



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



# **CS3109: Compiler Design**

## **Laboratory Manual**

### **Computer Engineering**



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



# **CS3109: Compiler Design**

## **Laboratory Manual**

### **Computer Engineering**

**Author : - Dr . Rushali A. Deshmukh**

**Creation Date: - Dec 2021**

**Last Updated : - Dec 2021**

**Version : - 1**

**© JSPM Group of Institutes, Pune. All Rights Reserved. All the information in this Course Manual is confidential. Participants shall refrain from copying, distributing, misusing or disclosing the content to any third parties any circumstances whatsoever.**



## Table of Contents

Sr . No.	Topic	Page. No.
A.	<b>Vision, Mission</b>	4
B.	<b>PEOs and POs</b>	5
C.	<b>PSOs</b>	7
D.	<b>Course Objectives and its mapping with POs and PSOs</b>	8
<b>F. List of Experiments</b>		
1.	Write a program to count number of lines, tabs, spaces, words, characters from a given text file.	9
2.	Implement the Lexical analyzer for the given language. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.	17
3.	Write a program for syntax checking of subset of given language using LEX and YACC.	24
4.	Write a program for syntax checking of control statements using LEX and YACC.	36
5.	Write a program to check syntax of declaration statement using LEX and YACC.	43
6.	Implement a desk calculator using LEX and YACC.	50
7.	Write a program to generate ICG using LEX and YACC.	57
8.	Write a program for code optimization.	70
9.	Write a program for code generation.	78
G.	<b>References</b>	89



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



### **Vision of Department**

To create quality computer professionals through an excellent academic environment.

### **Mission of Department**

1. To empower students with the fundamentals of Computer Engineering for being successful professionals.
2. To motivate the students for higher studies, research, and entrepreneurship by imparting quality education.
3. To create social awareness among the students.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



**Program Educational Objectives: -**

- PEO I**      Graduates shall have successful professional careers and lead and manage teams.
- PEO II**      Graduates shall exhibit disciplinary skills to resolve real-life problems.
- PEO III**     Graduates shall evolve as professionals or researchers and continue to learn emerging technologies.

**Program Outcomes: -**

Engineering Graduates will be able to:

- 1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- 2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- 3. 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 the public health and safety, and the cultural, societal, and environmental considerations.
- 4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



**5. 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.

**6. 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.

**7. 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.

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

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

**10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with 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.

**11. 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.

**12. 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.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



**Program Specific Outcomes (PSO) addressed by the Course:**

**A graduate of the Computer Engineering Program will demonstrate-**

**PSO1: Domain Specialization:**

Apply domain knowledge to develop computer based solutions for Engineering Applications.

**PSO2: Problem-Solving Skills:**

Find solutions for complex problems by applying problem solving skills and standard practices and strategies in software project development.

**PSO3: Professional Career and Entrepreneurship:**

Incorporate professional, social, ethical, effective communication, and entrepreneurial practices into their holistic development

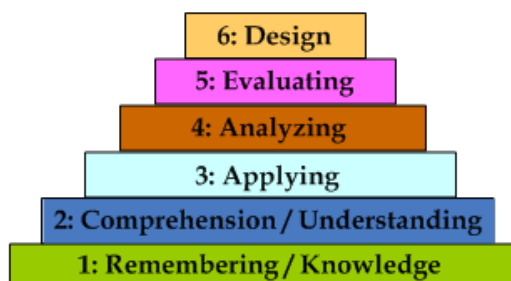


**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
 (Autonomous Institute Affiliated to Savitribai Phule Pune University)



CO	Description
CO1:	Analyse lexical structure of language and Design Lexical analyzer for given language. using tools. BL4
CO2:	Analyse syntactic structure of language and Design syntax analyzer for given language using tools. BL4
CO3:	Analyse semantic structure of language and implement symbol table. BL4
CO4:	Describe techniques for intermediate code and machine code optimisation. BL3

Sub code Subject	CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO 10	PO 11	PO 12	PSO1	PSO2	PSO3
<b>CS 3109</b>  <b>Compiler Design</b>	CO1	3	2	2	1	2	0	0	0	0	1	0	2	3	2	1
	CO2	3	2	2	1	2	0	0	0	0	2	0	2	3	2	1
	CO3	3	2	2	1	2	0	0	0	0	2	0	2	3	2	1
	CO4	3	2	2	1	2	0	0	0	0	2	0	2	3	2	1
	CO5	3	2	2	1	1	0	0	0	0	2	0	2	3	2	1
<b>Average Mapping</b>		<b>3</b>	<b>2</b>	<b>2</b>	<b>1</b>	<b>1.8</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>1.8</b>	<b>0</b>	<b>2</b>	<b>3</b>	<b>2</b>	<b>1</b>



	Blooms Taxonomy
	Remembering
	Understanding
	Applying





**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO.1**

**Count number of lines, tabs, spaces, words, characters from  
a given text file Using LEX tool**



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
 (Autonomous Institute Affiliated to Savitribai Phule Pune University)



## Session Plan

Time ( min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
15	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
15	Concept of Lex tool	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension
----	-------------	----------	--------------------	----------------------------------	---

**TITLE:** Program to count number of lines, tabs, spaces, words, characters from a given text file.

**OBJECTIVES:**

Understand the importance and usage of LEX automated tool.

**PROBLEM STATEMENT:**

Implement a lexical analyzer Program to count number of lines, tabs, spaces, words, characters from a given text file.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as text file in English.

**OUTPUT:** It will count number of lines, tabs, spaces, words, characters from a given text file.

**MATHEMATICAL MODEL:**

Let S be the solution such that

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

s=Start of program

e = the end of program

i=Sample file.

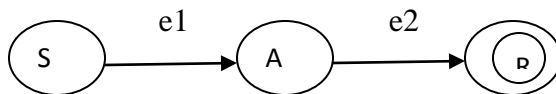
o=Result of statement

Success-token is generated.



Failure-token is not generated or forced exit due to system error.

Computational Model



Where,

S={ Start state }

A={ generate token() }

R={ Final Result }

## **THEORY:**

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex.

The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed

1) LEX Specifications :- Structure of the LEX Program is as follows

-----

Declaration Part



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



-----

%%

-----

Translation Rule

-----

%%

-----

Auxiliary Procedures

-----

2) Declaration part :- Contains the declaration for variables required for LEX program and C program.

Translation rules:- Contains the rules like

Reg. Expression            { action1 }

Reg. Expression            { action2 }

Reg. Expression            { action3 }

-----

Reg. Expression            { action-n }

3) Auxiliary Procedures :-

Contains all the procedures used in your C – Code.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



- Built-in Functions i.e. `yylex()` , `yyerror()` , `yywrap()` etc.
  - 1) `yylex()` :- This function is used for calling lexical analyzer for given translation rules.
  - 2) `yyerror()` :- This function is used for displaying any error message.
  - 3) `yywrap()` :- This function is used for taking i/p from more than one file.
- Built-in Variables i.e. `yylval`, `yytext`, `yyin`, `yyout` etc.
  - 1) `yylval` :- This is a global variable used to store the value of any token.
  - 2) `yytext` :- This is global variable which stores current token.
  - 3) `yyin` :- This is input file pointer used to change value of input file pointer. Default file pointer is pointing to `stdin` i.e. keyboard.
  - 4) `yyout` :- This is output file pointer used to change value of output file pointer. Default output file pointer is pointing to `stdout` i.e. Monitor.
- How to execute LEX program :- For executing LEX program follow the following steps
- Compile \*.l file with `lex` command

```
# lex *.l
```

It will generate `lex.yy.c` file for your lexical analyzer.
- Compile `lex.yy.c` file with `cc` command

```
# cc lex.yy.c
```

It will generate object file with name `a.out`.
- Execute the \*.out file to see the output

```
# ./a.out
```

## CONCLUSION

Thus we have studied Lex Tool syntax and applied to count tokens from given input file.

## OUTCOME

Upon completion Students will be able to:



1. Explain Lex tool

2. Explain lexical analysis

## QUESTIONS

1. What is compiler?

Ans: Compiler is a program which takes one language as input and translates into an equivalent another Language

2. What is token?

Ans: Token describes the class or category of input string. Eg identifier, constants are called tokens.

3. Define lexemes.

Ans: It is a sequence of character in source program that are matched with the pattern of token.

4. What is lex?

Ans: Lex is a tool for generating lexical analyzer.

5. What is lex.yy.c file?

Ans: The lex specification file contains the regular expression for tokens and lex.yy.c is a program that consist the tabular expression of the transition diagram constructed for regular expression of specification file.

6. What is the meaning of yytext?

Ans: When lexer matches or recognizes the token form input then the lexeme is stored in a null terminated string called yytext.

7. What is yylex() fuction?



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)




Ans: As soon as call to yylex() is encounter scanner starts scanning the source program.

8. whether lexical analyzer detects any error?

Ans: Yes. Following errors can be following errors

1. Spelling mistakes. Hence get incorrect tokens.
2. Exceeding length of identifier or numeric constants.
3. Appearance of illegal character.

