1 Lexing

1. Write regular expressions for each of the following languages, or explain why no such regular expression exists.

(a) Strings over the alphabet \{a, b, c\} where every b is immediately followed by at least one c.
   \((a \mid c \mid bc^+)\)\(^*\)

(b) Strings over the alphabet \{a, b, c\} where there are never two b’s in a row.
   \((b?a \mid b?c)^*b?\)
   or
   \(b?(ab? \mid cb?)^*\)

(c) Non-empty strings over the set of alphanumeric characters where the first character is a lowercase letter and the last character is a number.
   \([a−z][a−zA−Z0−9]^*[0−9]\)

(d) Strings over the set of alphanumeric characters where there are exactly as many letters as numbers.
   None exists; this would require remembering an arbitrary large number of letters/numbers seen so far. (Alternatively: this is equivalent to matching left- and right-parentheses.)

2. Convert each of the following regular expressions into a (possibly nondeterministic) finite-state automaton.

(a) \(xy(x^*) \mid yx(y^*)\)

(b) \((xyz)^*(yzx)^*\)
(c) $x^* (y \mid z)) \mid y^* x z$

3. Consider the following excerpt of a .lex file:

```
" " { continue; }
"int" { return INT; }
"bool" { return BOOL; }
"[" { return LBRACK; }
"]" { return RBRACK; }
"->" { return ARROW; }
"special:"[a-z]* { return SPECIAL; }
```

(a) Write the list of tokens this lexer would produce when given the following string: `int[] -> special:bool`

```
INT LBRACK RBRACK ARROW SPECIAL
```

(b) Suppose we wanted to extend the lexer so that if it sees the three-character string `end` at any point in the input, it discards those characters and any input that appears after them, producing no further tokens. Write one or more lexer rules that implement this behavior.

```
"end".* { continue; }
or
%s END

"end" { begin END; }
<END>.* { continue; }
```
2 Parsing

1. Write a context-free grammar for each of the following languages.

   (a) Sequences of 1 or more numbers separated by + signs. You may use the terminal symbol number to represent a number.

   \[ S \rightarrow \text{number} \mid S + S \]

   (b) Strings over the alphabet \{a, b, c\} that have the same number of a’s as b’s.

   \[ S \rightarrow \varepsilon \mid cS \mid aSbS \mid bSaS \]

   (c) The language of the regular expression \((xyz)^* (yzx)^*\)

   \[ S \rightarrow AB \]
   \[ A \rightarrow \varepsilon \mid xyzA \]
   \[ B \rightarrow \varepsilon \mid yzxB \]

2. Consider the following grammar, with the terminals true, false, &&, and ||:

   \[ T \rightarrow \text{true} \mid \text{false} \mid T \&\& T \mid T || T \]

   (a) Demonstrate that the grammar is ambiguous by showing at least two parse trees for the string \text{true} \&\& \text{false} || \text{true}.

   (b) Refactor the grammar so that 1) it is not ambiguous, 2) the && operator is right-associative, 3) the || operator is left-associative, and 4) && has higher precedence (binds tighter) than ||.

   \[ T \rightarrow T || A \mid A \]
   \[ A \rightarrow B \&\& A \mid B \]
   \[ B \rightarrow \text{true} \mid \text{false} \]
(c) Draw the parse tree for `true && false || true` in your refactored grammar.
3. Consider the following grammar, with nonterminals $\text{noun}$, $\text{verb}$, $\text{modifier}$, and $\$$:

$$
T \rightarrow \text{SVO}\$
$$
S \rightarrow \text{noun} \mid \text{MS}
$$
V \rightarrow \text{verb} \mid \text{MV}
$$
O \rightarrow \varepsilon \mid S
$$
M \rightarrow \text{modifier}
$$

(a) Write the FIRST and FOLLOW sets for this grammar.

