1. (a) [⋆] 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)))))`

Solution: `let x = 3 in let x = x + 1 in x + x`

ii. `(Plus (Let (Num 3) (x. (Let (Plus x x) x. (Plus x x)))) (Let (Num 2) (y. (Plus y (Num 4)))))`

Solution: `let x = 3 in x + x + (let y = 2 in y + 4)`

iii. `(Let (Num 2) (x. (Let (Num 1) (y. (Plus x y)))))`

Solution: `let x = 2 + (let y = 1 in x + y)`

(b) [⋆] Apply the substitution `x := (Plus z 1)` to the following expressions:

i. `(Let (Plus x z) (y. (Plus x y)))`

Solution: `(Let (Plus (Plus z 1) z) (y. (Plus (Plus z 1) y)))`

ii. `(Let (Plus x z) (x. (Plus z z)))`

Solution: `(Let (Plus (Plus z 1) z) (x. (Plus z z)))`

iii. `(Let (Plus x z) (z. (Plus x z)))`

Solution: Undefined without applying α-renaming first. Can safely substitute after renaming the bound `z` to `a`: `(Let (Plus (Plus z 1) z) (a. (Plus (Plus z 1) a)))`

(c) [⋆] 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)))))))`

Solution: The innermost let shadows the binding of `x` from the outermost let. The middle let shadows the free `y` mentioned in the outermost let.

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:

```
\( \begin{array}{cccc}
\top & \text{OUTPUT} & \bot & \text{OUTPUT} \\
\text{c INPUT} & t \text{ IEXPR} & e \text{ INPUT} & \alpha \text{ INPUT} \\
\text{if } c \text{ then } t \text{ else } e \text{ IEXPR} & \text{if } c \text{ then } t \text{ IEXPR} & x \text{ OUTPUT} & \beta \text{ INPUT} \\
\end{array} \)
```
If an else clause is omitted, the result of the expression if the condition is false is defaulted to $\bot$. For example, an AND or OR gate could be specified like so:

\[
\text{AND}: \text{if } \alpha \text{ then } (\text{if } \beta \text{ then } \top)
\]
\[
\text{OR}: \text{if } \alpha \text{ then } \top \text{ else } (\text{if } \beta \text{ then } \top)
\]

Or, a NAND gate:

\[
\text{if } \alpha \text{ then } (\text{if } \beta \text{ then } \bot \text{ else } \top) \text{ else } \top
\]

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

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Input} & \text{Output} & \text{Input}_a & \text{Input}_b \\
\hline
x & e & c & t & e & x & A \\
\hline
\end{array}
\]

(b) [*] Write rules for a parsing relation ($\leftarrow\to$) for this language.

\[
\begin{array}{|c|c|}
\hline
\text{Input} & \text{Output} \\
\hline
\top & \text{Top} \\
\bot & \text{Bot} \\
\text{c} & \text{c}' \\
\text{t} & \text{t}' \\
\text{e} & \text{e}' \\
\text{IF}_1 & \text{IF}_2 \\
\text{IF}_3 & \text{IF}_4 \\
\text{SHUNT}_1 & \text{SHUNT}_2 \\
\hline
\end{array}
\]

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

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Input} & \text{Output} & \text{Input}_a & \text{Input}_b \\
\hline
\alpha & \beta & \text{t} & \text{f} & \text{t} & \text{f} & \text{t} \\
\text{IF}_1 & \text{IF}_2 & \text{SHUNT}_1 & \text{SHUNT}_2 \\
\hline
\end{array}
\]

Fill in the right-hand side of this derivation tree with your parsing relation, labelling each step as you progress down the tree.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Input} & \text{Output} & \text{Input}_a & \text{Input}_b \\
\hline
\alpha & \beta & \text{t} & \text{f} & \text{t} & \text{f} & \text{t} \\
\text{IF}_1 & \text{IF}_2 & \text{SHUNT}_1 & \text{SHUNT}_2 \\
\hline
\end{array}
\]

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, $\text{lambda } x \ (\text{var } x)$ is the identity function, which simply returns its input.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Input} & \text{Output} & \text{Input}_a & \text{Input}_b \\
\hline
\alpha & \beta & \text{t} & \text{f} & \text{t} & \text{f} & \text{t} \\
\text{IF}_1 & \text{IF}_2 & \text{SHUNT}_1 & \text{SHUNT}_2 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Input} & \text{Output} & \text{Input}_a & \text{Input}_b \\
\hline
\alpha & \beta & \text{t} & \text{f} & \text{t} & \text{f} & \text{t} \\
\text{IF}_1 & \text{IF}_2 & \text{SHUNT}_1 & \text{SHUNT}_2 \\
\hline
\end{array}
\]
(a) [⋆] Give an example of name shadowing using an expression in this language, and provide an α-equivalent expression which does not have shadowing.

**Solution:** A simple example is \( \text{Lambda } x \ (\text{Lambda } x \ (\text{Var } x)) \). Here, the name \( x \) is shadowed in the inner binding.

An \( \alpha \)-equivalent expression without shadowing would use a different variable \( y \), i.e:

\[ \text{Lambda } x \ (\text{Lambda } y \ (\text{Var } y)) \]

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

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

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

**Solution:** The substitution doesn’t deal with name clashes. The rule for lambdas should look like this:

\[
(\text{Lambda } x \ e)[v := t] \mapsto \begin{cases} 
\text{Lambda } x \ (e[v := t]) & \text{if } x \neq v \text{ and } x \notin FV(t) \\
\text{Lambda } x \ e & \text{if } x = v \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

(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:

\[ \text{Lambda } x \ (\text{Lambda } y \ (\text{App } (\text{Var } x) \ (\text{Var } y))) \]

\[ \text{Lambda } a \ (\text{Lambda } b \ (\text{App } (\text{Var } a) \ (\text{Var } b))) \]

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.

**Solution:** Higher order abstract syntax encodes abstraction in the meta-logic level, or in the language implementation, rather than as a first-order abstract syntax construct.

First order abstract syntax might represent a term like \( \lambda x. x \) as something like \( \text{Lambda} \ "x" \ (\text{Var} \ "x") \), where literal variable name strings are placed in the abstract syntax directly.

Higher order abstract syntax, however, would place a function inside the abstract syntax, i.e \( \text{Lambda} \ (\lambda x. \ x) \), where the variable \( x \) is a meta-variable (or a variable in the language used to implement our interpreter, rather than the language being implemented). This function is (extensionally) equal to any other \( \alpha \)-equivalent function, and therefore we can consider two \( \alpha \)-equivalent terms to be equal with HOAS, assuming extensionality (that is, a function \( f \) equals a function \( g \) if and only if, for all \( x \), \( f(x) = g(x) \)).

For example, a first order Haskell implementation of the above syntax might look like this:

```haskell
type VarName = String
data AST = App AST AST
| Var VarName
| Lambda VarName AST
test = Lambda $ "x" (Lambda $ "y" (App $ "x" $ "y"))
```

Whereas a higher order syntax might look like this:

```haskell
data AST = App AST AST
| Lambda (AST -> AST)
test = Lambda $ \x -> Lambda $ \y -> App x y
```
There is no way in Haskell, for example, to determine that we used the names \( x \) and \( y \) for those function arguments. The only way for a Haskell function \( f \) to be distinguished from a function \( g \) is for \( f \ x \) to be different from \( g \ x \) for some \( x \) (i.e. extensionality). As \( \alpha \)-equivalent Haskell functions cannot be so distinguished, we must judge a term as equal to any other in its \( \alpha \)-equivalence class.