9. Explain the meaning of [], "", ?, | symbols. 

Ans: 1. [] – A character class which matches any character within the bracket.

2. "" – String written in quotes matches literally.

3. ? – Matches zero or more occurrences of preceding regular expression.

4. | - To represent the OR.





**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO.2**

### **Lexical Analyzer for Sample Language Using LEX**



**TITLE:** Lexical analyzer for a sample language using LEX

**OBJECTIVES:**

Understand the importance and usage of LEX automated tool.

Constraints: The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value.

**PROBLEM STATEMENT:**

Implement a lexical analyzer for a sample language using LEX Implementation.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as Sample program in given language.

**OUTPUT:** It will generate tokens for Sample program in given language.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of the class such that

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

s=Start of program

e = the end of program

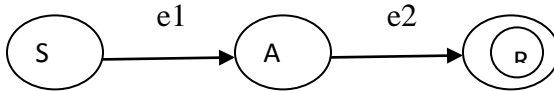
i=Sample language statement.

o=Result of statement

Success-token is generated.

Failure-token is not generated or forced exit due to system error.

Computational Model



Where,

S={Start state}

A={generate token() }

R={Final Result}

### **THEORY:**

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex.

The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed

1) LEX Specifications :- Structure of the LEX Program is as follows

```
-----  
  
Declaration Part  
  
-----  
  
%%  
  
-----
```



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



Translation Rule

-----

%%

-----

Auxiliary Procedures

-----

2) Declaration part :- Contains the declaration for variables required for LEX program and C program.

Translation rules:- Contains the rules like

Reg. Expression            { action1 }

Reg. Expression            { action2 }

Reg. Expression            { action3 }

-----

Reg. Expression            { action-n }

3) Auxiliary Procedures :-

Contains all the procedures used in your C – Code.

- Built-in Functions i.e. `yylex()` , `yyerror()` , `yywrap()` etc.
  - 1) `yylex()` :- This function is used for calling lexical analyzer for given translation rules.
  - 2) `yyerror()` :- This function is used for displaying any error message.
  - 3) `yywrap()` :- This function is used for taking i/p from more than one file.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



- Built-in Variables i.e. yylval, yytext, yyin, yyout etc.
  - 1) yylval :- This is a global variable used to store the value of any token.
  - 2) yytext :- This is global variable which stores current token.
  - 3) yyin :- This is input file pointer used to change value of input file pointer. Default file pointer is pointing to stdin i.e. keyboard.
  - 4) yyout :- This is output file pointer used to change value of output file pointer. Default output file pointer is pointing to stdout i.e. Monitor.
- How to execute LEX program :- For executing LEX program follow the following steps
- Compile \*.l file with lex command

```
# lex *.l
```

It will generate lex.yy.c file for your lexical analyzer.
- Compile lex.yy.c file with cc command

```
# cc lex.yy.c
```

It will generate object file with name a.out.
- Execute the \*.out file to see the output



```
# ./a.out
```

## CONCLUSION

Thus we have studied Lexical Analyzer for sample language.

## OUTCOME

**Upon completion Students will be able to:**

- 1.Explain Compiler concept 
2. Explain lexical analysis 

## QUESTIONS


9. What is compiler? 



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)




Ans: Compiler is a program which takes one language as input and translates into an equivalent another Language

10. What is token? 

Ans: Token describes the class or category of input string. Eg identifier, constants are called tokens.

11. Define lexemes.


Ans: It is a sequence of character in source program that are matched with the pattern of token.

12. What is lex? 

Ans: Lex is a tool for generating lexical analyzer.

13. What is lex.yy.c file? 

Ans: The lex specification file contains the regular expression for tokens and lex.yy.c is a program that consist the tabular expression of the transition diagram constructed for regular expression of specification file.

14. What is the meaning of yytext? 

Ans: When lexer matches or recognizes the token form input then the lexeme is stored in a null terminated string called yytext.

15. What is yylex() fuction? 

Ans: As soon as call to yylex() is encounter scanner starts scanning the source program.

16. whether lexical analyzer detects any error?

Ans: Yes. Following errors can be following errors

1.Spelling mistakes.Hence get incorrect tokens.


2.Exceeding length of identifier or numeric constants.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



3. Appearance of illegal character.

9. Explain the meaning of [], "", ?, | symbols. 

- Ans:
1. [] – A character class which matches any character within the bracket.
  2. "" – String written in quotes matches literally.
  3. ? – Matches zero or more occurrences of preceding regular expression.
  4. | – To represent the OR.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO.3**

**Program for syntax checking of subset of given language using  
LEX and YACC.**





## Session Plan

Time ( min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
15	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
15	Concept of YACC	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates  Monitors	Participates, Discusses	Comprehension,  Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension



**TITLE:** Program for syntax checking of subset of given language using LEX and YACC.

**OBJECTIVES:**

1. To understand Second phase of compiler: Syntax Analysis.
2. To learn and use compiler writing tools.
3. Understand the importance and usage of YACC automated tool.

**PROBLEM STATEMENT:**

Write a program for syntax checking of subset of given language using LEX and YACC.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as Sample language statement.

**OUTPUT:** It will generate parser for sample language.

**MATHEMATICAL MODEL:**

Let S be the solution such that

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

s=Start of program

e = the end of program

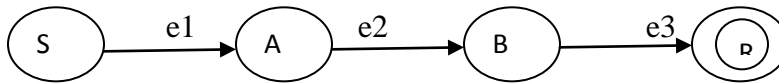
i=Arithmetic expression.

o=Result of arithmetic expression

Success-parser is generated.

Failure-parser is not generated or forced exit due to system error.

Computational Model



Where,

S={Start state}

A={Generate token()}

B={Parse\_token()}

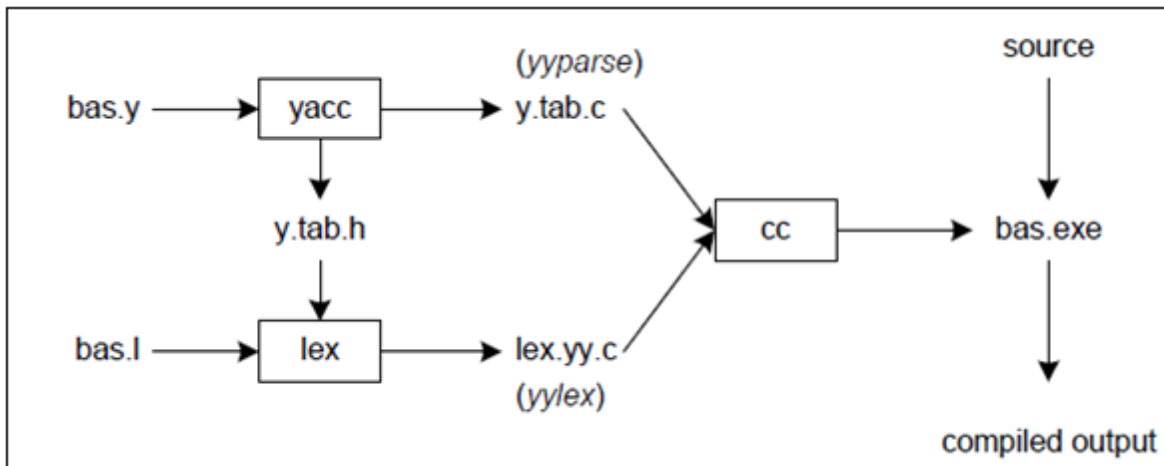
R={Final Result}

## **THEORY:**

- 1) YACC Specifications: - Parser generator facilitates the construction of the front end of a compiler. YACC is LALR parser generator. It is used to implement hundreds of compilers. YACC is command (utility) of the UNIX system. YACC stands for “Yet Another Compiler Compiler”. File in which parser generated is with .y extension. e.g. parser.y, which is containing YACC specification of the translator. After complete specification UNIX command. YACC transforms parser.y into a C program called y.tab.c using LR parser. The program y.tab.c is automatically generated. We can use command with -d option as yacc -d parser.y By using -d option two files will get generated namely y.tab.c and y.tab.h. The header file y.tab.h will store all the token information and so you need not have to create y.tab.h explicitly. The program y.tab.c is a representation of an LALR parser written in C, along with other C routines that the user may have prepared. By compiling y.tab.c with the ly library that contains the LR parsing program using the command. cc ytabc - ly . we obtain the desired object program a.out that perform the translation specified by the original program. If procedure is needed, they can be compiled or loaded with y.tab.c, just as with any C program. LEX recognizes regular expressions, whereas YACC recognizes entire grammar. LEX divides the input stream



into tokens, while YACC uses these tokens and groups them together logically. LEX and YACC work together to analyze the program syntactically. The YACC can report conflicts or ambiguities (if at all) in the form of error messages.



