
EXPERIMENT - III

LEXICAL ANALYSIS USING YACC

November 27, 2020

ADITHYA D RAJAGOPAL
ROLL NO : 9
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING TRIVANDRUM

AIM

To generate YACC specification for a few syntactic categories:

1. Program to recognize a valid arithmetic expression that uses operator +, −, * and /.
2. Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.
3. Implementation of Calculator using LEX and YACC.
4. Convert the BNF rules into YACC form and write code to generate abstract.

THEORY

Yacc

Grammar for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow id \end{aligned}$$

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E (expression) are nonterminals. Terms such as id (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

$$\begin{aligned} E &\Rightarrow E * E \text{ (r2)} \\ &\Rightarrow E * z \text{ (r3)} \\ &\Rightarrow E + E * z \text{ (r1)} \\ &\Rightarrow E + y * z \text{ (r3)} \\ &\Rightarrow x + y * z \text{ (r3)} \end{aligned}$$

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we need to do the reverse operation. Instead of starting with a single non-terminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single non-terminal. This is known as bottom-up or shift-reduce parsing and uses a stack for storing terms.

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

ALGORITHMS

Algorithm 1 Program to recognize a valid arithmetic expression that uses +, -, * and /.

- 1: Start the program.
 - 2: Read the expression.
 - 3: Check if the expression is valid according to the rule using yacc.
 - 4: Using the expression rules print the result of the given values.
 - 5: Stop the program.
-

Algorithm 2 Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

- 1: Start the program.
 - 2: Read the expression.
 - 3: Check if the expression is valid according to the rule using yacc.
 - 4: Using the expression rules print the result of the given values.
 - 5: Stop the program.
-

Algorithm 3 Implementation of Calculator using LEX and YACC.

- 1: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%. The format is as follows:
 - 2: Definitions
 - 3: %%
 - 4: Rules
 - 5: %%
 - 6: User_Subroutines
 - 7: In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %..%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.
 - 8: In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.
 - 9: Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.
 - 10: When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.
 - 11: In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.
 - 12: The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.
-

PROGRAMS AND OUTPUTS

1. Program to recognize a valid arithmetic expression that uses operator +, −, * and /.

Source Code

p3a.l

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

%%

[a-zA-Z][_a-zA-Z0-9]* {return ID;}
[ \t ]                {;}
[ \n ]                {return 0;}
.                      {return yytext[0];}

%%

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

p3a.y

```
%{
    #include<stdio.h>
    #include<stdlib.h>
    void yyerror();
    int yylex();
}%

%start stmt
```

```
%token ID NUM
```

```
%left '+' '-' '*' '/'
```

```
%%
```

```
stmt : expr | ID '=' expr ;
```

```
expr : expr '+' expr  
      | expr '-' expr  
      | expr '*' expr  
      | expr '/' expr  
      | '(' expr ')'  
      | ID  
      | NUM ;
```

```
%%
```

```
void main()  
{  
    printf("Enter an expression : ");  
    yyparse();  
    printf("Valid Expression Identified\n");  
}
```

```
void yyerror()  
{  
    printf("Expression Invalid\n");  
    exit(0);  
}
```

Output

```
user@adithya-d-rajagopal:~/s7/cd/Exp3$ lex p3a.l
user@adithya-d-rajagopal:~/s7/cd/Exp3$ yacc -d p3a.y
user@adithya-d-rajagopal:~/s7/cd/Exp3$ gcc lex.yy.c y.tab.c -ll
user@adithya-d-rajagopal:~/s7/cd/Exp3$ ./a.out
Enter an expression : i+i*i
Valid Expression Identified
user@adithya-d-rajagopal:~/s7/cd/Exp3$ ./a.out
Enter an expression : i++
Expression Invalid
user@adithya-d-rajagopal:~/s7/cd/Exp3$
```


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

Source Code

p3b.l

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

%%

[a-zA-Z][_a-zA-Z0-9]* {return ID;}
[ \t ]                {;}
[ \n ]                {return 0;}
.                     {return yytext[0];}