<table>
<thead>
<tr>
<th></th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T\rightarrow\text{SVO}$</td>
<td>noun, modifier</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>S</td>
<td>noun</td>
<td>verb, modifier, $$</td>
</tr>
<tr>
<td>S</td>
<td>modifier</td>
<td>verb, modifier, $$</td>
</tr>
<tr>
<td>V</td>
<td>verb</td>
<td>noun, modifier, $$</td>
</tr>
<tr>
<td>V</td>
<td>modifier</td>
<td>noun, modifier, $$</td>
</tr>
<tr>
<td>O</td>
<td>$\varepsilon$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>O</td>
<td>noun, modifier</td>
<td>$$</td>
</tr>
<tr>
<td>M</td>
<td>modifier</td>
<td>noun, verb, modifier</td>
</tr>
</tbody>
</table>

or

<table>
<thead>
<tr>
<th></th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>noun, modifier</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>S</td>
<td>noun, modifier</td>
<td>verb, modifier, $$</td>
</tr>
<tr>
<td>V</td>
<td>verb, modifier</td>
<td>noun, modifier, $$</td>
</tr>
<tr>
<td>O</td>
<td>noun, modifier</td>
<td>$$</td>
</tr>
<tr>
<td>M</td>
<td>modifier</td>
<td>noun, verb, modifier</td>
</tr>
</tbody>
</table>

(b) Fill in the LL(1) parse table for this grammar.

<table>
<thead>
<tr>
<th>noun</th>
<th>verb</th>
<th>modifier</th>
<th>$$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$T\rightarrow\text{SVO}$</td>
<td>$T\rightarrow\text{SVO}$</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>$S\rightarrow\text{noun}$</td>
<td>$S\rightarrow\text{MS}$</td>
<td></td>
</tr>
<tr>
<td>$V$</td>
<td>$V\rightarrow\text{verb}$</td>
<td>$V\rightarrow\text{MV}$</td>
<td></td>
</tr>
<tr>
<td>$O$</td>
<td>$O\rightarrow\text{S}$</td>
<td>$O\rightarrow\text{S}$</td>
<td></td>
</tr>
<tr>
<td>$M$</td>
<td>$M\rightarrow\text{modifier}$</td>
<td>$O\rightarrow\varepsilon$</td>
<td></td>
</tr>
</tbody>
</table>

(c) Is the grammar LL(1)? Why or why not?

Yes. There is at most one production in each cell of the table.

(d) If we were to build a LR(0) parser for this grammar, the start state would include the item $T\rightarrow\text{SVO}\$$

What other items would it include?

$$
S\rightarrow\text{.noun}
$$
$$
S\rightarrow\text{.MS}
$$
$$
M\rightarrow\text{.modifier}
$$
(e) Draw the start state of the LR(0) parser for the grammar, including its transition edges and the first item in each of its successor states.

```
T → .SVO$
S → .noun
S → .MS
M → .modifier

S → M,S
```

3 Semantic Analysis

1. Suppose you were writing a type checker for a language with the following grammar:

\[
E → \text{number} | \text{true} | \text{false} | \text{ident} := E
\]

The type of ASTs for the language is defined as:

```c
typedef struct A_exp{
    enum {E_num, E_true, E_false, E_assign} kind;
    union {int ival; struct {char *id; struct A_exp *rhs;} operands;} u;
} *A_exp;
```

Fill in the code for the `E_assign` case of the type checker, where an assignment typechecks if the type of the identifier on the left-hand side is the same as the type of the expression on the right-hand side, and the return type of the assignment is the type of the expression. You may assume the existence of a function `Ty lookup(char *id)` that looks up the type of an identifier in the environment. If the assignment has a type error, return the value `Ty_err`.

```c
Ty typecheck(A_exp e){
    switch(e->kind){
        case E_num:
            return Ty_int;
        case E_true:
            return Ty_bool;
        case E_false:
            return Ty_bool;
        case E_assign:
            Ty t = lookup(e->u.operands.id);
            if(t && t == typecheck(e->u.operands.rhs)) return t;
            else return Ty_err;
    }
}
```