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

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that Ms./Mr. ……...……..

Reg. No. …..…… Section: …… Roll No: ……... has satisfactorily completed the lab exercises prescribed for Compiler Design Lab [CSE 3211] of Third Year B. Tech. (Computer Science and Engineering) Degree at MIT, Manipal, in the academic year 2018-2019.

Date: …….....

Signature
Faculty in Charge

CONTENTS

Course Objectives

- Apply theory to the various challenges involved in design and development of a compiler.
- Understand theory and practical aspects of modern compiler design.
- Understand implementation detail for a small subset of a language applying different techniques studied in this course.
- Study both implementation and automated tools for different phases of a compiler.

Course Outcomes

At the end of this course, students will gain the

- Ability to create preliminary scanning applications and to classify different lexemes.
- Ability to create a lexical analyzer without using any lexical generation tools.
- Ability to implement suitable data structure for a symbol table.
- Ability to implement a recursive descent parser for a given grammar with/without using any parser generation tools.
- Ability to implement a recursive descent parser for a given grammar of C programming language with/without using any parser generation tools.
- Ability to design a code generator.

Evaluation plan

- Internal Assessment Marks : 60 Marks

Scanning and Tokenizing	:	5 Marks
Lexical analyzer	:	10 Marks
Parser	:	20 Marks
Code generation	:	10 Marks
Project Work	:	15 Marks

- End semester assessment : 40 Marks

✓ Duration: 2 hours

✓ Total marks : Write up: 15 Marks

Execution: 25 Marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Class notes, Lab Manual and the required stationery to every lab session
2. Be in time and follow the Instructions from Lab Instructors
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 given in Lab Manual
- 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

- 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.
- Copying from others is strictly prohibited and would invite severe penalty during evaluation.
- The exercises for each week are divided under three sets:
  - 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 at students end or in a repetition class (if available) with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- 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...

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

LAB No.: 1 **Date:**

PRELIMINARY SCANNING APPLICATIONS

Objectives:

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

Prerequisites:

- Knowledge of file pointers and string handling functions.
- Knowledge of the C programing language.

I. INTRODUCTION:

FILE OPERATIONS:

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 a storage device like disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure created by an Operating System. 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);

Table 1.2: Various modes present 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.

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);

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(fileprinter,"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. 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 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 endof-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.

PRELIMINARY SCANNING:

In this process, we mainly deal 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. Scanning is part of lexical analyzer. Scanning is built within a lexical analyzer.

II. SOLVED EXERCISE:

Write a program in 'C', that removes single and multiline comments from a given input 'C' file.

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 exists.
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

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=='/')
			{
			  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; } }
```

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 );
   }
```

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 );
   }
```

III. 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 and writes to an output file.
3. That takes C program as input, recognizes all the keywords and prints them in upper case along with their line numbers, column numbers.

IV. ADDITIONAL EXERCISES:

1. Write a program that inserts line number to the source file and prints them

LAB No.: 2 **Date:**

LEXICAL ANALYZER

Objectives:

- To perform a lexical analysis of a program.
- To recognize the various lexemes and generate its corresponding tokens.

Prerequisites:

- Knowledge of the C programing language.
- Knowledge of file pointers and string handling functions.

II. INTRODUCTION

Fig. 2.1 shows the various phases involved in the process of compilation. The process of compilation starts with the first phase called lexical analysis. The overview of scanning process is shown in Fig. 2.2. 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 and 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, and foo123bar matches the pattern for id.

This token sequence represents almost all the important information from the input program required by the syntax analyzer. Whitespace (newlines, spaces and tabs), although often important in separating lexemes, is usually not returned as a token. Also, when outputting an id or literal token, the lexical analyzer must also return the value of the matched lexeme (shown in parentheses) or else this information would be lost.

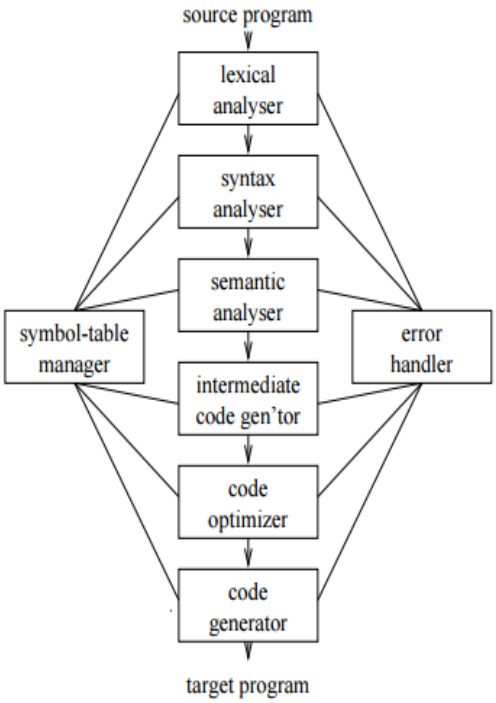


Fig. 2.1 Different Phases of Compiler

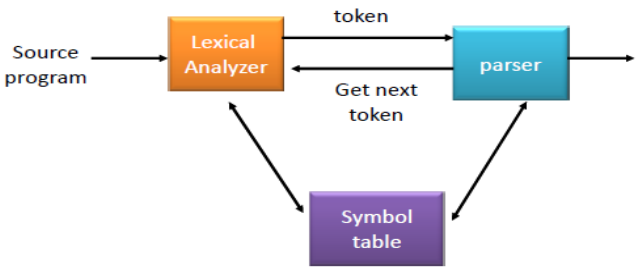


Fig. 2.2 Overview of Scanning Process

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 lexemename [ ];
        int index;
unsigned int row,col;  //Line numbers.
          unsigned int type;
 }
```

