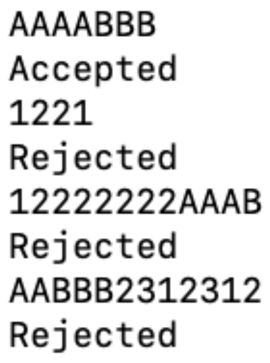
Compiler Lab Assignment 2

Ritabroto Ganguly
001910501090
BCSE-III A3

1. Design a grammar to recognise a string of the form AA...ABB...B, i.e. any number of As followed by any number of Bs. Use LEX or YACC to recognise it. Which one is a better option?

anbm.l

```
%%
^[A]+[B]+ {printf("Accepted");}
.+ {printf("Rejected");}
%%

int yywrap(){return 1;}
int main(){yylex(); return 0;}
```



```
AAAABBB
Accepted
1221
Rejected
12222222AAAB
Rejected
AABBB2312312
Rejected
```

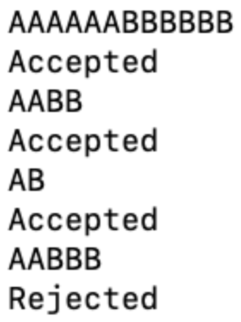
Change your grammar to recognise strings with equal numbers of As and Bs - now which one is better?

anbn.y

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

%token A B NL

%%
stmt: S NL {printf("Accepted\n");}
    | stmt S NL {printf("Accepted\n");}
    ;
S: A S B |/* empty string */
 ;
%%

void yyerror(char* s){printf("Rejected\n");exit(1);}
int main(){yyparse(); return 0;}
```

anbn.l

```
%{
#include "y.tab.h"
void yyerror(char* s);
%}

%%
A {return A;}
B {return B;}
\n {return NL;}
. {return yytext[0];}
%%

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



```
AAAAAABBBBBB
Accepted
AABB
Accepted
AB
Accepted
AABBB
Rejected
```

For the first part, lex is a better option than yacc, since there we are not concerned about keeping track of the number of As and Bs, hence our grammar can be easily represented using a regular expression and lexical analysers are designed exactly for this task of recognising regular expressions using finite automata logic where the current state can only hold the current symbol and check the next symbol.

On the other hand, yacc parsers can recognise context-free grammars using pushdown automata logic which have a much bigger memory than finite automata by using a stack, they can keep track of the number of As and Bs and hence they are used for the second part of our problem. Using yacc in the first part would have been an overkill.

2. Write the lex file and the yacc grammar for an expression calculator.
You need to deal with
i) binary operators '+', '*', '-';
ii) uniary operator '-';
iii) boolean operators '&', '|'
iv) Expressions will contain both integers and floating point numbers (up to 2 decimal places).
Consider left associativity and operator precedence by order of specification in yacc.

Along with the above mentioned features, I have a also added the features for

- ***Assignment of values to variables (variable names can be single letters only) using "=" . E.g. a=2; A=2;***
- ***Printing of arithmetic expressions and variables' values using "print" keyword. E.g. print a; print A;***

<u>Note</u>:
All statements in this language end with a ";"

<u>calc.y</u>

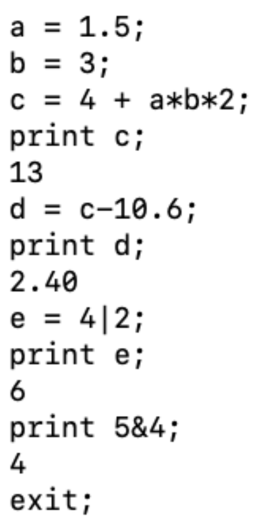
```
%{
void yyerror (char *s);
int yylex();
#include <stdio.h>     /* C declarations used in actions */
#include <stdlib.h>
#include <ctype.h>
double symbols[52];/*0 to 25 for uppercase letters and 26 to 51 for lowercase
letters*/
double symbolVal(char symbol);
void updateSymbolVal(char symbol, double val);
void printNumber(double num);
%}

%error-verbose
%union {double num; char id;}        /* Yacc definitions */
%token print
%token exit_command
%token <num> number
%token <id> identifier
%type <num> line exp term val
%type <id> assignment

/* descriptions of expected inputs     corresponding actions (in C) */
%%
line: assignment ';' {/*printf("Assigned\n")*/;}
    | exit_command ';' {exit(0);}
    | print exp ';' {printNumber($2);}
    | line assignment ';' {/*printf("Assigned\n")*/;}
    | line print exp ';' {printNumber($3);}
    | line exit_command ';' {exit(0);}
    ;

assignment: identifier '=' exp {updateSymbolVal($1,$3);}
          ;

exp: term {$$ = $1;}
   | exp '+' term {$$ = $1 + $3;}
   | exp '-' term {$$ = $1 - $3;}
   | exp '&' term {$$ = (int)$1 & (int)$3;}
   ;

term: val {$$ = $1;}
    | term '*' val {$$ = $1 * $3;}
    | term '|' val {$$ = (int)$1 | (int)$3;}
```

```
val: number {$$ = $1;}
   | identifier {$$ = symbolVal($1);}
   | '-' val {$$ = - $2;}
   ;
%%

void printNumber(double num){
        //printf("Printing ");
        int i = (int)num;
        if(i==num)
                printf("%d\n",i);
        else
                printf("%0.2lf\n",num);
}

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

/* returns the value of a given symbol */
double symbolVal(char symbol)
{
        int bucket = computeSymbolIndex(symbol);
        return symbols[bucket];
}

/* updates the value of a given symbol */
void updateSymbolVal(char symbol, double val)
{
        int bucket = computeSymbolIndex(symbol);
        symbols[bucket] = val;
}
```

```
int main (void) {
        /* init symbol table */
        int i;
        for(i=0; i<52; i++) {
                symbols[i] = 0;
        }

        return yyparse ( );
}

void yyerror (char *s) {fprintf (stderr, "%s\n", s);}
```

calc.l

```
%{
#include "y.tab.h"
void yyerror(char *s);
%}

%%
"print" {return print;}
"exit" {return exit_command;}
[a-zA-Z] {yylval.id = yytext[0]; return identifier;}//single lettered identifier
[0-9]+\.?[0-9]* {yylval.num = strtod(yytext,NULL); /*printf("Sending
%lf\n",yylval.num)*/;return number;}
[&|*+-=;] {/*printf("Sending %c\n",yytext[0])*/;return yytext[0];}
[ \t\n] ;
. {yyerror("Invalid token in lex");}
%%

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



```
a = 1.5;
b = 3;
c = 4 + a*b*2;
print c;
13
d = c-10.6;
print d;
2.40
e = 4|2;
print e;
6
print 5&4;
4
exit;
```