%%

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

p3b.y

```
%{
    #include<stdio.h>
    #include<stdlib.h>
    void yyerror();
    int yylex();
}%

%start stmt
%token ID
```

```
%%

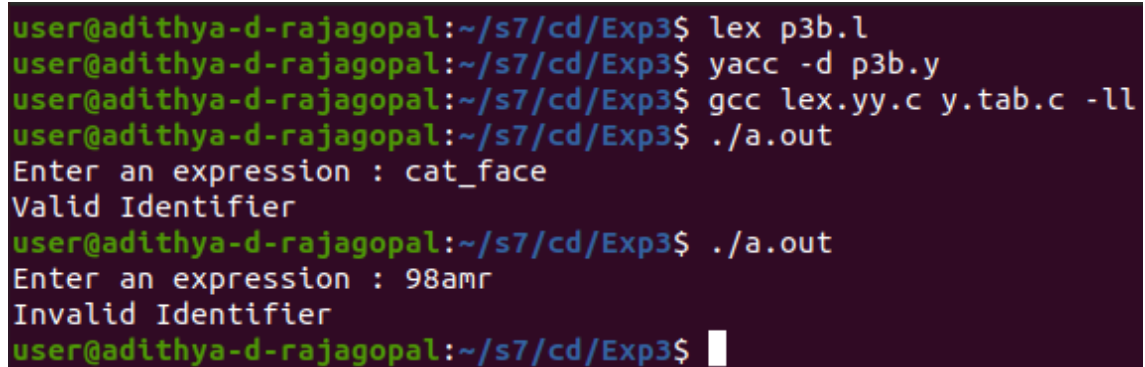
stmt : ID;

%%

void main()
{
    printf("Enter an expression : ");
    yyparse();
    printf("Valid Identifier\n");
}

void yyerror()
{
    printf("Invalid Identifier\n");
    exit(0);
}
```

Output



```
user@adithya-d-rajagopal:~/s7/cd/Exp3$ lex p3b.l
user@adithya-d-rajagopal:~/s7/cd/Exp3$ yacc -d p3b.y
user@adithya-d-rajagopal:~/s7/cd/Exp3$ gcc lex.yy.c y.tab.c -ll
user@adithya-d-rajagopal:~/s7/cd/Exp3$ ./a.out
Enter an expression : cat_face
Valid Identifier
user@adithya-d-rajagopal:~/s7/cd/Exp3$ ./a.out
Enter an expression : 98amr
Invalid Identifier
user@adithya-d-rajagopal:~/s7/cd/Exp3$
```

3. Implementation of Calculator using LEX and YACC.

Source Code

p3c.l

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

%option noyywrap

%%

"print"      {return print;}
"exit"       {return end;}
[a-zA-Z]     {yylval.id=yytext[0]; return identifier;}
[0-9]+       {yylval.n=atoi(yytext); return num;}
[ \t\n]      ;
[-=+ /;()]   {return yytext[0];}
"*"          {return yytext[0];}
"."          {;}
%%
```

p3c.y

```
%{
    #include<stdio.h>
    #include<stdlib.h>
    #include<ctype.h>
    void yyerror();
    int yylex();
    int sym[52];
    int value(char c);
    void update(char s,int val);
}%
```

```

%union{int n; char id;}
%start stmt
%token print end
%token <n> num
%token <id> identifier
%type <n> stmt exp term
%type <idnt> assign
%right '='
%left '+' '-' '*' '/'

%%

stmt : assign ';'          {};
      | end ';'            {exit(EXIT_SUCCESS);}
      | print exp ';'      {printf("Value is : %d\n",$2);}
      | stmt assign ';'    {};
      | stmt print exp ';' {printf("Value is : %d\n",$3);}
      | stmt end ';'       {exit(EXIT_SUCCESS);};

assign : identifier '=' exp {update($1,$3);};

exp : term          {$$ = $1;}
     | '(' exp ')'  {$$ = $2;}
     | exp '=' exp  {$$ = $3;}
     | exp '+' exp  {$$ = $1+$3;}
     | exp '-' exp  {$$ = $1-$3;}
     | exp '*' exp  {$$ = $1*$3;}
     | exp '/' exp  {$$ = $1/$3;};

term : identifier {$$ = value($1);}
      | num       {$$ = $1;};

