**(SUBJECT TITLE)**

**PRACTICAL**

(for which semester of B.Tech)

As per the curricullam and syllabus

of

**Bharath Institute of Higher Education & Research**

**(Subject Lab Manual)**



**PREPARED BY**

**PROF**

**ASSISTANT PROFESSOR**

NEW EDITION

**COMPILER DESIGN LABORATORY**

( VI semester of B.Tech)

As per the curricullam and syllabus

of

**Bharath Institute of Higher Education & Research**

**COMPILER DESIGN LAB MANUAL**



**PREPARED BY**

**DR. S.** **MEERA**

**M. SUSMITRA**

**(SUBJECT TITLE)**

**PRACTICAL**

(for which semester of B.Tech)

As per the curricullam and syllabus

of

**Bharath Institute of Higher Education & Research**

**(Subject Lab Manual)**



**PREPARED BY**

**PROF**

**ASSISTANT PROFESSOR**

**(SUBJECT TITLE)**

**PRACTICAL**

(for which semester of B.Tech)

As per the curricullam and syllabus

of

**Bharath Institute of Higher Education & Research**

**(Subject Lab Manual)**

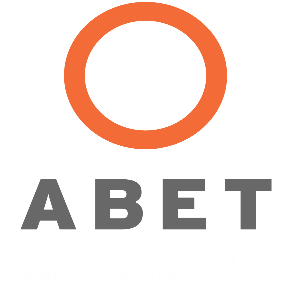


**PREPARED BY**

**PROF**

**ASSISTANT PROFESSOR**







**SCHOOL OF COMPUTING**

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**LAB MANUAL**

**SUBJECT NAME:COMPILER DESIGN LAB**

**SUBJECT CODE:U18PCCS6L1**

**R 2018**

**(2020-2021)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **U18PCCS6L1** | **COMPILER DESIGN LABORATORY** | **L** | **T** | **P** | **C** |
| Total Contact Hours - 45 | 0 | 0 | 3 | 2 |
| Prerequisite – Compiler Design, C Programming | | | | |
| Lab Manual Designed by – Dept. of Computer Science and Engineering. | | | | |
| **OBJECTIVES**  This laboratory course is intended to make the students experiment on the basic techniques of compiler construction and tools that can used to perform syntax-directed translation of a high-level programming language into an executable code. Students will design and implement language processors in C by using tools to automate parts of the implementation process. This will provide deeper insights into the more advanced semantics aspects of programming languages, code generation, machine independent optimizations, dynamic memory allocation, and object  orientation. | | | | | |

|  |  |
| --- | --- |
| **COURSE OUTCOMES (COs)** | |
| CO1 | To Understand the working of lex and yacc compiler for debugging of  programs. |
| CO2 | To Understand and define the role of lexical analyzer, use of regular expression  and transition diagrams. |
| CO3 | To Understand and use Context free grammar, and parse tree construction. |
| CO4 | To Learn & use the new tools and technologies used for designing a compiler. |
| CO5 | To Develop program for solving parser problems. |
| CO6 | To generate machine code from abstract syntax tree generated by the parser. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **MAPPING BETWEEN COURSE OUTCOMES & PROGRAM OUTCOMES (3/2/1 INDICATES STRENGTH OF CORRELATION) 3- High, 2- Medium, 1-Low** | | | | | | | | | | | | | | | |
| **COs** | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** | **PSO1** | **PSO2** | **PSO3** |
| **CO1** | 3 | 3 | 3 |  | 2 | 2 |  |  |  |  |  |  | 2 | 3 |  |
| **CO2** | 3 | 3 | 3 |  | 2 | 2 |  |  |  |  |  |  |  | 3 |  |
| **CO3** | 3 | 3 | 3 |  | 2 | 2 |  |  |  |  |  |  |  | 3 |  |
| **CO4** | 3 | 3 | 3 |  | 2 | 2 |  |  |  |  |  |  | 2 | 3 |  |
| **CO5** | 3 | 3 | 3 |  | 2 | 2 |  |  |  |  |  |  |  | 3 |  |
| **CO6** | 3 | 3 | 3 |  | 2 | 2 |  |  |  |  |  |  |  | 3 |  |
| Category | | Professional core (PC) | | | | | | | | | | | | | |
| Approval | | 47th Academic Council Meeting held in Aug, 2018 | | | | | | | | | | | | | |

