BATCH No.

Rashtreeya Sikshana Samithi Trust

R.V. College of Engineering®

(Autonomous Institution affiliated to VTU, Belagavi)

Department of Computer Science & Engineering

Bengaluru -560059

COMPILER DESIGN LAB

(18CS63)

VI Semester B.E

LABORATORY RECORD 2020 - 2021

Name of the Student:_____

U S N _____

Semester : _____ **Section :**_____

R.V. College of Engineering®

(Autonomous Institution affiliated to VTU, Belagavi)

Department of Computer Science & Engineering

COMPILER DESIGN

18CS63

Laboratory Record (Autonomous Scheme 2018)

R.V. College of Engineering®

(Autonomous Institution affiliated to VTU, Belagavi)

Department of Computer Science & Engineering

Laboratory Certificate

This is to certify that Mr. / Ms _____

_____has satisfactorily completed the course of Experiments in Practical_____
Prescribed by the Department of Computer Science and Engineering during the year _____

Name of the Candidate_____

USN No.:_____Semester:_____

Marks	
Maximum	**Obtained**

Signature of the staff in-charge Head of the Department

Date:

Department of Computer Science and Engineering R V College of Engineering, Bengaluru

Vision

To achieve leadership in the field of Computer Science & Engineering by strengthening fundamentals and facilitating interdisciplinary sustainable research to meet the ever growing needs of the society.

Mission

- To evolve continually as a center of excellence in quality education in computers and allied fields.
- To develop state-of-the-art infrastructure and create environment capable for interdisciplinary research and skill enhancement.
- To collaborate with industries and institutions at national and international levels to enhance research in emerging areas.
- To develop professionals having social concern to become leaders in top-notch industries and/or become entrepreneurs with good ethics.

Program Educational Objectives (PEOs):

PEO1: Develop Graduates capable of applying the principles of mathematics, science, core engineering and Computer Science to solve real-world problems in interdisciplinary domains.

PEO2: To develop the ability among graduates to analyze and understand current pedagogical techniques, industry accepted computing practices and state-of-art technology.

PEO3: To develop graduates who will exhibit cultural awareness, teamwork with professional ethics, effective communication skills and appropriately apply knowledge of societal impacts of computing technology.

PEO4: To prepare graduates with a capability to successfully get employed in the right role and achieve higher career goals or take-up higher education in pursuit of lifelong learning.

Program Outcomes (PO's):

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems

PO2: Problem analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and modeling to complex engineering activities, with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess Societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAMS

Student should be able to design phases of compiler by incorporating following features:

1 .Writing a scanner, writing predictive parser for a language construct.
2 . Experiment with scanner (lex/flex) and parser (yacc/byson) generators
3 . Writing scanner-parse specification for a simple language constructs.
4 . Translation of the language constructs to an intermediate form (e.g. three-address code),
5 . Generation of target code (in assembly language) using compiler construction tools.
6 . Code improvement and optimization using LLVM compiler.

RUBRICS

Sl. No	Criteria	Measuring Method	Excellent	Good	Poor
Lab Write-up and Execution rubrics (Max: 6 marks)					
a.	**Writing regular expression/Grammar for the given problem statement (CO3)** **(2 Marks)**	Analysis and design skills	Student exhibits thorough understanding of requirements and applies knowledge of regular expressions/ grammar using Lex and Yacc programming, and obtaining finite solutions to the given problem **(2M)**	Student has sufficient understanding of requirements and applies suitable understanding of regular expressions/ grammar using Lex and Yacc programming **(1M – 1.5M)**	Student does not have a clear understanding of requirements and is unable to apply suitable regular expressions/ grammar using Lex and Yacc programming.(**0M)**
b.	**Execution and Testing (CO4)** **(2 Marks)**	implementation skills	Program handles all possible conditions while execution of the code with satisfying results. **(2M)**	Average conditions are checked and verified in the code. **(1M – 1.5M)**	No Execution. **(0M)**
c.	**Results and Documentation (CO1)** **(2 Marks)**	Presentation skills	Meticulous documentation of changes made and the results obtained in data sheets. **(2M)**	Acceptable documentation shown in data sheets. **(1M – 1.5M)**	No Proper result and documentation. **(0M)**
Rubrics for viva voce (Max: 4 marks)					
a.	**Conceptual Understanding (CO1)** **(2 Marks)**	Viva Voce	Explains fully concepts of Lexical and/or Syntax Analyzer rules used in the given problem . **(2M)**	Adequately provides explanation on the lexical and/or syntax analyzer rules used in the given problem **(1M – 1.5M)**	Unable to explain concepts. **(0M)**
b.	**Analyzing the translation rules for solving the given problem (CO2)** **(1 Mark)**	Viva Voce	Insightful use of strategies and concepts to derive solutions to specified problem. **(1M)**	Intuitive use of strategies and concepts to derive solutions to specified problem. **(0.5M)**	Did not solve problem specified in the problem. **(0M)**
c.	**Communication of Idea (CO1)** **(1 Marks)**	Viva Voce	Communicates all thoughts clearly **(1M)**	Communicates thoughts moderately **(0.5M)**	Unable to communicate ideas **(0M)**

