

St. Thomas College of Engineering & Technology

Kozhuvalloor-Chengannur

**Department of Computer Science and Engineering**

**CSL 411 COMPILER LAB**

**LAB MANUAL**

**KTU 2019 Scheme SEMESTER – VII**

**INSTITUTE VISION AND MISSION**

# Vision

To be an institute of repute recognised for excellence in education, innovation andsocial contribution.

# Mission

M1: Infrastructural Relevance: Develop, maintain and manage our campus for our stakeholders.

M2: Life Long Learning: Encourage our stakeholders to participate in lifelong learning through industry and academic interactions.

M3: Social Connect: Organize socially relevant outreach programs for the benefit of humanity.

**DEPARTMENT VISION AND MISSION**

# Vision

To create industry ready and socially skilled Computer Science Engineers

# Mission

M1: Provide a learning platform that encourages thinking and analytical abilityin the area of computer software and hardware

M2: Inculcate lifelong and professional skills through the interaction of academicians and industrialists

M3: Engage with society through social programs in and out of campus

**PROGRAMME OUTCOMES(POs)**

## 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 engineeringproblems. **[PO1]**
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. **[PO2]**
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. **[PO3]**
4. **Conduct investigations of complex problems**: Use research-based knowledge and researchmethods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions. **[PO4]**
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activitieswith an understanding of the limitations. **[PO5]**
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 relevantto the professional engineering practice. **[PO6]**
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. **[PO7]**
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice. **[PO8]**
9. **Individual and team work**: Function effectively as an individual, and as a member or leaderin diverse teams, and in multidisciplinary settings. **[PO9]**
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. **[PO10]**
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. **[PO11]**
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. **[PO12]**

**PROGRAM SPECIFIC OUTCOMES (PSOs)**

**Engineering Graduates will be able to:**

1. **Professional Skills**: Ability to understand the architecture and working of computerhardware and software system. **[PSO1]**
2. **Design and Development Skills**: Ability to design and develop software for technology application to fulfill industrial and social needs. **[PSO 2]**

**COURSE OBJECTIVE**

To give students hands-on experience on compiler design concepts. Students will be able to familiarize with tools such as LEX and YACC and automate different phases of a compiler.

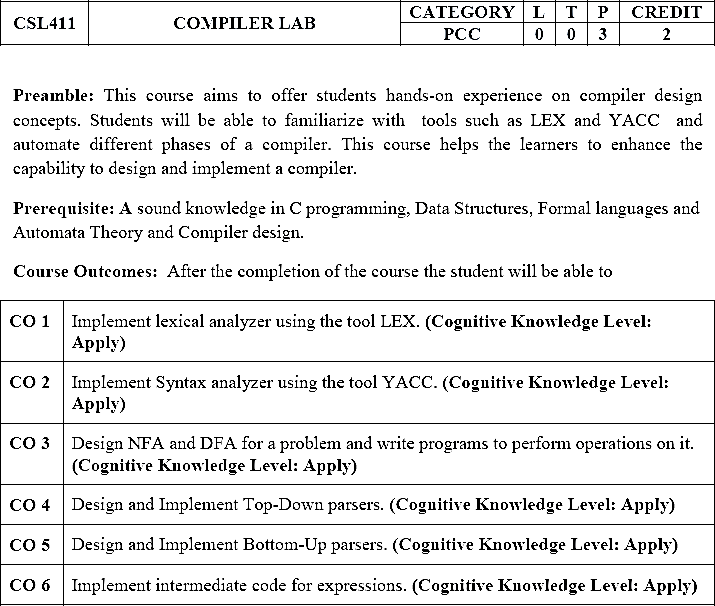
**COURSE OUTCOME(COs)**

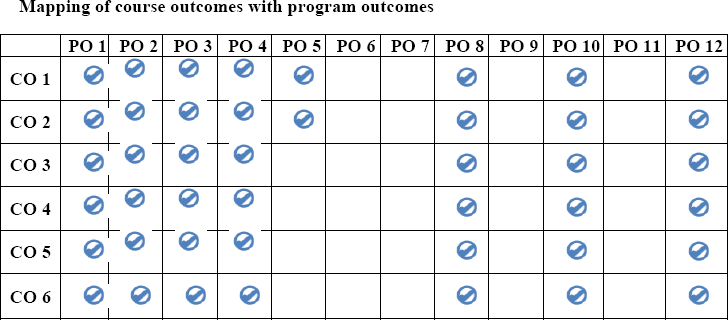
After the completion of the course the student will be able to

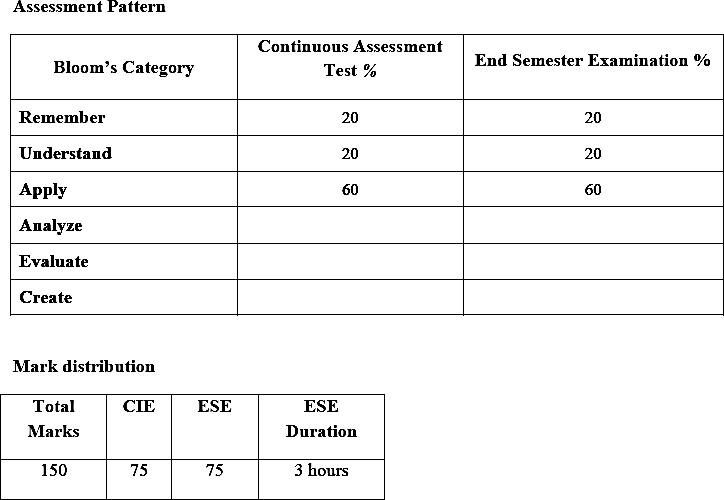
|  |  |
| --- | --- |
| **CO** | **DESCRIPTION** |
| **CSL 411.1** | Implement lexical analyzer using the tool LEX. (Cognitive Knowledge Level: Apply) |
| **CSL 411.2** | Implement Syntax analyzer using the tool YACC. (Cognitive Knowledge Level: Apply) |
| **CSL 411.3** | Design NFA and DFA for a problem and write programs to perform operations on it. (Cognitive Knowledge Level: Apply) |
| **CSL 411.4** | Design and Implement Top-Down parsers. (Cognitive Knowledge Level: Apply) |
| **CSL 411.5** | Design and Implement Bottom-Up parsers. (Cognitive Knowledge Level: Apply) |
| **CSL 411.6** | Implement intermediate code for expressions. (Cognitive Knowledge Level: Apply) |

