**GENERATE YACC SPECIFICATION FOR A FEW SYNTACTIC**

**CATEGORIES**

a) Program to recognize a valid arithmetic expression that uses operator +, –,

\*, and /.

b) Program to recognize a valid variable which starts with a letter followed by

any number of letters or digits.

c) Implementation of Calculator using LEX and YACC

d) Convert the BNF rules into YACC form and write code to generate

abstract syntax tree

**A) PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION**

**THAT USES OPERATOR +, - , \* AND /**

**AIM :** Program to recognize a valid arithmetic expression that uses operators

+, -, \* and /.

**ALGORITHM**

1. Start
2. Write regular expression for identifier and number
3. Write syntax for all possible arithmetic expression
4. Input the expression to be checked
5. Output whether the expression is valid or not
6. Stop

**PROGRAM**

**Lex**

%{

#include "y.tab.h"

%}

%%

[0-9]+ { return NUMBER; }

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

\n { return 0; }

. { return yytext[0]; }

%%

**Yacc**

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token NUMBER ID

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

%%

exp : exp'+'exp

| exp'-'exp

| exp'\*'exp

| exp'/'exp

| '('exp')'

| NUMBER

| ID ;

%%

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

printf("Enter the expression: ");

yyparse();

printf("Valid Expression!\n");

return 0;

}

int yyerror() {

printf("Invalid Expression!\n");

exit(1);

}

int yywrap() {

return 1;

}

**Execution**

lex 04aRecognizeOperators.l

yacc -dy 04aRecognizeOperators.y

gcc lex.yy.c y.tab.c

**Output**

**Test Case #1: Valid Expression**

Enter the expression: 12+23-8

Valid Expression!

**Test Case #2: Invalid Expression**

Enter the expression: a+\*

Invalid Expression!

**B) PROGRAM TO RECOGNIZE A VALID VARIABLE WHICH STARTS**

**WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR**

**DIGITS.**

**AIM:**

Program to recognize a valid variable, which starts with a letter, followed by

any number of letters or digits.

**ALGORITHM**

1. Start
2. Write the regular expression for identifier check
3. Input the variable
4. Stop

**PROGRAM**

**Lex**

%{

#include "y.tab.h"

%}

%%

[0-9] { return DIGIT; }

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

\n { return 0; }

. { return yytext[0]; }

%%

**Yacc**

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token DIGIT ALPHA

%%

var : ALPHA

| var ALPHA

| var DIGIT ;

%%

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

printf("Enter a variable name: ");

yyparse();

printf("Valid Variable!\n");

return 0;

}

int yyerror() {

printf("Invalid Variable!\n");

exit(1);

}

int yywrap() {

return 1;

}

**Execution**

lex 04bRecognizeVariable.l

yacc -dy 04bRecognizeVariable.y

gcc lex.yy.c y.tab.c

**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!

**C) IMPLEMENTATION OF CALCULATOR USING LEX AND YACC**

**AIM:** Implementation of calculator using lex and yacc

**ALGORITHM**

1. Start
2. Write function for each of the operator
3. Input the given expression
4. Perform lexical analyser
5. Print output
6. Stop

**PROGRAM:**

Lex<Cal.L>

%{

#include"y.tab.h"

#include<math.h>

%}

%%

([0-9]+|([0-9]\*\.[0-9]+)([eE][-+]?[0-9]+)?) {yylval.dval=atof(yytext);return

NUMBER;}

log |

LOG {return LOG;}

In {return nLOG;}

sin |

SIN {return SINE;}

cos |

COS {return COS;}

tan |

TAN {return TAN;}

mem {return MEM;}

[\t];

\$ return 0;

\n|. return yytext[0];

%%

**Yacc<Cal.Y>**

%{

double memvar;

%}

%union

{

double dval;

}

%token<dval>NUMBER

%token<dval>MEM

%token LOG SINE nLOG COS TAN

%left '-' '+'

%left '\*' '/'

%right '^'

%left LOG SINE nLOG COS TAN

%nonassoc UMINUS

%type<dval>expression

%%

start:statement'\n'

|start statement'\n'

;

statement:MEM'='expression {memvar=$3;}

| expression{printf("Answer=%g\n",$1);}

;

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

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

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

| expression '/' expression

{ if($3==0)

yyerror("divide by zero");

else

$$=$1/$3;

}

|expression'^'expression {$$=pow($1,$3);}

;

expression:'-'expression %prec UMINUS{$$=-$2;}

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

|LOG expression {$$=log($2)/log(10);}

|nLOG expression {$$=log($2);}

|SINE expression {$$=sin($2\*3.14/180);}

|COS expression {$$=cos($2\*3.14/180);}

|TAN expression {$$=tan($2\*3.14/180);}

|NUMBER {$$=$1;}

|MEM {$$=memvar;}

;

%%

main()

{

printf("Enter the expression");

yyparse();}

int yyerror(char \*error)

{

printf("%s\n",error);

}

**OUTPUT:**

[linuxpert@fosslab ~]$ vi cal.l

[linuxpert@fosslab ~]$ lex cal.l

[linuxpert@fosslab ~]$ yacc -d cal.y

[linuxpert@fosslab ~]$ cc lex.yy.c y.tab.c -ll -lm

[linuxpert@fosslab ~]$ ./a.out

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

Answer=7

**D. CONVERT THE BNF RULES INTO YACC FORM AND WRITE CODE**

**TO GENERATE ABSTRACT SYNTAX TREE.**

**AIM:** To Convert the BNF rules into Yacc form and write code to generate

Abstract Syntax Tree

**ALGORITHM:**

1. Start the program.

2. Include the header file.

3. In int code.l,declare the variable lie no as integer and assign it to be equal to ‘1’.

4. Start the int code.l with declarative section.

5. In translation rules section define keywords ,data types and integer along with

their actions .

6. Start the main block. In main block check the statement

7. 1.declarative 2.assignment 3.conditional 4.if and else 5.While assignment.

8. Perform the actions of that particular block.

9. In main program declare the parameters arg c as int end \*argv[] as char.

10. In main program open file in read mode.

11. Print the output in a file.

12. End the program.

**Lex<Bnf.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];

%%

**Yacc <Bnf.Y>**

%{

#include<string.h>

#include<stdio.h>

struct quad

{

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,"=");

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);

}

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");

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)

{s

tk.top++;

if(stk.top==100)

{

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

exit(0);

}s

tk.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])

{s

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

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

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:**

[linuxpert@fosslab ~]$ vi bnf.y

[linuxpert@fosslab ~]$ yacc -d bnf.y

[linuxpert@fosslab ~]$ gcc lex.yy.c y.tab.c -ll -lm

[linuxpert@fosslab ~]$ ./a.out test.c

----------------------------------------------------------------------------

**Pos Operator Arg1 Arg2 Result**

-----------------------------------------------------------------------------

0 < a b t0

1 == t0 FALSE 5

2 + a b t1

3 = t1 a

4 GOTO 5

5 < a b t2

6 == t2 FALSE 10

7 + a b t3

8 = t3 a

9 GOTO 5

10 <= a b t4

11 == t4 FALSE 15

12 - a b t5

13 = t5 c

14 GOTO 17

15 + a b t6

16 = t6 c