Structure of the YACC Program is as follows

```
-----
Declaration Section
-----
%%
-----

Translation Rule Section
-----
%%
-----

Auxiliary Procedures Section
-----
```



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



**Declaration Section :-**

The definition section can include a literal block, C code copied verbatim to the beginning of the generated C file, usually containing declaration and #include lines. There may be %union, %start, %token, %type, %left, %right, and %nonassoc declarations. (See "%union Declaration," "Start Declaration," "Tokens," "%type Declarations," and "Precedence and Operator Declarations.") It can also contain comments in the usual C format, surrounded by "/\*" and "\*/". All of these are optional, so in a very simple parser the definition section may be completely empty.

**Translation rule Section :-**

Contains the rules / grammars

Production	{ action1 }
Production	{ action2 }
Production	{ action3 }
-----	
Production	{ action-n }

**Auxiliary Procedure Section :-**

Contains all the procedures used in your C – Code.

**2) Built-in Functions i.e. yyparse() , yyerror() , yywrap() etc.**

**1) yyparse() :-**

This is a standard parse routine used for calling syntax analyzer for given translation rules.

**2) yyerror() :-**

This is a standard error routine used for displaying any error message.

**3) yywrap() :-**



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



This function is used for taking i/p from more than one file.

3) Built-in Types i.e. %token , %start , %prec , %nonassoc etc.

1) %token

Used to declare the tokens used in the grammar. The tokens that are declared in the declaration section will be identified by the parser.

Eg. :- %token NAME NUMBER

2) %start :-

Used to declare the start symbol of the grammar.

Eg.:- %start STMT

3) %left

Used to assign the associativity to operators.

Eg.:- %left '+' '-' - Assign left associatively to + & – with lowest precedence.

%left '\*' '/' - Assign left associatively to \* & / with highest precedence.

4) %right :-

Used to assign the associativity to operators.

Eg.:- 1) %right '+' '-'

- Assign right associatively to + & – with lowest precedence

2) %right '\*' '/'

- Assign right left associatively to \* & / with highest precedence.

5) %nonassoc :-



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



Used to un associate.

Eg.:- %nonassoc UMINUS

6) %prec :-

Used to tell parser use the precedence of given code.

Eg. :- %prec UMINUS

7) %type :-

Used to define the type of a token or a non-terminal of the production written in the rules section of the .Y file.

Eg.:- %type <name of any variable> exp

Let us see both LEX and YACC specification for writing a program for calculator.

LEX Specification : (lex.l file)

1) Declaration Section :

```
%{  
#include "y.tab.h"  
#include<math.h>  
extern int yylval;  
%}
```

Here, we include the header file that is generated while executing the .y file. We also include math.h as we will be using a function atoi (that type casts string to integer).

Lastly, when a lexical analyzer passes a token to the parser, it can also pass a value for the token. In order to pass the value that our parser can use (for the passed token), the lexical analyser has to store it in the variable yylval.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



Before storing the value in `yyval` we have to specify its data type. In our program we want to perform mathematical operations on the input, hence we declare the variable `yyval` as integer.

1) Rules Section :

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

```
[ \t] ; /* ignore white space */
```

```
\n return 0; /* logical EOF */
```

```
. return yytext[0];
```

In rules section, we match the pattern for numbers and pass the token `NUMBER` to the parser. As we know the matched string is stored in `yytext` which is a string, hence we type cast the string value to integer. We ignore spaces, and for all other input characters we just pass them to the parser.

YACC Specification : (yacc.y file)

1) Declaration Section:

```
%token NUMBER
```

```
%left '+' '-'
```

```
%left '/' '*'
```

```
%right '^'
```

```
%nonassoc UMINUS
```

In declaration section we declare all the variables that we will be using through out the program, also we include all the necessary files.

Apart from that, we also declare tokens that are recognized by the parser. As we are writing a parser specification for calculator, we have only one token that is `NUMBER`.

To deal with ambiguous grammar, we have to specify the associativity and precedence of the operators. As seen above `+`, `-`, `*` and `/` are left associative whereas the Unary minus and power symbol are non-associative.







## QUESTIONS

1. What is a parser?

**Ans:** Parse generator can be used to facilitate the construction of the front end of a compiler

2. Explain about yacc parser generator?

**Ans:** Yacc is a parser generator that is a program for converting a grammatical specification of a language like the one above into a parser that will parse statements in the language.

3. What is the difference between yylex() and scanf().

**Ans.** yylex() is used to accept input and call parser, but scanf() for only accepting data.

4. is Yacc a compiler!

**Ans:** No, Yacc is available as a command on the unix system and has been used to implement of hundred of compiler.

5. Explain construction of yacc prg.

**Ans:** a prg containing yacc specification is provided to yacc. This provides y.tab.c as a c prg. This is compiled using c compiler to get exe ie a.out.

6. What are different sections of yacc

**Ans:**Declarations, translation rule and support & routine sections.

7. Explain the grammar of yacc

**Ans:** <left side> : <alternate1> {semantic action1 }

| alternate2> {semantic action2 }

8. what is the difference between Lex and Yacc







**Ans:** lex prg to lex.yy.c to scanner to parser to exe

yac prg to y.tab.c to parser to exe



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



9. What is Yacc? What is the input to it? What is its output? 
10. What are regular expressions? 
11. What is a grammar? 
12. Explain this regular expression  $[\wedge \backslash t \backslash n]^+$  
13. What are token definitions? 
14. What does y.tab.h do? 

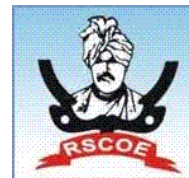


**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO.4**

**Program for syntax checking of control statements using  
LEX and YACC.**



## Session Plan

Time ( min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
15	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
15	Concept of YACC	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates  Monitors	Participates, Discusses	Comprehension,  Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



**TITLE:** Program for syntax checking of syntax checking of control statements using LEX and YACC.

**OBJECTIVES:**

1. To understand Second phase of compiler: Syntax Analysis.
2. To learn and use compiler writing tools.
3. Understand the importance and usage of LEX and YACC automated tool.

**PROBLEM STATEMENT:**

Write a program for syntax checking of control statements using LEX and YACC.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as control statements.

**OUTPUT:** It will generate parser for sample language.

**MATHEMATICAL MODEL:**

Let S be the solution such that

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

s=Start of program

e = the end of program

i=Arithmetic expression.

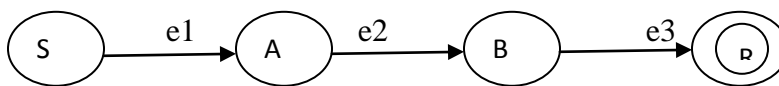


o=Result of arithmetic expression

Success-parser is generated.

Failure-parser is not generated or forced exit due to system error.

Computational Model



Where,

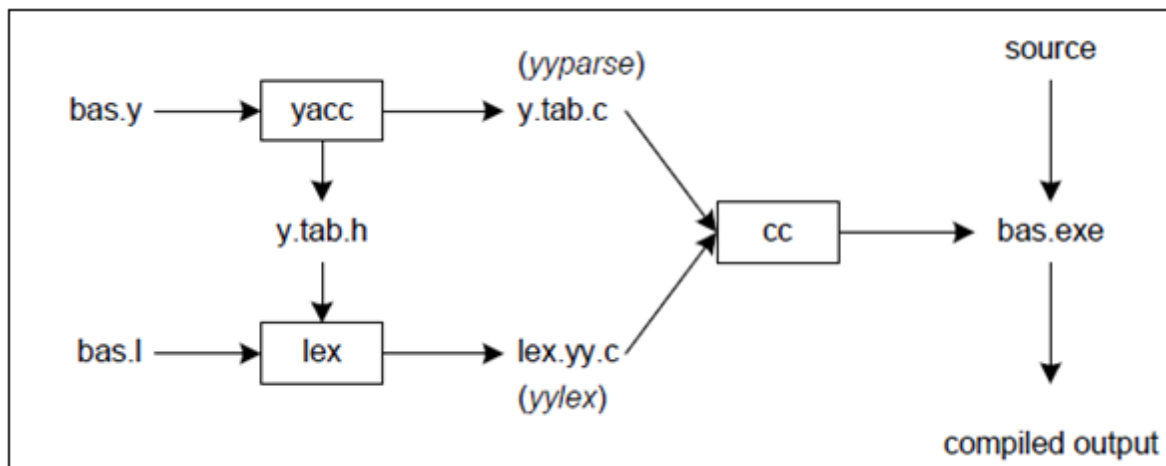
S={Start state}

A={Generate token() }

B={Parse\_token() }

R={Final Result}

## **THEORY:**



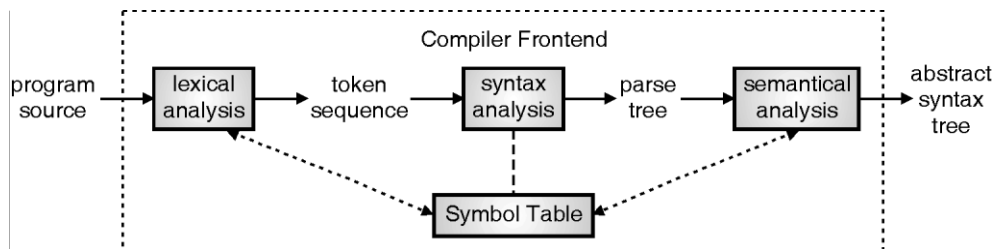
**The Role of the Parser :**



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.a grammar can be directly converted into a parser by some tools.