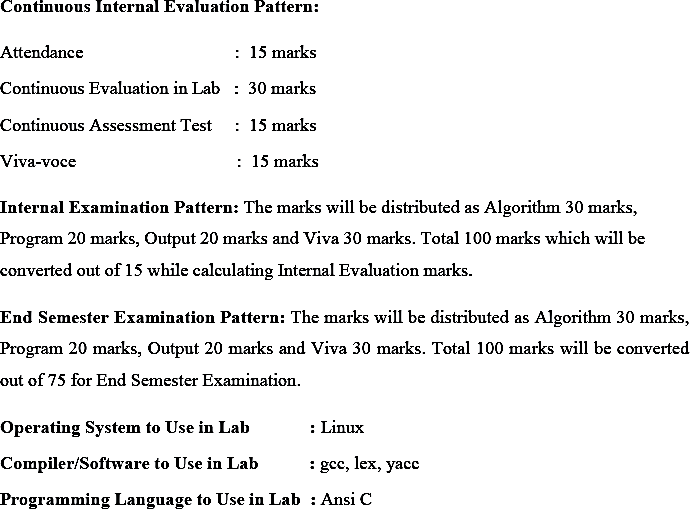
**CO-PO/PSO Mapping**

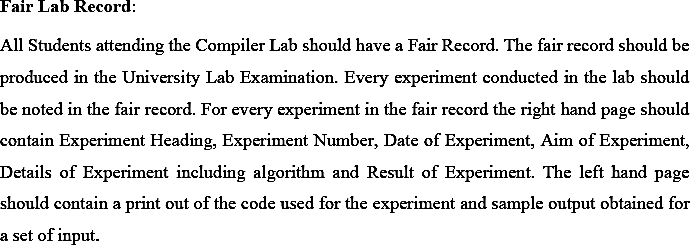
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CO** | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** | **PSO1** | **PSO2** |
| **CSL 411.1** | 3 | 3 | 3 | 3 | 2 |  |  | 2 |  | 2 |  | 2 | 2 | 3 |
| **CSL 411.2** | 3 | 3 | 3 | 3 | 2 |  |  | 2 |  | 2 |  | 2 | 2 | 3 |
| **CSL 411.3** | 3 | 3 | 3 | 3 |  |  |  | 2 |  | 2 |  | 2 | 2 | 3 |
| **CSL 411.4** | 3 | 3 | 3 | 3 |  |  |  | 2 |  | 2 |  | 2 | 2 | 3 |
| **CSL 411.5** | 3 | 3 | 3 | 3 |  |  |  | 2 |  | 2 |  | 2 | 2 | 3 |
| **CSL 411.6** | 3 | 3 | 3 | 3 |  |  |  | 2 |  | 2 |  | 2 | 2 | 3 |

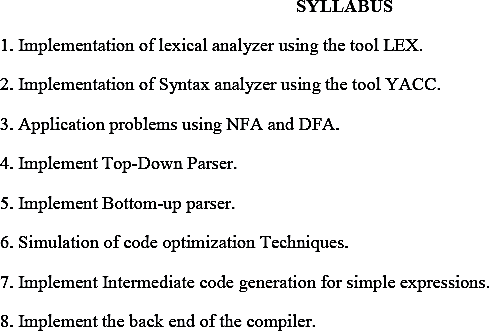


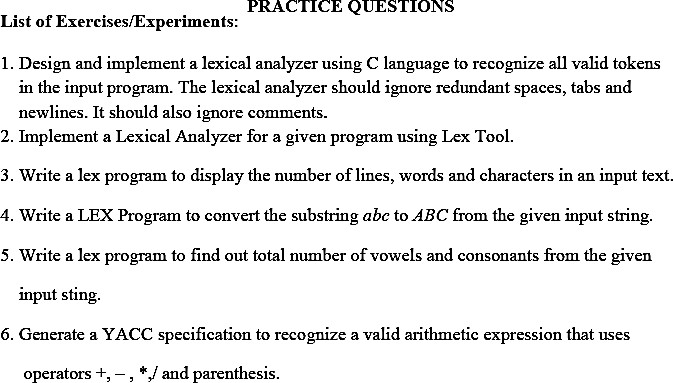


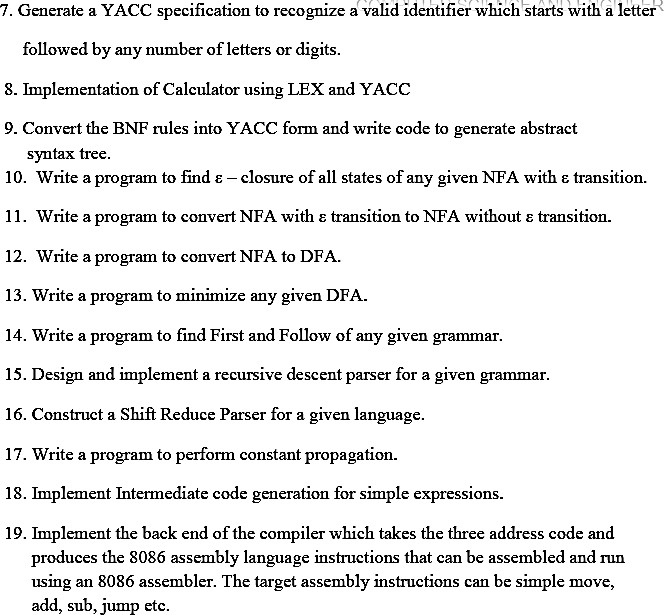












**LAB CYCLE**

**CSL 411 COMPILER LAB**

1. Design and implement a Lexical Analyzer using C.
2. Write a Lex program to display the number of lines, words and characters in an input text.
3. Write a Lex program to convert the substring “*abc*” to ABC from the given input string.
4. Write a lex program to find out total number of vowels and consonants from the given input string.
5. Write a program to recognize a valid arithmetic expression that uses operator +, - , \* and /
6. Generate a Yacc specification to recognize a valid identifier which starts with a letter followed by any number of letters or digits.
7. Implement a arithmetic calculator using Lex and Yacc.
8. Implement Recursive descent parser of a given grammar.
9. Write a program to find the FIRST of a given grammar.
10. Write a program to find the FOLLOW of a given grammar.
11. Implement operator precedence parser for a given language.
12. Implement shift reduce parser.
13. Write a program for constant propagation.
14. Implement the back end of the compiler

**INDEX**

|  |  |  |
| --- | --- | --- |
| **Exp No.** | **Name of Experiment** | **Page No.** |
| 1 | LEXICAL ANALYZER USING C | 1 |
| 2 | DISPLAY THE NUMBER OF LINES, WORDS AND CHARACTERS IN AN INPUT TEXT | 3 |
| 3 | CONVERT THE SUBSTRING “abc” TO “ABC” FROM THE GIVEN INPUT STRING. | 5 |
| 4 | FIND OUT TOTAL NUMBER OF VOWELS AND CONSONANTS FROM THE GIVEN INPUT STING. | 7 |
| 5 | RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +, - , \* AND / | 9 |
| 6 | RECOGNIZE A VALID IDENTIFIER WHICH STARTS WITH A  LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS. | 11 |
| 7 | ARITHMETIC CALCULATOR USING LEX AND YACC. | 13 |
| 8 | RECURSIVE DESCENT PARSER OF A GIVEN GRAMMAR | 15 |
| 9 | FIND THE FIRST OF A GIVEN GRAMMAR. | 19 |
| 10 | FIND THE FOLLOW OF A GIVEN GRAMMAR. | 23 |
| 11 | OPERATOR PRECEDENCE PARSER FOR A GIVEN LANGUAGE. | 26 |
| 12 | IMPLEMENTATION OF SHIFT REDUCE PARSER. | 32 |
| 13 | CONSTANT PROPAGATION. | 37 |
| 14 | IMPLEMENTATION OF BACK END OF THE COMPILER | 40 |

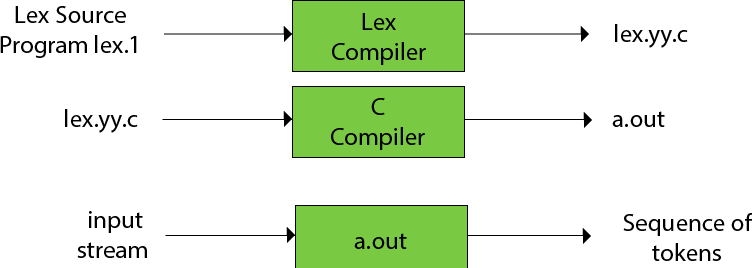
**STUDY OF LEX AND YACC TOOLS**

* + **LEX-**

A Lexical analyzer generator: Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). Lex is commonly used with the yacc parser generator. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language

A lexer or scanner is used to perform lexical analysis or the breaking up of an input stream into meaningful units, or tokens.

* Lex is officially known as a "Lexical Analyser".
* Its main job is to break up an input stream into more usable elements. Or in, other words, to identify the "interesting bits" in a text file.
* For example, if you are writing a compiler for the C programming language, the symbols { } ( ); all have significance on their own.
* The letter a usually appears as part of a keyword or variable name, and is not interesting on its own.
* Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes "like this"
* All of these things are handled by the Lexical Analyser.
* A tool widely used to specify lexical analyzers for a variety of languages
* We refer to the tool as Lex compiler, and to its input specification as the Lex language.