Input Program 2.1

```
1.      main()
2.      {
3.      int a,b,sum;
4.      a = 1;
5.      b = 1;
6.      sum = a + b;
7.      }
```

sample output file for program 2.1

<main,1,1,IDENTIFIER>,

<(,1,5,LB>,

<),1,6, RB>,

<{,2,1,LC>,

<int,3,1,KEYWORD>,

<a,3,5,IDENTIFIER>

<b,3,7,IDENTIFIER>

<sum,3,9,IDENTIFIER>

<;,3,12, SS>

<a,4,1,IDENTIFIER>

<=,4,3, ASSIGNMENT OPERTOR>

<1,4,5, NUMBER>

<;,4,6, SS>

<b,5,1,IDENTIFIER>

<=,5,3, ASSIGNMENT OPERATOR>

<1,5,5, NUMBER>

<;,5,6, SS>

<sum,6,1,IDENTIFIER>

<=,6,4, ASSIGNMENT OPERATOR>

<a,6,5,IDENTIFIER>

<+,6,6, ARITHMETIC OPERATOR>

<b,6,7,IDENTIFIER>

<;,6,8, SS>

<},7,1, LC>

In the above shown output, syntax of the token:

< lexeme name, row number, column number, token type>

Assumptions to be made:

- Scan for identifiers from the beginning, recording it as global. Once an entry named FUNC is created, all variables (except argument section) will be assumed to be local to that function.
- However, the scope ends when a second function declaration is made.

III. SOLVED EXERCISE:

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

Algorithm: Identification of relational operators from the given input file.

Step 1: Open the input 'C' file.

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

Step 3: Read each character from the input file.

Step 4: If character read is '/' and next character read is '/' or '*' it is considered as comments, then skip all the characters until the end of the comment.

Step 5: If character read is ", skip all the characters until the another " is encountered.

Step 6: 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 7: Repeat step 3, 4,5 and 6 until EOF is encountered.

Step 8: Stop.

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. Design a lexical analyzer which contains getNextToken( ) for a simple C program to create a structure of token each time and return, which includes row number, column number, token type and lexeme. The following tokens should be identified - arithmetic operators, relational operators, logical operators, special symbols, keywords, numerical constants, string literals and identifiers. Also, getNextToken() should ignore all the tokens when encountered inside single line or multiline comment or inside string literal. Preprocessor directive should also be ignored.

V. ADDITIONAL EXERCISES:

1. 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;
```

#! 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, foeach are considered as keywords.

@_, += are treated as single tokens.

Remaining symbols are tokenized accordingly.

LAB No.: 3 **Date :**

INTRODUCTION TO FLEX

Objectives:

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

Prerequisites:

- Knowledge of the C programing language.
- Knowledge of basic level regular expressions

I. INTRODUCTION

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. 3.1

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. Then, 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. (sample1.l, sample2.l)
- lex.yy.c defines a routine yylex() that uses the specification to recognize tokens.
- a.out is actually the scanner.

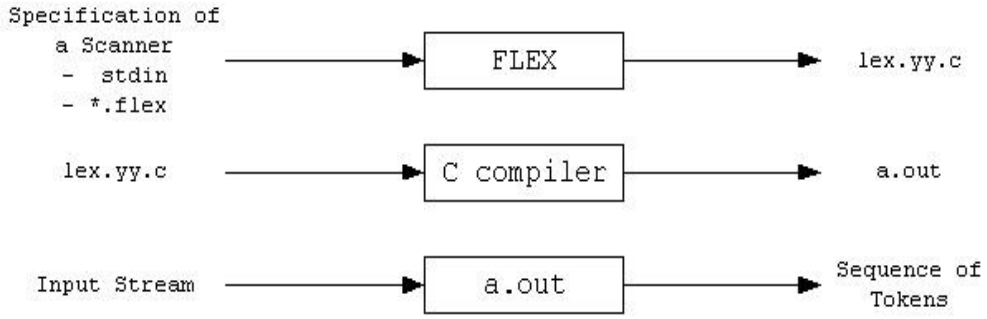


Fig. 3.1 Steps involved in generating Lexical Analyzer using Flex

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

The general format of Flex source program is:

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 consists of the two 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("%8d%8d%8d\n", lines, words, chars); }
```

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 thing, 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 %%.

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 the vast majority of 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.

Table 3.1 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 *yytext*. It is an array of pointer to char where *lex* places the current token's lexeme. The string is automatically null terminated.
yyleng	Stores the length or number of characters in the input string. The value of yyleng is same as strlen( ) functions. In other words it is a integer that holds strlen(yytext).
yyval	This variables 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 3.2 Regular Definitions in FLEX

Reg Expression	**Description**
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 newline character.
\t	matches the tab character
[^a-d]	matches any character other than a,b,c and d.

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.

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).

Tokens and Values

When a flex scanner returns a stream of tokens, each token actually has two parts, the token and the token's value. The token is a small integer. The token numbers are arbitrary, except that token zero always means end-of-file. At the time of parsing, the token numbers are assigned automatically starting at 258. But for now, we'll just define a few tokens by hand:

NUMBER = 258,

ADD = 259,

SUB = 260,

MUL = 261,

DIV = 262,

ABS = 263,

EOL = 264

A token's value identifies which of a group of similar tokens this one is. In our scanner, all numbers are NUMBER tokens, with the value saying what number it is. When parsing more complex input with names, floating-point numbers, string literals, and the like, the value says which name, number, literal, or whatever, this token is.

II. SOLVED EXERCISE:

Write a Flex program to recognize and print tokens for relational operators.