**COMPILER DESIGN LAB]-[U18PCCS6L1**

**LIST OF EXPERIMENTS**

|  |  |
| --- | --- |
|  | **NAME OF THE EXPERIMENT** |
| 1 | Design a lexical analyzer for given language and the lexical analyzer |
| 2 | Write a c program to recognize strings under 'a', 'a\*b+', 'abb'. |
| 3 | Write a c program to test whether a given identifier is valid or not. |
| 4 | Write a c program to simulate lexical analyzer for validating operators. |
| 5 | Implement the lexical analyzer using jlex, flex or other lexical analyzer generating tools |
| 6 | Write a c program for constructing of ll (1) parsing. |
| 7 | Write a c program to implement lalr parsing |
| 8 | Write a c program to implement operator precedence parsing. |
| 9 | Convert the bnf rules into yacc form and write code to generate abstract syntax tree for the mini language specified in note 1. |
| 10 | Simulation of basic memory mulation of basic memory management schemes |
| 11 | Simulation of virtual memory management schemes |
| 12 | Simulation of file system |

CONTENT

|  |  |  |
| --- | --- | --- |
| S.NO | NAME OF THE EXPERIMENT | PAGE NO |
| 1 | Design a lexical analyzer for given language and the lexical analyzer | 6 |
| 2 | Write a c program to recognize strings under 'a', 'a\*b+', 'abb'. | 11 |
| 3 | Write a c program to test whether a given identifier is valid or not. | 14 |
| 4 | Write a c program to simulate lexical analyzer for validating operators. | 16 |
| 5 | Implement the lexical analyzer using jlex, flex or other lexical analyzer generating tools | 18 |
| 6 | Write a c program for constructing of ll (1) parsing. | 20 |
| 7 | Write a c program to implement lalr parsing | 22 |
| 8 | Write a c program to implement operator precedence parsing. | 25 |
| 9 | Convert the bnf rules into yacc form and write code to generate abstract syntax tree for the mini language specified in note 1. | 29 |
| 10 | Simulation of basic memory mulation of basic memory management schemes | 32 |
| 11 | Simulation of virtual memory management schemes | 35 |
| 12 | Simulation of file system | 38 |

**Ex. No 1** **DESIGN A LEXICAL ANALYZER FOR GIVEN LANGUAGE AND THE LEXICAL ANALYZER**

**Aim:**

Write a C/C++ program to implement the design of a Lexical analyzer to recognize

the tokens defined by the given grammar.

# Algorithm:

We make use of the following two functions in the process.

look up() – it takes string as argument and checks its presence in the symbol table. If the string is found then returns the address else it returns NULL.

insert() – it takes string as its argument and the same is inserted into the symbol table and the corresponding address is returned.

1. Start

2. Declare an array of characters, an input file to store the input;

3. Read the character from the input file and put it into character type of variable, say ‘c’.

4. If ‘c’ is blank then do nothing.

5. If ‘c’ is new line character line=line+1.

6. If ‘c’ is digit, set token Val, the value assigned for a digit and return the ‘NUMBER’.

7. If ‘c’ is proper token then assign the token value.

8. Print the complete table with

Token entered by the user,

Associated token value.

9. Stop

# 

# Program:

#include<string.h>

#include<ctype.h>

#include<stdio.h>

void keyword(char str[10])

{

if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int",str )==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("static",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)

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

else

printf("\n%s is an identifier",str);

}

main()

{ 8

FILE \*f1,\*f2,\*f3;

char c, str[10], st1[10];

int num[100], lineno=0, tokenvalue=0,i=0,j=0,k=0;

printf("\n Enter the c program : ");/\*gets(st1);\*/

f1=fopen("input","w");

while((c=getchar())!=EOF)

putc(c,f1);

fclose(f1);

f1=fopen("input","r");

f2=fopen("identifier","w");

f3=fopen("specialchar","w");

while((c=getc(f1))!=EOF)

{

if(isdigit(c))

{

tokenvalue=c-'0';

c=getc(f1);

while(isdigit(c))

{

tokenvalue\*=10+c-'0';

c=getc(f1);

}

num[i++]=tokenvalue;

ungetc(c,f1);

}

else

if(isalpha(c))

{

putc(c,f2);

c=getc(f1);

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

{

putc(c,f2);

c=getc(f1);

}

putc(' ',f2); 9

ungetc(c,f1);

}

else

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

printf(" ");

else

if(c=='\n')

lineno++;

else

putc(c,f3);

}

fclose(f2);

fclose(f3);

fclose(f1);

printf("\n The no's in the program are :");

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

printf("%d", num[j]);

printf("\n");

f2=fopen("identifier", "r");

k=0;

printf("The keywords and identifiers are:");

while((c=getc(f2))!=EOF)

{

if(c!=' ')

str[k++]=c;

else

{

str[k]='\0';

keyword(str);

k=0;

}

}

fclose(f2);

f3=fopen("specialchar","r");

printf("\n Special characters are : ");

while((c=getc(f3))!=EOF) 10

printf("%c",c);

printf("\n");

fclose(f3);

printf("Total no. of lines are:%d", lineno);

}

**Output 1:**

Enter the C program: a+b\*c

Ctrl-D

The no’s in the program are:

The keywords and identifiers are:

a is an identifier and terminal

b is an identifier and terminal

c is an identifier and terminal

Special characters are:

+ \*

Total no. of lines are: 1

# Result:

Thus program to design a lexical analyzer for given language and the lexical analyzer is executed successfully.

