Practical No. 2

Aim: Write YACC specification
A) to check syntax of an arithmetic expression involving various operations such as addition, multiplication, subtraction, division. Also, convert this expression to postfix form.
B) To validate syntax of programming language constructs.
(Batch A1: switch-case, A2: do-while loop, A3: for loop, A4: if-then-else)

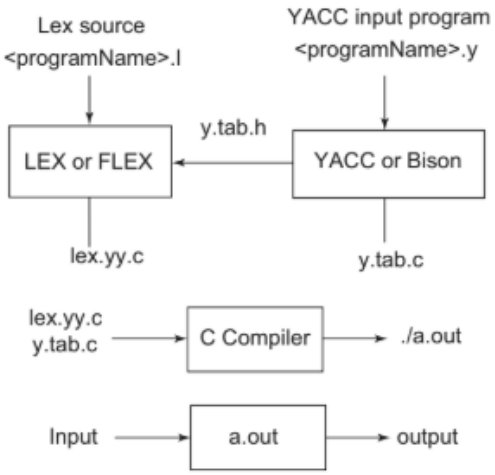
Theory:

YACC

- Yacc generates C code for syntax analyzer, or parser.
- Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

Yacc (Yet Another Compiler-Compiler) is a computer program for the Unix operating system developed by Stephen C. Johnson. It is a Look Ahead Left-to-Right (LALR) parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to Backus–Naur Form (BNF). Yacc is supplied as a standard utility on BSD and AT&T Unix. GNU-based Linux distributions include Bison, a forward-compatible Yacc replacement.
The input to Yacc is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse trees. Using an example from Johnson, if the call node (label, left, right) constructs a binary parse tree node with the specified label and children, then the rule. recognizes summation expressions and constructs nodes for them. The special identifiers $$, $1 and $3 refer to items on the parser's stack.
Yacc produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper.

Yacc Functioning



Format for Yacc file

A full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
Rules
```

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

If there are several grammar rules with the same left hand side, the vertical bar ``|'' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to Yacc as

```
A : B C D
  | E F
  | G
  ;
```

Inclusions in yacc declaration

%start	Specify the grammar's start symbol
%token	Declare a terminal symbol (token type name) with no precedence or associativity specified
%type	Declare the type of semantic values for a nonterminal symbol
%right	Declare a terminal symbol (token type name) that is right-associative
%left	Declare a terminal symbol (token type name) that is left-associative
%nonassoc	Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error, ex: x *op*. y *op*. z is syntax error)

Pseudovariables

- To facilitate easy communication between the actions and the parser, the action statements are altered slightly. '$' is used
- To return a value, the action normally sets the pseudovariable '$$' to some value.
- For example, {$$ = 1;}
- To obtain the values returned by previous actions and the lexical analyser, the action may use the pseudovariables $1, $2, . . .
- For example,
  A : B C D ;

$1 has the value returned by B,
$2 has the value returned by C
$3 the value returned by D.

Steps to execute YACC programs in Linux

flex *pgmname*.l
bison -d *pgm name*.y
gcc -c lex.yy.c *pgm name*.tab.c
gcc -o a.out lex.yy.o *pgm name*.tab.o -lfl
./a.out

Questions:

1. *What is the role of tab.h file and when is it created?*

Ans.1) Contains definitions for token names and it is created when yacc file is compiled.

2. *What is the role of tab.c file and when is it created?*

Ans.2) Contains an output file and it is created when lex is compiled.

3. *Where is main written?*

Ans.3) main() is written in yacc file.

4. *How is the program run by calling yylex internally from main?*

Ans.4) In the YACC file, you write your own main() function, which calls yyparse() at one point. The function yyparse() is created for you by YACC, and ends up in y.tab.c.
yyparse() reads a stream of token/value pairs from yylex(), which needs to be supplied.
Each call to yylex() returns an integer value which represents a token type. This tells YACC what kind of token it has read. The token may optionally have a value, which should be placed in the variable yylval.
The Lexer needs to be able to access yylval. In order to do so, it must be declared in the scope of the lexer as an extern variable. The original YACC neglects to do this for you, so you should add the following to your lexter, just beneath #include <y.tab.h>:
extern YYSTYPE yylval.

5. *Use of option –lfl and option –d*

Ans.5) –d : produces the y.tab.h file
-lfl : produces the executable file

6. *Which parsing technique does YACC use internally?*

Ans.6) Shift – Reduce parsing techniques for ex. Operator Precedence

Code:

Practical 2(a):
Lex:

```
%{
#include "practical2.tab.h"
%}
%%
[0-9]+ { yylval=atoi(yytext);return NUMBER; }
[a-zA-Z][a-zA-Z0-9_]* { return ID; }
\n { return NL ;}
. { return yytext[0]; }
%%
```

Yacc:

```
%token NUMBER ID NL
%left '+' '-'
%left '*' '/'
%%
stmt : exp NL { printf("\nValid Expression"); exit(0);}
;
exp : exp '+' exp { printf("+");}
| exp '-' exp { printf("-");}
| exp '*' exp { printf("*");}
| exp '/' exp { printf("/");}
| '(' exp ')' { printf("("); printf(")");}
| ID
| NUMBER  { printf("%d", $1);}
;
%%
int yyerror(char *msg)
{
printf("Invalid Expression\n");
exit(0);
}
main ()
{
printf("Enter the expression\n");
yyparse();
}
```

OUTPUT:

```
Enter the expression
1+2
12+
Valid Expression
```

Practical 2(b):

Lex:

```
%{
#include "practical2b.tab.h"
%}
%%
"for" { return F;}
"(" {return OB;}
")" {return CB;}
[a-z]+ {return VAR;}
";" {return SEM;}
"{" {return OCB;}
"}" {return CCB;}
[0-9]+ {return NUMBER;}
["<""">""<=""">=""="==""] {return OP;}
("++"|"--") {return INC;}
\n { return NL ;}
. { return yytext[0]; }
%%
```

Yacc:

```
%token NUMBER NL F OB CB OCB CCB VAR SEM OP INC
%left '+' '-'
%left '*' '/'
%%
stmt : F a b b d { printf("Valid Expression"); exit(0);}
;
a : OB
;
b : VAR OP NUMBER c
;
c : SEM
;
d : VAR INC e OCB NL exp SEM NL CCB
;
exp : exp '+' exp
| exp '-' exp
| exp '*' exp
| exp '/' exp
| '(' exp ')'
| exp '='
| VAR
;
e : CB
%%
```

```
int yyerror(char *msg)
{
printf("Invalid Expression\n");
exit(0);
}
main ()
{
yyparse();
}
```

Input.txt :

```
for(int i=0; i<10; i++){
a+b;
}
```

OUTPUT:
Valid Expression