A Lex program (the .l file) consists of three parts, separated by %% delimiters. The formal of Lex source is as follows:

***declarations***

**%%**

***translation rules***

## %%

### auxiliary procedures

* + The declarations section includes declarations of constant, variable and regular definitions.
  + The translation rules of a Lex program are statements of the form : p1 {action1}

p2 {action2}

....

pn {action n}

Where each p is a regular expression and each action is a program fragment describing what action the lexical analyzer should take when a pattern p matches a lexeme. In Lex the actions are written in C.

* + The third section holds whatever auxiliary procedures are needed by the actions. Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

In the lex program, a main () function is generally included as: main ()

{

yyin = fopen (filename,”r”); while (yylex ());

}

Here filename corresponds to input file and the yylex routine is called which returns the tokens.

Lex makes the lexeme available to the routines appearing in the third section through two variables yytext and yyleng

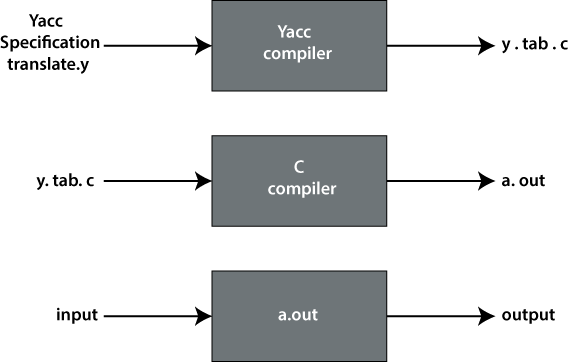
1. yytext is a variable that is a pointer to the first character of the lexeme.
2. yyleng is an integer telling how long the lexeme is.

# YACC: Yet Another Compiler-Compiler

* + YACC provides a tool to produce a parser for a given grammar.
  + A parser generator is a program that takes as input a specification of syntax, and produces as output a procedure for recognizing that language.
  + YACC is a program designed to compile a LALR (1) grammar.
  + It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
  + The input of YACC is the rule or grammar and the output is a C program.
  + YACC stands for "Yet another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers.

•

The construction of translation using YACC is illustrated in the figure below:



A YACC source program contains three parts:

***Declarations***

**%%**

***Translation rules***

## %%

***Supporting C rules***

## Declarations Part

This part of YACC has two sections; both are optional. The first section has ordinary C declarations, which are delimited by %{ and %}. Any temporary variable used by the second and third sections will be kept in this part.

Example :

%{

#include <ctype.h>

%}

The second section contains the declaration of grammar tokens. This part defined the tokens that can be used in the later parts of a YACC specification.

Example :

%token DIGIT

**Translation Rule Part**

After the first %% pair in the YACC specification part, we place the translation rules. Every rule has a grammar production and the associated semantic action.

A set of productions would be written in YACC as

|  |  |  |
| --- | --- | --- |
| <head> :  | | <body>1  <body>2  ..... | {<semantic action>1}  {<semantic action>2} |
| | | <body>n | {<semantic action>n} |
| ; |  |  |

The semantic action of YACC is a set of C statements. In a semantic action, the symbol $$ considered to be an attribute value associated with the head's non-terminal. While $i considered as the

value associated with the ith grammar production of the body. If we have left only with associated production, the semantic action will be performed. The value of $$ is computed in terms of $i's by semantic action.

## Supporting C–Rules

It is the last part of the YACC specification and should provide a lexical analyzer named yylex().

The attribute value which is associated with a token will communicate to the parser through a variable called **yylval**.

# Using the Lex Program with the Yacc Program:

The Lex program recognizes only extended regular expressions and formats them into character packages called tokens, as specified by the input file. When using the Lex program to make a lexical analyzer for a parser, the lexical analyzer (created from the Lex command) partitions the input stream. The parser(from the yacc command) assigns structure to the resulting pieces. You can also use other programs along with programs generated by Lex or yacc commands.

The yacc program looks for a lexical analyzer subroutine named yylex, which is generated by the lex command. Normally the default main program in the Lex library calls the yylex subroutines. However if the yacc command is loaded and its main program is used, yacc calls the yylex subroutines. In this case each Lex rule should end with: return (token); Where the appropriate token value is returned. The yacc command assigns an integer value to each token defined in the yacc grammar file through a # define preprocessor statement. The lexical analyzer must have access to these macros to return the tokens to the parser. Use the yacc –d option to create a y.tab.h file and include the y.tab.h file in the Lex specification file by adding the following lines to the definition section of the Lex specification file:

%{

#include “y.tab.h”

%}

Alternatively we can include the lex.yy.c file the yacc output file by adding the following lines after the second %% (percent sign, percent sign) delimiter in the yacc grammar file: #include”lex.yy.c”. The yacc library should be loaded before the Lex library to get a main program that invokes the yacc parser. You can generate Lex and yacc programs in either order

Experiment No. 1

**AIM:**

# LEXICAL ANALYZER USING C

Design and implement a lexical analyzer for given language using C.

## PROGRAM:

#include<stdio.h> #include<string.h> #include<ctype.h> void main()

{

FILE \*f1;

char c;

char str[20];

int i=0,num,linecount=1; f1=fopen("input.txt","r"); while((c=getc(f1))!=EOF)

{

if(isdigit(c))

{

num=c-48; c=getc(f1); while(isdigit(c))

{

num=num\*10+(c-48); c=getc(f1);

}

printf("%d is a number \n",num);

}

else if(isalpha(c))

{

str[i++]=c; c=getc(f1);

while(isdigit(c)||isalpha(c)||c=='\_'||c=='$')

{

str[i++]=c; c=getc(f1);

}

str[i++]='\0';

if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strc mp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("d ouble",str)==0||strcmp("static",str)==0||strcmp("switch",str)==0||strcmp(" case",str)==0)

{

printf("%s is a keyword\n",str);

}

else

printf("%s is a identifier\n",str); ungetc(c,f1);

i=0;

}

else if(c==' '||c=='\t')

{ }

else if(c=='\n') linecount++;

else

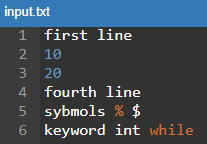
printf("%c is a special symbol\n",c);

}

printf("Total no. of lines are: %d\n",linecount); fclose(f1);

}

## OUTPUT:



|  |  |
| --- | --- |
| first is a identifier line is a identifier 10 is a number 20 is a number  fourth is a identifier line is a identifier  sybmols is a identifier | % is a special symbol  $ is a special symbol keyword is a identifier int is a keyword  while is a keyword Total no. of lines are: 6 |

**RESULT:**

A lexical analyzer for a given language has been implemented using C.

## QUESTIONS:

* 1. Is it necessary to translate a HLL program? Explain.
  2. List various language Translators.
  3. Differentiate tokens
  4. What is a compiler?
  5. List the various phases of a compiler.

Experiment No. 2

# DISPLAY THE NUMBER OF LINES, WORDS AND CHARACTERS IN AN INPUT TEXT

## AIM:

Write a Lex program to display the number of lines, words and characters in an input text.

## ALGORITHM:

Step 1. Start

Step 2. Read the input file/text

Step 3. Initialize the counters for characters, words, lines to zero Step 4. Scan the characters, words, lines and

Step 5. increment the respective counters Step 6. Display the counts

Step 7. Stop

## PROGRAM:

%{

#include<stdio.h>

int sc=0,wc=0,lc=0,cc=0;

%}

%%

[\n] { lc++; wc++;}

[\t] { sc++; wc++;}

[" "] { sc++; wc++;}

[^\n\t ]+ {cc+=yyleng;}

%%

void main()

{

printf("Enter the input:\n"); yylex();

printf("The number of lines=%d\n",lc); printf("The number of spaces=%d\n",sc); printf("The number of words=%d\n",wc); printf("The number of characters are=%d\n",cc);

}

int yywrap( )

{

return 1;

}

## OUTPUT: (Note: After input string press enter and ctrl+d)

Enter the input:

s7 cse stc cngr

^d

The number of lines=2 The number of spaces=2 The number of words=4

The number of characters are=12

## RESULT:

Lex program to display the number of lines, words and characters in an input text is implemented.