**Ex. No 2**  **WRITE A C PROGRAM TO RECOGNIZE STRINGS UNDER 'A', 'A\*B+', 'ABB'.**

# AIM:

To Write a C program to recognize strings under 'a', 'a\*b+', 'abb'

# ALGORITHM:

1. By using transition diagram we verify input of the state.
2. If the state recognize the given pattern rule.
3. Then print string is accepted under a\*/ a\*b+/ abb.
4. Else print string not accepted.

# PROGRAM:

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

void main()

{

char s[20],c;

int state=0,i=0;

printf("\n Enter a string:");

gets(s);

while(s[i]!='\0')

{

switch(state)

{

case 0: c=s[i++];

if(c=='a')

state=1;

else if(c=='b')

state=2;

else

state=6;

break;

case 1: c=s[i++];

if(c=='a')

state=3;

else if(c=='b')

state=4;

else

state=6;

break;

case 2: c=s[i++];

if(c=='a')

state=6;

else if(c=='b')

state=2;

else

state=6;

break;

case 3: c=s[i++];

if(c=='a')

state=3;

else if(c=='b')

state=2;

else

state=6;

break;

case 4: c=s[i++];

if(c=='a')

state=6;

else if(c=='b')

state=5;

else

state=6;

break;

case 5: c=s[i++];

if(c=='a')

state=6;

else if(c=='b')

state=2;

else

state=6;

break;

case 6: printf("\n %s is not recognised.",s);

exit(0);

}

}

if(state==1)

printf("\n %s is accepted under rule 'a'",s);

else if((state==2)||(state==4))

printf("\n %s is accepted under rule 'a\*b+'",s);

else if(state==5)

printf("\n %s is accepted under rule 'abb'",s);

getch();

}

**OUTPUT:**

Enter a String: aaaabbbbb

aaaabbbbb is accepted under rule 'a\*b+'

Enter a string: cdgs

cdgs is not recognized

# RESULT:

Thus the C program to recognize strings under 'a', 'a\*b+', 'abb' is successfully executed.

**Ex. No 3** **WRITE A C PROGRAM TO TEST WHETHER A GIVEN IDENTIFIER IS VALID OR NOT.**

**Aim:**

To Write a C program to test whether a given identifier is valid or not.

# Algorithm:

1. Read the given input string.
2. Check the initial character of the string is numerical or any special character except ‘\_’ then print it is not a valid identifier.
3. Otherwise print it as valid identifier if remaining characters of string doesn’t contains any special characters except ‘\_

**Program:**

#include <stdio.h>

#include <conio.h>

void main()

{

int i=0,flag=0;

char keyw[10][10]={"int","float","break","long","char","for","if","switch","else","while"},a[10];

printf("Enter Identifier : ");

gets(a);

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

{

if((strcmp(keyw[i],a)==0))

{

flag=1;

}

}

if(flag==1)

{

printf("\n%s is Keyword.",a);

}

else

{

flag=0;

if((a[0]=='\_')||(isalpha(a[0])!=0))

{

for(i=1;a[i]!='\0';i++)

{

if((isalnum(a[i])==0)&&(a[i]!='\_'))

{

flag=1;

}

}

}

else

{

flag=1;

}

}

if(flag==0)

{

printf("\n%s is an Identifier.",a);

}

else

{

printf("\n%s is Not an Identifier.",a);

}

}

**OUTPUT:**

Enter an identifier: first

Valid identifier

Enter an identifier:1aqw

Not a valid identifier

# Result:

Thus, the Program to test whether a given identifier is valid or not.

**Ex. No: 4**  **WRITE A C PROGRAM TO SIMULATE LEXICAL ANALYZER FOR VALIDATING OPERATORS.**

**Aim:**

To Write a C program to test whether a given identifier is valid or not

# Algorithm:

1. Read the given input.
2. If the given input matches with any operator symbol.
3. Then display in terms of words of the particular symbol.
4. Else print not a operator.

