)* | equivalent to [a-c]* |
| X? | X is optional (zero or one occurrence) |
| [A-Za-z] | matches any alphabetical character |

| | |
|--------|---|
| • | matches any character except newline |
| \• | matches the • character |
| \n | Matches the new line character |
| \t | Matches the tab character |
| [^a-d] | Matches any character other than a,b,c and d. |

E. Example for abbreviation:

```
%%
[a-zA-Z_][a-zA-Z0-9_]* return IDENTIFIER;
%%
```

We may write

```
id [a-zA-Z_][a-zA-Z0-9_]*
%%
{id} return IDENTIFIER;
%%
```

- **r/s**: Match an r but only if it is followed by an s. The text matched by s is included when determining whether this rule is the longest match but is then returned to the input before the action is executed. So the action only sees the text matched by r. This type of pattern is called trailing context.
- **^r** : Match an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).
- **r\$** : Match an r, but only at the end of a line (i.e., just before a newline).

F. Installing FLEX:

Steps to download, compile, and install on linux platform using following command

\$sudo apt-get install flex

Steps to execute:

- Type Flex program and save it using .l extension.
- Compile the flex code using
\$ flex filename.l
- Compile the generated C file using
\$ gcc lex.yy.c -o output
- This gives an executable output
- Run the executable using \$./output

III. Solved Exercise:

Write a Flex program to recognize identifiers.

Program:

```
% { #include<stdio.h>
% }
%%
[a-zA-Z_][a-zA-Z0-9_]* {printf(" Identifier");}
%%
int yywrap() { return 1;}
int main()
{
char stat[20];
printf("Enter the valid C statement");
scanf("%s",stat);
yylex();
}
```

IV. Lab Exercises:

Write a FLEX program to

1. Count the number of vowels and consonants in the given input.
2. Count the number of words, characters, blanks and lines in a given text.
3. Find the number of positive integer, negative integer, positive floating positive number and negative floating point number
4. Given a input C file, replace all scanf with READ and printf with WRITE statements also find the number of scanf and printf in the file.
5. That changes a number from decimal to hexadecimal notation.
6. Convert uppercase characters to lowercase characters of C file excluding the characters present in the comment.

V. Additional Exercises:

1. Generate tokens for a simple C program. (Tokens to be considered are: Keywords, Identifiers, Special Symbols, arithmetic operators and logical operators).
2. Write a FLEX program to identify verb, noun and pronoun.

RECURSIVE DESCENT PARSER FOR SIMPLE GRAMMER

I. Objectives:

In this lab, student will be able to:

- To understand implementation of recursive descent parser (RD) for simple grammars.

II. Description:

A. Top-Down Parsers

Syntax analysis is the second phase of the compiler. Output of lexical analyzer – tokens and symbol table are taken as input to the syntax analyzer. Syntax analyzer is also called as parser. Top Down parsers come in two forms:

- Backtracking parsers
- Predictive parsers

Predictive parsers are categorized into

- Recursive Descent parsers
- Table driven or LL (1) parser

B. Recursive Descent Parsers:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

III. Solved Exercise:

```

/*
E --> TEprime
E' --> +TE' / ε
T --> FT'
T' --> *FT' / ε
F --> (E) / i
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int curr = 0;
char str[100];
// -----
void E();
void Eprime();
void T();
void Tprime();
void F();

```

```

// -----
void invalid()
{
    printf("-----ERROR!-----\n");
    exit(0);
}
void valid()
{
    printf("-----SUCCESS!-----\n");
    exit(0);
}
// -----
void E()
{
    T();
    Eprime();
}
// -----
void Eprime()
{
    if(str[curr] == '+')
    {
        curr++;
        T();
        Eprime();
    }
}
// -----
void T()
{
    F();
    Tprime();
}
// -----
void Tprime()
{
    if(str[curr] == '*')
    {
        curr++;
        F();
        Tprime();
    }
}
// -----
void F()
{
    if(str[curr] == '(')
    {
        curr++;

```

```

    E();
    if(str[curr] == ')')
    {
        curr++;
        return;
    }
    else
        invalid();
}
else if(str[curr] == 'i')
{
    curr++;
    return;
}
else
    invalid();
}
// -----

int main()
{
    printf("Enter String: ");
    scanf("%s", str);
    E();
    if(str[curr] == '$')
        valid();
    else
        invalid();
}

```

Sample input and Output:

```

Enter string: i+i$
-----SUCCESS!-----

```

IV. Lab Exercises:

Write a recursive descent parser for the following simple grammars.