## QUESTIONS:

1. What is the output of lexical analyzer?
2. What are the classifications of a compiler?
3. List the phases that constitute the front end of a compiler.
4. Differentiate patterns
5. Differentiate LEXEME.

Experiment No. 3

# CONVERT THE SUBSTRING “abc” TO “ABC” FROM THE GIVEN

**INPUT STRING**

## AIM:

Write a Lex program to convert the substring “abc” to “ABC” from the given input string.

## ALGORITHM:

Step 1. Start

Step 2. Read the input text

Step 3. If the first three characters are “a”, “b”, and “c” then Replace the corresponding characters by “A”,”B” and “C”.

Step 4. Print the updated text Step 5. Stop

## PROGRAM:

%{

#include<stdio.h> int i;

%}

%%

[a-z A-Z]\* { for(i=0;i<=yyleng;i++)

{

if((yytext[i]=='a')&&(yytext[i+1]=='b')&&(yytext[i+2]=='c'))

{

yytext[i]='A';

yytext[i+1]='B';

yytext[i+2]='C';

}

}

printf("%s",yytext);

}

. {ECHO;}

\n {ECHO;}

%%

void main()

{

yylex();

}

int yywrap()

{

return 1;

}

## OUTPUT:

abc ABC

## RESULT:

Lex program to convert the substring “abc” to “ABC” from the given input string is implemented.

## QUESTIONS:

* 1. What is bootstrapping in compiler design?
  2. What is yacc?
  3. What is Relocatable Machine Code?
  4. What is Linker?
  5. What is Cross Compiler?

Experiment No. 4

# FIND OUT TOTAL NUMBER OF VOWELS AND CONSONANTS FROM THE GIVEN INPUT STRING.

**AIM:**

Write a Lex program to find out total number of vowels and consonants from the given input string.

**ALGORITHM:**

Step 1. Start

Step 2. Take the string as input

Step 3. Take each character from this string to check

Step 4. If this character is a vowel, increment the count of vowels Step 5. Else increment the count of consonants.

Step 6. Print the total count of vowels and consonants in the end. Step 7. Stop

## PROGRAM:

%{

int vcount=0; int ccount =0;

%}

%%

[aeiouAEIOU] {vcount++;} [a-zA-Z] {ccount++;}

%%

void main()

{

printf("Enter the string of vowels and consonents:"); yylex();

printf("Number of vowels are: %d\n", vcount); printf("Number of consonants are: %d\n", ccount);

}

int yywrap()

{

return 1;

}

## OUTPUT:

Enter the string of vowels and consonents:Hello everyone

^Z

Number of vowels are: 6 Number of consonants are: 7

## RESULT:

Lex program to find out total number of vowels and consonants from the given input string is implemented.

## QUESTIONS:

1. Define compiler-compiler
2. Write a regular expression for an identifier.
3. What is Ambiguous grammar?
4. Define parser.
5. What is Symbol Table?

Experiment No. 5

**AIM:**

# RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +, - , \* AND /

Write a program to recognize a valid arithmetic expression that uses operator +, - , \* and /

## ALGORITHM:

Step1: Start

Step2: Reading an expression.

Step3: Checking the validating of the given expression according to the rule using yacc. Step4: Using expression rule print the result of the given values

Step5: Stop

## PROGRAM:

### Lex Program:

%{

#include "pgm5.tab.h"

%}

%%

[0-9]+ { return NUMBER; }

[\_a-zA-Z][\_a-zA-Z0-9]\* { return ID; }

\n { return 0; }

. { return yytext[0]; }

%%

int yywrap()

{

return 1;

}

### Yacc Program:

%{

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

%}

%token NUMBER ID

%left '+''-''\*''/'

%%

exp : exp'+'exp

| exp'-'exp

| exp'\*'exp

| exp'/'exp

| '('exp')'

| NUMBER

| ID

;

%%

void main()

{

printf("Enter the expression: "); yyparse();

printf("Valid Expression!\n");

}

void yyerror()

{

printf("Invalid Expression!\n"); exit(0);

}

## OUTPUT:

### Execution:

lex exp5.l yacc -d exp5.y

gcc lex.yy.c y.tab.c

./a.out

**Test Case #1: Valid Expression** Enter the expression: 12+23-8 Valid Expression!

## Test Case #2: Invalid Expression

Enter the expression: a+\* Invalid Expression!

## RESULT:

Lex and Yacc program to recognize a valid arithmetic expression that uses operator +, - , \* and / has been implemented.

## QUESTIONS:

1. What is a linear analysis called in a compiler?
2. Which of the compiler allows the only modified sections of source code to be recompiled?
3. Explain the structure of a lex program
4. Explain the structure of a yacc program
5. How lex passing the tokens to yacc

Experiment No. 6

# PROGRAM TO RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF

**LETTERS OR DIGITS**

## AIM:

Write a program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.

## ALGORITHM:

Step1: Start

Step2: Reading an expression.

Step3: Checking the validating of the given expression according to the rule using yacc. Step4: Using expression rule print the result of the given values

Step5: Stop

## PROGRAM:

### Lex Program:

%{

#include "pgm6.tab.h"

%}

%%

[0-9] { return DIGIT; }

[a-zA-Z] { return ALPHA; }

\n { return 0; }

. { return yytext[0]; }

%%

### Yacc Program:

%{

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

%}

%token DIGIT ALPHA

%%

var : ALPHA

| var ALPHA

| var DIGIT ;

%%

int main()

{

printf("Enter a variable name: "); yyparse();

printf("Valid Variable!\n"); return 0;

}

int yyerror()

{

printf("Invalid Variable!\n"); exit(1);

}

int yywrap()

{

return 1;

}

## OUTPUT:

**Test Case #1: Valid Variable** Enter a variable name: a123 Valid Variable!

**Test Case #2: Invalid Variable** Enter a variable name: 12aa Invalid Variable!

## RESULT:

Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits is implemented.

## QUESTIONS:

1. Which class of statement when compiled does not produce any executable code.
2. Define a context free grammar.
3. What is regular expression?
4. What is finite automaton?
5. What is an auxiliary definition?

Experiment No. 7

# ARITHMETIC CALCULATOR USING LEX AND YACC

## AIM:

Write a program to implement an arithmetic calculator using Lex and Yacc.

## PROGRAM:

### Lex Program:

%{

#include "pgm7.tab.h" #include<stdio.h> extern int yylval;

%}

%%

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

. {return yytext[0];} [\t]+ ;

\n {return 0;}

%%

### Yacc Program:

%{

#include<stdio.h>

%}

%token NUMBER

%left '+' '-'

%left '\*' '/'

%%

st: exp {printf(“Answer=%d\n”,$$);}

;

exp: exp '+' exp {$$ = $1 + $3;}

|exp '-' exp {$$ = $1 - $3;}

|exp '\*' exp {$$ = $1 \* $3;}

|exp '/' exp {$$ = $1 / $3;}

|'('exp')' {$$ = $2;}

|NUMBER {$$ = $1;}

;

%%

int main()

{

printf("Enter the expression"); yyparse();

return 0;

}

int yywrap()

{

return 0;

}

yyerror(char \*s)

{

printf("Error:%s",s);

}

## OUTPUT:

Enter the expression(5+2)\*(3-1)/(2) Answer=7

## RESULT:

Program to implement an arithmetic calculator using Lex and Yacc is implemented.

## QUESTIONS :

* 1. What does a bottom-up parser generate?
  2. What is bottom-up parser?
  3. What is the bottom-up parsing method also known as?
  4. Identify the method which merges the bodies of two loops?
  5. Which parsing technique avoids backtracking?