**Program :**

#include<stdio.h>

#include<conio.h>

void main()

{

char s[5];

printf("\n Enter any operator:");

gets(s);

switch(s[0])

{

case'>': if(s[1]=='=')

printf("\n Greater than or equal");

else

printf("\n Greater than");

break;

case'<': if(s[1]=='=')

printf("\n Less than or equal");

else

printf("\nLess than");

break;

case'=': if(s[1]=='=')

printf("\nEqual to");

else

printf("\nAssignment");

break;

case'!': if(s[1]=='=')

printf("\nNot Equal");

else

printf("\n Bit Not");

break;

case'&': if(s[1]=='&')

printf("\nLogical AND");

else

printf("\n Bitwise AND");

break;

case'|': if(s[1]=='|')

printf("\nLogical OR");

else

printf("\nBitwise OR");

break;

case'+': printf("\n Addition");

break;

case'-': printf("\nSubstraction");

break;

case'\*': printf("\nMultiplication");

break;

case'/': printf("\nDivision");

break;

case'%': printf("Modulus");

break;

default: printf("\n Not a operator");

}

}

**OUTPUT:**

Enter any operator: \*

Multiplication

# Result:

Thus the C program to test whether a given identifier is valid or not is executed successfully.

**Ex. No: 5 IMPLEMENT THE LEXICAL ANALYZER USING JLEX, FLEX OR OTHER LEXICAL ANALYZER GENERATING TOOLS**

**Aim:**

Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.

# Algorithm:

1. Read the given input.
2. If the given input matches with any operator symbol.
3. Then display in terms of words of the particular symbol.
4. Else print not a operator.

**Program:**

{

/\* program name is lexp.l \*/

%{ /\* program to recognize a c program

\*/ int COMMENT=0;

%}

identifier [a-zA-Z][a-zA-Z0-9]\* %% #.\*

{ printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);

} int |float |char |double |while |for |do |if |break |continue |void |switch |case |long |struct |const |typedef |return |else |goto {printf("\n\t%s is a KEYWORD",yytext);} "/\*" {COMMENT = 1;} /\*{printf("\n\n\t%s is a COMMENT\n",yytext);}

\*/ "\*/" {COMMENT = 0;} /\* printf("\n\n\t%s is a COMMENT\n",yytext);}\*/ {

Identifier

}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);} { {if(!COMMENT) printf("\n BLOCK BEGINS");} } {if(!COMMENT) printf("\n BLOCK ENDS");}

{identifier}(\[[0-9]\*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);} ".\*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}

[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);} {

if(!COMMENT) printf("\n\t");ECHO;printf("\n");} ( ECHO; {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);

}<= |>= |< |== |> {

if(!COMMENT)

printf("\n\t%s is a RELATIONAL OPERATOR",yytext);} %% int main(int argc,char \*\*argv) { if (argc > 1)

{

FILE \*file; file = fopen(argv[1],"r"); if(!file) { printf("could not open %s \n",argv[1]); exit(0); } yyin = file; } yylex(); 12

printf("\n\n"); return 0; } int yywrap() { return 0; }

else {

printf(“\n the child process ID is %d\n”, pid2); pid2=getpid();

int main()

{

    if (fork()== 0)

        printf("HC: hello from child\n");

    else

    {

        printf("HP: hello from parent\n");

        wait(NULL);

        printf("CT: child has terminated\n");

    }

    printf("Bye\n");

    return 0;

}

**Output**

$lex lex.l $cc lex.yy.c $./a.out var.c

#include<stdio.h> is a PREPROCESSOR DIRECTIVE FUNCTION main ( ) BLOCK BEGINS int is a KEYWORD a IDENTIFIER b IDENTIFIER BLOCK ENDS

# Result:

Thus the program implements the lexical analyzer using JLex, flex or other lexical analyzer generating tools.

is executed and verified successfully.

# Ex. No 6

**DATE: WRITE A C PROGRAM FOR CONSTRUCTING OF LL (1) PARSING.**

**Aim:**

# Write a C program for constructing of LL (1) parsing.

# Algorithm:

Read the input string.

Using predictive parsing table parse the given input using stack .

If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to $.

Go to debug -> run or press CTRL + F9 to run the program.

