1. Free and Bound Variables

(a) Write the following λ-terms in their fully-parenthesized, curried forms. Identify the bound and free variables. For the bound variables, indicate which abstraction operator binds them.

(i) $\lambda xyz.\, z\, x\, \lambda x.\, x$

(ii) $\lambda xy.\, (\lambda z.\, z\, y)\, \lambda x.\, z\, x$

(iii) $(\lambda x.\, y\, \lambda y.\, x\ y)\, \lambda z.\, y\ z$

(b) Reduce (a)(iii) to normal form using α- and β-reduction. In each step, underline the redex and indicate which rule is being applied.

Two of the six rules for safe substitution are

$$(\lambda y.\, e_0)\,\{e_1/x\} \;\triangleq\; \lambda y.\, (e_0\,\{e_1/x\}) \qquad \text{where } y \neq x \text{ and } y \notin FV(e_1)$$
$$(\lambda y.\, e_0)\,\{e_1/x\} \;\triangleq\; \lambda z.\, (e_0\,\{z/y\}\,\{e_1/x\}) \qquad \text{where } y \neq x,\ z \neq x,\ z \notin FV(e_0), \text{ and } z \notin FV(e_1).$$

We also defined α-reduction as

$$\lambda x.\, e \;\xrightarrow{\alpha}\; \lambda y.\, (e\{y/x\}) \quad \text{where } y \notin FV(e).$$

(c) These rules contain a number of side-conditions of the form $y \notin FV(e)$ whose purpose may not be immediately apparent. Show by counterexample that each side-condition is independently necessary to avoid variable capture. Each counterexample should involve a pair of terms that either converge to different normal forms or such that one converges and the other diverges.

2. Encoding Lists

We would like to encode lists $[x_1;\, \ldots;\, x_n]$ and list operators corresponding to the OCaml operators `::`, `List.hd`, and `List.tl` as λ-terms. We could use pair, first, and second defined in lecture, but with that encoding, there seems to be no reasonable encoding of the empty list or a way to check for it.

Here is an alternative approach. Define

$$\mathsf{true} \;\triangleq\; \lambda xy.\, x \qquad\qquad \mathsf{nil} \;\triangleq\; \lambda x.\, x\ \mathsf{false}$$
$$\mathsf{false} \;\triangleq\; \lambda xy.\, y \qquad\qquad \mathsf{list} \;\triangleq\; \lambda h.\, \lambda t.\, \lambda x.\, x\ \mathsf{true}\ h\ t$$

We will consider $[x_1;\, x_2;\, \ldots;\, x_n]$ as syntactic sugar for $(\mathsf{list}\ x_1\ (\mathsf{list}\ x_2\ (\,\cdots\ (\mathsf{list}\ x_n\ \mathsf{nil})\ \cdots\,)))$.

(a) Write λ-terms head and tail such that $\mathsf{head}\ (\mathsf{list}\ e_1\ e_2) = e_1$ and $\mathsf{tail}\ (\mathsf{list}\ e_1\ e_2) = e_2$. Show the reductions that verify this. The functions head and tail are not required to do anything sensible when applied to nil or to any non-list. Show that $(\mathsf{head}\ (\mathsf{tail}\ [x_1;\, x_2])) = x_2$.

(b) Write a λ-term empty that returns true on input nil and false on any other list. It need not do anything sensible when applied to non-lists.

(c) Write a λ-term curry that converts a function that takes inputs of the form $[x;\, y]$ to a function that does the same thing, but takes its arguments one at a time. Write a λ-term uncurry that converts in the opposite direction. For the uncurried function, you need not worry about inputs not of the form $[x;\, y]$.

(d) Write a λ-term map that accepts a function and a list and returns a new list containing the results of the function applied to each element of the input list. For example, if succ is the successor function, then

$$\mathsf{map\ succ}\ [\bar{1};\ \bar{2};\ \bar{3};\ \bar{4}] \ \rightarrow\ [\bar{2};\ \bar{3};\ \bar{4};\ \bar{5}].$$

(*Hint.* Use the fixpoint combinator Y defined in lecture.)

3. Implementing the λ-Calculus

The archive `lambda.zip` contains a partial implementation of some useful λ-calculus mechanisms. In particular, it contains an implementation of the call-by-value reduction strategy, and you can use it to try out evaluation. The syntax is similar to OCaml: $\lambda x.\, e$ is written `fun x -> e`. The parser also accepts multiple arguments, as in `fun x y -> e` for $\lambda xy.\, e$.

There is a read-eval-print loop that will accept a sequence of lines that you type in, terminated by a blank line. The program will then parse your input and display the current expression. You can then hit enter repeatedly to step through the CBV evaluation sequence.

```
?  (fun x y -> x) (fun z -> z) (fun u v -> u)
(fun x y -> x) (fun z -> z) fun u v -> u
>>
(fun y -> (fun z -> z)) fun u v -> u
>>
Result:  fun z -> z
?
```

(a) The construct `let x = e1 in e2` is syntactic sugar for `(fun x -> e2) e1`. Implement the `let` statement. The parser already accepts it, you just have to fill out the stubs in `lambda.ml`.

(b) The substitution function `subst` does not check for the capture of free variables.

```
?  (fun x -> fun y -> x) (fun z -> y)
(fun x -> (fun y -> x)) fun z -> y
>>
Result:  fun y -> fun z -> y
?
```

Make it do so.