EVALUATION

Program No.	Date of Submission	Lab Write-up and Execution marks(6 marks)			Viva voce Marks (4 marks)			Total Marks	Signature
		a	b	c	a	b	c		
1 a									
1 b									
2 a									
2 b									
3 a									
3 b									
4									
5									
6									
7									
8									
Total Marks									

REDUCED MARKS = (Total/100)*40 = _____

Program Structure of Lex

The input program structure of lex contains three parts namely

1. Declaration section (or definition section starts with "%{" and ends with "%}")
2. Rules section (starts with "%%" and ends with "%%")
3. Subroutine section (or c code section)

- A lex file would have the extension ".l"
- The ".l " file is given as input to the lex it will generate a file with extension ".c" named as "lex.yy.c"

To put it in a more precise way, Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

1. A simple pattern: **letter(letter|digit)***
2. Regular expressions are translated by lex to a computer program that mimics an FSA.(Finite State Automata)
3. This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits.

Table 1: Pattern Matching Primitives

Meta character	Matches
.	any character except newline
\n	newline
*****	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	**a** or **b**
(ab)+	one or more copies of **ab** (grouping)
"a+b"	literal "**a+b**" (C escapes still work)
[]	character class

Table 2: Pattern Matching Examples

Expression	Matches
abc	**abc**
abc*	**ab abcabccabccc ...**
abc+	**abc, abcc, abccc, abcccc, ...**
a(bc)+	**abc, abcbc, abcbcbc, ...**
a(bc)?	**a, abc**
[abc]	one of: **a, b, c**
[a-z]	any letter, a through z
[a\-z]	one of: **a, -, z**
[-az]	one of: **- a z**
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: **a, b**
[a^b]	**a, ^, b**
[a|b]	**a, |, b**
a|b	**a, b**

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2.Predefined variables are shown in Table 3. Within a character class normal operators lose their meaning. Two operators allowed in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression.

Table 3: Lex Predefined Variables

Name	Function
Int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
Int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

Exercise Problems on LEX

1. Write a LEX program to count the number of vowels and consonants in a given string.

```
%{
int c=0,v=0;
%}
%%
[aeiouAEIOU] v++;
[a-zA-Z] c++;
. ;
\n return 0;
%%
int main()
{
printf("Enter the string\n");
yylex();
printf("no. of vowels=%d\nno. of consonants=%d\n",v,c);
}
```

----------OUTPUT----------

```
1.Enter the string
aeiou
no. of vowels=5
no. of consonants=0
```

2. Enter the string

R V College of engineering

no. of vowels=9

no. of consonants=13

2. Write a LEX program to count the number of 'scanf' and 'printf' statements in a C program. Replace them with 'readf' and 'writef' statements respectively.

```
%{
#include<stdio.h>
#include<math.h>
int p=0,s=0;
%}
%%

"printf" {p++;fprintf(yyout,"writef");}
"scanf" {s++;fprintf(yyout,"readf");}
; {fprintf(yyout,"%s",yytext);}
\n {fprintf(yyout,"\n");}
%%

int main()
{
yyin=fopen("c.c","r+");
yyout=fopen("d.c","r+");
yylex();
printf("The no. of printf's are %d\n\t\tscanf's are%d\n",p,s);
}
```

input file c.c

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

```
printf("hfgjxfdj");
scanf("khfg");
printf("kdgfghj");
```

----------OUTPUT----------

The no. of
printf's are 2
scanf's are1

output file c.c

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

writef("hfgjxfdj");
readf("khfg");
writef("kdgfghj");
```

Program Structure for YACC

A YACC source program is structurally similar to a LEX one.