**Program:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

char s[20],stack[20];

void main()

{

char m[5][6][3]={"tb"," "," ","tb"," "," "," ","+tb"," "," ","n","n","fc"," "," ","fc"," "," "," ","n","\*fc"," a ","n","n","i"," "," ","(e)"," "," "};

int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};

int i,j,k,n,str1,str2; clrscr();

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

scanf("%s",s);

strcat(s,"$");

n=strlen(s);

stack[0]='$';

stack[1]='e';

i=1; j=0;

printf("\nStack Input\n");

printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

while((stack[i]!='$')&&(s[j]!='$'))

{

if(stack[i]==s[j])

{

i--; j++;

}

switch(stack[i])

{

case 'e': str1=0; break; case 'b': str1=1; break; case 't': str1=2; break;

case 'c': str1=3; break; case 'f': str1=4; break;

}

switch(s[j])

{

case 'i': str2=0; break; case '+': str2=1;

break;

case '\*': str2=2; break; case '(': str2=3; break; case ')': str2=4; break; case '$': str2=5; break;

}

if(m[str1][str2][0]=='\0')

{

printf("\nERROR"); exit(0);

}

else if(m[str1][str2][0]=='n') i--;

else if(m[str1][str2][0]=='i')

stack[i]='i';

else

{

for(k=size[str1][str2]-1;k>=0;k--)

{

stack[i]=m[str1][str2][k]; i

} i--;

}

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

printf(" %c",stack[k]); printf(" ");

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

printf("%c",s[k]); printf(" \n ");

}

printf("\n SUCCESS"); getch();

}

**INPUT & OUTPUT:**

Enter the input string:

i\*i+i Stack INPUT $bt i\*i+i$ $bcf i\*i+i$ $bci i\*i+i$ $bc \*i+i$ $bcf\* \*i+i$ $bcf i+i$ $bci i+i$ $bc +i$ $b +i$ $bt+ +i$ $bt i$ $bcf i$ $ bci i$ $bc $ $b $ $ $ success

**Result:**

Thus the program to write a c program for constructing of ll (1) parsing is executed successfully.

**EX. NO 7 WRITE A C PROGRAM TO IMPLEMENT LALR PARSING**

# DATE:

**Aim :**

To Design and implement an LALR bottom up Parser for checking the syntax of the

Statements in the given language.

# Algorithm:

Read the input string.

Push the input symbol with its state symbols in to the stack by referring lookaheads

We perform shift and reduce actions to parse the grammar.

Parsing is completed when we reach $ symbol.

Go to debug -> run or press CTRL + F9 to run the program.

**Program:**

<parser.l>

%{

#include<stdio.h>

#include "y.tab.h"

%}

%%

[0-9]+ {yylval.dval=atof(yytext);

return DIGIT;

}

\n|. return yytext[0];

%%

<parser.y>

%{

/\*This YACC specification file generates the LALR parser for the program

considered in experiment 4.\*/

#include<stdio.h>

%}

%union

{

double dval;

}

%token <dval> DIGIT

%type <dval> expr

%type <dval> term

%type <dval> factor

%%

line: expr '\n' {

;

printf("%g\n",$1);

}

expr: expr '+' term {$$=$1 + $3 ;}

| term

;

term: term '\*' factor {$$=$1 \* $3 ;}

| factor

; 20

factor: '(' expr ')' {$$=$2 ;}

| DIGIT

;

%%

int main()

{

yyparse();

}

yyerror(char \*s)

{

printf("%s",s);

}

**Output 1:**

$lex parser.l

$yacc –d parser.y

$cc lex.yy.c y.tab.c –ll –lm

$./a.out

2+3

5.0000

# Result:

Thus the program for implementing the ls command and grep command was executed and the output was verified successfully.

**EX. NO 8 WRITE A C PROGRAM TO IMPLEMENT OPERATOR PRECEDENCE PARSING.**

**Aim:**

To write a C program to perform Stack Implementation of Operator Precedence Parser.

# Algorithm:

A string w and a table of precedence relations and functions for grammar G.

If w is well formed, a skeletal parse tree, with a place holder non terminal E labeling all interior nodes; otherwise an error indication.

**Program:**

**Operator precedence parser for the grammar**

**E->E+E | E-E | E\*E | E/E | E^E | (E) | a**

#include<stdio.h>

#include<conio.h>

int f(char x)

{

int p;

switch(x)

{

case '+': p=2;break;

case '-': p=2;break;

case '\*': p=4;break;

case '/': p=4;break;

case '^': p=4;break;

case '(': p=0;break;

case ')': p=6;break;

case 'a': p=6;break;

case '$': p=0;break;

}

return p;

}

int g(char x)

{

int q;

switch(x)

{

case '+': q=1;break;

case '-': q=1;break;

case '\*': q=3;break;

case '/': q=3;break;

case '^': q=5;break;

case '(': q=5;break;

case ')': q=0;break;

case 'a': q=5;break;

case '$': q=0;break;

}

return q;

}

void main()

