1. Regular expressions

a. Lex regular expression for all lowercase English words with the five vowels in order. Throughout we assume that each input line consists of a single word, that the five vowels are a, e, i, o, u, and that “in order” means that the word is of the form uavewixoyuz, where u, v, w, x, y, and z are any strings of lowercase letters (that may contain vowels). One Lex regular expression definition is:

```
L   [a-z]*
W   ^{L}a{L}e{L}i{L}o{L}u{L}$
```

Lex program:

```
{%
    #include <stdio.h>
    #include <string.h>
    int maxlen = 0;
    char maxstr[100];
%
%}
L     [a-z]*
W     {L}a{L}e{L}i{L}o{L}u{L}
%
{W}    {if (yyleng > maxlen)
            maxlen = yyleng;
            strcpy(maxstr, yytext);}

|.

int main()
{
    yylex();
    if (maxlen == 0)
        printf("no word was found\n");
    else
        printf("longest word is %s\n", maxstr);
}
```

First longest word in /usr/dict/words: adventitious

b. Lex regular expression for all lowercase English words with beginning and ending with the substring ad:
W \(^{ad([a-z]*ad)?\}$

First longest word in /usr/dict/words: ad

c. Lex regular expression for all lowercase English words with exactly one vowel:

C \([b-df-hj-np-tv-z]*\)
W \(^{[C][aeiou]}{[C]}\$

First longest word in /usr/dict/words: dystrophy
Note: Technically we should also include y as a vowel.

d. Lex regular expression for all lowercase English words in which the letters are in strictly increasing alphabetic order:

w \(^{a\?b\?c\?d\?e\?f\?g\?h\?i\?j\?k\?l\?m\?n\?o\?p\?q\?r\?s\?t\?u\?v\?w\?x\?y\?z}\$

First longest word in /usr/dict/words: almost

The lex programs for (b)-(d) are similar to that for (a).

2. Let L be the language \{ abx | x is any string of a's and b's not containing the substring ab \}.

a. A regular expression for L: ab\*a\*

b. From your regular expression construct a nondeterministic finite automaton (NFA) for L using the McNaughton-Yamada-Thompson algorithm:
c. From your NFA construct an equivalent deterministic finite automaton (DFA).

Here $A = \{1\}$, $B = \{2\}$, $C = \{3, 4, 6, 7, 9\}$, $D = \{8, 7, 9\}$, $E = \{5, 4, 6, 7, 9\}$. In addition, there is a “dead” state $F$ to which there is a transition on any unspecified input, and there is a transition from $F$ to $F$ on $a$ and $b$.

d. Minimize the number states in your DFA.

Here state $G$ is the merger of the two equivalent states $C$ and $E$ of the DFA in (c).
e. Show how your regular expression generates abbba, and how each of your finite automata recognizes abbba:

Let $R$ be the regular expression $abb^*a$. The first $a$ in $R$ generates an $a$, then the first $b$ generates a $b$, then $b^*$ generates two $b$’s, and the final $a$ generates an $a$. Thus $R$ generates the string $abbba$.

The NFA in (b) makes the following sequence of state transitions:

```
1   2   3   5   5   8
4   4   4   7
6   6   6   9
7   7   7
9   9   9
```

The DFA in (c) makes the following sequence of state transitions:

```
a   b   b   b   a
A   B   C   E   E   D
```

The DFA in (d) makes the following sequence of state transitions:

```
a   b   b   b   a
A   B   G   G   G   D
```

3. Sequences of boolean expressions

a. Construct a grammar that generates newline-terminated sequences of boolean expressions whose value is true. The expressions can contain the logical constants TRUE and FALSE, and the boolean operators AND, OR, and NOT. The expressions can contain parentheses. For example, your grammar should generate the expression TRUE OR FALSE, but not TRUE AND FALSE.
%left OR
%left AND
%right NOT

lines → lines true NL | ε
true → true AND true
      | true OR false | false OR true
      | true OR true
      | NOT false | (true) | TRUE

false → false OR false
      | false AND true | true AND false
      | false AND false
      | NOT true | (false) | FALSE

b. Show the parse tree according to your grammar for the input
   TRUE AND NOT FALSE OR FALSE:
c. Implement an interpreter that takes as input lines of boolean expressions and produces as output the truth value of each expression. You can use lex and yacc or their equivalents to implement your interpreter. Print the source code for your interpreter and the sequences of commands you used to create it and test it. The program below is patterned after the desk calculator in ALSU, Figs. 4.60 and 4.61, pp. 295-6.