**Position of parser**

- Parser works on a stream of tokens.
- The smallest item is a token.
- We categorize the parsers into two groups:
  1. **Top-down parser :**

The parse tree is created top to bottom, starting from the root.
  2. **Bottom-up parser :**

The parse is created bottom to top; starting from the leaves.
- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing.
  - LR for bottom-up parsing.





**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



**Conditional Statements :**

$S \rightarrow \text{if } E \text{ then } S_1$

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

$S \rightarrow \text{While } E \text{ do } S_1$

- We consider Boolean Expressions with the following grammar:

$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id rel op id} \mid \text{true} \mid \text{false}$

$S \rightarrow \text{id} := E$

$E \rightarrow E + E$

$\mid E * E$

$\mid - E$

$\mid ( E )$

$\mid \text{id}$

**CONCLUSION:**

Thus we have studied Parser for sample language using YACC.

**OUTCOME**

**Upon completion Students will be able to:**

1. Explain different parsing technique
2. What is YACC

**QUESTIONS**

1. What is a parser?

**Ans:** Parse generator can be used to facilitate the construction of the front end of a compiler

2. Explain about yacc parser generator?

**Ans:** Yacc is a parser generator that is a program for converting a grammatical specification of a language like the one above into a parser that will parse statements in the language.



3. What is the difference between yylex() and scanf().

**Ans.** yylex() is used to accept input and call parser, but scanf() for only accepting data.

4. is Yacc a compiler!

**Ans:** No, Yacc is available as a command on the unix system and has been used to implement of hundred of compiler.

5. Explain construction of yacc prg.

**Ans:** a prg containing yacc specification is provided to yacc. This provides y.tab.c as a c prg. This is compiled using c compiler to get exe ie a.out.

6. What are different sections of yacc

**Ans:**Declarations, translation rule and support & routine sections.

7. Explain the grammar of yacc

**Ans:** <left side> : <alternate1> {semantic action1 }

| alternate2> {semantic action2 }

8. what is the difference between Lex and Yacc

**Ans:** lex prg to lex.yy.c to scanner to parser to exe

yac prg to y.tab.c to parser to exe

9. What is Yacc? What is the input to it? What is its output?

10. What are regular expressions?

11. What is a grammar?

12. Explain this regular expression  $[\backslash t\backslash n]^+$

13. What are token definitions?

14. What does y.tab.h do?



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO.5**

**Program for Syntax checking of declaration statement  
using LEX and YACC.**



## Session Plan

Time ( min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
15	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
15	Concept of YACC	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension



**TITLE:** Program for syntax checking of declaration statements using LEX and YACC.

### **OBJECTIVES:**

1. To understand Second phase of compiler: Syntax Analysis.
2. To learn and use compiler writing tools.
3. Understand the importance and usage of LEX and YACC automated tool.

### **PROBLEM STATEMENT:**

Write a program for syntax checking of declaration statements using LEX and YACC.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as control statements.

**OUTPUT:** It will generate parser for sample language.

### **MATHEMATICAL MODEL:**

Let S be the solution such that

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

s=Start of program

e = the end of program

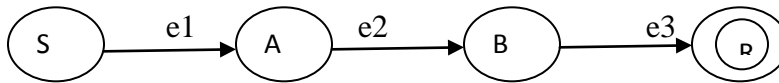
i=Arithmetic expression.

o=Result of arithmetic expression

Success-parser is generated.

Failure-parser is not generated or forced exit due to system error.

Computational Model



Where,

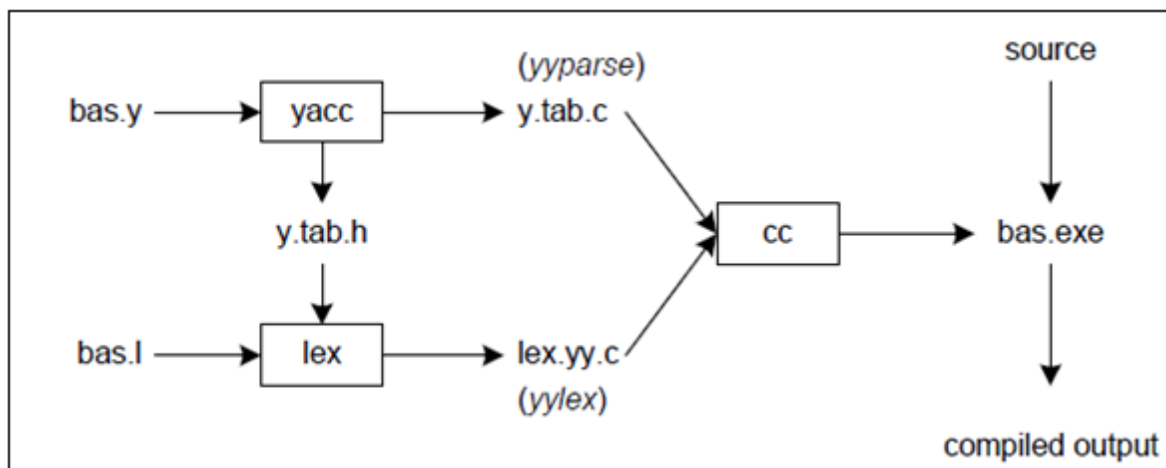
S={ Start state }

A={ Genrate token() }

B={ Parse\_token() }

R={ Final Result }

## THEORY:

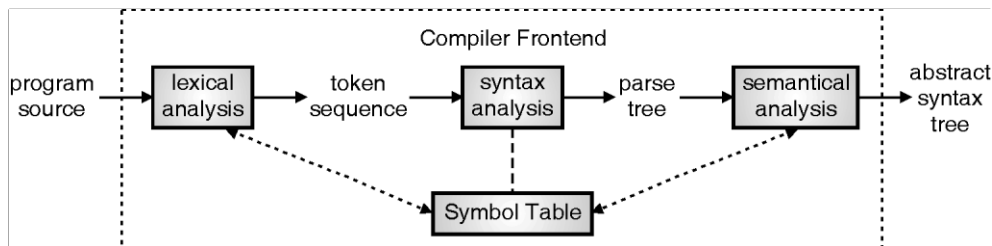


### The Role of the Parser :

- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as **parser**.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.



- If it satisfies, the parser creates the parse tree of that program.
- Otherwise the parser gives the error messages.
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools.



**Position of parser**

- Parser works on a stream of tokens.
- The smallest item is a token.
- We categorize the parsers into two groups:
  1. **Top-down parser :**

The parse tree is created top to bottom, starting from the root.
  2. **Bottom-up parser :**

The parse is created bottom to top; starting from the leaves.
- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing.
  - LR for bottom-up parsing.

#### **Declaration Statements Grammar :**

Sample Grammar for integer statements is given below. Same way grammar for other type of declaration statements can be written.

$S \rightarrow S \text{ DL} \mid \text{DL}$

$\text{DL} \rightarrow \text{INTV} ;$

$\text{INTV} \rightarrow \text{INT IV}$

$\text{IV} \rightarrow \text{I} \mid \text{IV}, \text{id} \mid \text{IV}, \text{id} = \text{INUM} \mid \text{id} = \text{INUM}$



I->id

## CONCLUSION:

Thus we have studied Parser for sample language using YACC.

## OUTCOME

Upon completion Students will be able to:

1. Explain different parsing technique
2. What is YACC

## QUESTIONS

1. What is a parser?

**Ans:** Parse generator can be used to facilitate the construction of the front end of a compiler

2. Explain about yacc parser generator?

**Ans:** Yacc is a parser generator that is a program for converting a grammatical specification of a language like the one above into a parser that will parse statements in the language.

3. What is the difference between yylex() and scanf().

**Ans.** yylex() is used to accept input and call parser, but scanf() for only accepting data.

4. is Yacc a compiler!

**Ans:** No, Yacc is available as a command on the unix system and has been used to implement of hundred of compiler.

5. Explain construction of yacc prg.

**Ans:** a prg containing yacc specification is provided to yacc. This provides y.tab.c as a c prg. This is compiled using c compiler to get exe ie a.out.