Program:

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define YY_DECL struct token *yylex(void)
```

```
enum tokenType { EOFILE=-1, LESS_THAN,
LESS_THAN_OR_EQUAL,GREATER_THAN,GREATER_THAN_OR_EQUAL,EQUAL,NO
T_EQUAL};
struct token
{
        char *lexeme;
        int index;
        unsigned int rowno,colno;  //row number, column number.
          enum tokenType type;
};
int lineno=1, colno=1;
struct token *tk;
struct token * allocToken() {
                struct token *tk;
                tk=(struct token *)malloc(sizeof(struct token));
                tk->lexeme = (char *)malloc(sizeof(char)*3); //maximum two characters in this
case
                tk->index=-1;
                tk->type=EOFILE;
                return tk;
}

void setTokenArgs(struct token *tk, char *lexeme, int index, int rowno,int colno,enum
tokenType type)
 {
        if(tk==NULL)
                return;
        strcpy(tk->lexeme,lexeme);
        tk->index=index;
        tk->rowno=rowno;
        tk->colno=colno;
        tk->type=type;
```

```
}
%}
%%
"/*".*"*/" {int i = 0;
                while (yytext[i]!='\0') {
                        if(yytext[i]=='\n')
        {
                                lineno++;
                                colno=1;
                        }
                        else
                                colno++;
                        i++;
                }
    }

"//".*"\n" { lineno++;   colno=1;}
(\"(.)*\")  {colno+=strlen(yytext);}
(\'(.)\')     {colno+=strlen(yytext);}
\n          {lineno++; colno=1;}
"<"         {tk=allocToken();
              setTokenArgs(tk,yytext,-1,lineno,colno,LESS_THAN);
               colno++;
               return tk; }
"<="        {tk=allocToken();
setTokenArgs(tk,yytext,-1,lineno,colno,LESS_THAN_OR_EQUAL);
colno+=2;
             return tk; }
">"         {tk=allocToken();
            setTokenArgs(tk,yytext,-1,lineno,colno,GREATER_THAN);
              colno++;
              return tk;
```

```
            }
">="    {tk=allocToken();
          setTokenArgs(tk,yytext,-1,lineno,colno,GREATER_THAN_OR_EQUAL)
colno+=2;
            return tk;}

"=="    {
                  tk=allocToken();
                  setTokenArgs(tk,yytext,-1,lineno,colno,EQUAL);
                  colno+=2;
                  return tk;}
"!="    {         tk=allocToken();
                  setTokenArgs(tk,yytext,-1,lineno,colno,NOT_EQUAL);
                  colno+=2;
                  return tk;}
"\t"              {colno+=8;}
.                 {colno++;}
%%
main(argc,argv)
int argc;
char **argv;
{
        if(argc<2) {
                printf("This program requires name of one C file");
                exit(0);
        }
        yyin=fopen(argv[1],"r");
        int cnt=0;
        while((tk=yylex())) {
                printf("%d %d  %d %s\n",cnt,tk->rowno,tk->colno,tk->lexeme);
                cnt++;
        }
```

```
        return 0;
}
int yywrap() {
        return 1;
}
```

We define the token numbers in a C enum. Then we make yylval, the variable that stores the token value, an integer, which is adequate for the first version of our calculator. For each of the tokens, the scanner returns the appropriate code for the token; for numbers, it turns the string of digits into an integer and stores it in yylval before returning. The pattern that matches whitespace doesn't return, so the scanner just continues to look for what comes next.

Installing FLEX:

Steps to download, compile, and install are as follows. Note: Replace 2.5.33 with your version number:

Downloading Flex (The fast lexical analyzer):

- **Run the command below,**

  wget http://prdownloads.sourceforge.net/flex/flex 2.5.33.tar.gz?download

- **Extracting files from the downloaded package:**

```
tar -xvzf flex-2.5.33.tar.gz
```

- **Now, enter the directory where the package is extracted.**

```
cd flex-2.5.33
```

- **Configuring flex before installation:**

If you haven't installed m4 yet then please do so. Click here to read about the installation instructions for m4. Run the commands below to include m4 in your PATH variable.

```
PATH=$PATH:/usr/local/m4/bin/
```

NOTE: Replace '/usr/local/m4/bin' with the location of m4 binary. Now, configure the source code before installation.

```
./configure --prefix=/usr/local/flex
```

Replace "/usr/local/flex" above with the directory path where you want to copy the files and folders. Note: check for any error message.

Compiling flex:

```
make
```

Note: check for any error message.

Installing flex:

As root (for privileges on destination directory), run the following.

With sudo,