{

char stack[20],input[20];

int i,j,l,x,y;

char pop;

clrscr();

printf("Enter the input string to be parsed : ");

scanf("%s",input);

l=strlen(input);

input[l]='$';

input[l+1]='\0';

stack[0]='$';

stack[1]='\0';

i=0;j=0;

printf("\n\n");

printf("------------------------------------------\n");

printf("STACK\tINPUT\tACTION\n");

printf("------------------------------------------\n");

printf("%s\t%s\tINITIAL CONFIGURATION\n",stack,input);

while(!(stack[i]=='$'&&input[0]=='$'))

{

if((stack[i]=='a')&&(input[0]=='a'))

{

printf("Not a sentence\n");

goto a;

}

else if(stack[i]=='a'&&input[0]=='(')

{

printf("Not a sentence\n");

goto a;

}

else if(stack[i]==')'&&input[0]=='a')

{

printf("Not a Sentence\n");

goto a;

}

else if(stack[i]==')'&&input[0]=='(')

{

printf("Not a Sentence\n");

goto a;

}

else if(stack[i]=='('&&input[0]=='$')

{

printf("Not a Sentence\n");

goto a;

}

else if(stack[i]=='$'&&input[0]==')')

{

printf("Not a Sentence\n");

goto a;

}

else

{

if(f(stack[i])<g(input[0])||f(stack[i])==g(input[0]))

{

i++;

stack[i]=input[0];

stack[i+1]='\0';

l=strlen(input);

for(j=0;j<l-1;j++)

{

input[j]=input[j+1];

}

input[l-1]='\0';

printf("%s\t%s\tSHIFT\n",stack,input);

}

else

{

do

{

pop=stack[i];

i--;

}while(!(f(stack[i])<g(pop)));

stack[i+1]='\0';

printf("%s\t%s\tREDUCE\n",stack,input);

}

}

}

printf("ACCEPT\n");

a:

getch();

}

# Result:

Thus, the program perform Stack Implementation of Operator Precedence Parser is executed successfully.

**EX. NO 9 CONVERT THE BNF RULES INTO YACC FORM AND WRITE CODE TO GENERATE ABSTRACT SYNTAX TREE FOR THE MINI LANGUAGE SPECIFIED IN NOTE 1.**

**Aim:**

To Implement the process of conversion from BNF rules to Yacc form and generate Abstract Syntax Tree.

Algorithm:

STEP 1: Start the program.

STEP 2: Declare the required variables.

STEP 3: Initialize the buffer size and get maximum item you want to produce.

STEP 4: Get the option, which you want to do either producer, consumer or exit from the operation.

STEP 5: If you select the producer, check the buffer size if it is full the producer should not produce the item or otherwise produce the item and increase the value buffer size.

STEP 6: If you select the consumer, check the buffer size if it is empty the consumer should not consume the item or otherwise consume the item and decrease the value of buffer size.

STEP 7: If you select exit come out of the program.

STEP 8: Stop the program.

**Program**:

#include<stdio.h>

int mutex=1,full=0,empty=3,x=0; main()

{

int n; void producer();

void consumer();

int wait(int); int signal(int); printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");

while(1)

{

printf("\nENTER YOUR CHOICE\n"); scanf("%d",&n); switch(n)

{

case 1: if((mutex==1)&&(empty!=0)) producer(); else printf("BUFFER IS FULL");

break;

case 2: if((mutex==1)&&(full!=0)) consumer(); else printf("BUFFER IS EMPTY");

break;

case 3: exit(0); break;

}

}

}

int wait(int s)

{

return(--s);

}

int signal(int s)

{

return(++s);

}

void producer()

{

mutex=wait(mutex); full=signal(full);

empty=wait(empty);

x++; printf("\nproducer produces the item%d",x); mutex=signal(mutex);

}

void consumer()

{

mutex=wait(mutex); full=wait(full); empty=signal(empty);

printf("\n consumer consumes item%d",x); x--; mutex=signal(mutex);

}

# Output:

# Result:

# 

# Thus, the program to synchronization problems using Semaphore has been executed successfully

# Ex. No: 10 SIMYLATION OF BASIC MEMORY MULATION OF BASIC MEMORY MANAGEMENT SCHEMES

**Aim:**

To write a C program to implement memory management using paging technique.

# Algorithm:

STEP 1 : Start the program.

STEP 2 : Read the base address, page size, number of pages and memory unit.

STEP 3 : If the memory limit is less than the base address display the memory

limit is less than limit.

STEP 4 : Create the page table with the number of pages and page address.

STEP 5 : Read the page number and displacement value.

STEP 6 : If the page number and displacement value is valid, add the displacement

value with the address corresponding to the page number and display the result.

STEP 7 : Display the page is not found or displacement should be less than page

size.

STEP 8 : Stop the program.