```
declarations
%%
rules
%%
routines
```

- The declaration section may be empty. Moreover, if the routines section is omitted, the second %% mark may be omitted also.
- Blanks, tabs, and newlines are ignored except that they may not appear in names.

THE DECLARATIONS SECTION may contain the following items.

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword ***%token***.
- Declaration of the start symbol using the keyword ***%start***
- C declarations: included files, global variables, types.
- C code between ***%{*** and ***%}***.

RULES SECTION.

A rule has the form:

```
nonterminal : sentential form
        | sentential form
        .................
        | sentential form
        ;
```

Actions may be associated with rules and are executed when the associated sentential form is matched.

To Run the above program:
$lex 1b.l => generates lex.yy.c file
$yacc –d 1b.y => generates y.tab.c and y.tab.h(file of token definitions)
$cc lex.yy.cy.tab.c –ly –ll
$./a.out=> to run OR you can create your own executable file

Exercise Problems on YACC

1. Write a YACC program to recognize strings of the form $a^n b^n$, n>=0.

Lex Program

```
%{
#include "y.tab.h"
%}
%%

"a" { return 'a';}
"b" { return 'b';}
. return yytext[0];
\n return 0;
%%
```

Yacc Program

```
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
%}
%%
S:'a'S'b'
|
;
%%

int main()
{
yyparse();
printf("\n Valid string\n");
}

int yyerror()
{
printf("INVALID!!!\n");
exit(0);
}
```

----------OUTPUT----------

Valid string
aabb
Valid string

aaabb
INVALID!!!

abbb
INVALID!!!

a
INVALID!!!

2. YACC program that reads the input expression and convert it to post fix expression.

gram.l

```
%{ #include"y.tab.h"
extern int yylval;
%}
%%
[0-9]+  {yylval=atoi(yytext); return NUM;}
\n  return 0;
.   return *yytext;
%%
int yywrap()
{
return 1; }
```

gram.y

```
%{
#include<stdio.h>
%}
%token NUM
%left '+' '-'
%left '*' '/'
%right NEGATIVE
```

```
%%
S: E {printf("\n");}  ;
E: E '+' E {printf("+");}
   | E '*' E {printf("*");}
   | E '-' E {printf("-");}
   | E '/' E {printf("/");}
   | '(' E ')'  | '-' E %prec NEGATIVE {printf("-");}
   | NUM  {printf("%d", yylval);}  ;
 %%
int main()
{
    yyparse();
 }
int yyerror (char *msg)
{
  return printf ("error YACC: %s\n", msg);
 }
```

Input: 2+6*2-5/3

Output: 262*+53/-

Brief Introduction on LLVM

The LLVM Project is a collection of modular and reusable compiler and tool chain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project. LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM[1] has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research.

The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular. These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.

Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles (e.g. about 3x faster than GCC when compiling Objective-C code in a debug configuration), extremely useful error and warning messages and to provide a platform for building great source level tools. The Clang Static Analyzer is a tool that automatically finds bugs in your code, and is a great example of the sort of tool that can be built using the Clang frontend as a library to parse C/C++ code.

Try out some of the commands(assuming you add llvm/build/bin to your path):[2]

$clang --help

$clang file.c -fsyntax-only (check for correctness)

$clang file.c -S -emit-llvm -o - (print out unoptimized llvm code)

$clang file.c -S -emit-llvm -o - -O3

$clang file.c -S -O3 -o - (output native machine code)

The clang tool is the compiler driver and front-end, which is designed to be a drop-in replacement for the gcc command. Here are some examples of how to use the high-level driver:

```
$ cat t.c
    #include <stdio.h>
    int main(int argc, char **argv) { printf("hello world\n"); }
$ clang t.c
$ ./a.out
    hello world
```

The 'clang' driver is designed to work as closely to GCC as possible to maximize portability. The only major difference between the two is that Clang defaults to gnu99 mode while GCC defaults to gnu89 mode. If you see weird link-time errors relating to inline functions, try passing -std=gnu89 to clang.

Examples of using Clang [4]

```
$ cat ~/t.c
    typedef float V __attribute__((vector_size(16)));
    V foo(V a, V b) { return a+b*a; }
```

Preprocessing:

```
$ clang t.c -E
    # 1 "/Users/sabre/t.c" 1
    typedef float V __attribute__((vector_size(16)));
    V foo(V a, V b) { return a+b*a; }
```

Type checking:

```
$ clang -fsyntax-only t.c
```

GCC options:

```
$ clang -fsyntax-only t.c -pedantic

    /Users/sabre/t.c:2:17: warning: extension used

    typedef float V __attribute__((vector_size(16)));
                    ^1 diagnostic generated.
```

Pretty printing from the AST:

Note, the -cc1 argument indicates the compiler front-end, and not the driver, should be run. The compiler front-end has several additional Clang specific features which are not exposed through the GCC compatible driver interface.