1. $S \rightarrow a \mid > \mid (T)$
 $T \rightarrow T, S \mid S$

2. $S \rightarrow UVW$
 $U \rightarrow (S) \mid aSb \mid d$
 $V \rightarrow aV \mid \epsilon$
 $W \rightarrow cW \mid \epsilon$

3. $S \rightarrow aAcBe$
 $A \rightarrow Ab \mid b$
 $B \rightarrow d$

V. Additional Exercises:

1. Write a program with functions `first(X)` and `follow(X)` to find first and follow for X where X is a non-terminal in a grammar.
2. Write a program to remove left recursion from the grammar

RECURSIVE DESCENT PARSER FOR DECLARATION AND EXPRESSION STATEMENTS

I. Objectives:

In this lab, student will be able to:

- To design RD parser for simple variable declaration statements of a 'C' program.
- To be able to perform error detection and correction in variable declarations.
- To design RD parser for array declaration and expression statements of a 'C' program

II. Description:

A simple 'C' language grammar is given. Student should write/update RD parser for subset of grammar and integrate it lexical analyzer. Before parsing the input file, remove ambiguity and left recursion, if present and perform left factoring on subset of grammar given. Include the functions first(X) and follow(X) which already implemented in previous week. Lexical analyzer code should be included as header file in parser code. Parser program should make a function call getNextToken() of lexical analyze which generates a token. Parser parses it according to given grammar. The parser should report syntax errors if any (for e.g.: Misspelling an identifier or keyword, Undeclared or multiply declared identifier, Arithmetic or Relational Expressions with unbalanced parentheses and Expression syntax error etc.) with appropriate line-no.

Sample C grammar:

| | | |
|---------------------|---|---------------------------|
| Data Types | : | int, char |
| Arrays | : | 1-dimensional |
| Expressions | : | Arithmetic and Relational |
| Looping statements | : | for, while |
| Decision statements | : | if, if – else |

```

Program → main () { declarations statement-list }
Declarations → data-type identifier-list; declarations | ∈
data-type → int | char
identifier-list → id | id, identifier-list | id[number] , identifier-list | id[number]
statement_list → statement statement_list | ∈
statement → assign_stat; | decision_stat | looping_stat
assign_stat → id = expn
expn → simple-expn eprime
epime → relop simple-expn | ∈
simple-exp → term seprime
seprime → addop term seprime | ∈
term → factor tprime
tprime → mulop factor tprime | ∈
factor → id | num
decision_stat → if ( expn ) {statement_list} dprime
dprime → else {statement_list} | ∈
looping_stat → while (expn) {statement_list} | for (assign_stat ; expn ; assign_stat )
{statement_list}
relop → == | != | <= | >= | > | <
addop → + | -
mulop → * | / | %

```

III. Lab Exercises:

1. For given subset of Grammar 7.1, design RD parser with appropriate error messages with expected character and row and column number.

```
Program → main () { declarations assign_stat }  
declarations → data-type identifier-list; declarations | ∈  
data-type → int | char  
identifier-list → id | id, identifier-list  
assign_stat → id=id; | id = num;
```

2. Design the recursive descent parser to parse array declarations and expression statements with error reporting. Subset of grammar 7.1 is as follows:

```
Program → main () { declarations statement-list }  
identifier-list → id | id, identifier-list | id[number], identifier-list | id[number]  
statement_list → statement statement_list | ∈  
statement → assign-stat;  
assign_stat → id = expn  
expn → simple-expn eprime  
epime → relop simple-expn | ∈  
simple-exp → term seprime  
seprime → addop term seprime | ∈  
term → factor tprime  
tprime → mulop factor tprime | ∈  
factor → id | num  
relop → == | != | <= | >= | > | <  
addop → + | -  
mulop → * | / | %
```

IV. Additional Exercises:

1. Write a program to parse pointer declarations.
2. Write a program to detect errors in the variable declarations. Error report should contain line numbers.
3. Modify the RD parser to handle compound expressions present in C program.
4. Modify the RD parser to handle ternary statements present in C program.