**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



6. What are different sections of yacc

**Ans:** Declarations, translation rule and support & routine sections.

7. Explain the grammar of yacc

**Ans:** <left side> : <alternate1> {semantic action1}  
| alternate2> {semantic action2}

8. what is the difference between Lex and Yacc

**Ans:** lex prg to lex.yy.c to scanner to parser to exe

yac prg to y.tab.c to parser to exe

9. What is Yacc? What is the input to it? What is its output?

10. What are regular expressions?

11. What is a grammar?

12. Explain this regular expression  $[^ \backslash t \backslash n]^+$

13. What are token definitions?

14. What does y.tab.h do?



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO.6**

**Program for a desk calculator using LEX and YACC.**



## Session Plan

Time ( min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
15	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
15	Concept of YACC	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



**TITLE:** Program for desk calculator using LEX and YACC.

**OBJECTIVES:**

4. To understand Second phase of compiler: Syntax Analysis.
5. To learn and use compiler writing tools.
6. Understand the importance and usage of LEX and YACC automated tool.

**PROBLEM STATEMENT:**

Write a program for desk calculator using LEX and YACC.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as control statements.

**OUTPUT:** It will generate parser for sample language.

**MATHEMATICAL MODEL:**

Let S be the solution such that

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

s=Start of program

e = the end of program

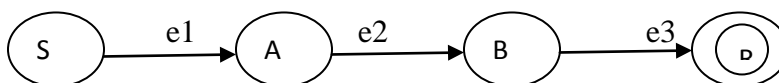
i=Arithmetic expression.

o=Result of arithmetic expression

Success-parser is generated.

Failure-parser is not generated or forced exit due to system error.

Computational Model





Where,

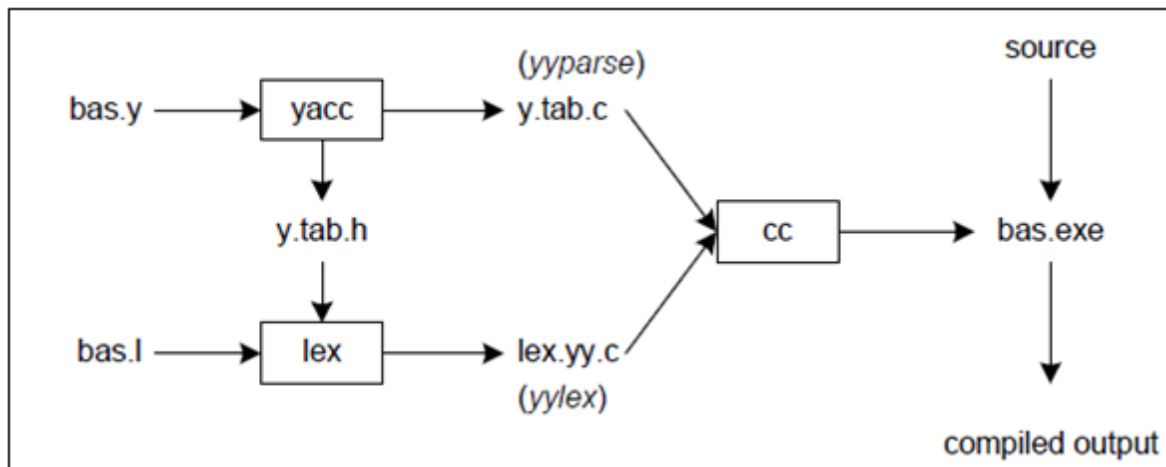
$S = \{\text{Start state}\}$

$A = \{\text{Generate token()}\}$

$B = \{\text{Parse\_token()}\}$

$R = \{\text{Final Result}\}$

## THEORY:

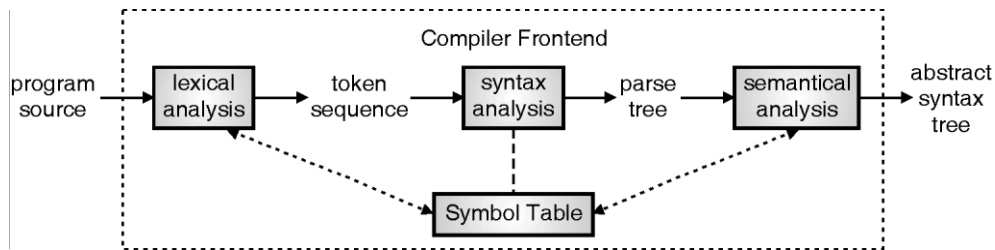


### The Role of the Parser :

- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as ***parser***.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.



- Otherwise the parser gives the error messages.
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools.



**Position of parser**

- Parser works on a stream of tokens.
- The smallest item is a token.
- We categorize the parsers into two groups:
  1. **Top-down parser :**

The parse tree is created top to bottom, starting from the root.
  2. **Bottom-up parser :**

The parse is created bottom to top; starting from the leaves.
- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing.
  - LR for bottom-up parsing.

#### **Syntax-Directed Definition for Desk Calculator:**

<b>Production</b>	<b>Semantic Rules</b>
$L \rightarrow E \text{ return}$	$\text{print}(E.\text{val})$
$E \rightarrow E1 + T$	$E.\text{val} = E1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T1 * F$	$T.\text{val} = T1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow ( E )$	$F.\text{val} = E.\text{val}$



$F \rightarrow \text{digit} \quad F.val = \text{digit}.lexval$

Symbols E, T, and F are associated with a synthesized attribute *val*.

The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

## CONCLUSION:

Thus we have studied Parser for sample language using YACC.

## OUTCOME

Upon completion Students will be able to:

1. Explain different parsing technique
2. What is YACC

## QUESTIONS

1. What is a parser?

**Ans:** Parse generator can be used to facilitate the construction of the front end of a compiler

2. Explain about yacc parser generator?

**Ans:** Yacc is a parser generator that is a program for converting a grammatical specification of a language like the one above into a parser that will parse statements in the language.

3. What is the difference between yylex() and scanf().

**Ans.** yylex() is used to accept input and call parser, but scanf() for only accepting data.

4. is Yacc a compiler!

**Ans:** No, Yacc is available as a command on the unix system and has been used to implement of hundred of compiler.



5. Explain construction of yacc prg.

**Ans:** a prg containing yacc specification is provided to yacc. This provides y.tab.c as a c prg. This is compiled using c compiler to get exe ie a.out.

6. What are different sections of yacc

**Ans:**Declarations, translation rule and support & routine sections.

7. Explain the grammar of yacc

**Ans:** <left side> : <alternate1> {semantic action1}  
| alternate2> {semantic action2}

8. what is the difference between Lex and Yacc

**Ans:** lex prg to lex.yy.c to scanner to parser to exe

yac prg to y.tab.c to parser to exe

9. What is Yacc? What is the input to it? What is its output?

10. What are regular expressions?

11. What is a grammar?

12. Explain this regular expression [<sup>^</sup>t\|n]<sup>+</sup>

13. What are token definitions?

14. What does y.tab.h do?





**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO.7**

**Program to generate Intermediate Code Generation  
using LEX and YACC.**



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
 (Autonomous Institute Affiliated to Savitribai Phule Pune University)



## Session Plan

Time ( min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
15	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
15	Concept of Intermediate Language	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension



**TITLE:** Int. code generation for sample language using LEX and YACC.

**OBJECTIVES:**

1. To understand fourth phase of compiler: Intermediate code generation.
2. To learn and use compiler writing tools.
3. To learn how to write three address code for given statement.

**PROBLEM STATEMENT:**

Write an attributed translation grammar to recognize declarations of simple variables, "for", assignment, if, if-else statements as per syntax of C and generate equivalent three address code for the given input made up of constructs mentioned above using LEX and YACC .

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC

**INPUT:** Input data as Sample language.

**OUTPUT:** It will generate Intermediate language for sample language.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of the class Weather Report such that

$S = \{s, e, i, o, f, \text{success}, \text{failure}\}$

s=initial state of grammar

e = the end state of grammar.

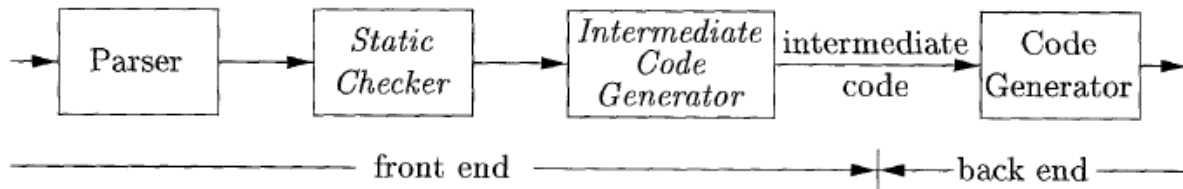
i=Sample language Statement.

o=Intermediate code for language statement

Success-Intermediate code is is generated.

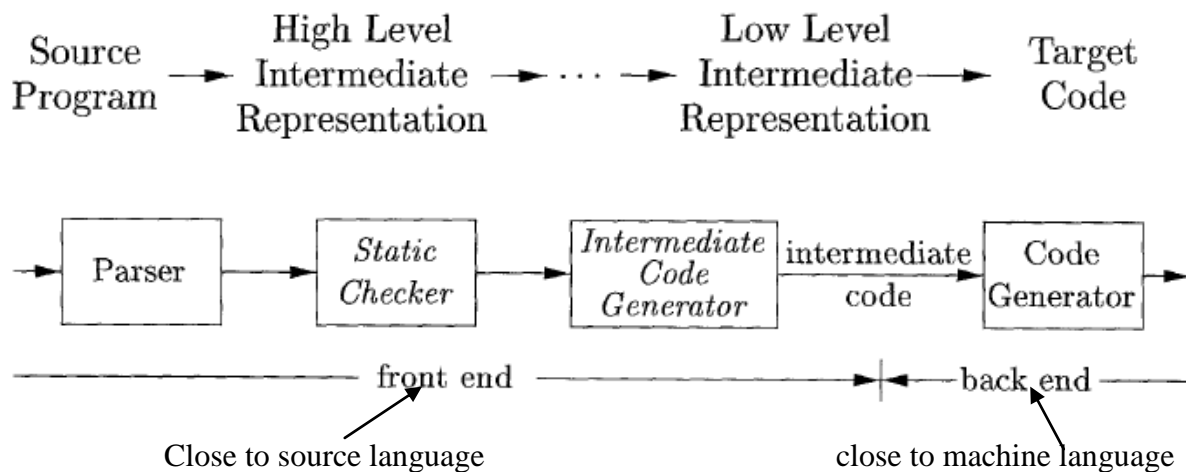
Failure-Intermediate code is not generated or forced exit due to system error.

Mathematical model for above system.



### THEORY:

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates retargeting: enables attaching a back end for the new machine to an existing front end.



A compiler front end is organized, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

### Static Checking

This includes type checking which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing like

- flow-of-control checks
  - Ex: Break statement within a loop construct
- Uniqueness checks



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



– Labels in case statements

- Name-related checks

#### Intermediate Representations

We could translate the source program directly into the target language. However, there are benefits to having an intermediate, machine-independent representation.

- A clear distinction between the machine-independent and machine-dependent parts of the compiler
- Retargeting is facilitated; the implementation of language processors for new machines will require replacing only the back-end
- We could apply machine independent code optimisation techniques

Intermediate representations span the gap between the source and target languages.

- High Level Representations

- closer to the source language
- easy to generate from an input program
- code optimizations may not be straightforward

- Low Level Representations

- closer to the target machine
- Suitable for register allocation and instruction selection

#### Intermediate Languages

Three ways of intermediate code representation:

Syntax tree

Postfix notation

Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

#### Graphical Representations

Syntax tree



A syntax tree depicts the natural hierarchical structure of a source program. A dag (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement

$a := b * -c + b * -c$

are as follows:

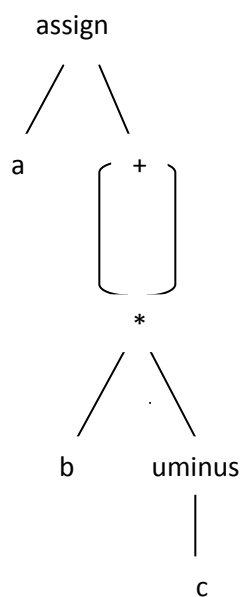
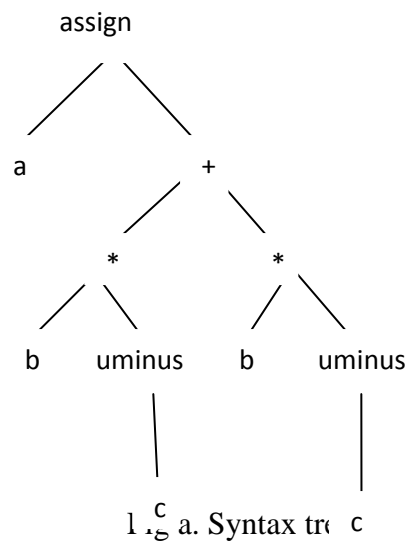




Fig b. DAG

#### Postfix notation

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is a b c uminus \* b c uminus \* + assign

#### Three-Address Code

Three-address code is a sequence of statements of the general form  $x := y \text{ op } z$  where  $x$ ,  $y$  and  $z$  are names, constants, or compiler-generated temporaries;  $\text{op}$  stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean valued data. Thus a source language expression like  $x + y * z$  might be translated into a sequence

$t1 := y * z$

$t2 := x + t1$

where  $t1$  and  $t2$  are compiler-generated temporary names. The reason for the term “three-address code” is that each statement usually contains three addresses, two for the operands and one for the result.

#### Advantages of three-address code

The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.

The use of names for the intermediate values computed by a program allows three address code to be easily rearranged – unlike postfix notation. Three-address code is a liberalized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three address statements.

#### Types of Three-Address Statements

The common three-address statements are:



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



Assignment statements of the form  $x := y \text{ op } z$ , where  $\text{op}$  is a binary arithmetic or logical operation.

Assignment instructions of the form  $x := \text{op } y$ , where  $\text{op}$  is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number. Copy statements of the form  $x := y$  where the value of  $y$  is assigned to  $x$ . The unconditional jump  $\text{goto } L$ . The three-address statement with label  $L$  is the next to be executed.

Conditional jumps such as  $\text{if } x \text{ relop } y \text{ goto } L$ . This instruction applies a relational operator ( $<$ ,  $=$ ,  $\geq$ , etc.) to  $x$  and  $y$ , and executes the statement with label  $L$  next if  $x$  stands in relation  $\text{relop}$  to  $y$ . If not, the three-address statement following  $\text{if } x \text{ relop } y \text{ goto } L$  is executed next, as in the usual sequence.

$\text{param } x$  and  $\text{call } p, n$  for procedure calls and  $\text{return } y$ , where  $y$  representing a returned value is optional. For example,

$\text{param } x_1$

$\text{param } x_2$

.....

$\text{param } x_n$

$\text{call } p, n$  generated as part of a call of the procedure  $p(x_1, x_2, \dots, x_n)$ .

Indexed assignments of the form  $x := y[i]$  and  $x[i] := y$ .

Address and pointer assignments of the form  $x := \&y$ ,  $x := *y$ , and  $*x := y$ .

**Implementation of Three-Address Statements:** A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are: Quadruples, Triples, Indirect triples.

#### A. Quadruples

A quadruple is a record structure with four fields, which are,  $\text{op}$ ,  $\text{arg1}$ ,  $\text{arg2}$  and  $\text{result}$ . The  $\text{op}$  field contains an internal code for the operator. The 3 address statement  $x = y \text{ op } z$  is represented





**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



by placing y in arg1, z in arg2 and x in result. The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created. Fig a) shows quadruples for the assignment  $a : b * c + b * c$

**B. Triples:**

To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it. If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2. The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure ( for temporary values ). Since three fields are used, this intermediate code format is known as triples. Fig b) shows the triples for the assignment statement  $a : = b * c + b * c$ .

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t2	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Fig. a. Quadraples

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Fig.b. Triples



### Quadruples & Triple representation of three-address statement

#### C. Indirect triples:

Indirect triple representation is the listing pointers to triples rather-than listing the triples themselves. Let us use an array statement to list pointers to triples in the desired order. Fig c) shows the indirect triple representation.

Statement	op		
(1)	(15)		
(2)	(16)		
(3)	(17)		
(4)	(18)		
(5)	(19)		

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Fig c): Indirect triples representation of three address statements

Steps to execute the program

```
$ lex filename.l (eg: comp.l)
```

```
$ yacc -d filename.y (eg: comp.y)
```

```
$cc lex.yy.c y.tab.c -ll -ly -lm
```

```
$/a .out
```

#### ALGORITHM:

Write a LEX and YACC program to generate Intermediate Code for arithmetic expression

LEX program



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



1. Declaration of header files specially y.tab.h which contains declaration for Letter, Digit, expr.
2. End declaration section by %%
3. Match regular expression.
4. If match found then convert it into char and store it in yylval.p where p is pointer declared in YACC
5. Return token
6. If input contains new line character (\n) then return 0
8. If input contains „.“ then return yytext[0]
9. End rule-action section by %%
10. Declare main function
  - a. open file given at command line
  - b. if any error occurs then print error and exit
  - c. assign file pointer fp to yyin
  - d. call function yylex until file ends
11. End

YACC program

1. Declaration of header files
2. Declare structure for three address code representation having fields of argument1, argument2, operator, result.
3. Declare pointer of char type in union.
4. Declare token expr of type pointer p.
5. Give precedence to „\*,/“.
6. Give precedence to „+,-“.
7. End of declaration section by %%.
8. If final expression evaluates then add it to the table of three address code.
9. If input type is expression of the form.
  - a. exp+exp then add to table the argument1, argument2, operator.
  - b. exp-exp then add to table the argument1, argument2, operator.



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



- c. exp\*exp then add to table the argument1, argument2, operator.
- d.exp/exp then add to table the argument1, argument2, operator.
- e. „(,exp) then assign \$2 to \$\$.
- f. Digit OR Letter then assigns \$1 to \$\$.
- 10. End the section by % %.
- 11. Declare file \*yyin externally.
- 12. Declare main function and call yyparse function untill yyin ends
- 13. Declare yyerror for if any error occurs.
- 14. Declare char pointer s to print error.
- 15. Print error message.
- 16. End of the program.

In short:

Addtotable function

It will add the argument1, argument2, operator and temporary variable to the structure array of threeaddresscode.

Threeaddresscode function

It will print the values from the structure in the form first temporary variable, argument1, operator, argument2

Quadruple Function

It will print the values from the structure in the form first operator, argument1, argument2, result field

Triple Function

It will print the values from the structure in the form first argument1, argument2, and operator.

The temporary variables in this form are integer / index instead of variables.

**CONCLUSION:**



Hence, we have successfully studied concept of Intermediate code generation of sample language

## OUTCOME

Upon completion Students will be able to:

1. Explain Intermediate code generation
2. Explain DAG

## QUESTIONS:

1. What is intermediate code generation?
2. Explain the different forms of ICG?
3. What are the difference between syntax tree and DAG?
4. What are advantages of 3-address code?
5. Which representation of 3-address code is better than other and why? Justify.
6. What is role of Intermediate code in compiler?
7. Explain quadruple, triples and indirect tuples representation for  $a : b * c + b * c$
8. What are the advantages of three-address code?
9. Compare different types of Three-Address Statements?



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO. 8**

### **Program for Code Optimization**



## Session Plan

Time ( min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
10	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
15	Explanation of Problem statement	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
20	Concept of Code Optimization, DAG	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension



**TITLE:** Program for Code optimization.

**OBJECTIVES:**

1. To express concept of code optimization.
2. To implement code optimization.

**PROBLEM STATEMENT:**

Implement code optimization.

**SOFTWARE REQUIRED:** Linux Operating Systems, GCC.

**INPUT:** Input data as intermediate code.

**OUTPUT:** It will create a optimized code or optimized intermediate code.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of optimized code

$S = \{s, e, i, o, f, DD, NDD, \text{success}, \text{failure}\}$

s=initial state that is intermediate or un-optimized code

e = the end state or optimized code.

i= input of the system.

o=output of the system.

DD-deterministic data it helps identifying the load store functions or assignment functions.

NDD- Non deterministic data of the system S to be solved.

**THEORY:**





**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects. In the code optimization phase the intermediate code is improved to run the output faster and occupies the lesser space.

Output of this phase is another intermediate code to improve the efficiency. The basic requirement of optimization methods should comply with is that an optimized program must have the same output and side effects as its non-optimized version. This requirement, however, may be ignored in case the benefit from optimization is estimated to be more important than probable consequences of a change in the program behavior.

Optimization can be performed by automatic optimizers or programmers. An optimizer is either a specialized software tool or a built-in unit of a compiler (the so-called optimizing compiler). Optimizations are classified into high-level and low-level optimizations. High-level optimizations are usually performed by the programmer who handles abstract entities (functions, procedures, classes, etc.) and keeps in mind the general framework of the task to optimize the design of a system. Optimizations performed at the level of elementary structural blocks of source code - loops, branches, etc. - are usually referred to as high-level optimizations too. Low-level optimizations are performed at the stage when source code is compiled into a set of machine instructions, and it is at this stage that automated optimization is usually employed. Assembler programmers, though, believe that no machine, however perfect, can do this better than a skilled programmer.

### **Control-Flow Analysis:-**

In control-flow analysis, the compiler figures out even more information about the program does its work, only now it can assume that there are no syntactic or semantic errors in the code.

Control-flow analysis begins by constructing a control-flow graph, which is a graph of the different possible paths program flow could take through a function. To build the graph, we first divide the code into basic blocks. A basic block is a segment of the code that a program must enter at the beginning and exit only at the end. This means that only the first statement can be reached from outside the block (there are no branches into the middle of the block) and all



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



statements are executed consecutively after the first one is (no branches or halts until the exit). Thus a basic block has exactly one entry point and one exit point. If a program executes the first instruction in a basic block, it must execute every instruction in the block sequentially after it. Optimizations performed exclusively within a basic block are called "local optimizations".

**Constant Folding :-** Constant folding refers to the evaluation at compile-time of expressions whose operands are known to be constant.

**Constant Propagation:-** If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable.

**Code Motion :-** Code motion (also called code hoisting ) unifies sequences of code common to one or more basic blocks to reduce code size and potentially avoid expensive re-evaluation.

**Peephole Optimizations :-** Peephole optimization is a pass that operates on the target assembly and only considers a few instructions at a time (through a "peephole") and attempts to do simple, machine-dependent code improvements.

**Redundant instruction elimination:-**At source code level, the following can be done by the user

int add_ten(int x)	int add_ten(int x)	int add_ten(int x)	int add_ten(int x)
{	{	{	{
int y, z;	int y;	int y = 10;	return x + 10;
y = 10;	y = 10;	return x + y;	}
z = x + y;	y = x + y;	}	
return z;	return y;		
}	}		

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

MOV x, R1

**Unreachable code:-**Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidentally written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
    return x + 10;
    printf("value of x is %d", x);
}
```

In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

### Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:...

```
MOV R1, R2
GOTO L1
...
```



L1 : GOTO L2

L2 : INC R1

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:...

MOV R1, R2

GOTO L2

...

L2 : INC R1

### **Algebraic expression simplification**

There are occasions where algebraic expressions can be made simple. For example, the expression  $a = a + 0$  can be replaced by  $a$  itself and the expression  $a = a + 1$  can simply be replaced by `INC a`.

### **Strength reduction**

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result. For example,  $x * 2$  can be replaced by  $x \ll 1$ , which involves only one left shift. Though the output of  $a * a$  and  $a^2$  is same,  $a^2$  is much more efficient to implement.

### **Accessing machine instructions**

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.



## Optimization Phases

The phase which represents the pertinent, possible flow of control is often called *control flow analysis*. If this representation is graphical, then a *flow graph* depicts *all* possible execution paths. Control flow analysis simplifies the data flow analysis. Data flow analysis is the process of collecting information about the *modification, preservation*, and *use* of program "quantities"--usually *variables* and *expressions*. Once control flow analysis and data flow analysis have been done, the next phase, the improvement phase, improves the program code so that it runs faster or uses less space. This phase is sometimes termed *optimization*. Thus, the term optimization is used for this final code improvement, as well as for the entire process which includes control flow analysis and data flow analysis. Optimization algorithms attempt to remove useless code, eliminate redundant expressions, move invariant computations out of loops, etc.

## CONCLUSION:






Hence, we have successfully implemented code optimization using directed acyclic graph.

## OUTCOME

**Upon completion Students will be able to:**

1. Optimize the code.

## QUESTIONS:

1. What is code optimization? 
2. How DAG is used for code optimization? 
3. Define peephole optimization and list its characteristics. 
4. Mention the issues to be considered while applying the techniques for code optimization.
5. What do you mean by machine dependent and machine independent optimization? 
6. What are the criteria that need to be considered while applying the code optimization techniques? 



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



## **EXPERIMENT NO. 9**

### **Code Generation using Labeled Tree**



## Session Plan

Time ( min)	Content	Learning Aid / Methodology	Faculty Approach	Typical Student Activity	Skill / Competency Developed.
05	Relevance and significance of Problem statement	Chalk & Talk , Presentation	Introduces, Explains	Listens, Participates, Discusses	Knowledge, intrapersonal
10	Concept of DAG & Labeling Algorithm	Chalk & Talk , Presentation	Introduces, Facilitates, Explains	Listens, Participates,	Knowledge, intrapersonal, Application
25	Explanation of code generation algorithm	Demonstration, Presentation	Explains, Facilitates, Monitors	Listens, Participates, Discusses	Knowledge, intrapersonal, interpersonal Application
60	Implementation of problem statement	N/A	Guides, Facilitates Monitors	Participates, Discusses	Comprehension, Hands on experiment
10	Assessment	N/A	Monitors	Participates, Discusses	Knowledge, Application
10	Conclusions	Keywords	Lists, Facilitates	Listens, Participates, Discusses	Knowledge, intrapersonal, Comprehension



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



**TITLE:** Code Generation using Labeled Tree

**OBJECTIVES:**

1. To express concept of Labeled Tree.
2. To apply the code generation algorithm to generate target code.

**PROBLEM STATEMENT:**

Accept tree for the given expression. Apply Labeling algorithm to tree and then apply code generation algorithm to generate target code from labeled tree.

**SOFTWARE REQUIRED:** 64-bit Fedora or equivalent OS, LEX, YACC.

**INPUT:** Input data as Labeled Tree which is generated from given expression.

**OUTPUT:** It will create a target code.

**MATHEMATICAL MODEL:**

Let S be the solution perspective of the code generation such that

$S = \{s, e, i, o, f, DD, NDD, success, failure\}$

s=initial state that is start of program

e = the end state or end of program

i= postfix expression





o=target code

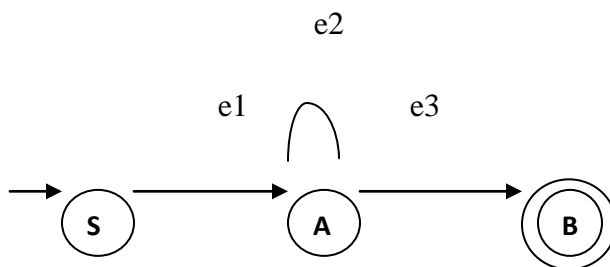
f= {ComputeLabel ( ), PostTraverse( ), Gencode ( )}

ComputeLabel ( )={Applies Labeling Algorithm to tree generated from postfix expression}

PostTraverse( )={ displays tree in postorder}

Gencode() = { Handles different cases of tree & accordingly generate target code}

COMPUTATIONAL MODEL:



Where S : Initial State

A : DAG /Labeled Tree

B: Target Code

Edge e1 : postfix expression

e2 : Compute Label

e3: root node of labeled tree

System Accepts postfix expression & enters into A state . In that state, it generates tree, at the same time, calculates label of tree. After that, it passes root node of Label tree to gencode function & enters into B state. In B state it applies code generation algorithm & generate target code.



Success- desired output is generated as target code in assembly language form

Failure- desired output is not generated as target code in assembly language form

### **THEORY:**

The Labeling Algorithm

The labeling can be done by visiting nodes in a bottom-up order so that a node is not visited until all its children are labeled.

Fig.1.1 gives the algorithm for computing label at node  $n$ .

```
if  $n$  is a leaf then  
    if  $n$  is the leftmost child of its parent then  
         $label(n) := 1$   
    else  $label(n) := 0$   
else begin /*  $n$  is an interior node */  
    let  $n_1, n_2, \dots, n_k$  be the children of  $n$  ordered by  $label$ ,  
    so  $label(n_1) \geq label(n_2) \geq \dots \geq label(n_k)$ ;  
     $label(n) := \max_{1 \leq i \leq k} (label(n_i) + i - 1)$   
end
```

Fig.1.1 The Labeling Algorithm

In the important special case that  $n$  is binary node and its children have labels  $l_1$  and  $l_2$ , the above formula reduces to

$$label(n) = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

Example:

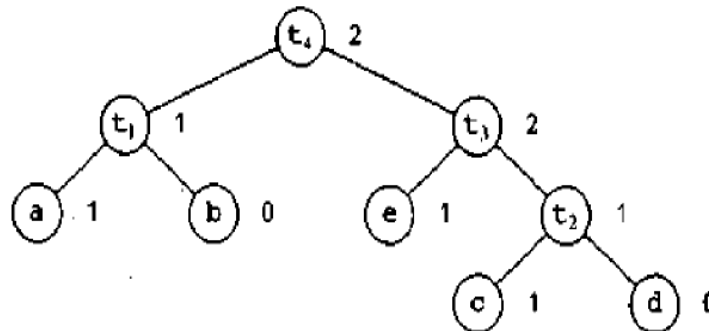


Fig. 1.2 Labeled Tree

Node a is labeled 1 since it is left leaf. Node b is labeled 0 since it is right leaf. Node t1 is labeled 1 because the labels of its children are unequal and the maximum label of a child is 1. Fig.1.2 shows labeled tree that results.

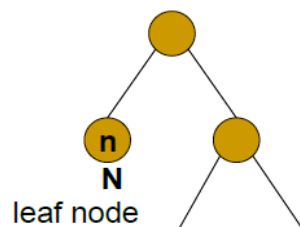
Code generation from a Labeled Tree

Procedure GENCODE(n)

- RSTACK –stack of registers,  $R_0, \dots, R_{(r-1)}$
- TSTACK –stack of temporaries,  $T_0, T_1, \dots$
- A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into the register  $\text{top}(\text{RSTACK})$ , and
  - the rest of RSTACK remains in the same state as the one before the call
- A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child

Procedure gencode(n);

{ /\* case 0 \*/





If

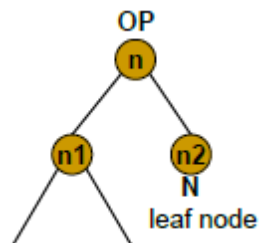
n is a leaf representing

operand N and is the

leftmost child of its parent

then

print(LOAD N, top(RSTACK))



/\* case 1 \*/

else if

n is an interior node with operator OP, left child n1, and right child n2

then

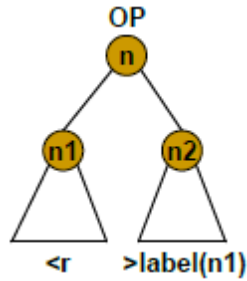
if label(n2) == 0 then {

let N be the operand for n2;

gencode(n1);



```
print(OP N, top(RSTACK));  
  
}
```

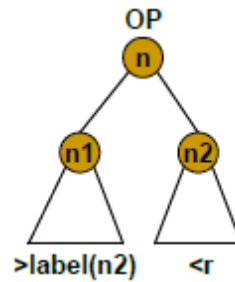


```
/* case 2 */  
  
else if ((1 < label(n1) < label(n2))  
and( label(n1) < r))  
then {  
    swap(RSTACK); gencode(n2);  
  
    R := pop(RSTACK); gencode(n1);  
  
    /* R holds the result of n2 */  
  
    print(OP R, top(RSTACK));  
  
    push (RSTACK,R);  
  
    swap(RSTACK);  
  
}
```

The swap() function ensures that a node is evaluated into the same register as its left child



**JSPM's**  
**RAJARSHI SHAHU COLLEGE OF ENGINEERING**  
**TATHAWADE, PUNE-33**  
(Autonomous Institute Affiliated to Savitribai Phule Pune University)



/\* case 3 \*/

else if ((1 < label(n2) < label(n1))

and( label(n2) < r))

then {

gencode(n1);

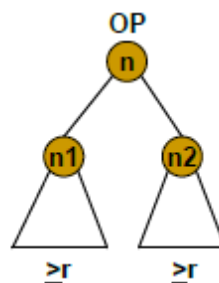
R := pop(RSTACK); gencode(n2);

/\* R holds the result of n1 \*/

print(OP top(RSTACK), R);

push (RSTACK,R);

}



/\* case 4, both labels are > r \*/



```
else {  
  
gencode(n2); T:= pop(TSTACK);  
  
print(LOAD top(RSTACK), T);  
  
gencode(n1);  
  
print(OP T, top(RSTACK));  
  
push(TSTACK, T);  
  
}  
  
}
```

Example:

Consider Fig.1.2 Labeled Tree with rstack = R0, R1 initially. The sequence of calls to gencode and code printing steps is shown below.

```
gencode(t4)    [R1R0]          /* case 2 */  
  gencode(t3)    [R0R1]          /* case 3 */  
    gencode(e)    [R0R1]          /* case 0 */  
      print MOV e, R1  
        gencode(t2)    [R0]          /* case 1 */  
          gencode(c)    [R0]          /* case 0 */  
            print MOV c, R0  
              print ADD d, R0  
                print SUB R0, R1  
                  gencode(t1)    [R0]          /* case 1 */  
                    gencode(a)    [R0]          /* case 0 */  
                      print MOV a, R0  
                        print ADD b, R0  
                          print SUB R1, R0
```

Fig.1.3 Trace of gencode routine



## CONCLUSION:

Hence, we have successfully studied DAG, Labeling algorithm and code generation from Labeled Tree.

## OUTCOME

Upon completion Students will be able to:

- ☐ Explain the labeling algorithm.
- ☐ Apply code generation algorithm to generate target code from Labeled Tree.

## QUESTIONS:

1. Describe issues in code generation phase
2. Difference between a AST and DAG
3. What are the applications of DAG
4. Explain Labeling Algorithm
5. Explain code generation algorithm for DAG
6. How to Generate DAG from postfix expression.
7. What are possible inputs to code generation phase.
8. Compute DAG for following
  - (i)  $X1 = a+b*12$
  - (ii)  $X2 = a/2+b*12$
9. For following expression
$$(a+b) - (c - (d+e))$$
  - (i) Draw expression tree and Apply labeling algorithm to generate labeled tree





(ii) Show optimal code for labeled expression tree



### **PRACTICE ASSIGNMENT:**

1. Accept input as 3-address code & generate target code for same.
2. Implement a program to generate DAG and target code for following piece of code

Repeat {

$p = p + x[i] / y[i]$

}

Until  $i > 100$

### **REFERENCES**

1. A V Aho, R Sethi, J D Ullman, "Compilers: Principles, Techniques, and Tools", Pearson Edition, ISBN 81-7758-590-8.
2. J R Levin, T Mason, D Brown, "Lex and Yacc", O'Reilly, 2000 ISBN 81-7366-061-X
3. D. M. Dhamdhere, "Compiler Construction: Principles and Practice", Macmillan India, 1983.
4. S. Chattopadhyay, "Compiler Design", Prentice-Hall of India, 2005, ISBN 81-203-2725-X.