```
$ clang -cc1 t.c -ast-print

    typedef float V __attribute__(( vector_size(16) ));

    V foo(V a, V b) {

      return a + b * a;

    }
```

Code generation with LLVM:

```
$ clang t.c -S -emit-llvm -o -

    define <4 x float> @foo(<4 x float> %a, <4 x float> %b) {

    entry:

        %mul = mul <4 x float> %b, %a

        %add = add <4 x float> %mul, %a

        ret <4 x float> %add

    }

$ clang -fomit-frame-pointer -O3 -S -o - t.c # On x86_64

    ...
```

```
_foo:
Leh_func_begin1:
        mulps   %xmm0, %xmm1
        addps   %xmm1, %xmm0
        ret
Leh_func_end1:
```

Reference websites for more on llvm/ clang:

1. http://llvm.org/
2. http://clang.llvm.org/
3. https://www.youtube.com/watch?v=E6i8jmiy8MY
4. http://clang.llvm.org/get_started.html

Experiment No-1(a)

Writing a scanner

A Scanner should create a token stream from the source code.

Step1- Write down the regular expression for recognising characters, words, white spaces, lines etc.

Step2- Declare the variables.

Step 3- Write down the necessary auxiliary functions.

OUTPUT:

Experiment No-1(b)

Writing A Parser

Write a YACC program to recognize strings of the given form i.e $a^n b^{n+m} c^m$, n,m>=0

Step1- Write the Lex Part to identify the Tokens

Step2-Declare the variables

Step3- Write a CFG to generate the strings of the form given and convert it to a YACC program

Step4– Write down the Auxiliary procedures

OUTPUT:

Experiment No-2(a)

Write a LEX program to count number of Positive & negative integers and Positive & negative fractions

Step1- Write down the regular expression for Positive & negative integers and Positive & negative fractions (for eg. Fractions may include 2.3 or 2/3 with combinations of –ve and +ve signs)
Step2- Declare the variables.
Step3- Convert the expression into a lex program with the translation rules.
Step4- Write down the auxiliary functions.

OUTPUT:

Experiment No-2(b)

Write a YACC program to validate and evaluate a simple expression involving operators +,- , * and /.

Step1- Write the Lex Part identify to identify numbers or characters as operands and operators.

Step2-Declare the variables

Step3- Write and Convert the CFG to YACC program

Step4– Auxiliary procedures

GRAMMAR:

OUTPUT:

Experiment No- 3

Writing scanner-parse specification for a simple language construct.

A). Write a Parser program to recognize a nested (minimum 3 levels) FOR loop statement for C language.

Step1- Write a LEX program having the regular expressions to recognize keyword FOR, different operators, parenthesis, numbers etc. Consider simple expressions inside the for loop.

Step2- Declare the variables.

Step3- Write down a CFG using the generated tokens in the lex file and convert the grammar into a YACC program.

Step4- Write down the auxiliary functions.

OUTPUT:

/

Experiment No-3(b)

Write a YACC program that identifies Function Definition of C language

Step1- Write the Lex Part to identify the keywords and variables

Step2-Declare the variables

Step3- Write and Convert the CFG to YACC program to identify function definition

Step4– Auxiliary procedures

GRAMMAR:

OUTPUT:

Experiment No-4
Translation of the language constructs to an intermediate form (e.g. three-address code)

Write a YACC program that reads the C statements from an input file and converts them into three address code representation.

Step 1: write a lex file to recognise characters and numbers

Step 2 : Write a Yacc file to convert the expression into Quadruple and three address code representation

Input: Valid expression

Output: Three address code ,Quadruple and triples representation

Output:

Experiment No-5

Generation of target code (in assembly language) using compiler construction tools.

Write a program to generate the assembly code for the given source code. You can use C language or any compiler construction tools.

```
#include<stdio.h>

int main()

{

 int a,b,c;

 a=45;

 b=25;

 c=a+b;

 printf("Sum %d",c);

 return 0;

}
```

N: B: Students can try with different Source code

OUTPUT:

INNOVATIVE EXPERIMENTS

Experiment No-6

Code improvement and optimization using LLVM compiler.

Step 1: Take the source code of Bubble sort in C language program (or any sorting algorithm).
Step 2: Run it through LLVM Compiler
Step 3: Print the unoptimized assembly code
Step 4: Apply Optimization techniques and print the optimized assembly code
Step 5: Justify the optimized code

OUTPUT:

Experiment No-7

Code improvement and optimization using CLANG compiler.

Step 1: Take the source code of Binary Search sort in C language program (or any searching algorithm).
Step 2: Run it through CLANG Compiler
Step 3: Print the unoptimized assembly code
Step 4: Apply Optimization techniques and print the optimized assembly code
Step 5: Justify the optimized code

OUTPUT:

Experiment No-8

Loop Unrolling using LLVM Compiler

For the given loop apply loop unrolling optimization technique in LLVM and write down the output in assembly code.

```
For (i=0; i<N;i++)
  For(j=0;j<N;j++)
    c[i]=a[i,j] * b[i];
```

OUTPUT: