CS 421, Fall 2012

Sample Final Questions & Solutions

You should review the questions from the sample midterm exams, the practice midterm exams, and the assignments (MPs, MLs and WAs), as well as these question.

1. Write a function `get_primes : int -> int list` that returns the list of primes less than or equal to the input. You may use the built-in functions `/` and `mod`. You will probably want to write one or more auxiliary functions. Remember that 0 and 1 are not prime.

Solution:
```
let rec every p l = match l with [] -> true | x::xs -> p x && every p xs
let not_divides n q = ((q = 0) || not(n mod q = 0))
let rec get_primes n =
match n with 0 -> []
| 1 -> []
| _ -> let primes = get_primes (n-1) in
   if every (not_divides n) primes then n::primes else primes
```

2. Write a tail-recursive function `largest: int list -> int option` that returns `Some` of the largest element in a list if there is one, or else `None` if the list is empty.

Solution:
```
let rec largest_aux lgst list =
    match list with [] -> lgst
    | x :: xs -> match lgst with None -> largest_aux (Some x) xs
                 | Some l ->
                   largest_aux (if l > x then lgst else (Some x)) xs

let largest = largest_aux None

(* I would also accept *)
let largest list =
List.fold_left
 (fun lgst x -> match lgst with None -> Some x
                | Some l -> if l > x then lgst else Some x)
 None
 list
```

3. Write a function `dividek: int -> int list -> (int -> 'a) -> 'a`, that is in full Continuation Passing Style (CPS), that divides `n` successively by every number in the list, starting from the *last* element in the list. If a zero is encountered in the list, the function should pass 0 to `k` immediately, *without doing any divisions*. You should use

```
# let divk x y k = k(x/y);;
val divk : int -> int -> (int -> 'a) -> 'a = <fun>
```

for the divisions. An example use of `dividek` would be

```
# let report n = print_string "Result: "; print_int n; print_string "\n";;
val report : int -> unit = <fun>
# dividek 6 [1;3;2] report;;
Result: 1
- : unit = ()
```

Solution:

```
let eqk a b k = k(a = b)
let rec dividek n list k =
    match list
     with [] -> k n
        | 0::xs -> k 0
        | x::xs ->
          dividek n xs
          (fun r -> eqk r 0 (fun b -> if b then k 0 else divk r x k)
```

4. a. Give most general (polymorphic) types for following functions (you don't have to derive them):

```
let first lst = match lst with
   | a:: aa -> a;;

let rest lst = match lst with
   | [] -> []
   | a:: aa -> aa;;
```

Solution: `first` : $\forall$ 'a. a' list $\rightarrow$ 'a

`rest` : $\forall$ 'a. a' list $\rightarrow$ 'a list

b. Use these types (i.e., start in an environment with these identifiers bound to these types) to give a polymorphic type derivation for:

```
let rec foldright f lst z =
         if lst = [] then z
            else (f (first lst) (foldright f (rest lst) z))
in foldright (+) [2;3;4] 0
```

You should use the following types: `[]` : $\forall$'a. 'a list, and `(::)` : $\forall$'a. 'a $\rightarrow$ 'a list $\rightarrow$ 'a list. Assume that the Relation Rule is extended to allow equality at all types.

Solution: Let us use LR for the Let Rec rule, F for the Function rule, A for the Application rule, If for the If_then_else rule, C for the Constants rule, V for the variable rule, and R for the Relations rule. Let

$$\begin{aligned}
&\Gamma = \{\texttt{first} : \forall\ \texttt{'a. a' list} \to \texttt{'a}; \texttt{rest} : \forall\ \texttt{'a. a' list} \to (\texttt{'a list})\} \\
&\Gamma_1 = \{\texttt{foldright} : (\texttt{'a} \to \texttt{'b} \to \texttt{'b}) \to (\texttt{'a list}) \to \texttt{'b} \to \texttt{'b}\} \cup \Gamma \\
&\Gamma_2 = \{\texttt{foldright} : \forall\ \texttt{'a 'b.}\ (\texttt{'a} \to \texttt{'b} \to \texttt{'b}) \to (\texttt{'a list}) \to \texttt{'b} \to \texttt{'b}\} \cup \Gamma \\
&\Gamma_3 = \{\texttt{f} : \texttt{'a} \to \texttt{'b} \to \texttt{'b}\} \cup \Gamma_1 \\
&\Gamma_4 = \{\texttt{lst} : \texttt{'a list}\} \cup \Gamma_3\} \\
&\Gamma_5 = \{\texttt{z} : \texttt{'b}\} \cup \Gamma_4
\end{aligned}$$

Let *Tree1* =

$$\dfrac{\dfrac{}{\Gamma_5 \vdash \texttt{f: 'a} \to \texttt{'b} \to \texttt{'b}}\,\textsf{V} \qquad \dfrac{\dfrac{}{\Gamma_5 \vdash \texttt{first: 'a list} \to \texttt{'a}}\,\textsf{V} \quad \dfrac{}{\Gamma_5 \vdash \texttt{lst: 'a list}}\,\textsf{V}}{\Gamma_5 \vdash \texttt{first lst: 'a}}\,\textsf{A}}{\Gamma_5 \vdash \texttt{f (first lst): 'b} \to \texttt{'b}}\,\textsf{A}$$

Let *Tree2* =

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_5 \vdash \texttt{foldright: ('a} \to \texttt{'b} \to \texttt{'b)} \to \texttt{('a list)} \to \texttt{'b} \to \texttt{'b}}\,\textsf{V} \quad \dfrac{}{\Gamma_5 \vdash \texttt{f: ('a} \to \texttt{'b} \to \texttt{'b)}}\,\textsf{V}}{\Gamma_5 \vdash \texttt{foldright f: ('a list)} \to \texttt{'b} \to \texttt{'b}}\,\textsf{A} \quad \dfrac{\dfrac{}{\Gamma_5 \vdash \texttt{rest: 'a list} \to \texttt{'a list}}\,\textsf{V} \quad \dfrac{}{\Gamma_5 \vdash \texttt{lst: 'a list}}\,\textsf{V}}{\Gamma_5 \vdash \texttt{rest lst: 'a list}}\,\textsf{A}}{\Gamma_5 \vdash \texttt{foldright f (rest lst): 'b} \to \texttt{'b}}\,\textsf{A} \quad \dfrac{}{\Gamma_5 \vdash \texttt{z: 'b}}\,\textsf{V}}{\Gamma_5 \vdash \texttt{foldright f (rest lst) z: 'b}}\,\textsf{A}$$

The type variable `'a` in each isntance of the Variable rule for `first` and `rest` , we speciallize `'a` to `int`.

Let *Tree3* =

$$\dfrac{\dfrac{\dfrac{}{\texttt{(::):int} \to \texttt{int list} \to \texttt{int}}\,\textsf{C} \quad \dfrac{}{2 : \texttt{int}}\,\textsf{C}}{\texttt{(::)2:int list} \to \texttt{int}}\,\textsf{A} \quad \dfrac{\dfrac{\dfrac{}{\texttt{(::):int} \to \texttt{int list} \to \texttt{int}}\,\textsf{C} \quad \dfrac{}{3 : \texttt{int}}\,\textsf{C}}{\texttt{(::)3:int list} \to \texttt{int}}\,\textsf{A} \quad \dfrac{\dfrac{\dfrac{}{\texttt{(::):int} \to \texttt{int list} \to \texttt{int}}\,\textsf{C} \quad \dfrac{}{4 : \texttt{int}}\,\textsf{C}}{\texttt{(::)4:int list} \to \texttt{int}}\,\textsf{A} \quad \dfrac{}{\texttt{[]:int list}}\,\textsf{C}}{\texttt{(::)4 []:int list}}\,\textsf{A}}{\texttt{(::)3((::)4 []):int list}}\,\textsf{A}}{\Gamma_1 \vdash \texttt{[2;3;4] : int list}}\,\textsf{A}$$

In each instance of the Constant Rule for `(::)` and `[]`, we speciallize `'a` to `int`.

Let *Tree4* =

```
 ------------------------------- V
 Γ2 ⊢foldright:
 (int → int → int) →                          ------------------------------ C
 (int list) → int → int                      Γ2 ⊢(+): int → int → int
 -------------------------------------------------------------------------- A
       Γ2 ⊢foldright (+): int list → int → int                  Tree3
       ------------------------------------------------------------------- A   ---------- C
                 Γ2 ⊢foldright (+) [2;3;4]: int → int                          Γ2 ⊢0: int
                 -------------------------------------------------------------------------- A
                                  Γ2 ⊢foldright (+) [2;3;4] 0: int
```

In the Variable Rule we specialize both 'a and 'b to int.

