1. (a) [$\star$] Consider the following expressions in Higher Order abstract syntax. Convert them to concrete syntax.
   i. (Let (Num 3) (x. (Let (Plus x (Num 1)) (x. (Plus x x)))))
   ii. (Plus (Let (Num 3) (x. (Plus x x))) (Let (Num 2) (y. (Plus y (Num 4)))))
   iii. (Let (Num 2) (x. (Let (Num 1) (y. (Plus x y)))))

(b) [$\star$] Apply the substitution $x := (Plus z 1)$ to the following expressions:
   i. (Let (Plus x z) (y. (Plus x y)))
   ii. (Let (Plus x z) (x. (Plus z z)))
   iii. (Let (Plus x z) (z. (Plus x z)))

(c) [$\star$] Which variables are shadowed in the following expression and where?
   (Let (Plus y 1) (x. (Let (Plus x 1) (y. (Let (Plus x y) (x. (Plus x y)))))))

2. Here is a concrete syntax for specifying binary logic gates with convenient if – then – else syntax. Note that the else clause is optional, which means we must be careful to avoid ambiguity – we introduce mandatory parentheses around nested conditionals:

<table>
<thead>
<tr>
<th>⊤ Output</th>
<th>⊥ Output</th>
<th>α Input</th>
<th>β Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>c Input</td>
<td>t IExpr</td>
<td>e Expr</td>
<td>c Input</td>
</tr>
<tr>
<td>if c then t else e Expr</td>
<td>if c then t Expr</td>
<td>c IExpr</td>
<td>x Output</td>
</tr>
<tr>
<td>e Expr</td>
<td>(c) IExpr</td>
<td>e IExpr</td>
<td>c IExpr</td>
</tr>
<tr>
<td>⊤ Output</td>
<td>⊤ IExpr</td>
<td>⊤ Expr</td>
<td>⊤ IExpr</td>
</tr>
</tbody>
</table>

If an else clause is omitted, the result of the expression if the condition is false is defaulted to ⊥. For example, an AND or OR gate could be specified like so:

**AND**: if $\alpha$ then (if $\beta$ then ⊤)

**OR**: if $\alpha$ then ⊤ else (if $\beta$ then ⊤)

Or, a NAND gate:

if $\alpha$ then (if $\beta$ then ⊥ else ⊤) else ⊤

(a) [$\star\star$] Devise a suitable abstract syntax $A$ for this language.

(b) [$\star$] Write rules for a parsing relation (⇒⇒) for this language.

(c) [$\star$] Here’s the parse derivation tree for the NAND gate above:

<table>
<thead>
<tr>
<th>β Input</th>
<th>⊥ Output</th>
<th>⊤ Output</th>
<th>⊥ IExpr</th>
</tr>
</thead>
<tbody>
<tr>
<td>if β then ⊥ else</td>
<td>IExpr</td>
<td>(if β then ⊥ else) IExpr</td>
<td>⊤ IExpr</td>
</tr>
<tr>
<td>α Input</td>
<td>if α then (if β then ⊥ else) IExpr</td>
<td>⊥ IExpr</td>
<td></td>
</tr>
<tr>
<td>if α then (if β then ⊥ else) IExpr</td>
<td>⊤ IExpr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in the right-hand side of this derivation tree with your parsing relation, labelling each step as you progress down the tree.
3. Here is a first order abstract syntax for a simple functional language, Lc. In this language, a lambda term defines a function. For example, \texttt{lambda } x \ (\texttt{var } x) \texttt{ is the identity function, which simply returns its input.}

\[
\begin{array}{cccc}
  e_1 \mathrm{Lc} & e_2 \mathrm{Lc} & x \mathrm{ VarName} & e \mathrm{ Lc} & x \mathrm{ VarName} \\
  \text{App } e_1 \ e_2 \mathrm{Lc} & \text{Lambda } x \ e \mathrm{Lc} & \text{Var } x \mathrm{Lc} \\
\end{array}
\]

(a) [⋆] Give an example of name shadowing using an expression in this language, and provide an \(\alpha\)-equivalent expression which does not have shadowing.

(b) [⋆⋆] Here is an incorrect substitution algorithm for this language:

\[
\begin{align*}
  (\text{App } e_1 \ e_2)[v := t] & \Rightarrow \text{App} \ (e_1[v := t]) \ (e_2[v := t]) \\
  (\text{Var } v)[v := t] & \Rightarrow t \\
  (\text{Lambda } x \ e)[v := t] & \Rightarrow \text{Lambda } x \ (e[v := t])
\end{align*}
\]

What is wrong with this algorithm? How can you correct it?

(c) [⋆⋆] Aside from the difficulties with substitution, using arbitrary strings for variable names in first-order abstract syntax means that \(\alpha\)-equivalent terms can be represented in many different ways, which is very inconvenient for analysis. For example, the following two terms are equivalent, but have different representations:

\[
\begin{align*}
  \text{Lambda } x \ (\text{Lambda } y \ (\text{App } x \ (\text{Var } y))) \\
  \text{Lambda } a \ (\text{Lambda } b \ (\text{App } x \ (\text{Var } a) \ (\text{Var } b)))
\end{align*}
\]

One technique to achieve canonical representations (i.e \(\alpha\)-equivalence is the same as equality) is called higher order abstract syntax (HOAS). Explain what HOAS is and how it solves this problem.