RECURSIVE DESCENT PARSER FOR DECISION MAKING AND LOOPING STATEMENTS

I. Objectives:

In this lab, student will be able to:

- To design RD parser for decision making statements of a 'C' program

II. Lab Exercises:

1. Modify the Recursive Descent parser implemented in the previous lab to parse decision making with error reporting. Subset of grammar 7.1 is as follows:

$$\begin{aligned} \text{statement} &\rightarrow \text{assign-stat;} \mid \text{decision_stat} \\ \text{decision-stat} &\rightarrow \text{if (expn) \{statement_list\} dprime} \\ \text{dprime} &\rightarrow \text{else \{statement_list\} } \mid \epsilon \end{aligned}$$

2. Modify the Recursive Descent parser implemented in the previous lab to parse looping statements with error reporting. Subset of grammar 7.1 is as follows:

$$\begin{aligned} \text{statement} &\rightarrow \text{assign-stat;} \mid \text{decision_stat} \mid \text{looping-stat} \\ \text{looping-stat} &\rightarrow \text{while (expn) \{statement_list\} } \mid \text{for (assign_stat ; expn ; assign_stat)} \\ &\quad \text{\{statement_list\} } \end{aligned}$$

III. Additional Exercises:

1. Design a grammar for defining a switch block in 'C' and implement a RD parser to parse the same.
2. Write a grammar for defining a structure in 'C' and implement a RD parser to parse the same.

RD PARSER FOR SIMPLE GRAMMARS USING YACC

I. Objectives:

In this lab, student will be able to:

- To understand bison tool.
- To implement the parser using bison

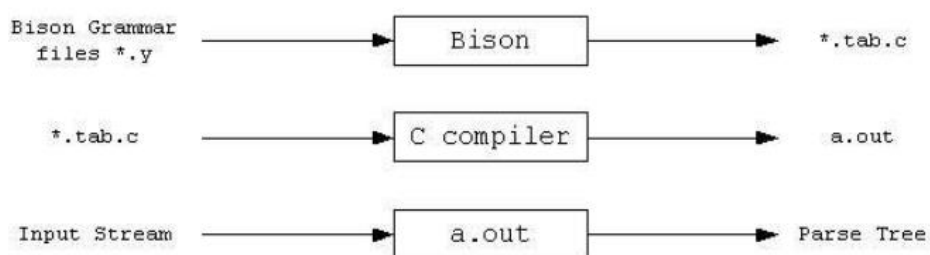
II. Description:

A. Introduction:

Parsing is the process of matching grammar symbols to elements in the input data, according to the rules of the grammar. The parser obtains a sequence of tokens from the lexical analyzer and recognizes its structure in the form of a parse tree. The parse tree expresses the hierarchical structure of the input data and is a mapping of grammar symbols to data elements. Tree nodes represent symbols of the grammar (non-terminals or terminals), and tree edges represent derivation steps.

There are two basic parsing approaches: top-down and bottom-up. Intuitively, a top-down parser begins with the start symbol. By looking at the input string, it traces a leftmost derivation of the string. By the time it is done, a parse tree is generated top-down. While a bottom-up parser generates the parse tree bottom-up. Given the string to be parsed and the set of productions, it traces a rightmost derivation in reverse by starting with the input string and working backwards to the start symbol.

Bison is a tool for building programs that handle structured input. The parser's job is to figure out the relationship among the input tokens. A common way to display such relationships is a parse tree. Bison is a general-purpose parser generator that converts a grammar description (Bison Grammar Files) for an LALR(1) context-free grammar into a C program to parse that grammar. The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.



Grammar 9.1

B. Matching an Input using Bison Parser:

A grammar is a series of rules that the parser uses to recognize syntactically valid input.

Statement: NAME '=' expression

Expression: NUMBER '+' NUMBER

| NUMBER '-' NUMBER

The vertical bar, |, means there are two possibilities for the same symbol; that is, an expression can be either an addition or a subtraction. The symbol to the left of the: is known as the left-

hand side of the rule, often abbreviated LHS, and the symbols to the right are the right-hand side, usually abbreviated RHS. Several rules may have the same left-hand side; the vertical bar is just shorthand for this. Symbols that actually appear in the input and are returned by the lexer are terminal symbols or tokens, while those that appear on the left-hand side of each rule are nonterminal symbols or non-terminals. Terminal and nonterminal symbols must be different; it is an error to write a rule with a token on the left side.