**Program :**

#include<stdio.h>

#include<unistd.h>

void main()

{

int b[20],n,i,pa,p,a,d;

printf(“\nProgram for paging”);

scanf(“%d”,&n);

printf(“\nEnter the base address:”);

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

{

scanf(“%d”,&b[i]);

}

printf(“\nEnter the logical address:”);

scanf(“%d”,&p);

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

{

if(i==p)

{

pa=b[i]+d;

a=b[i];

printf(“\n\tPageNo.\t BaseAdd. PhysicalAdd. \n\t %d \t %d \t %d \t ”,p,a,pa);

# }

}

printf(“\nInvalid page”);

# }

# Output:

Program for paging

Enter the number of pages:2

Enter the base address:

100

150

Enter the Logical address:50

# Enter the page number:1

PageNo. BaseAdd. PhysicalAdd.

1 150 200

Program for paging

Enter the number of pages:1

Enter the base address:

100

Enter the Logical address:2

Enter the page number:2

Invalid page.

# Result:

Thus the program for implementing the paging concept was executed and the output was verified successfully.

# Ex. No:11 SIMULATION OF VIRTUAL MEMORY MANAGEMENT SCHEMES

**Aim:**

To write a C program to implement memory management using segmentation

# Algorithm:

STEP 1 : Start the program.

STEP 2 : Read the base address, number of segments, size of each segment, memory limit.

STEP 3 : If memory address is less than the base address display “invalid memory limit”.

STEP 4 : Create the segment table with the segment number and segment address and display it.

STEP 5 : Read the segment number and displacement.

STEP 6 : If the segment number and displacement is valid compute the real address and display the same.

STEP 7 : Stop the program.

**Program:**

#include<stdio.h>

#include<unistd.h>

void main()

{

int b[20],l[20],n,i,pa,s,a,d;

printf(“\nProgram for segmentation”);

printf(“\nEnter the number of segments:”);

scanf(“%d”,&n);

printf(“\nEnter the base address and limit register:”);

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

{

scanf(“%d”,&b[i]);

scanf(“%d”,&l[i])

}

printf(“\nEnter the logical address:”);

scanf(“%d”,&d);

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

{

if(i==s)

{

if(d<l[i])

{

pa=b[i]+d;

a=b[i];

printf(“(“\n\tPageNo.\t BaseAdd. PhysicalAdd. \n\t %d \t %d \t %d \t ”,s,a,pa);

exit(0);

}

else

{

printf(“\nPage size exceeds”);

exit(0);

}

}

}

printf(“\nInvalid segment”);

}

# Output:

Program for segmentation

Enter the number of segments:3

Enter the base address and limit register:

100 50

150 20

130 34

Enter the Logical address:25

Enter the segment number:1

PageNo. BaseAdd. PhysicalAdd.

2 130 155

# Result:

Thus the program for implementing the segmentation concept was executed and the output was verified successfully.

# Ex. No 12 SIMULATION OF FILE SYSTEM

# Date:

**Aim:**

To write C program to organize the file using single level directory.

# Algorithm:

Read the input string.

Push the input symbol with its state symbols in to the stack by referring lookaheads

We perform shift and reduce actions to parse the grammar.

Parsing is completed when we reach $ symbol.

Go to debug -> run or press CTRL + F9 to run the program.

**Program:**

**<int.l>**

%{

#include"y.tab.h"

#include<stdio.h>

#include<string.h>

int LineNo=1;

% }

identifier [a-zA-Z][\_a-zA-Z0-9]\*

number [0-9]+|([0-9]\*\.[0-9]+)

%%

main\(\) return MAIN;

if return IF;

else return ELSE;

while return WHILE;

int |

char |

float return TYPE;

{identifier} {strcpy(yylval.var,yytext);

return VAR;}

{number} {strcpy(yylval.var,yytext);

return NUM;}

\< |

\> |

\>= |

\<= |

== {strcpy(yylval.var,yytext);

return RELOP;}

[ \t] ;

\n LineNo++;

. return yytext[0];

%%

<int.y>

%{

#include<string.h>

#include<stdio.h>

struct quad 22

{

char op[5];

char arg1[10];

char arg2[10];

char result[10];

}QUAD[30];

struct stack

{

int items[100];

int top;

}stk;

int Index=0,tIndex=0,StNo,Ind,tInd;

extern int LineNo;

%}

%union

{

char var[10];

}

%token <var> NUM VAR RELOP

%token MAIN IF ELSE WHILE TYPE

%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP

%left '-' '+'

%left '\*' '/'

%%

PROGRAM : MAIN BLOCK

;

BLOCK: '{' CODE '}'

;

CODE: BLOCK

| STATEMENT CODE

| STATEMENT

;