%%

```

```
int idx(char s)
{
    int i=-1;
    if(islower(s))
        i=s-'a'+26;
    else if(isupper(s))
        i=s-'A';
    return i;
}

int value(char s)
{
    int i=idx(s);
    return sym[i];
}

void update(char s,int val)
{
    int i=idx(s);
    sym[i]=val;
}

int main(void)
{
    int j;
    for(j=0;j<52;j++)
        sym[j]=0;
    return yyparse();
}

void yyerror() {}
```

Input

```
user@adithya-d-rajagopal:~/s7/cd/Exp3$ cat p3c.txt
a=5;
b=10;
c=a+b;
d=a-b;
e=a/b;
f=a*b;
g=f+5+3;
h=a+(a+b+c);
print a;
print b;
print c;
print d;
print e;
print f;
print g;
print h;
print 1+2+3*4;
end;
user@adithya-d-rajagopal:~/s7/cd/Exp3$
```

Output

```
user@adithya-d-rajagopal:~/s7/cd/Exp3$ lex p3c.l
user@adithya-d-rajagopal:~/s7/cd/Exp3$ yacc -d p3c.y
user@adithya-d-rajagopal:~/s7/cd/Exp3$ gcc lex.yy.c y.tab.c -ll
user@adithya-d-rajagopal:~/s7/cd/Exp3$ ./a.out < p3c.txt
Value is : 5
Value is : 10
Value is : 15
Value is : -5
Value is : 0
Value is : 50
Value is : 165
Value is : 755
Value is : 24
user@adithya-d-rajagopal:~/s7/cd/Exp3$
```

4. Convert the BNF rules into YACC form and write code to generate abstract.

Source Code

p3d.l

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

identifier [a-zA-Z][_a-zA-Z0-9]*
number [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] {yylineno++;}
. return yytext[0];

%%

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

p3d.y

```

%{
    #include<string.h>
    #include<stdio.h>
    #include<stdlib.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 yylineno;
    void yyerror();
    int yylex();
    void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10]);
    int pop();
    void push(int data);
}%

%union{char var[10];}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+' '*' '/'

%%

```



```
PROGRAM : MAIN BLOCK;
```

```
BLOCK : '{' CODE '}';
```

```
CODE : BLOCK
      | STATEMENT CODE
      | STATEMENT;
```

```
STATEMENT : DESCT ';'
           | ASSIGNMENT ';'
           | CON DST
           | 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 ( "\t \t \t Pos \t Operator \t Arg1 \t Arg2 \t Result \n " );
    for ( i = 0 ; i < Index ; i ++ )
        printf ( "\t \t \t %d \t %s \t \t \t %s \t %s \t %s \n " , i , QUAD [ i ] . op ,
                QUAD [ i ] . arg1 , QUAD [ i ] . arg2 , QUAD [ i ] . result );

```