A bison specification has the same three-part structure as a flex specification. (Flex copied its structure from the earlier lex, which copied its structure from YACC, the predecessor of bison.) The first section, the definition section, handles control information for the parser and generally sets up the execution environment in which the parser will operate. The second section contains the rules for the parser, and the third section is C code copied verbatim into the generated C program.

```
... definition section ...
%%
... rules section ...
%%
... user subroutines section ...
```

The declarations here include C code to be copied to the beginning of the generated C parser, again enclosed in `%{` and `%}`. Following that are `%token` token declarations, telling bison the names of the symbols in the parser that are tokens. By convention, tokens have uppercase names, although bison doesn't require it. Any symbols not declared as tokens have to appear on the left side of at least one rule in the program. The second section contains the rules in simplified BNF. Bison uses a single colon rather than `::=`, and since line boundaries are not significant, a semicolon marks the end of a rule. Again, like flex, the C action code goes in braces at the end of each rule.

Bison creates the C program by plugging pieces into a standard skeleton file. The rules are compiled into arrays that represent the state machine that matches the input tokens. The actions have the `$N` and `@N` values translated into C and then are put into a switch statement within `yyparse()` that runs the appropriate action each time there's a reduction.

C. Abstract Syntax Tree :

One of the most powerful data structures used in compilers is an abstract syntax tree. An AST is basically a parse tree that omits the nodes for the uninteresting rules. A bison parser doesn't automatically create this tree as a data structure. Every grammar includes a start symbol, the one that has to be at the root of the parse tree.

D. Download and Install bison on Ubuntu

Use the following command :

```
$sudo apt-get install bison
```

III. SOLVED EXERCISE:

Write a Bison program to check the syntax of a simple expression involving operators +, -, * and /.

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token NUMBER ID NL
%left '+'
```

```

%left '*'
%%
stmt : exp NL { printf("Valid Expression"); exit(0); }
;
exp : exp '+' term
    | term
term: term '*' factor
factor
factor: ID
    | NUMBER
;
%%
int yyerror(char *msg)
{
printf("Invalid Expression\n");
exit(0);
}
void main ()
{
printf("Enter the expression\n");
yyparse();
}

```

Flex Part

```

% {
#include "y.tab.h" //filename.tab.h : here both flex and bison
% }
%%
[0-9]+ {return NUMBER; }
\n {return NL ;}
[a-zA-Z][a-zA-Z0-9_]* {return ID; }
. {return yytext[0]; }
%%

```

Steps to execute:

- Type Flex program and save it using .l extension.
- Type the bison program and save it using .y extension.
- Compile the bison code using `$ bison -d filename.y`
The option -d Generates the file y.tab.h with the #define statements that associate the yacc user-assigned "token codes" with the user-declared "token names." This association allows source files other than y.tab.c to access the token codes.
- This command generates two files
filename.tab.h and filename.tab.c
- Compile the flex code using
`$ flex filename.l`
- Compile the generated C file using
`$ gcc lex.yy.c filename.tab.c -o output`
- This gives an executable output.out
- Run the executable using `$./output`

III. Lab Exercises:

Write a bison program,

- To check a valid declaration statement.

2. To check a valid decision making statements.
3. To evaluate an arithmetic expression involving operations +,-,* and /.
4. To validate a simple calculator using postfix notation. The grammar rules are as follows –

input \rightarrow input line | ϵ

line \rightarrow '\n' | exp '\n'

exp \rightarrow num | exp exp '+'
 | exp exp '-'
 | exp exp '*'
 | exp exp '/'
 | exp exp '^'
 | exp 'n'

IV. Additional Exercises:

1. Write a grammar to recognize strings 'aabb' and 'ab' ($a^n b^n$, $n \geq 0$). Write a Bison program to validate the strings for the derived grammar.
2. Write a grammar to recognize ($a^n b$, $n \geq 10$). Write a Bison program to validate the strings for the derived grammar.

CODE GENERATION

I. Objectives:

In this lab, student will be able to:

- To understand the intermediate code generation and code generation phase of compilation.
- To generate machine code from the intermediate representation i.e. postfix expression

II. Description:

Code Generation is the last phase of the compilation process. It takes any of the intermediate representation format as input and produces equivalent Assembly Code as output. Here we consider postfix expression and Three Address Code as the intermediate representations to generate the basic level assembly code.

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

$a = b + c * d;$

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;  
r2 = b + r1;  
a = r2
```

‘r’ being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression.

