**Ex.No:**

**YACC Programs**

**Aim:**

To write a YACC program

* 1. to recognize a valid arithmetic expression that usesoperator +, - , \* and /.
  2. to recognize a valid variable which starts with a letterfollowed by any number of letters or digits
  3. Implementation of Calculator using LEX and YACC

**Description:**

YACC – yet another compiler–compiler converts a context–free grammar and translation code into a set of tables for an LR (1) parser and translator. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

*Structure of Source Program*

declarations

%%

rules

%%

routines

* The declaration section may be empty. Moreover, if the routines section is omitted, the second %% mark may be omitted also.

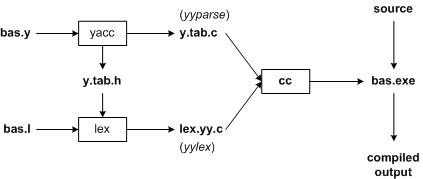
The declarations section may contain the following items.

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

yyparse() calls yylex() when it needs a new token.

|  |  |
| --- | --- |
| LEX | YACC |
| return(TOKEN) | %token TOKEN |
|  | TOKEN is used in production |

Extern variable yylvalis used in a LEX source program to return values of lexemes



**/\*YACC program\*/**

**1. To recognize a valid arithmetic expression that usesoperator +, - , \* and /.**

exp.l

=====

%{

#include "y.tab.h"

%}

**%%**

[0-9]+ { return NUMBER;}

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

\n {return NL;}

. {return yytext[0];}

**%%**

int yywrap()

{

return 1;

}

exp.y

======

%token NUMBER ID NL

%left '+''-'

%left '\*''/'

**%%**

stmt :E NL {printf("valid \n"); exit(0);}

;

E :E'+'E

|E'-'E

|E'\*'E

|E'/'E

|'('E')'

|ID

|NUMBER

;

%%

int yyerror(char \*msg)

{

printf(“Invalid\n%s",msg);

exit(0);

}

main()

{

yyparse();

}

**/\*OUTPUT\*/**

[student@localhost 201301084]$ yacc -d exp.y

[student@localhost 201301084]$ cc lex.yy.c y.tab.c

[student@localhost 201301084]$ ./a.out

8+9

valid

[student@localhost 201301084]$ ./a.out

89+

invalidsyntax error

**2. To recognize a valid variable which starts with a letterfollowed by any number of letters or digits**

%{

#include<stdio.h>

#include<ctype.h>

#include<stdlib.h>

%}

%token let dig

%%

sad : let recld '\n' {printf("accepted\n"); exit(0);}

| let '\n' {printf("accepted\n"); exit(0);}

|

|error {yyerror("rejected\n");exit(0);}

;

recld : let recld

| dig recld

| let

| dig

;

%%

yylex(){

char ch;

while((ch=getchar())==' ');

if(isalpha(ch))

return let;

if(isdigit(ch))

return dig;

return ch;

}

yyerror(char \*s){

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

exit(0);

}

main(){

printf("ENTER A variable : ");

yyparse();

}

**/\*OUTPUT\*/**

[student@localhost]$ yacc valid.y

[student@localhost]$ cc y.tab.c

[student@localhost]$ ./a.out

ENTER A variable : count5

accepted

[student@localhost]$ ./a.out

ENTER A variable : 5count

syntax error

***3. Implementation of Calculator using LEX and YACC***

calc.l:

=====

%{

#include "y.tab.h"

extern intyylval;

%}

%%

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

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

\n {return NL;}

. {returnyytext[0];}

%%

intyywrap()

{

return 1;

}

calc.y:

=====

%{

#include <stdio.h>

%}

%token NUMBER ID NL

%left '+''-'

%left '\*''/'

%%

stmt : E NL { printf("valid=%d\n",$1); exit(0); }

;

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

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

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

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

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

| ID { $$=$1; }

| NUMBER { $$=$1; }

;

%%

int yyerror(char \*msg)

{

printf("invalid%s\n",msg);

exit(0);

}

main()

{

yyparse();

}

**/\*OUTPUT\*/**

[student@localhost]$ lex calc.l

[student@localhost]$ yacc -d calc.y

[student@localhost]$ cc lex.yy.c y.tab.c

[student@localhost]$ ./a.out

6+7\*2

valid=20

[student@localhost 201301084]$

**ABSTRACT SYNTAX TREE**

**Aim:**

To write a program to Convert the BNF rules into Yacc form and write code to generate Abstract Syntax Tree

**Description**

An abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches. Abstract syntax trees are also used in program analysis and program transformation systems.

**/\*Abstract Syntax Tree\*/**

ast.l:

======

%{

#include "y.tab.h"

extern int yylval;

%}

**%%**

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

[ \t\n] ;

"+" return(PLUS);

"-" return(MINUS);

"\*" return(TIMES);

"(" return(LEFT\_PARENTHESIS);

")" return(RIGHT\_PARENTHESIS);

";" return(END);

**%%**

int yywrap ( )

{

return 1;

}

ast.y:

======

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct node

{

struct node \*left;

struct node \*right;

char \*token;

} node;

node \*mknode(node \*left, node \*right, char \*token);

void printtree(node \*tree);

#define YYSTYPE struct node \*

%}

%start lines

%token NUMBER PLUS MINUS TIMES L\_PAREN R\_PAREN END

%left PLUS MINUS

%left TIMES

%%

lines: /\* empty \*/

| lines line /\* do nothing \*/

line : exp END { printtree($1); printf("\n");}

;

exp : term {$$ = $1;}

| exp PLUS term {$$ = mknode($1, $3, "+");}

| exp MINUS term {$$ = mknode($1, $3, "-");}

;

term : factor {$$ = $1;}

| term TIMES factor {$$ = mknode($1, $3, "\*");}

;

factor : NUMBER {$$ = mknode(0,0,(char \*)yylval);}

| L\_PAREN exp R\_PAREN {$$ = $2;}

;

%%

int main ()

{

returnyyparse ( );

}

node \*mknode(node \*left, node \*right, char \*token)

{

node \*newnode = (node \*)malloc(sizeof(node));

char \*newstr = (char \*)malloc(strlen(token)+1);

strcpy(newstr, token);

newnode->left = left;

newnode->right = right;

newnode->token = newstr;

return(newnode);

}

void printtree(node \*tree)

{

inti;

if (tree->left || tree->right)

printf("(");

printf(" %s ", tree->token);

if (tree->left)

printtree(tree->left);

if (tree->right)

printtree(tree->right);

if (tree->left || tree->right)

printf(")");

}

int yyerror (char \*s)

{

fprintf (stderr, "%s\n", s);

}

**/\*OUTPUT\*/**

[student@localhost]$ lex ast.l

[student@localhost]$ yacc -d ast.y

[student@localhost]$ cc lex.yy.c y.tab.c

[student@localhost]$ ./a.out

8+9\*2

;

( + 8 ( \* 9 2 ))

3+4

;

( + 3 4 )

[student@localhost]$