```
    return 0;
}

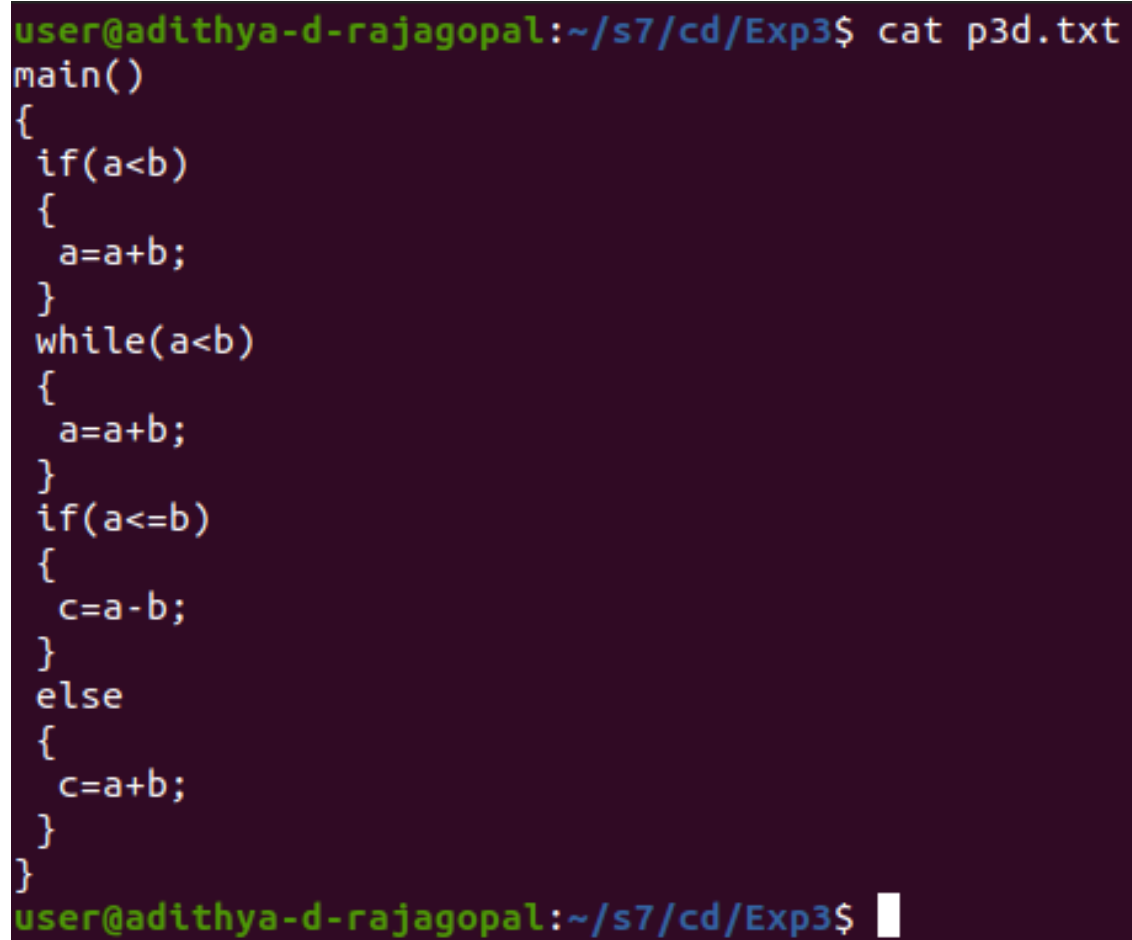
void push(int data)
{
    stk.top++;
    if (stk.top==100)
    {
        printf("\n Stack overflow\n");
        exit(0);
    }
    stk.items[stk.top]=data;
}

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);
    strcpy(QUAD[Index].arg2,arg2);
    sprintf(QUAD[Index].result," t%d",tIndex++);
    strcpy(result,QUAD[Index++].result);
}
```

```
void yyerror()  
{  
    printf("\n Error on line no:%d",yylineno);  
    exit(0);  
}
```

Input



```
user@adithya-d-rajagopal:~/s7/cd/Exp3$ cat p3d.txt  
main()  
{  
    if(a<b)  
    {  
        a=a+b;  
    }  
    while(a<b)  
    {  
        a=a+b;  
    }  
    if(a<=b)  
    {  
        c=a-b;  
    }  
    else  
    {  
        c=a+b;  
    }  
}  
user@adithya-d-rajagopal:~/s7/cd/Exp3$
```

Output

```
user@adithya-d-rajagopal:~/s7/cd/Exp3$ lex p3d.l
user@adithya-d-rajagopal:~/s7/cd/Exp3$ yacc -d p3d.y
user@adithya-d-rajagopal:~/s7/cd/Exp3$ gcc lex.yy.c y.tab.c -ll
user@adithya-d-rajagopal:~/s7/cd/Exp3$ ./a.out p3d.txt
```

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

```
user@adithya-d-rajagopal:~/s7/cd/Exp3$
```

RESULT

The yacc specifications for the given syntactic categories were generated and the outputs were verified.