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.

   (b) Strings over the alphabet \{a, b, c\} where there are never two b’s in a row.

   (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.

   (d) Strings over the set of alphanumeric characters where there are exactly as many letters as numbers.

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

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

   (b) \( (xyz)^*(yzx)^* \)

   (c) \( x^*(y \mid z) ) \mid y^*xz \)
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`

(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.

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.

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

   (c) The language of the regular expression `(xyz)∗(yzx)∗`
2. Consider the following grammar, with the nonterminals `true`, `false`, `&&`, and `||`:

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

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

(c) Draw the parse tree for `true && false || true` in your refactored grammar.
3. Consider the following grammar, with terminals noun, verb, and modifier:

\[
\begin{align*}
T & \rightarrow SVO$
\S & \rightarrow \text{noun} \mid MS \\
V & \rightarrow \text{verb} \mid MV \\
O & \rightarrow \varepsilon \mid S \\
M & \rightarrow \text{modifier}
\end{align*}
\]

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

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

<table>
<thead>
<tr>
<th></th>
<th>noun</th>
<th>verb</th>
<th>modifier</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>S</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>V</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>O</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>M</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

(d) If we were to build a LR(0) parser for this grammar, the start state would include the item

\[ T \rightarrow .SVO$ \]

What other items would it include?
(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.

3 Semantic Analysis

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

\[ E \rightarrow \text{number} \mid \text{true} \mid \text{false} \mid \text{ident} := E \]

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:
```

```c
```