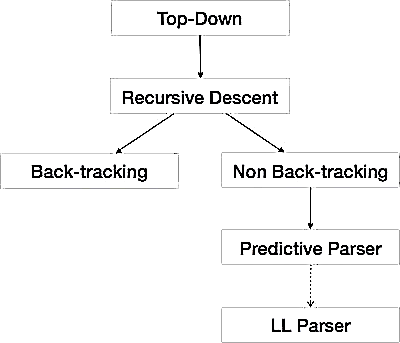
Experiment No. 8

# RECURSIVE DESCENT PARSER OF A GIVEN GRAMMAR

## AIM:

Write a program to implement Recursive descent parser of a given grammar

## DESCRIPTION:

We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes.

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

### Back-tracking:

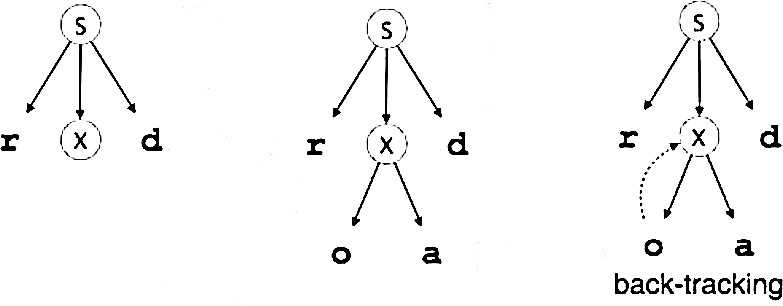
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).To understand this, take the following example of CFG.

S → rXd | rZd X → oa | ea

Z → ai

For an input string: read, a top-down parser will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‟r‟. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. „e‟). The parser tries to expand non-terminal „X‟ and checks its

production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).Now the parser matches all the input letters in an ordered manner. The string is accepted.

### For factoring:

Give a set of productions of the form A->αβ1|αβ2|…|αβn

Invent a new nonterminal Z and replace this set by the collection: A->αZ

Z->β1|β2|…βn

### For substitution:

If we are given a grammar containing A->Bα and if all of the productions with B on the left side are: B->β1|β2|…βnα

## ALGORITHM:

Step 1: start.

Step 2: Declare the prototype functions E() , Eprime (),T(), Tprime (),F() Step 3: Read the string to be parsed.

Step 4: Check the productions

Step 5: Compare the terminals and Non-terminals Step 6: Read the parse string.

Step 7: stop the production

## PROGRAM:

#include<stdio.h> #include<string.h> char input[10]; int i=0,error=0; void E();

void T();

void Eprime(); void Tprime(); void F();

void main()

{

printf("Enter an arithmetic expression :\n"); gets(input);

E();

if(strlen(input)==i&&error==0) printf("\nAccepted..!!!");

else

}

printf("\nRejected..!!!");

void E()

{

T();

Eprime();

}

Void Eprime()

{

if(input[i]=='+')

{

i++;

T();

Eprime();

}

}

void T()

{

F();

Tprime();

}

Void Tprime()

{

if(input[i]=='\*')

{

i++;

F();

Tprime();

}

}

void F()

{

if(input[i]=='(')

{

i++;

E();

if(input[i]==')') i++;

}

else if(isalpha(input[i]))

{

}

else

}

i++;

while(isalnum(input[i])||input[i]=='\_') i++;

error=1;

## OUTPUT:

Enter an arithmetic expression: sum+month\*interest Accepted..!!!

Enter an arithmetic expression: sum+avg\*+interest Rejected..!!!

## RESULT:

Program to implement Recursive descent parser of a given grammar has been implemented.

## QUESTIONS:

1. What type of parsing is Recursive Descent Parser?
2. What is Recursive descent parser?
3. What is Backtracking?
4. What is left recursion?
5. What is Left factoring?

Experiment No. 9

# FIRST OF A GIVEN GRAMMAR

## AIM:

Write a program to find the FIRST of a given grammar.

## DESCRIPTION:

The construction of a predictive parser is aided by two functions associated with a grammar

G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible.

## FIRST(α):

If α is any string of grammar symbols, let FIRST (α) be the set of terminals that begin the strings derived from α. If α→є then є is also in FIRST (α).

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or є can be added to any FIRST set:

1. If x is terminal, and then FIRST(X) is {X}.
2. If X →є is a production, then add є to FIRST(X).
3. If X is non terminal and X →Y1 Y2 ... YK. is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi ), and є is in all of FIRST(Y1), ... , FIRST(Yi-1); that is, Y1,

... ,Yi-1 → є. If є is in FIRST (Yj ) for all j = 1, 2, ... , k, then add є to FIRST(X). For example, everything in FIRST (Y1) is surely in FIRST(X). If Y1 does not derive є, then we add nothing more to FIRST(X), but if Y1→є, then we add FIRST (Y2) and so on.

## ALGORITHM:

1. **Compute FIRST(X) as follows:**
   * if X is a terminal, then FIRST(X)={X}
   * if X is a production, then add  to FIRST(X)
   * if X is a non-terminal and XY1Y2...Yn is a production, add FIRST(Yi) to FIRST(X) if the preceding Yjs contain  in their FIRSTs

## PROGRAM:

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

void FIRST(char[],char ); void result(char[],char); int nop;

char prod[10][10]; void main()

{

int i;

char choice; char c;

char res1[20]; clrscr();

printf("How many number of productions ? :"); scanf(" %d",&nop);

printf("enter the production string like E=E+T\n"); for(i=0;i<nop;i++)

{

printf("Enter productions Number %d : ",i+1); scanf(" %s",prod[i]);

}

do

{

printf("\n Find the FIRST of :"); scanf(" %c",&c); memset(res1,’0’,sizeof(res)); FIRST(res1,c);

printf("\n FIRST(%c)= { ",c);

for(i=0;res1[i]!='\0';i++)

printf(" %c ",res1[i]);

printf("}\n");

printf("press 'y' to continue : "); scanf(" %c",&choice);

}

while(choice=='y'||choice =='Y');

}

void FIRST(char res[],char c)

{

inti,j,k;

char subres[5]; int eps; subres[0]='\0';

res[0]='\0'; memset(res,’0’,sizeof(res)); memset(subres,’0’,sizeof(res)); if(!(isupper(c)))

{

result(res,c); return ;

}

for(i=0;i<nop;i++)

{

if(prod[i][0]==c)

{

if(prod[i][2]=='$')

result(res,'$');

else

{ j=2;

while(prod[i][j]!='\0')

{

eps=0; FIRST(subres,prod[i][j]);

for(k=0;subres[k]!='\0';k++) result(res,subres[k]); for(k=0;subres[k]!='\0';k++) if(subres[k]=='$')

{

eps=1; break;

}

if(!eps) break; j++;

}

}

}

}

return ;

}

void result(char res[],char val)

{

int k;

for(k=0 ;res[k]!='\0';k++) if(res[k]==val)

return; res[k]=val; res[k+1]='\0';

}

## OUTPUT:

How many number of productions ?:8 enter the production string like E=E+T Enter productions Number 1 : E=TX Enter productions Number 2 : X=+TX Enter productions Number 3 : X=$ Enter productions Number 4 : T=FY Enter productions Number 5 : Y=\*FY Enter productions Number 6 : Y=$ Enter productions Number 7 : F=(E) Enter productions Number 8 : F=i Find the FIRST of :X

FIRST(X)= { + $ }

press 'y' to continue : Y

Find the FIRST of :F FIRST(F)= { ( i }

press 'y' to continue : Y Find the FIRST of :Y FIRST(Y)= { \* $ }

press 'y' to continue : Y Find the FIRST of :E FIRST(E)= { ( i }

press 'y' to continue : Y Find the FIRST of :T FIRST(T)= { ( i }

press 'y' to continue : N

## RESULT:

Program to find the FIRST of a given grammar is implemented and executed successfully.