--------bool.l lexical analyzer--------
%}
#include <ctype.h>
#include <stdio.h>
%}
%token AND FALSE NOT
%left OR
%right AND
%
lines : lines expr '
' { $2?printf("true\n"); printf("false\n"); } 
| lines "\n" /* empty */ 
| error "\n" { yyerror("reenter last line:"); y yerrok; } 
;
expr : expr OR expr { $$ = $1 || $3; } 
| expr AND expr { $$ = $1 && $3; } 
| NOT expr { $$ = !$2; } 
| '(' expr ')' { $$ = $2; } 
| TRUE { $$ = 1; } 
| FALSE { $$ = 0; } 
;

--------bool.y translator--------
{%
#include <ctype.h>
#include <stdio.h>
%}{
%token AND FALSE NOT OR TRUE
%left OR
%right NOT
%
lines : lines expr 'n' { $2?printf("true\n"); printf("false\n"); } 
| lines 'n' /* empty */ 
| error 'n' { yyerror("reenter last line:"); y yerrok; } 
;
expr : expr OR expr { $$ = $1 || $3; } 
| expr AND expr { $$ = $1 && $3; } 
| NOT expr { $$ = !$2; } 
| '(' expr ')' { $$ = $2; } 
| TRUE { $$ = 1; } 
| FALSE { $$ = 0; } 
;

--------Compile bool--------
>lex bool.l
>yacc bool.y
>gcc y.tab.c -ly -ll -o bool
d. Run your interpreter on the inputs
   TRUE OR FALSE
   TRUE AND FALSE
   TRUE AND (TRUE OR FALSE)

   TRUE OR FALSE
   result: true

   TRUE AND FALSE
   result: false

   TRUE AND (TRUE OR FALSE)
   result: true

4. Let \( L \) be the language consisting of all strings of \( a \)'s and \( b \)'s having the same number of \( a \)'s as \( b \)'s.

   a. Construct an LL(1) grammar for \( L \).

   \[
   S \rightarrow aAbS \mid bBaS \mid \varepsilon \\
   A \rightarrow aAbA \mid \varepsilon \\
   B \rightarrow bBaB \mid \varepsilon
   \]

   b. Construct a leftmost derivation for the string \( abbaab \).

   \[
   S \Rightarrow aAbS \quad // \quad S \rightarrow aAbS \\
   \Rightarrow abS \quad // \quad A \rightarrow \varepsilon \\
   \Rightarrow abbBaS \quad // \quad S \rightarrow bBaS \\
   \Rightarrow abbaS \quad // \quad B \rightarrow \varepsilon \\
   \Rightarrow abbaaAbS \quad // \quad S \rightarrow aAbS \\
   \Rightarrow abbaabS \quad // \quad A \rightarrow \varepsilon \\
   \Rightarrow abbaab \quad // \quad S \rightarrow \varepsilon
   \]
c. Construct a predictive parsing table for your grammar.

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>a</td>
<td>$</td>
</tr>
<tr>
<td>S → aAbS</td>
<td>S → bBaS</td>
<td>S → ε</td>
</tr>
<tr>
<td>A</td>
<td>a → aAbA</td>
<td>ε</td>
</tr>
<tr>
<td>B</td>
<td>B → ε</td>
<td>B → bBaB</td>
</tr>
</tbody>
</table>

d. Show how your predictive parser parses the string \textit{abbaab}. (Top of stack is on the left.)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>S$</td>
<td>abbaab$</td>
<td>S → aAbS</td>
</tr>
<tr>
<td>aAbS$</td>
<td>abbaab$</td>
<td>match a</td>
</tr>
<tr>
<td>AbS$</td>
<td>bbaab$</td>
<td>A → ε</td>
</tr>
<tr>
<td>bS$</td>
<td>bbaab$</td>
<td>S → bBaS</td>
</tr>
<tr>
<td>S$</td>
<td>baab$</td>
<td>match b</td>
</tr>
<tr>
<td>bBaS$</td>
<td>baab$</td>
<td>S → bBaS</td>
</tr>
<tr>
<td>BaS$</td>
<td>aab$</td>
<td>match b</td>
</tr>
<tr>
<td>aS$</td>
<td>aab$</td>
<td>B → ε</td>
</tr>
<tr>
<td>S$</td>
<td>ab$</td>
<td>match a</td>
</tr>
<tr>
<td>aAbS$</td>
<td>ab$</td>
<td>S → aAbS</td>
</tr>
<tr>
<td>AbS$</td>
<td>b$</td>
<td>match a</td>
</tr>
<tr>
<td>bS$</td>
<td>b$</td>
<td>A → ε</td>
</tr>
<tr>
<td>S$</td>
<td>$</td>
<td>match b</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>S → ε</td>
</tr>
</tbody>
</table>

5. Augmented grammar

(0) $S' \rightarrow S$

(1) $S \rightarrow SA$

(2) $S \rightarrow A$

(3) $A \rightarrow a$
LR(0) automaton:

$I_0$:
- $S' \rightarrow •S$
- $S \rightarrow •SA$
- $S \rightarrow •A$
- $A \rightarrow •a$

$I_1$:
- $S' \rightarrow S•$
- $S \rightarrow S•A$
- $A \rightarrow •a$

$I_2$:
- $S \rightarrow A•$

$I_3$:
- $A \rightarrow a•$

$I_4$:
- $S \rightarrow SA•$

SLR(1) parsing table:

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$a$</td>
<td>$S$</td>
</tr>
<tr>
<td>1</td>
<td>$s3$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$r2$</td>
<td>$r2$</td>
</tr>
<tr>
<td>3</td>
<td>$r3$</td>
<td>$r3$</td>
</tr>
<tr>
<td>4</td>
<td>$r1$</td>
<td>$r1$</td>
</tr>
</tbody>
</table>

Since each entry in this table is uniquely defined, this grammar is SLR(1).

However, this grammar is not LL(1) since $S \rightarrow SA|A$ and $SA$ and $A$ can both derive strings beginning with the terminal $a$. 