```
sudo make install
```

Without sudo,

```
make install
```

Note: check for any error messages.

Flex has been successfully installed.

Steps to execute:

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

III. LAB EXERCISES

Write a FLEX program to

1. Find 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. Replace scanf with READ and printf with WRITE statement also find the number of scanf and printf in the given input file.
5. Find whether the given sentence is simple or compound for example, if input statement is "My name is John and I study in MIT" then the program should display it as compound statement.
6. Generate tokens for a simple C program. (Tokens to be considered are: Keywords, Identifiers, Special Symbols, arithmetic operators and logical operators)

IV. ADDITONAL EXERCISES:

1.Write a FLEX program that changes a number from decimal to hexadecimal notation.

2.Write a LEX program to convert uppercase characters to lowercase characters of C file excluding the characters present in the comment.

LAB No.: 4 **Date:**

IMPLEMENTATION OF SYMBOL TABLE

Objectives:

- To implement symbol table for the compiler.
- To store the tokens in symbol table.

Prerequisites:

- Knowledge of the C programing language and FLEX.
- Knowledge of file pointers.
- Knowledge of data structures.

I. INTRODUCTION:

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler. A symbol table may serve the following purposes depending upon the language in hand:

- ✓ To store the names of all entities in a structured form at one place.
- ✓ To verify if a variable has been declared.
- ✓ To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- ✓ To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table.

Using a symbol table

- The symbol table is a data structure that is globally accessible (or at least accessible by the parser); it stores the text of most or all of the tokens found in a particular input and associates them with identifying ``indexes".
- The function put_in_tabl() helps maintain the symbol table. This function may be defined in the subroutines section of the lex specification, or in an included file.

- When a token is recognized, put_in_tabl() takes the value in yytext and compares it with all the strings currently in the symbol table.
- If it finds that value already in the table, put_in_tabl() returns the index of the value. The index is a unique identifying integer: no other identifier may have the same index. The index could be an array subscript, but the details of this are unimportant to us as long as the value is unique.
- If the current token does not already appear in the symbol table, a new entry is created for it and an index is generated. (The function may store some other information in the symbol table such as whether the token is an identifier, constant, or other token type.) put_in_tabl() then returns the index.

Structure of symbol table:

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.

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 functions. It stores identifier details used inside the function.

II. SOLVED EXERCISE :

```
/*
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 collosion.
	}
	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;
	}}
```

III. LAB EXERCISES:

1. Implement symbol table to store all the identifiers and user defined function names of a C program.

IV. ADDITIONAL EXERCISES:

1. For the given code snippet, store all the identifiers identified into a symbol table.

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

LAB No.: 5 **Date:**

RECURSIVE DESCENT PARSER FOR A MINIATURE GRAMMAR

Objective:

- To understand implementation of a recursive descent parser for a miniature grammar.
- To write Context-Free Grammar for parsing

Prerequisites:

- Knowledge of top down parsing.
- Knowledge on removal of left recursion from the grammar.
- Knowledge on performing left factoring on the grammar.
- Understanding about computation of first and follow for a grammar.

I. 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) parsers

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.

II. SOLVED EXERCISE:

E→ num T

T→ *num T|ε

Grammar 5.1

[**Note**: Token generated for num is NUMBER]

Algorithm: RD parser for grammar 5.1

Step 1: Read the input

Step 2: Make a function for the start symbol 'E' of the grammar.

Step 3: E( )

```
if lookahead  = num then
                        lookahead=getNextToken();
                        T();
        else
                error();
if lookahead =EOL
  Declare success
else
error()
```

Step 4: T( )

```
if lookahead ='*'
        lookahead=getNextToken();
        if lookahead = 'num'
                lookahead=getNextToken();
            T();
        else
          error();
else  NULL
```

Step 5: Stop

Sample Code :

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

```
#include “lexel.c”  // lexel.c is lexical analyzer program which is coded in previous lab
#include<string.h>
struct token lookahead;
int i=0;
void proc_t();
void proc_e()
{
lookahead=getNexttoken();
if(strcmp(lookahead.lexemename,”NUMBER”)==0)
{
lookahead=getNexttoken();
proc_t();
}
else
{
printf("\n Error");
}
if(strcmp(lookahead.lexemename,”EOL”)==0)
printf("\nSuccessful");
else
printf("\n Error");
}
void proc_t()
  {
lookahead=getNexttoken();
if(strcmp(lookahead.lexemename,”MUL”)==0)
{
lookahead=getNexttoken();
if(strcmp(lookahead.lexemename,”NUMBER”)==0)
{
lookahead=getNexttoken();
          proc_t();
```

```
 }
else
 {
        printf("Error");
  }  } }
 int main()
{
 lookahead=getNextToken();
 proc_e();
 return 0;
 }
```

If the input file contains “NUMBER*NUMBER$”, then token generated is returned by getNextToken( ) which is parsed by the above program. And as input matches the grammar, it displays a success message.

III. LAB EXERCISES:

Write a recursive descent parser for the following simple grammars.

1. S→a | > | (T)
   T→T, S|S

2. E→E+T | T
   T→T*F |F
   F→(E) | id

3. S→aAcBe
   A→Ab|b
   B→d

4. lexp →aterm | list
   aterm →number|identifier
   list→(lexp_seq)
   lexp_seq→ lexp_seqlexp | lexp

IV. 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.
3. Write a program to perform left factoring.

LAB No.: 6 **Date:**

RECURSIVE DESCENT PARSER FOR C CONSTRUCTS

Objectives:

- To design RD parser for various constructs of a 'C' program.
- To be able to perform error detection.

Prerequisites:

- Acquaintance of top down parsing.
- Knowledge on removal of left recursion from the grammar and performing left factoring on the grammar.
- Knowledge on computation of first and follow.
- Basic understanding about error detection and correction methods.

I. RECURSIVE DESCENT PARSER FOR C GRAMMAR:

A simple 'C' language grammar is given. Student should write/update RD parser for subset of grammar each week and integrate it lexical analyzer. Before parsing the input file, remove ambiguity and left recursion, if present and also 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.

II. LAB EXERCISE:

1. Design an unambiguous grammar for defining a switch block in C. Also implement a RD parser to parse the same. In case of any syntactic errors, the parser is expected to report the error along with error message and the line number and column number.

2. Design an unambiguous grammar for defining a looping statement in C. Also implement a RD parser to parse the same. In case of any syntactic errors, the parser is expected to report the error along with error message and the line number and column number.

III. ADDITIONAL EXERCISES:

1. Design a grammar for defining a decision making statement in ‘C’ and implement a RD parser to parse the same.

LAB No. : 7 **Date:**

INTRODUCTION TO BISON

Objectives:

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

Prerequisites:

- Knowledge of the C programing language.
- Knowledge of basic level of context free and EBNF grammars.

I. 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.

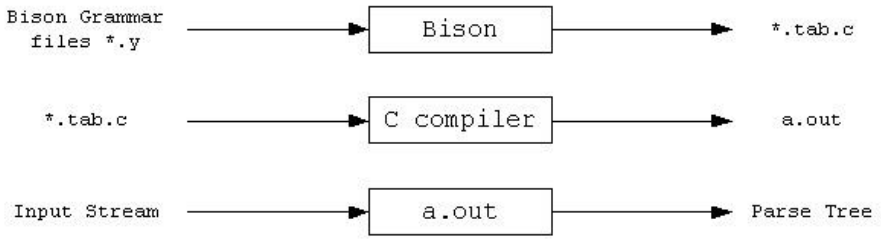


Fig. 7.1 Working of Bison

How a Bison Parser Matches its Input

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.

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.

II. 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"
%}

%%
[0-9]+ {return NUMBER; }
\n {return NL ;}
[a-zA-Z][a-zA-Z0-9_]* {return ID; }
. {return yytext[0]; }
%%
```