III. SOLVED EXERCISE:

Write a C program to implement the intermediate code for the given postfix expression.

```
/*  
Store in  
Int -code -gen.c  
*/  
#include<stdio.h>  
#include<string.h>  
#define MAX_STACK_SIZE 40  
#define MAX_IDENTIFIER_SIZE 64  
/*Assume variables are separated by a space*/  
char *str="a b c d * + =";  
/*Expected output  
temp1=c*d  
temp2=b+temp1  
a=temp2
```

```

*/
//implementation using stack
char **stack=NULL;
int top=-1;
int tcounter=1;
int push(char *str) {
    int k;
    if(!((top+1)<MAX_STACK_SIZE))
        return 0;
    strcpy(stack[top+1],str);
    top=top+1;
    return 1;
}
char *pop() {
    if(top < 0 )
        return NULL;
    top=top
    -1;
    return stack[top+1];
}
char *dec_to_str(int num) {
    char numstr[MAX_IDENTIFIER_SIZE];
    int count=0,i=0;
    int rem;
    while(num > 0 ) {
        rem=num%10;
        numstr[count++]=(char)rem+48;
        num=num/10;
    }
    numstr[count]='\0'; //reverse the string
    for(i=0;i<count/2;i++) {
        char temp=numstr[i];
        numstr[i]=numstr[count-i-1];
        numstr[count-i-1]=temp;
    }
    return numstr;
}
void parseAndOutput() {
    int i;
    stack=malloc(MAX_STACK_SIZE* sizeof(char *));
    for(i=0;i<MAX_STACK_SIZE;i++)
    {
        stack[i]=malloc(MAX_IDENTIFIER_SIZE*sizeof(char));
    }
    char op[MAX_IDENTIFIER_SIZE];
    char *pop1,*pop2;
    int kop=0;
    for(i=0;i<strlen(str);i++) {
        //is it an identifier ?
        if((str[i]>='A' && str[i]<='Z') || (str[i]>='a' && str[i]<='z') || str[i]=='_' || (str[i]>='0' &&
        str[i]<='9')) {
            op[kop++]=str[i];
        }
        //is it a space ?
        else if(str[i]==' ') {
            op[kop]='\0';

```

```

kop=0;
if(strcmp(op,"")!=0)
push(op);
}
//has to be any operator namely +, -, *, /, % etc
else {
//check if previous identifier is stored in stack
if(kop > 0) {
op[kop]='\0';
kop=0;
if(strcmp(op,"")!=0)
push(op);
}
pop2=pop();
pop1=pop();
int k;
//check for = operator
if(str[i]=='=') {
printf("%s = %s\n",pop1,pop2);
push(pop1);
}
else { //could be any +,-,*,/
char tempStr[MAX_IDENTIFIER_SIZE];
char *numStr;
strcpy(tempStr,"temp");
//convert tcounter number to string
numStr=dec_to_str(tcounter);
int j;
int ts=strlen(tempStr);
for(j=strlen(tempStr);j<strlen(tempStr)+strlen(numStr);j++)
tempStr[j]=numStr[j-ts];
tempStr[j]='\0';
printf("%s = %s %c %s\n",tempStr,pop1,str[i],pop2);
tcounter=tcounter+1;
push(tempStr);
}
}
}
//free the memory allocated
for(i=0;i<MAX_STACK_SIZE;i++) {
free(stack[i]);
}
free(stack);
}
int main() {
parseAndOutput();
return 0;
}

```

The assembly code corresponding to the following C program is can be obtained using the option `-S` with gcc command.

Consider a code snippet.

Program- z.c

```
#include<stdio.h>
int main() {
int x;
x=3+2;
printf("%d",x);
return 0;
}
```

Steps to obtain the assembly output from the program z.c

Step 1: Generate the equivalent TAC for the program and the store in a file.