## QUESTIONS:

1. Define parser.
2. Mention the basic issues in parsing.
3. Why lexical and syntax analyzers are separated out?
4. Define a context free grammar.
5. Write the algorithm for FIRST

Experiment No. 10

# FOLLOW OF A GIVEN GRAMMAR

## AIM:

Write a program to find the Follow of a given grammar.

## DESCRIPTION:

For nonterminal A, if there are set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form S→αAaβ for some α and β. Note that there may, at some time during the derivation, have been symbols between A and a, but if so, they derived є and disappeared. If A can be the rightmost symbol in some sentential form, then $, representing the input right endmarker, is in FOLLOW (A).

To compute FOLLOW (A) for all no terminals A, apply the following rules until nothing can be added to any FOLLOW set:

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right end marker.
2. If there is a production A→αBβ, then everything in FIRST(β), except for є, is placed in FOLLOW(B).
3. If there is a production A → αB, or a production A →αBβ where FIRST(β) contains є, then everything in FOLLOW(A) is in FOLLOW(B).

## ALGORITHM:

1. **Compute FOLLOW as follows:**
   * FOLLOW(S) contains $, If S is the starting symbol of the grammar.
   * For productions AB, everything in FIRST() except  goes into FOLLOW(B)
   * For productions AB or AB where FIRST() contains , FOLLOW(B) contains everything that is in FOLLOW(A)

## PROGRAM:

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

int nop,m=0,p,i=0,j=0; char prod[10][10],res[10]; void FOLLOW(char c);

void first(char c); void result(char); void main()

{

int i;

int choice; char c,ch;

printf("Enter the no.of productions: "); scanf("%d", &nop); printf("enter the production string like E=E+T\n"); for(i=0;i<nop;i++)

{

printf("Enter productions Number %d : ",i+1); scanf(" %s",prod[i]);

}

do

{ m=0;

printf("Find FOLLOW of -->"); scanf(" %c",&c); FOLLOW(c);

printf("FOLLOW(%c) = { ",c); for(i=0;i<m;i++) printf("%c ",res[i]);

printf(" }\n");

printf("Do you want to continue(Press 1 to continue )?");

scanf("%d%c",&choice,&ch);

} while(choice==1);

}

void FOLLOW(char c)

{

if(prod[0][0]==c)

result('$'); for(i=0;i<nop;i++)

{

for(j=2;j<strlen(prod[i]);j++)

{

if(prod[i][j]==c)

{

if(prod[i][j+1]!='\0')

first(prod[i][j+1]); if(prod[i][j+1]=='\0'&&c!=prod[i][0])

FOLLOW(prod[i][0]);

}

}

}

}

void first(char c)

{

int k; if(!(isupper(c))) result(c); for(k=0;k<nop;k++)

{

if(prod[k][0]==c)

{

if(prod[k][2]=='$')

FOLLOW(prod[i][0]);

else if(islower(prod[k][2])) result(prod[k][2]); else first(prod[k][2]);

}

}

}

void result(char c)

{

int i;

for( i=0;i<=m;i++) if(res[i]==c)

if(res[i]==c) return; res[m++]=c;

}

## OUTPUT:

Enter the no.of productions: 8

enter the production string like E=E+T Enter productions Number 1 : E=TX Enter productions Number 2 : X=+TX Enter productions Number 3 : X=$ Enter productions Number 4 : T=FY Enter productions Number 5 : Y=\*FY Enter productions Number 6 : Y=$ Enter productions Number 7 : F=[E] Enter productions Number 8 : i

Find FOLLOW of -->X FOLLOW(X) = { $ ] }

Do you want to continue(Press 1 to continue )?1

Find FOLLOW of -->E FOLLOW(E) = { ] }

Do you want to continue(Press 1 to continue )?1

Find FOLLOW of -->Y FOLLOW(Y) = { + $ ] }

Do you want to continue(Press 1 to continue )?1

Find FOLLOW of -->T FOLLOW(T) = { $ ] }

Do you want to continue(Press 1 to continue )?1

Find FOLLOW of -->F FOLLOW(F) = { \* + $ ] }

Do you want to continue(Press 1 to continue )?2

## RESULT:

Program to find the FOLLOW of a given grammar is implemented and executed successfully.

## QUESTIONS:

1. Briefly explain the concept of derivation.
2. Define a context free grammar.
3. Define ambiguous grammar.
4. What are the problems with top down parsing?
5. Write the algorithm for FOLLOW

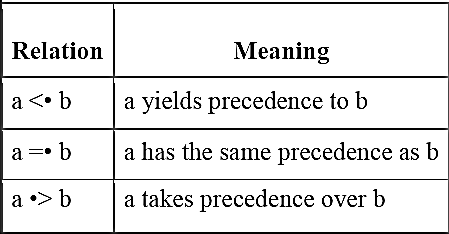
Experiment No. 11

# OPERATOR PRECEDENCE PARSER FOR A GIVEN LANGUAGE

## AIM:

Develop an operator precedence parser for a given language

## DESCRIPTION:

Operator precedence grammars rely on the following three precedence relations between the terminals:

These operator precedence relations allow delimiting the handles in the right sentential forms: <• marks the left end, =• appears in the interior of the handle, and •> marks the right end. Contrary to other shift-reduce parsers, all non-terminals are considered equal for the purpose of identifying handles. The relations do not have the same properties as their un-dotted counterparts; e. g. a =• b does not generally imply b =• a, and b •> a does not follow from a <• b. Furthermore, a =• a does not generally hold, and a •> a is possible.

Let us assume that between the terminals ai and ai+1 there is always exactly one precedence relation. Suppose that $ is the end of the string. Then for all terminals b we define: $ <• b and b •> $. If we remove all non-terminals and place the correct precedence relation: <•, =•, •> between the remaining terminals, there remain strings that can be analyzed by an easily developed bottom-up parser.

## Example:

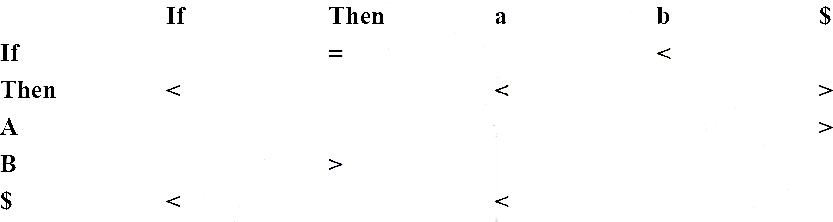
If the grammar is

S→if C then S|a C→b

Then LEADING(S)={if,a} LEADING(C)={b} TRAILING(S)=then,a}

TRAILING(C)={b}

## Precedence table:



**ALGORITHM:**

**LEADING:**

If a is in LEADING (A) if there is a production of the form A→ Ɣa𝛿 where Ɣ is є or a single non-terminal.

**TRAILING:**

If a is in TRAILING (B) if there is a production of the form A→ Ɣa𝛿 where 𝛿 is є or a single non-terminal.

**TABLE DESIGNING:**

Execute the following for each production A→x1x2x3….xn do

1. If xi and xi+1 are terminals then set xi=xi+1
2. If i<=n-2 and xi and xi+2 are terminals and xi+1 is a single non-terminal then set xi=xi+2
3. If xi is a terminal and xi+1 is a non-terminal then for all a in LEADING(xi+1) do set xi

<a

1. If xi is a non-terminal and xi+1 is a terminal then for all a in TRAILING(xi) do set a>xi+1
2. Set $<a for all a in LEADING(S)

And set b>$ for all b in TRAILING(S) where S is start symbol of G.