Steps to execute:

a. Type Flex program and save it using .l extension.

b. Type the bison program and save it using .y extension.

c. 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.

d. This command generates two files

   filename.tab.h and filename.tab.c

e. Compile the flex code using

$ flex filename.l

f. Compile the generated C file using

$ gcc lex.yy.c filename.tab.c - o output

g. This gives an executable output.out

h. Run the executable using **$ ./output.out**

III. LAB EXERCISES:

Write a bison program,

1. To check a valid declaration statement.
2. To check a valid decision making statement and display the nesting level.
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 → input line | ε

line → '\n' | exp '\n'

exp → 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^nb^n, n>=0). Write a Bison program to validate the strings for the derived grammar.
2. Write a grammar to recognize (a^nb, n>=10). Write a Bison program to validate the strings for the derived grammar.

Lab No. : 8 **Date:**

CONSTRUCTION OF PARSER FOR A MINIATURE C GRAMMAR USING BISON

Objectives:

- To build the parser for C.
- To understand the usage of bison tool.

Prerequisites:

- Knowledge of the C programing language.
- Knowledge of basic level of context free and EBNF grammars.

I. INTRODUCTION

Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression El + E2 has label + and two children representing the subexpressions El and E2.We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an op field that is the label of the node. The objects will have additional fields as follows: If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function Leaf (op, val) creates a leaf object. Alternatively, if nodes are viewed as records, then Leaf returns a pointer to a new record for a leaf. If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function Node takes two or more arguments: Node (op, cl, ca, . . . , ck) creates an object with first field op and k additional fields for the k children cl, . . . , ck.

II. SOLVED EXERCISE

Write a BISON code for arithmetic calculator.

Calculator Flex code :

```
/* recognise tokens for the calculator */
%option noyywrap nodefault yylineno
%{
#include "fun.h"
#include "cal.tab.h"
%}
```

```
/* float exponent */
EXP     ([Ee][-+]?[0-9]+)
%%
"+" |
"-" |
"*" |
"/" |
"|" |
"(" |
")"     { return yytext[0]; }
[0-9]+"."[0-9]*{EXP}? |
"."?[0-9]+{EXP}? { yylval.d = atof(yytext); return NUMBER; }
\n      { return EOL; }
"//".*
[ \t]   { /* ignore whitespace */ }
.       { yyerror("Mystery character %c\n", *yytext); }
%%
```

Calculator Bison code :

```
/* calculator with AST */
%{
#include <stdio.h>
#include <stdlib.h>
#include "fun.h"
%}
%union {
   struct ast *a;
   double d;
}
/* declare tokens */
%token <d> NUMBER
%token EOL
%type <a> exp factor term
%%
calclist: /* nothing */
```

```
| calclist exp EOL {
   printf("=%4.4g\n",eval($2));
   //printf("%c",$2->nodetype);
   postfix($2);
treefree($2);
   printf("> ");
  }
 | calclist EOL { printf("> "); } /* blank line or a comment */
 ;
exp: factor
 | exp '+' factor { $$ = newast('+', $1, $3); }
 | exp '-' factor { $$ = newast('-', $1, $3); }
 ;
factor: term
 | factor '*' term { $$ = newast('*', $1, $3); }
 | factor '/' term { $$ = newast('/', $1, $3); }
 ;
term: NUMBER { $$ = newnum($1); }
 | '|' term { $$ = newast('|', $2, NULL); }
 | '(' exp ')' { $$ = $2; }
 | '-' term { $$ = newast('M', $2, NULL); }
 ;
%%
```

Generation of Abstract Syntax Tree :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "fun.h"
struct ast * newast(int nodetype, struct ast *l, struct ast *r)
{
   struct ast *a = malloc(sizeof(struct ast));

   if(!a) {
      yyerror("out of space");
```

```
        exit(0);
    }
   a->nodetype = nodetype;
    a->l = l;
    a->r = r;
    //printf("--- %d----%d",eval(l),eval(r));
    return a;
}
struct ast * newnum(double d)
{
    struct numval *a = malloc(sizeof(struct numval));
    if(!a) {
        yyerror("out of space");
        exit(0);
    }
   a->nodetype = 'K';
    a->number = d;
    return (struct ast *)a;
}
double eval (struct ast *a)
{
    double v;
    switch(a->nodetype) {
       case 'K': v = ((struct numval *)a)->number; break;

        case '+': v = eval(a->l) + eval(a->r); break;
        case '-': v = eval(a->l) - eval(a->r); break;
        case '*': v = eval(a->l) * eval(a->r); break;
        case '/': v = eval(a->l) / eval(a->r); break;
        case '|': v = eval(a->l); if(v < 0) v = -v; break;
        case 'M': v = -eval(a->l); break;
        default: printf("internal error: bad node %c\n", a->nodetype);
    }
    return v;
```