Step 2: The TAC obtained in Step 1 is taken as input.

Step 3: Run the command `gcc -S z.c`. This will automatically generate a file z.s with corresponding assembly code.

Step 4: The assembly code is run using the command `gcc z.s -o z` and `./z`

The assembly code generated is as shown below

```
.file "z.c"
.section .rodata
.LC0:
.string "%d"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
movl $5, -4(%rbp)
movl -4(%rbp), %eax
movl %eax, %esi
movl $.LC0, %edi
movl $0, %eax
call printf
movl $0, %eax
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
.section .note.GNU-stack,"",@progbits
```

III. Lab Exercises:

1. Write a program to generate Assembly Level code from the given Postfix expression.

IV. Additional Exercises:

1. Write a program to generate Three Address Code for a C program with decision making constructs.

Mini Project

Objective:

- Design, implement and document an entire compiler chain [Lexical Analyzer, Parser, Intermediate Code Generator and Code Generator] for the any one option listed below.
 - programming language described on the following pages.
 - programming language as per your choice.
- The compiler will produce a listing file which lists and numbers each line, interleaved with any error messages following the source file. Each error message should refer to a line number. Even if errors are present in the source code, your compiler should continue to analyze the input source code. Target code will be generated only if there are no errors.
- The entire project is due at the last lab class of the semester. You need to demonstrate the compiler. You also need to submit
 - Source code of your compiler
 - Project Report: Documentation describing the internal operation of the compiler. Describe any limitations of your compiler, the algorithms and data structures which you used, the purpose of major routines, inputs etc.

DESCRIPTION OF THE PROGRAMMING LANGUAGE

Lexical Units and Spacing

The lexical units of a program are identifiers, reserved words, numbers, strings, and delimiters.

A delimiter is any one of the following special characters:

() [] ; : . , * - + / < = >

or one of the following compound symbols:

<> := <= >=

Spaces may be inserted freely with no effect on meaning between lexical units. At least one space must separate adjacent identifiers or numbers. The end of a line is equivalent to a space except (1) it terminates comments and (2) is illegal in quoted character strings. Thus each lexical unit must fit on one line.

IDENTIFIERS

Identifiers (names) consist of a letter followed by a sequence of letters or digits. Upper and lower case are considered equivalent. An identifier may be any length but must be distinguishable within the first 32 characters. Thus the two names

a111111111111111111111111111111112222 and

a111111111111111111111111111111119999 are not distinguishable.

NUMBERS

The only kind of number is the integer. An integer is a sequence of digits. Maximum and minimum values of integers are determined by our implementation.

STRINGS

A character string is a sequence of characters prefixed and terminated by the apostrophe character. The backslash character (\) acts as an escape. When followed by an **n** or a **t**, the backslash denotes a new line (CR) or tab. When followed by another character it denotes that character. Thus \' denotes the single quote. For example, the string DON'T would be represented as 'DON\'T'.

COMMENTS

A comment starts with an exclamation point (!) and is terminated by the end of a line. It may appear following a lexical unit or at the beginning of a program.

!This is a comment

RESERVED WORDS

The identifiers listed below are reserved words and may not be declared by the programmer:

and
array
begin
integer
do
else
end
function
if
of
or
not
procedure
program
read
then
var
while
write

Syntax :

the production rules are as follows:

program -> program id;
type -> standartype | array[num]
standard~type -> integer
statement -> variable assignop expression
 | procedure_statement
 | compound_statement
 | if expression then statement else statement
 | if expression then statement
 | while expression do statement
 | read_statement
 | write_statement

arguments -> (parameter_list) | ()
read_statement -> read (identifier_list)
identifier__list -> variable,identifier_list | variable
write_statement -> write(outputlist)
outputlist -> outputitem |outputitem,outputlist
outputitem -> string | expression_list
factor -> id[expression]

In addition: You can think about different types of operators, Arrays.

Guidelines:

All the students are instructed to form a team of four members. All the teams need to submit synopsis by 21/08/2023. Once the synopsis is submitted, no change of team member(s) shall be entertained.

Code of the mini project should be unique among the teams.

Synopsis Format:

- Title, Team Members
- Abstract (Briefly describe the language features that will be planned to be compiled, which phases team want to implement)
- Expected output of the project