Using these proofs we have:

```
----------------- V  ----------------- C
Γ5 ⊢lst: 'a list     Γ5 ⊢ [] : 'a list                            Tree1            Tree2
----------------------------------------- R  --------- V  ------------------------------------------------------- A
        Γ5 ⊢lst = []:bool                    Γ5 ⊢z: 'b    Γ5 ⊢f (first lst) (foldright f (rest lst) z): 'b
        ---------------------------------------------------------------------------------------------------------- If
                Γ5 ⊢if lst = [] then z else (f (first lst) (foldright f (rest lst) z)): 'b
       ------------------------------------------------------------------------------------------------------------ F
       Γ4 ⊢  fun z -> if lst = [] then z else (f (first lst) (foldright f (rest lst) z)):'b → 'b
       ------------------------------------------------------------------------------------------------------------ F
                Γ3 ⊢  fun lst -> fun z ->
                      if lst = [] then z else (f (first lst) (foldright f (rest lst) z)):
                      ('a list) → 'b → 'b
                --------------------------------------------------------------------------------------- F
                Γ1 ⊢  fun f -> fun lst -> fun z−>
                      if lst = [] then z else (f (first lst) (foldright f (rest lst) z)):
                      ('a → 'b → 'b) → ('a list) → 'b → 'b                                        Tree4
                ------------------------------------------------------------------------------------------------- LR
                   Γ ⊢  let rec foldright = fun f -> fun lst -> fun z ->
                                if lst = [] then z else (f (first lst) (foldright f (rest lst) z))
                        in foldright (+) [2;3;4] 0 :  int
```

This time, the 'a in the Constant Rule for [] is specialized to 'a.

5. Use the unification algorithm described in class and in ML4 to give a most general unifier for the following set of equations (unification problem). Capital letters (A, B, C, D, E) denote variables of unification. The lower-case letters (f, l, n, p) are constants or term constructors. (f and p have arity 2 - i.e., take 2 arguments, l has arity 1, and n has arity 0 - i.e. it is a constant.) Show all your work by listing the operations performed in each step of the unification and the result of that step.

$$\{(f(A, f(B, B)) = f(p(C, D), f(p(E, F), p(l(C), l(D))))); (p(l(p(D, n)), C) = p(l(A), C))\}$$

Solution:

Rule Resulting Equations / Substitution

Given

$\mathsf{Unify}\{(f(A, f(B,B)) = f(p(C,D), f(p(E,F), p(l(C), l(D))))); (p(l(p(D,n)), C) = p(l(A), C))\}$

by Decompose $(f(A, f(B,B)) = f(p(C,D), f(p(E,F), p(l(C), l(D)))))$

$= \mathsf{Unify}\{(A = p(C,D)); (f(B,B) = f(p(E,F), p(l(C), l(D)))); (p(l(p(D,n)), C) = p(l(A), C))\}$

by Eliminate $(A = p(C,D))$

$= \mathsf{Unify}\{(f(B,B) = f(p(E,F), p(l(C), l(D)))); (p(l(p(D,n)), C) = p(l(p(C,D)), C))\}\text{o } \{A \mapsto p(C,D)\}$

by Decompose $(f(B,B) = f(p(E,F), p(l(C), l(D))))$

$= \mathsf{Unify}\{(B = p(E,F)); (B = p(l(C), l(D))); (p(l(p(D,n)), C) = p(l(p(C,D)), C))\}\text{o } \{A \mapsto p(C,D)\}$

by Eliminate $(B = p(E,F))$

$= \mathsf{Unify}\{(p(E,F) = p(l(C), l(D))); (p(l(p(D,n)), C) = p(l(p(C,D)), C))\}\text{o } \{A \mapsto p(C,D); B \mapsto p(E,F)\}$

by Decompose $(p(E,F) = p(l(C), l(D)))$

$= \mathsf{Unify}\{(E = l(C)); (F = l(D)); (p(l(p(D,n)), C) = p(l(p(C,D)), C))\}\text{o } \{A \mapsto p(C,D); B \mapsto p(E,F)\}$

by Eliminate $(E = l(C))$

$= \mathsf{Unify}\{(F = l(D)); (p(l(p(D,n)), C) = p(l(p(C,D)), C))\}\text{o } \{A \mapsto p(C,D); B \mapsto p(l(C), F); E \mapsto l(C)\}$