```
}
void treefree(struct ast *a)
{
   switch(a->nodetype)
   {
      /* two subtrees */
      case '+':
      case '-':
      case '*':
      case '/':
         treefree(a->r);
      /* one subtree */
      case '|':
      case 'M':
         treefree(a->l);

      /* no subtree */
      case 'K':
         free(a);
         break;
      default: printf("internal error: free bad node %c\n", a->nodetype);
   }
}
void yyerror(char *s, ...)
{
   va_list ap;
   va_start(ap, s);
   fprintf(stderr, "%d: error: ", yylineno);
   vfprintf(stderr, s, ap);
   fprintf(stderr, "\n");
}
void postfix(struct ast *a) {
        struct ast *temp=a;
        //printf("%c ",a->nodetype);
```

```
    if(temp !=NULL) {
        if(temp->nodetype!='K')
                postfix(temp->l);
        if(temp->nodetype!='K')
                postfix(temp->r);
        if(temp->nodetype=='K')
                printf("%f ",eval(temp));
        else
                printf("%c",temp->nodetype);
    }
    return ;
}
int main ()
{
    printf("> ");
    return yyparse();
}
```

Header File:

```
/*
 * Declarations for calculator  fb3-1
 */
/* Interface to the lexer */
extern int yylineno; /* from lexer */
void yyerror(char *s, ...);
/* nodes in the abstract syntax tree */
struct ast {
    int nodetype;
    struct ast *l;
    struct ast *r;
};
struct numval {
    int nodetype;   /* type K for constant */
    double number;
```

```
};

/* build an AST */
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newnum(double d);
/* evaluate an AST */
double eval(struct ast *);
/* delete and free an AST */
void treefree(struct ast *);
void postfix(struct ast *);
```

III. LAB EXERCISES:

1. Write a bison code to check validity of relational, logical and arithmetic expressions. Here, an expression may have variable names or constant names apart from numbers.
2. Write a bison code to implement C parser. (Appendix)

IV. ADDITIONAL EXERCISE:

1. Write a bison code to implement structures and unions for C constructs.

LAB NO: 9 **Date:**

INTERMEDIATE CODE GENERATION

Objectives:

- To understand the intermediate code generation phase of compilation.
- To generate the intermediate representation i.e. three address code from simple C code.

Prerequisites:

- Knowledge of three address code statements.

I. INTRODUCTION

This phase of compiler construction takes postfix notation generated from the syntax tree in the previous phase as input and produces intermediate representation. There are wide range of intermediate representations and in this lab, we mainly consider an intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done. Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some three-address instructions have fewer than three operands.

II. 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;
}
```

III. LAB EXERCISES:

1. Write a C program to generate Three Address Code statements for arithmetic expressions including parenthesis. Also store the generated TAC statements to a file.
2. Write a C program to generate Three Address Code statements for decision statements.