STATEMENT: DESCT ';'

| ASSIGNMENT ';'

| CONDST

| WHILEST

;

DESCT: TYPE VARLIST

;

VARLIST: VAR ',' VARLIST

| VAR

;

ASSIGNMENT: VAR '=' EXPR{

strcpy(QUAD[Index].op,"="); 23

strcpy(QUAD[Index].arg1,$3);

strcpy(QUAD[Index].arg2,"");

strcpy(QUAD[Index].result,$1);

strcpy($$,QUAD[Index++].result);

}

;

EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}

| EXPR '-' EXPR {AddQuadruple("-",$1,$3,$$);}

| EXPR '\*' EXPR { AddQuadruple("\*",$1,$3,$$);}

| EXPR '/' EXPR { AddQuadruple("/",$1,$3,$$);}

| '-' EXPR { AddQuadruple("UMIN",$2,"",$$);}

| '(' EXPR ')' {strcpy($$,$2);}

| VAR

| NUM

;

CONDST: IFST{

Ind=pop();

sprintf(QUAD[Ind].result,"%d",Index);

Ind=pop();

sprintf(QUAD[Ind].result,"%d",Index);

}

| IFST ELSEST

;

IFST: IF '(' CONDITION ')' {

strcpy(QUAD[Index].op,"==");

strcpy(QUAD[Index].arg1,$3);

strcpy(QUAD[Index].arg2,"FALSE");

strcpy(QUAD[Index].result,"-1");

push(Index);

Index++;

}

BLOCK {

strcpy(QUAD[Index].op,"GOTO");

strcpy(QUAD[Index].arg1,"");

strcpy(QUAD[Index].arg2,"");

strcpy(QUAD[Index].result,"-1");

push(Index);

Index++;

};

ELSEST: ELSE{

tInd=pop();

Ind=pop();

push(tInd);

sprintf(QUAD[Ind].result,"%d",Index); 24

BLOCK{

Ind=pop();

sprintf(QUAD[Ind].result,"%d",Index);

};

CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);

StNo=Index-1;

}

| VAR

| NUM

;

WHILEST: WHILELOOP{

Ind=pop();

sprintf(QUAD[Ind].result,"%d",StNo);

Ind=pop();

sprintf(QUAD[Ind].result,"%d",Index);

}

;

WHILELOOP: WHILE '(' CONDITION ')' {

strcpy(QUAD[Index].op,"==");

strcpy(QUAD[Index].arg1,$3);

strcpy(QUAD[Index].arg2,"FALSE");

strcpy(QUAD[Index].result,"-1");

push(Index);

Index++;

}

BLOCK {

strcpy(QUAD[Index].op,"GOTO");

strcpy(QUAD[Index].arg1,"");

strcpy(QUAD[Index].arg2,"");

strcpy(QUAD[Index].result,"-1");

push(Index);

Index++;

}

;

%%

extern FILE \*yyin;

int main(int argc,char \*argv[])

{

FILE \*fp;

int i;

if(argc>1)

{

fp=fopen(argv[1],"r"); 25

if(!fp)

{

printf("\n File not found");

exit(0);

}

yyin=fp;

}

yyparse();

printf("\n\n\t\t ----------------------------""\n\t\t Pos Operator Arg1 Arg2 Result" "\n\t\t

--------------------");

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

{

printf("\n\t\t %d\t %s\t %s\t %s\t

%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);

}

printf("\n\t\t -----------------------");

printf("\n\n");

return 0;

}

void push(int data)

{

stk.top++;

if(stk.top==100)

{

printf("\n Stack overflow\n");

exit(0);

}

stk.items[stk.top]=data;

}

int pop()

{

int data;

if(stk.top==-1)

{

printf("\n Stack underflow\n");

exit(0);

}

data=stk.items[stk.top--];

return data;

}

void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])

{

strcpy(QUAD[Index].op,op);

strcpy(QUAD[Index].arg1,arg1); 26

strcpy(QUAD[Index].arg2,arg2);

sprintf(QUAD[Index].result,"t%d",tIndex++);

strcpy(result,QUAD[Index++].result);

}

yyerror()

{

printf("\n Error on line no:%d",LineNo);

}

Input:

$vi test.c

main()

{

int a,b,c;

if(a<b)

{

a=a+b;

}

while(a<b)

{

a=a+b;

}

if(a<=b)

{

c=a-b;

}

else

{

c=a+b;

}

}

**Output 1:**

$ lex int.l

$ yacc –d int.y

$ gcc lex.yy.c y.tab.c –ll –lm

$ ./a.out test.c 27

# Result:

Thus the program f organize the file using single level directory.

was verified successfully.