## PROGRAM:

#include<stdio.h> #include<string.h> int main()

{

char stack[20],ip[20],opt[10][10][1],ter[10]; int i,j,k,n,top=0,col,row;

for(i=0;i<10;i++)

{

stack[i]=NULL; ip[i]=NULL;

for(j=0;j<10;j++)

{

opt[i][j][0]=NULL;

}

}

printf("Enter the no.of terminals:"); scanf("%d",&n);

printf("\nEnter the terminals:"); scanf(" %s",ter);

printf("\nEnter the table values:\n"); for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

printf("Enter the value for %c %c:",ter[i],ter[j]); scanf(" %s",opt[i][j]);

}

}

printf("\nOPERATOR PRECEDENCE TABLE:\n");

for(i=0;i<n;i++)

{

printf("\t%c",ter[i]);

}

printf("\n"); for(i=0;i<n;i++)

{

printf("\n%c",ter[i]); for(j=0;j<n;j++)

{

printf("\t%c",opt[i][j][0]);

}

}

stack[top]='$';

printf("\nEnter the input string:"); scanf(" %s",ip);

i=0;

printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n"); printf("\n%s\t\t\t%s\t\t\t",stack,ip); while(i<=strlen(ip))

{

for(k=0;k<n;k++)

{

if(stack[top]==ter[k]) row=k;

if(ip[i]==ter[k]) col=k;

}

if((stack[top]=='$')&&(ip[i]=='$'))

{

printf("String is accepted"); break;

}

else if((opt[row][col][0]=='<') ||(opt[row][col][0]=='='))

{

stack[++top]=opt[row][col][0]; stack[++top]=ip[i]; printf("Shift %c",ip[i]);

i++;

}

else

{

if(opt[row][col][0]=='>')

{

while(stack[top]!='<')

--top; top=top-1;

printf("Reduce");

}

else

{

printf("\nString is not accepted"); break;

}

}

printf("\n"); for(k=0;k<=top;k++)

printf("%c",stack[k]); printf("\t\t\t"); for(k=i;k<strlen(ip);k++)

printf("%c",ip[k]); printf("\t\t\t");

}

}

## OUTPUT:

Enter the no.of terminals:4 Enter the terminals: i+\*$ Enter the table values:

Enter the value for \* \*:> Enter the value for \* $:> Enter the value for $ i:< Enter the value for $ +:< Enter the value for $ \*:< Enter the value for $ $:accept

\*\*\*\* OPERATOR PRECEDENCE TABLE \*\*\*\*

i + \* $

i e > > >

+ < > < >

\* < > > >

$ < < < a

\*/

Enter the input string:

i\*i

STACK INPUT STRING ACTION

|  |  |  |
| --- | --- | --- |
| $ | i\*i | Shift i |
| $<i | \*i | Reduce |
| $ | \*i | Shift \* |
| $<\* | i | Shift i |
| $<\*<i |  |  |

String is not accepted

Enter the no.of terminals:4

Enter the terminals:+\*i$ Enter the table values:

Enter the value for + +:> Enter the value for + \*:< Enter the value for + i:< Enter the value for + $:> Enter the value for \* +:> Enter the value for \* \*:> Enter the value for \* i:< Enter the value for \* $:> Enter the value for i +:> Enter the value for i \*:> Enter the value for i i:= Enter the value for i $:> Enter the value for $ +:< Enter the value for $ \*:< Enter the value for $ i:< Enter the value for $ $:a

OPERATOR PRECEDENCE TABLE:

+ \* i $

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| + | > | < | < | > |
| \* | > | > | < | > |
| i | > | > | = | > |
| $ | < | < | < | a |

Enter the input string:i+i\*i$

STACK INPUT STRING ACTION

|  |  |  |
| --- | --- | --- |
| $ | i+i\*i$ | Shift i |
| $<i | +i\*i$ | Reduce |
| $ | +i\*i$ | Shift + |
| $<+ | i\*i$ | Shift i |
| $<+<i | \*i$ | Reduce |
| $<+ | \*i$ | Shift \* |
| $<+<\* | i$ | Shift i |
| $<+<\*<i | $ | Reduce |
| $<+<\* | $ | Reduce |
| $<+ | $ | Reduce |
| $ | $ |  |

String is accepted

## RESULT:

Developed an operator precedence parser for a given language

## QUESTIONS:

1. What is top-down parsing?
2. What is an operator precedence parser?
3. List the advantages and disadvantages of operator precedence parsing.
4. List the properties of LR parser.
5. What is phrase level error recovery?
6. Briefly explain the concept of derivation.

Experiment No. 12

# SHIFT REDUCE PARSER

## AIM:

Implement of shift-reduce parsing algorithm.

## DESCRIPTION:

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols. During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the non-terminal occurring on the LHS of the rule. This shift- reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input.

The parser is nothing but a stack automaton which may be in one of several discrete states. A state is usually represented simply as an integer. In reality, the parse stack contains states, rather than grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

### The operation of the parser is controlled by a couple of tables:

**Action Table**: The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state s and the current lookahead terminal is *t*, the action taken by the parser depends on the contents of action[s][t], which can contain four different kinds of entries:

### Shift s'

Shift state s‟ onto the parse stack.

### Reduce r

Reduce by rule r. This is explained in more detail below.

### Accept

***Error***

Terminate the parse with success, accepting the input.

Signal a parse error.

# ALGORITHM:

Step 1 : Initialize the parse stack to contain a single state *s0*, where *s0* is the distinguished *initial state* of the parser.

Step 2 : Use the state *s* on top of the parse stack and the current lookahead *t* to consult the action table entry *action*[*s*][*t*]:

* 1. If the action table entry is *shift s'* then push state *s'* onto the stack and advance the input so that the lookahead is set to the next token.
  2. If the action table entry is *reduce r* and rule *r* has *m* symbols in its RHS, then pop *m* symbols off the parse stack. Let *s'* be the state now revealed on top of the parse stack and *N* be the LHS nonterminal for rule *r*. Then consult the goto table and push the state given by *goto*[*s'*][*N*] onto the stack. The lookahead token is not changed by this step.
  3. If the action table entry is *accept*, then terminate the parse with success.
  4. If the action table entry is *error*, then signal an error. Step 3 : Repeat step (2) until the parser terminates.

## PROGRAM:

#include<stdio.h> #include<stdlib.h> #include<string.h> #include<ctype.h>

char ip\_sym[15],stack[15]; int ip\_ptr=0,st\_ptr=0,len,i; char temp[2],temp2[2];

char act[15]; void check(); int main()

{

printf("\n\t\t SHIFT REDUCE PARSER\n"); printf("\n Grammer\n");

printf("\n E->E+E\n E->E/E"); printf("\n E->E\*E\n E->a/b"); printf("\n Enter the input symbol:\t"); gets(ip\_sym);

printf("\n\t Stack Implementation Table"); printf("\n Stack\t\t Input symbol\t\t Action"); printf("\n \t\t \t\t \n"); printf("\n $\t\t%s$\t\t\t--",ip\_sym); strcpy(act,"shift ");

temp[0]=ip\_sym[ip\_ptr]; temp[1]='\0'; strcat(act,temp); len=strlen(ip\_sym);

for(i=0;i<=len-1;i++)

{

stack[st\_ptr]=ip\_sym[ip\_ptr]; stack[st\_ptr+1]='\0'; ip\_sym[ip\_ptr]=' ';

ip\_ptr++;

printf("\n $%s\t\t%s$\t\t\t%s",stack,ip\_sym,act); strcpy(act,"shift ");

temp[0]=ip\_sym[ip\_ptr]; temp[1]='\0'; strcat(act,temp); check();

st\_ptr++;

}

check();

}

void check()

{

int flag=0; temp2[0]=stack[st\_ptr]; temp2[1]='\0'; if((isalpha(temp2[0])))

{

stack[st\_ptr]='E';

printf("\n $%s\t\t%s$\t\t\tE->%s",stack,ip\_sym,temp2); flag=1;

}

if((!strcmp(temp2,"+"))||(!strcmp(temp2,"\*"))||(!strcmp(temp2,"/"))

)

{

flag=1;

}

if((!strcmp(stack,"E+E"))||(!strcmp(stack,"E/E"))||(!strcmp(stack," E\*E")))