by Eliminate $(F = l(D))$

$= \mathsf{Unify}\{(p(l(p(D,n)), C) = p(l(p(C,D)), C))\}\text{o } \{A \mapsto p(C,D); B \mapsto p(l(C), l(D)); E \mapsto l(C); F \mapsto l(D)\}$

by Decompose $(p(l(p(D,n)), C) = p(l(p(C,D)), C))$

$= \mathsf{Unify}\{(l(p(D,n)) = l(p(C,D))); (C = C)\}\text{o } \{A \mapsto p(C,D); B \mapsto p(l(C), l(D)); E \mapsto l(C); F \mapsto l(D)\}$

by Decompose $(l(p(D,n)) = l(p(C,D)))$

$= \mathsf{Unify}\{(p(D,n) = p(C,D)); (C = C)\}\text{o } \{A \mapsto p(C,D); B \mapsto p(l(C), l(D)); E \mapsto l(C); F \mapsto l(D)\}$

by Decompose $(p(D,n) = p(C,D))$

$= \mathsf{Unify}\{(D = C); (n = D); (C = C)\}\text{o } \{A \mapsto p(C,D); B \mapsto p(l(C), l(D)); E \mapsto l(C); F \mapsto l(D)\}$

by Eliminate $(D = C)$

$= \mathsf{Unify}\{(n = C); (C = C)\}\text{o } \{A \mapsto p(C,C); B \mapsto p(l(C), l(C)); E \mapsto l(C); F \mapsto l(C); D \mapsto C\}$

by Orient $(n = C)$

$= \mathsf{Unify}\{(C = n); (C = C)\}\text{o } \{A \mapsto p(C,C); B \mapsto p(l(C), l(C)); E \mapsto l(C); F \mapsto l(C); D \mapsto C\}$

by Eliminate $(C = n)$

$= \mathsf{Unify}\{(n = n)\}\text{o } \{A \mapsto p(n,n); B \mapsto p(l(n), l(n)); E \mapsto l(n); F \mapsto l(n); D \mapsto n; C \mapsto n\}$

by Delete $(n = n)$

$= \mathsf{Unify}\{\ \}\text{o } \{A \mapsto p(n,n); B \mapsto p(l(n), l(n)); E \mapsto l(n); F \mapsto l(n); D \mapsto n; C \mapsto n\}$

The final unifying substitution is $\{A \mapsto p(n,n); B \mapsto p(l(n), l(n)); E \mapsto l(n); F \mapsto l(n); D \mapsto n; C \mapsto n\}$.

6. For each of the regular expressions below (over the alphabet {a,b,c}), give a right regular gramar that derives exactly the same set of strings as the set of strings generated by the given regular expression.

   i) a*∨b*∨c*

   ii) ((aba∨bab)c(aa∨bb))*

   iii) (a*b*)*(c∨ϵ)(b*a*)*

Solution: i) a*∨b*∨c*

$$\begin{array}{lcl} S & ::= & \varepsilon \mid aA \mid bB \mid cC \\ A & ::= & \varepsilon \mid aA \\ B & ::= & \varepsilon \mid bB \\ C & ::= & \varepsilon \mid cC \end{array}$$

ii) ((aba∨bab)c(aa∨bb))*

$$\begin{array}{lcl} S & ::= & \varepsilon \mid aA \mid bB \\ A & ::= & bC \\ B & ::= & aD \\ C & ::= & aE \\ D & ::= & bE \\ E & ::= & cF \\ F & ::= & aG \mid bH \\ G & ::= & aS \\ H & ::= & bS \end{array}$$

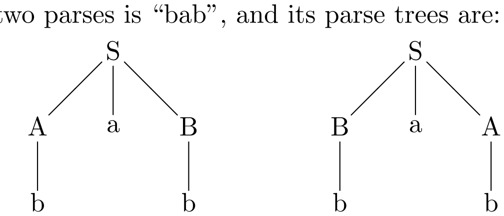
iii) (a*b*)*(c∨ϵ)(b*a*)*

$$\begin{array}{lcl} S & ::= & \varepsilon \mid aS \mid bS \mid cA \\ A & ::= & \varepsilon \mid aA \mid bA \end{array}$$

7. Consider the following ambiguous grammar (Capitals are nonterminals, lowercase are terminals):

   S ::= A a B | B a A
   A ::= b | c
   B ::= a | b

   a. Give an example of a string for which this grammar has two different parse trees, and give their parse trees.
      Solution: A string with two parses is "bab", and its parse trees are:



   b. Disambiguate this grammar.

