CSB 353: Compiler Design

LAB 10-11

Submitted By:

Name: PREM KUMAR

Roll No: 191210037

Branch: CSE

Semester: 6 th

Submitted To: Dr. Shelly Sachdeva

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

2019-2023

Ques 1. Consider the example of simple desk calculator that performs simple operations on integer expressions with the grammar:

exp -> exp addop term | term addop -> + | -

term -> term mulop factor | factor mulop -> *

factor -> (exp) | number

number -> number digit | digit

digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

You are required to write YACC specifications for this grammar so that the parser evaluates any arithmetic expressions and the output shows each grammar rule as it is applied in the parsing process. Show your parsing sequence for the input string: (2+(3*4)).

Code:

- prog1.l (Lex File)

Prog1.l

Lab10-11 > Prog1.l

```
%{
    #include "y.tab.h"
    extern yylval;

%}

%%
[0-9] {return digit;}
[\t] ;
\n {return 0;}
. {return yytext[0];}
%%

int yywrap()
{
return(1);
}
```

- prog1.y (Yacc File)

Prog1.y

Lab10-11 > Prog1.y

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
%}

%token digit
%left '+' '-'
%left '*' '/'

%%
exp     :     exp addop term      {printf("Rule 1: exp -> exp addop term\n");}
        |     term                {printf("Rule 2: exp -> term\n");}
        ;
addop   :     '+'                 {printf("Rule 3: addop -> +\n");}
        |     '-'                 {printf("Rule 4: addop -> -\n");}
        ;
term    :     term mulop factor   {printf("Rule 5: term -> term mulop factor\n");}
        |     factor              {printf("Rule 6: term -> factor\n");}
        ;
mulop   :     '*'                 {printf("Rule 7: mulop -> *\n");}
        ;
factor  :     '(' exp ')'         {printf("Rule 8: factor -> ( exp ) \n");}
        |     number              {printf("Rule 9: factor -> number\n");}
        ;
number  :     number digit        {printf("Rule 10: number -> number digit\n");}
        |     digit               {printf("Rule 11: number -> digit\n");}
        ;
%%

main(){
    printf("Enter the input expression: ");
    yyparse();
    printf("Expression is Valid");
    exit(0);

}

int yyerror(char *s)
{
printf("\nExpression is invalid");
exit(0);
}
```

Output:

```
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11> .\a.exe
Enter the input expression: (2+(3*4))
Rule 11: number -> digit
Rule 9: factor -> number
Rule 6: term -> factor
Rule 2: exp -> term
Rule 3: addop -> +
Rule 11: number -> digit
Rule 9: factor -> number
Rule 6: term -> factor
Rule 7: mulop -> *
Rule 11: number -> digit
Rule 9: factor -> number
Rule 5: term -> term mulop factor
Rule 2: exp -> term
Rule 8: factor -> ( exp )
Rule 6: term -> factor
Rule 1: exp -> exp addop term
Rule 8: factor -> ( exp )
Rule 6: term -> factor
Rule 2: exp -> term
Expression is Valid
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11>
```

Ques 2. The following grammar describes a boolean expression (exp) consisting of operators "&", "|", "!", "==", "! =", brackets "(" and ")", and identifiers.

exp : exp_2 | exp '&' exp_2 ;

exp_2 : exp_3 | exp_3 '|' exp_2 ;

exp_3 : exp_4 | exp_4 '==' exp_4 | exp_4 '!=' exp_4 ;

exp_4 : exp_5 | '!' exp_5 ;

exp_5 : identifier | '(' exp ')' ;

Write Yacc code to recognize a series of expressions, each on a new line. Each expression should unambiguously demonstrate precedence & associativity of the operators, by showing the orders in which operators would be evaluated. Also, give a parse sequence for each of the expressions.

Code:

- prog2.l (Lex File)

Prog2.l

Lab10-11 > Prog2.l

```
%{
    #include "y.tab.h"
    extern yylval;

%}

%%
[a-zA-Z][a-zA-Z0-9]* {return ID;}
[0-9]+ {return NUM;}
"&"|"|"|"=="|"!=" {return BINARY_OP;}
"!" {return UNARY_OP;}
[\t] ;
\n {return 0;}
. {return yytext[0];}
%%
int yywrap()
{
return(1);
}
```

- prog2.y (Yacc File)

Prog2.y

Lab10-11 > Prog2.y

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
%}

%left '&'
%left '|'
%token ID NUM BINARY_OP UNARY_OP

%%
exp     :     exp_2               {printf("Rule 1: exp -> exp_2\n");}
        |     exp '&' exp_2       {printf("Rule 2: exp -> exp & exp_2\n");}
        ;
exp_2   :     exp_3               {printf("Rule 3: exp_2 -> exp_3\n");}
        |     exp_3 '|' exp_2     {printf("Rule 4: exp_2 -> exp_3 | exp_2\n");}
        ;
exp_3   :     exp_4               {printf("Rule 5: exp_3 -> exp_4\n");}
        |     exp_4 '=' '=' exp_4 {printf("Rule 6: exp_3 -> exp_4 == exp_4\n");}
        |     exp_4 '!' '=' exp_4 {printf("Rule 7: exp_3 -> exp_4 != exp_4\n");}
        ;
exp_4   :     exp_5               {printf("Rule 8: exp_4 -> exp_5\n");}
        |     '!' exp_5           {printf("Rule 9: exp_4 -> ! exp_5\n");}
        ;
exp_5   :     ID                  {printf("Rule 10: exp_5 -> ID\n");}
        |     '(' exp ')'         {printf("Rule 11: exp_5 -> ( exp )\n");}
        ;
%%

main(){
    printf("Enter the input expression: ");
    yyparse();
    printf("Expression is Valid");
    exit(0);

}

int yyerror(char *s)
{
printf("\nExpression is Invalid");
exit(0);
}
```