{

if(!strcmp(stack,"E+E"))

{

strcpy(stack,"E");

printf("\n $%s\t\t%s$\t\t\tE->E+E",stack,ip\_sym);

}

else if(!strcmp(stack,"E/E"))

{

strcpy(stack,"E");

printf("\n $%s\t\t %s$\t\t\tE->E/E",stack,ip\_sym);

}

else

{

strcpy(stack,"E");

printf("\n $%s\t\t%s$\t\t\tE->E\*E",stack,ip\_sym);

}

flag=1; st\_ptr=0;

}

if(!strcmp(stack,"E")&&ip\_ptr==len)

{

printf("\n $%s\t\t%s$\t\t\tACCEPT",stack,ip\_sym); exit(0);

}

if(flag==0)

{

printf("\n $%s\t\t%s$\t\t\tREJECT",stack,ip\_sym); exit(0);

}

}

## OUTPUT:

SHIFT REDUCE PARSER

Grammar

E->E+E E->E/E E->E\*E

E->a/b

Enter the input symbol: a+b\*c Stack Implementation Table

Stack Input Symbol Action

$ a+b\*c$ --

$a +b\*c$ shift a

$E +b\*c$ E->a

$E+ b\*c$ shift +

$E+b \*c$ shift b

$E+E \*c$ E->b

$E \*c$ E->E+E

$E\* c$ shift \*

$E\*c $ shift c

$E\*E $ E->c

$E $ E->E\*E

$E $ ACCEPT SHIFT REDUCE PARSER

GRAMMER

E->E+E E->E/E E->E\*E

E->a/b

Enter the input symbol: a+b\*+c

Stack Implementation Table

Stack Input Symbol Action

|  |  |  |
| --- | --- | --- |
| $  $a | a+b\*+c$  +b\*+c$ | --  shift a |
| $E | +b\*+c$ | E->a |
| $E+ | b\*+c$ | shift + |
| $E+b | \*+c$ | shift b |
| $E+E | \*+c$ | E->b |
| $E | \*+c$ | E->E+E |
| $E\* | +c$ | shift \* |
| $E\*+ | c$ | shift + |
| $E\*+c | $ | shift c |
| $E\*+E | $ | E->c |
| $E\*+E | $ | REJECT |

## RESULT:

Shift-reduce parsing algorithm has been implemented and verified the output.

## QUESTIONS:

1. What type parser is SR parser?
2. What is SHIFT and REDUCE?
3. What is ACCEPT?
4. List the properties of LR parser.
5. Mention the types of LR parser.

Experiment No. 13

# CONSTANT PROPAGATION

**AIM:**

Write a program for constant propagation.

## DESCRIPTION:

Constant Propagation is one of the local code optimization technique in Compiler Design. It can be defined as the process of replacing the constant value of variables in the expression. Constant propagation is executed using reaching definition analysis results in compilers, which means that if reaching definition of all variables has same assignment which assigns a same constant to the variable, then the variable has a constant value and can be substituted with the constant. Also, Constant propagation reduces the number of cases where values are directly copied from one location or variable to another, in order to simply allocate their value to another variable.

## PROGRAM:

#include<stdio.h> #include<string.h> #include<ctype.h> #include<stdlib.h> struct expr

{

char op[2],op1[5],op2[5],res[5]; int flag;

}arr[10]; int n;

void input()

{

int i;

printf("\n\nEnter the maximum number of expressions : "); scanf("%d",&n);

printf("\nEnter the input : \n"); for(i=0;i<n;i++)

{

scanf("%s",arr[i].op);

scanf("%s",arr[i].op1);

scanf("%s",arr[i].op2);

scanf("%s",arr[i].res); arr[i].flag=0;

}

}

void change(int p,char \*res)

{

int i; for(i=p+1;i<n;i++)

{

if(strcmp(arr[p].res,arr[i].op1)==0) strcpy(arr[i].op1,res);

else if(strcmp(arr[p].res,arr[i].op2)==0) strcpy(arr[i].op2,res);

}

}

void constant()

{

int i;

int op1,op2,res; char op,res1[5]; for(i=0;i<n;i++)

{

if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"=")==0)

{

op1=atoi(arr[i].op1); op2=atoi(arr[i].op2); op=arr[i].op[0]; switch(op)

{

case '+': res=op1+op2; break;

case '-': res=op1-op2; break;

case '\*': res=op1\*op2; break;

case '/': res=op1/op2; break;

case '=': res=op1; break;

}

sprintf(res1,"%d",res); arr[i].flag=1; change(i,res1);

}

}

}

void output()

{

int i=0;

printf("\nOptimized code is : "); for(i=0;i<n;i++)

{

if(!arr[i].flag)

{

printf("\n%s %s %s %s", arr[i].op, arr[i].op1,arr[i].op2, arr[i].res);

}

}

}

void main()

{

input(); constant(); output();

}

## OUTPUT:

Enter the maximum number of expressions : 4

Enter the input :

= 3 . a

+ a b t1

+ a c t2

+ t1 t2 t3

Optimized code is :

+ 3 b t1

+ 3 c t2

+ t1 t2 t3

## RESULT:

Program for constant propagation is implemented successfully.

## QUESTIONS:

1. What is constant propagation?
2. What is code motion?
3. What are the properties of optimizing compiler?
4. What is meant by Dead code? Give an example.
5. Mention the issues to be considered while applying the techniques for code optimization.

Experiment No. 14

**AIM:**

# BACK END OF THE COMPILER

Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions

## DESCRIPTION:

Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly language instructions can be simple move, add, sub, jump etc.

The main phases of the back end include the following:

* **Analysis:** This is the gathering of program information from the intermediate representation derived from the input; data-flow analysis is used to build use-define chains, together with dependence analysis, alias analysis, pointer analysis, escape analysis etc.
* **Optimization**: The intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are expansion, dead, constant, propagation, loop transformation, register allocation and even automatic parallelization.
* **Code generation**: The transformed language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated modes. Debug data may also need to be generated to facilitate debugging.

## ALGORITHM :

Step 1. Start

Step 2. Open the source file and store the contents as quadruples.

Step 3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.

Step 4. Write the generated code into output Step 6. Stop

## PROGRAM:

#include<stdio.h> #include<stdio.h> #include<string.h> void main()

{

char icode[10][30],str[20],opr[10]; int i=0;

printf("\n Enter the set of intermediate code (terminated by exit):\n"); do

{

scanf("%s",icode[i]);

}while(strcmp(icode[i++],"exit")!=0);

printf("\n Target Code Generation"); printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"); i=0;

do

{

strcpy(str,icode[i]); switch(str[3])

{

case '+':

case '-':

case '\*':

case '/':

}

strcpy(opr,"ADD"); break;

strcpy(opr,"SUB"); break;

strcpy(opr,"MUL"); break;

strcpy(opr,"DIV"); break;

printf("\n\tMov %c,R%d",str[2],i);

printf("\n\t%s%c,R%d",opr,str[4],i);

printf("\n\tMov R%d,%c",i,str[0]);

}while(strcmp(icode[++i],"exit")!=0);

}

## OUTPUT:

Enter the set of intermediate code (terminated by exit): a=a\*b

c=f\*h g=a\*h f=Q+w t=q-j exit

Target Code Generation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Mov a,R0

MUL b,R0

Mov R0,a Mov f,R1 MUL h,R1

Mov R1,c Mov a,R2 MUL h,R2

Mov R2,g Mov Q,R3 ADD w,R3

Mov R3,f Mov q,R4 SUB j,R4

Mov R4,t

## RESULT:

Implemented a back end of the compiler which takes the three address code and produces the 8086 assembly language instructions.

## QUESTIONS:

* 1. Write intermediate code for a=b+c\*d.
  2. Mention The Back-end Phases of A Compiler.
  3. Define code optimization.
  4. List out the criteria for code improving transformations.
  5. What are the basic goals of code movement?