IV. ADDITIONAL EXERCISES:

1. Write a C program to generate Three Address Code statements for the following sample C code snippet.

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

}
```

LAB No.: 10 **Date:**

CODE GENERATION

Objectives:

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

I. INTRODUCTION:

Code Generation is the next 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.

The assembly code has three different kinds of elements:

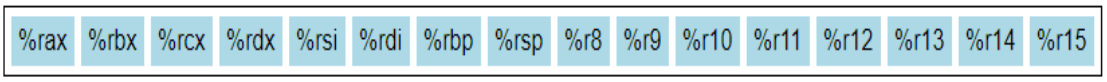
- **Directives** begin with a dot and indicate structural information useful to the assembler, linker, or debugger, but are not in and of themselves assembly instructions. For example, .file simply

records the name of the original source file. .data indicates the start of the data section of the program, while .text indicates the start of the actual program code. .string indicates a string constant within the data section, and .globl main indicates that the label main is a global symbol that can be accessed by other code modules. (You can ignore most of the other directives.)

- **Labels** end with a colon and indicate by their position the association between names and locations. For example, the label .LC0: indicates that the immediately following string should be called .LC0. The label main: indicates that the instruction pushq %rbp is the first instruction of the main function. By convention, labels beginning with a dot are temporary local labels generated by the compiler, while other symbols are user-visible functions and global variables.
- **Instructions** are the actual assembly code (pushq %rbp), typically indented to visually distinguish them from directives and labels.

Registers

We say almost general purpose because earlier versions of the processors intended for each register to be used for a specific purpose, and not all instructions could be applied to every register.



A few remaining instructions, particularly related to string processing, require the use of %rsi and %rdi. In addition, two registers are reserved for use as the stack pointer (%rsp) and the base pointer (%rbp). The final eight registers are numbered and have no specific restrictions. The architecture has expanded from 8 to 16 to 32 bits over the years, and so each register has some internal structure that you should know about:

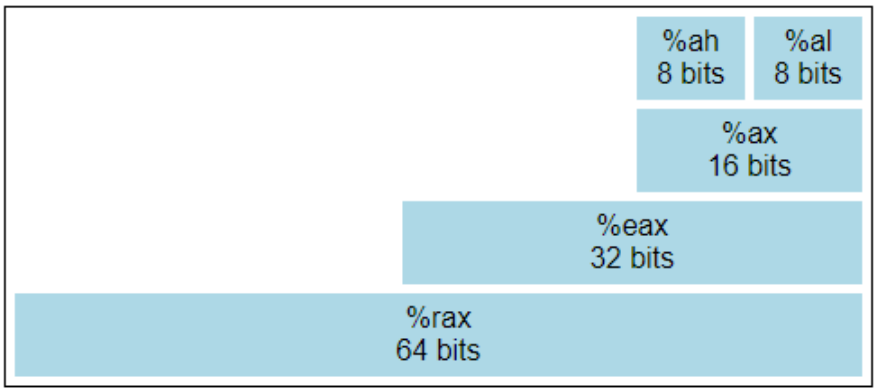


Fig. 10.1 Registers

The lowest 8 bits of the %rax register are an 8-bit register %al, and the next 8 bits are known as %ah. The low 16 bits are collectively known as %ax, the low 32-bits as %eax, and the whole 64 bits as %rax.

Addressing modes

As the design developed, new instructions and addressing modes were added to make the various registers almost equal. The first instruction that you should know about is the MOV instruction, which moves data between registers and to and from memory. X86-64 is a complex instruction set (CISC), so the MOV instruction has many different variants that move different types of data between different cells.

MOV, like most instructions, has a single letter suffix that determines the amount of data to be moved.

Table 10.1: Describe data values of various sizes

Suffix	Name	Size
B	BYTE	1 byte (8 bits)
W	WORD	2 bytes (16 bits)
L	LONG	4 bytes (32 bits)
Q	QUADWORD	8 bytes (64 bits)

The arguments to MOV can have one of several addressing modes. A global value is simply referred to by an unadorned name such as x or printf An immediate value is a constant value indicated by a dollar sign such as $56 A register value is the name of a register such as %rbx. An indirect refers to a value by the address contained in a register. For example, (%rsp)refers to the value pointed to by %rsp. A base-relative value is given by adding a constant to the name of a register. For example, -16(%rcx) refers to the value at the memory location sixteen bytes below the address indicated by %rcx. This mode is important for manipulating stacks, local values, and function parameters. There are a variety of complex variations on base-relative, for example -

16(%rbx,%rcx,8) refers to the value at the address -16+%rbx+%rcx*8. This mode is useful for accessing elements of unusual sizes arranged in arrays.

Table 10.2: Addressing modes

Mode	**Example**
Global Symbol	MOVQ x, %rax
Immediate	MOVQ $56, %rax
Register	MOVQ %rbx, %rax
Indirect	MOVQ (%rsp), %rax
Base-Relative	MOVQ -8(%rbp), %rax
Offset-Scaled-Base-Relative	MOVQ -16(%rbx,%rcx,8), %rax

Basic Arithmetic

You will need four basic arithmetic instructions for your compiler: ADD, SUB, IMUL, and IDIV. ADD and SUB have two operands: a source and a destructive target. For example, this instruction:

```
ADDQ %rbx, %rax
```

adds %rbx to %rax, and places the result in %rax, overwriting what might have been there before. This requires that you be a little careful in how you make use of registers. For example, suppose that you wish to translate c = b*(b+a), where a and b are global integers. To do this, you must be careful not to clobber the value of b when performing the addition. Here is one possible translation:

```
MOVQ  a, %rax
MOVQ  b, %rbx
ADDQ  %rbx, %rax
IMULQ %rbx
```

```
MOVQ  %rax, c
```

The IMUL instruction is a little unusual: it takes its argument, multiplies it by the contents of %rax, and then places the low 64 bits of the result in %rax and the high 64 bits in %rdx. (Multiplying two 64-bit numbers yields a 128-bit number, after all.)

The IDIV instruction does the same thing, except backwards: it starts with a 128 bit integer value whose low 64 bits are in %rax and high 64 bits in %rdx, and divides it by the value give in the instruction. (The CDQO instruction serves the very specific purpose of sign-extending %rax into %rdx, to handle negative values correctly.) The quotient is placed in %rax and the remainder in %rdx. For example, to divide a by five:

```
MOVQ a,  %rax    # set the low 64 bits of the dividend
CDQO             # sign-extend %rax into %rdx
IDIVQ $5         # divide %rdx:%rax by 5, leaving result in %eax
```

(Note that the modulus instruction found in most languages simply exploits the remainder left in %rdx.)

The instructions INC and DEC increment and decrement a register destructively. For example, the statement a = ++b could be translated as:

```
MOVQ b, %rax
INCQ %rax
MOVQ %rax, a
```

Boolean operations work in a very similar manner: AND, OR, and XOR perform destructive boolean operations on two operands, while NOT performs a destructive boolean-not on one operand.

Like the MOV instruction, the various arithmetic instructions can work on a variety of addressing modes. However, for your compiler project, you will likely find it most convenient to use MOV to load values in and out of registers, and then use only registers to perform arithmetic.

Comparisons and Jumps

Using the JMP instruction, we may create a simple infinite loop that counts up from zero using the %eax register:

```
 MOVQ $0, %rax
loop:
    INCQ %rax
    JMP loop
```

To define more useful structures such as *terminating* loops and if-then statements, we must have a mechanism for evaluating values and changing program flow. In most assembly languages, these are handled by two different kinds of instructions: compares and jumps.

All comparisons are done with the CMP instruction. CMP compares two different registers and then sets a few bits in an internal EFLAGS registers, recording whether the values are the same, greater, or lesser. You don't need to look at the EFLAGS register directly. Instead a selection of conditional jumps examine the EFLAGS register and jump appropriately:

Table 10.3: Jumps

Instruction	Meaning
JE	Jump If Equal
JNE	Jump If Not Equal
JL	Jump If Less Than
JLE	Jump If Less or Equal
JG	Jump if Greater Than
JGE	Jump If Greater or Equal

For example, here is a loop to count %rax from zero to five:

```
    MOVQ $0, %rax
loop:
    INCQ %rax
```

```
    CMPQ $5, %rax
    JLE  loop
```

And here is a conditional assignment: if global variable x is greater than zero, then global variable y gets ten, else twenty:

```
    MOVQ x, %rax
    CMPQ $0, %rax
    JLE  twenty
ten:
    MOVQ $10, %rbx
    JMP  done
twenty:
    MOVQ $20, %rbx
    JMP  done
done:
    MOVQ %ebx, y
```

Note that jumps require the compiler to define target labels. These labels must be unique and private within one assembly file, but cannot be seen outside the file unless a `.globl` directive is given. In C parlance, a plain assembly label is `static`, while a `.globl` label is extern.

II. SOLVED EXERCISE:

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

Program:

```
#include<stdio.h>
#include<string.h>
/*
//This program works for the following input.
```

```
#include<stdio.h>
int main(){
	int a=3,b=2;
	int c;
	c=a+b;   // or c=a-b
	printf("%d",c);
	return 0;
}
*/

char *three_address_input="c = a + b";

char *code_prefix=
"	.file	\"input.c\"			\n"
"	.section	.rodata		\n"
".LC0:						\n"
"	.string	\"output=%d\\n\"	\n"
"	.text					\n"
"	.globl	main			\n"
"	.type	main, @function	\n"
"main:						\n"
".LFB0:						\n"
"	.cfi_startproc		\n"
"	pushq	%rbp		\n"
"	.cfi_def_cfa_offset 16	\n"
"	.cfi_offset 6, -16		\n"
"	movq	%rsp, %rbp	\n"
"	.cfi_def_cfa_register 6	\n"
"	subq	$16, %rsp	\n"
"	movl	$3, -12(%rbp)	\n"
"	movl	$2, -8(%rbp)	\n"
"	movl	-8(%rbp), %eax	\n"
"	movl	-12(%rbp), %edx	\n";
```

```
//      subl    %eax, %edx
char *code_suffix=
"       movl    %eax, -4(%rbp)                                  \n"
"       movl    -4(%rbp), %eax                                  \n"
"       movl    %eax, %esi                                      \n"
"       movl    $.LC0, %edi                                     \n"
"       movl    $0, %eax                                        \n"
"       call    printf                                              \n"
"       movl    $0, %eax                                        \n"
"       leave                                                   \n"
"       .cfi_def_cfa 7, 8                                       \n"
"       ret                                                     \n"
"       .cfi_endproc                                                                    \n"

".LFE0:                                                         \n"
"       .size   main, .-main                                    \n"
"       .ident  \"GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4\"  \n"
"       .section        .note.GNU-stack,\"\",@progbits                  \n";
int main() {
        int i;
        unsigned char afterEqual=0;  // boolean flag

        FILE *fp;
        fp=fopen("input.s","w");
        fprintf(fp,"%s",code_prefix);
        for(i=0;i<strlen(three_address_input);i++) {
                if(!afterEqual &&three_address_input[i] !=' ')
                        continue;
                if(three_address_input[i]==' ') {
                        afterEqual=1;
                }
          if(three_address_input[i]=='+')
                        fprintf(fp,"\taddl\t %%edx,%%eax\n");
                else if(three_address_input[i]=='-')
```

```
                fprintf(fp,"\tsubl\t %%eax,%%edx\n\tmovl\t%%edx,%%eax\n");
        }
        fprintf(fp,"%s",code_suffix);
        fclose(fp);
        /*system("gcc input.s -o input.o");
        system("gcc input.o -o input");
        system("./input");*/
        return 0;
}
```

The assembly code corresponding to the following C program is can also 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 C program that takes a file containing TAC for expression statements as input and generates the assembly level code for the same.

2. Write a C program that takes a file containing TAC for decision making statements as input and generates the assembly level code for the same.

IV. ADDITIONAL EXERCISES

1. Write a C program that takes a file containing TAC for expression involving arrays as input and generates the assembly level code for the same.

2. Write a C program that takes a file containing TAC for switch statement as input and generates the assembly level code for the same.

LAB No.: 11 & 12 **Date:**

MINI PROJECT

Objectives:

- To implement the various phases of compiler for different programming language.

Project Submission guidelines:

- Student can form a group of atmost 3 members.
- Student must do a mini project using Flex and Bison.
- Student must subit the synopsis in the 6th lab.
- Complete the mini project and demonstrate in the 12th lab.
- Student must submit a report in the 12th lab.

Project Synopsis format

- Synopsis should contain the project title, grammar of the language constructs selected and team members name along with section and roll number.
- Refer the IEEE report format for final report.

REFERENCES:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques and Tools, Pearson Education, 2nd Edition, 2010.
2. D M Dhamdhere, "Systems Programming and Operating Systems", Tata McGraw Hill, 2nd Revised Edition, 2001.
3. Kenneth C. Louden, "Compiler Construction- Principles and Practice", Thomson, India Edition, 2007.
4. "Keywords and Identifiers",https://www.programiz.com/c-programming/c-keywords-identifier.
5. Behrouz A. Forouzan, Richard F. Gilberg " A Structured Programming Approach Using C", 3rd edition, Cengage Learning India Private Limited, India,2007.
6. Debasis Samanta, "Classic Data Structures", 2nd Edition, PHI Learning Private Limited, India,2010.
7. File handling ,http://iiti.ac.in/people/~tanimad/FileHandlinginCLanguage.pdf
8. https://www3.nd.edu/~dthain/courses/cse40243/fall2015/intel-intro.html

Appendix

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

eprime→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 → * | / | %