Output:

```
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11> .\a.exe
Enter the input expression: a & b
Rule 10: exp_5 -> ID
Rule 8: exp_4 -> exp_5
Rule 5: exp_3 -> exp_4
Rule 3: exp_2 -> exp_3
Rule 1: exp -> exp_2

Expression is Valid
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11>
```

Ques 3. Using YACC, write semantic actions that translate arithmetic expressions (generated from the given grammar) from infix into postfix notation

E -> E + T E -> T

T-> T * F T -> F

F -> (E) F -> num.

Code:

- prog3.l (Lex File)

Prog3.l

Lab10-11 > Prog3.l

```
%{
    #include <stdio.h>
    #include "y.tab.h"
%}

%%
[0-9]+ {
    yylval=atoi(yytext);
    return num;
}
"+"   return *yytext;
"-"   return *yytext;
[()]  return *yytext;
[\t]  ;
\n    {return 0;}
.     {return yytext[0];}
%%

yywrap(){
    return 1;
}
```

- prog3.y (Yacc File)

Prog3.y

Lab10-11 > Prog3.y

```
%token num
%left '+'
%left '*'

%%
E    :    E '+' T    {printf("+ ");}
     |    T
     ;
T    :    T '*' F    {printf("* ");}
     |    F
     ;
F    :    '(' E ')'
     |    num        {printf("%d ", $1);}
%%

yyerror(char const *s){
    printf("\nInvalid expression\n");
    printf("Something went wrong:  %s\n",s);
    exit(0);
}

int main(){
    printf("Enter the input expression: ");
    yyparse();
    printf("\nValid expression");
}
```

Output:

```
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11> .\a.exe
Enter the input expression: (2+(3*4))
2 3 4 * +
Valid expression
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11>
```

Ques 4. For the grammar below, write a YACC program to compute the decimal value of an input string in binary. For example the translation of the string 101.101 should be the decimal number 5.625.

S→L.L|L L→LB|B B→0|1.

Code:

- prog4.l (Lex File)

Prog4.l

Lab10-11 > Prog4.l

```
%{
    #include <stdio.h>
    #include "y.tab.h"
%}

%%
[ \t] ;
\n {return 0;}
. {return yytext[0];}
%%

yywrap(){
    return 1;
}
```

- prog4.y (Yacc File)

Prog4.y

Lab10-11 > Prog4.y

```
%{
  #include<stdio.h>
  #include<stdlib.h>
  void yyerror(char *s);
%}

%union
{
    struct node
    {
        float val;
        int pos;
    }bit;
}
%type <bit> S L B

%%
S   :   L '.' L     { float base=1; for(int i=0;i<$3.pos;i++)base*=2;
                      printf("Value = %f", $1.val + $3.val/base);}
    |   L           { printf("Value = %f", $$.val);};
L   :   L B         { $$.val = $1.val*2+$2.val; $$.pos = $1.pos + $2.pos; }
    |   B           { $$.val = $1.val; $$.pos = $1.pos; };
B   :   '0'        { $$.val = 0; $$.pos = 1; }
    |   '1'         { $$.val = 1; $$.pos = 1; }
    ;
%%

main(){
    printf("Enter the input expression: ");
    yyparse();
    printf("\nValid expression");
}

yyerror(char *s){
    printf("\nInvalid expression: %s", s);
    exit(0);
}
```

Output:

```
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11> .\a.exe
Enter the input expression: 101.101
Value = 5.625000
Valid expression
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11>
```

Ques 5. Write a YACC program that accepts valid strings as per the grammar given below and produces as output a linear representation of the same list

Grammar:

S (L)| char

L L,S|S

and char is any character between a-z

e.g. ((a,c),d,(h)) Produces acdh but (a,c),d,(h)) is an invalid input.

Code:

- prog5.l (Lex File)

Prog5.l

Lab10-11 > Prog5.l

```
%{
    #include "y.tab.h"
%}

%%
[a-z] {
    yylval=yytext[0];
    return character;
}
[\t] ;
\n {return 0;}
. {return yytext[0];}
%%

int yywrap()
{
return(1);
}
```

- prog5.y (Yacc File)

Prog5.y

Lab10-11 > Prog5.y

```
%{
    #include <stdio.h>
    #include <stdlib.h>
%}

%token character

%%
S    :    '(' L ')'
     |    character        {printf("%c", $1);}
     ;
L    :    L ',' S
     |    S
     ;
%%

main(){
    printf("Enter the input expression: ");
    yyparse();
    printf("\nValid expression");
}

yyerror(){
    printf("\nInvalid expression");
    exit(0);
}
```

Output:

```
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11> .\a.exe
Enter the input expression: ((a,c),d,(h))
acdh
Valid expression
PS C:\Users\Prem\Desktop\6thSem\CSB353\Lab10-11>
```