Solution: S ::= b a a | b a b | c a a | c a b | a a b | a a c | b a c

8. **I did not mean to include this this year**
Write a unambiguous grammar for regular expressions over the alphabet $\{\mathbf{a}, \mathbf{b}\}$. The Kleene star binds most tightly, followed by concatenation, and then choice. Here we will have concatenation and choice associate to the right. Write an Ocaml datatype corresponding to the tokens for parsing regular expressions, and one for capturing the abstract syntax trees corresponding to parses given by your grammar.

Solution:

$$reg ::= a \mid b \mid \varepsilon \mid (\ reg\ ) \mid reg \vee reg \mid reg\ reg \mid reg*$$

$$\begin{array}{ll} Atom ::= & \mathbf{a}|\mathbf{b}|\varepsilon|(RegExp) \\ Star ::= & Atom|Star\ \texttt{*} \\ Concat ::= & Star|Star\ Concat \\ RegExp ::= & Concat|Concat \vee RegExp \end{array}$$

```
type tokens = A_tk | B_tk | Epsilon_tk | LParen | RParen | Star_tk | Or_tk
type atom = A | B | Epsilon | Paren of regexp
and star = Atom of atom | Star of star
and concat = StarAsConcat of star | Concat of (star * concat)
and regexp = ConcatAsRegExp of concat | Choice of (concat * regexp)
```

9. a. Write the transition semantics rules for `if _ then _ else` and `repeat _ until _`. (A repeat _ until _ executes the code in the body of the loop and then checks the condition, exiting if the condition is true.)

Solution: Let m represent the current state. If_then_else rules:

$$\frac{}{(\texttt{if true then } C_1 \texttt{else } C_2 \texttt{ fi }, m) \rightarrow (C_1, m)}$$

$$\frac{}{(\texttt{if false then } C_1 \texttt{else } C_2, m) \texttt{ fi } \rightarrow (C_2, m)}$$

$$\frac{(B, m) \rightarrow (B', m)}{(\texttt{if } B \texttt{ then } C_1 \texttt{else } C \texttt{ fi }, m) \rightarrow (\texttt{if } B' \texttt{ then } C_1 \texttt{else } C_1 \texttt{ fi }, m)}$$

$$\frac{}{(\texttt{repeat } C \texttt{ until } B, m) \rightarrow (C; \texttt{if } B \texttt{ then skip else (repeat } C \texttt{ until } B\texttt{) fi }, m)}$$

b. Assume we have an OCaml type `bexp` with constructors `True` and `False` corresponding to true and false, and other constructors representing the syntax trees of non-value boolean expressions. Futher assume we have a type `mem` of memory associating variables (represented by strings) with values, a type `exp` for integer expressions in our language, a type `comm` for language commands with constructors including `IfThenElse of bexp * comm * comm`, `RepeatUntil of comm * bexp`, and `Seq: comm * comm`, and type

```
type eval_comm_result = Mid of (comm * mem) | Done of mem
```

Further suppose we have a function `eval_bexp : (bexp * mem) -> (bexp * mem)` that returns the result of one step of evaluation of an expression.

Write Ocaml clauses for a function `eval_comm : (comm*mem) -> eval_comm_result` for the case of `IfThenElse` and `RepeatUntil`. You may assume that all other needed clauses of `eval_comm` have been defined, as well as the function `eval_bexp: (bexp*mem) -> (bexp*mem)`.

Solution:
```
let rec eval_comm (comm, mem) =
    match comm with
     . . .
    | IfThenElse (True, thenclause, elseclause) -> Mid (thenclause, mem)
    | IfThenElse (False, thenclause, elseclause) -> Mid (elseclause, mem)
    | IfThenElse (b, thenclause, elseclause) ->
      (match eval_bexp (b, mem) with (new_b, mem') ->
           Mid (IfThenElse (new_b, thenclause, elseclause), mem') )
     . . .
    | RepeatUntil(c,b) ->
       Mid (Seq (c, IfThenElse (b, Skip, RepeatUntil(c,b))), mem)
```

10. Assume you are given the OCaml types `exp`, `bool_exp` and `comm` with (partially given) type definitions:

```
type comm = ... | If of (bool_exp * comm * comm) | ...
type bool_exp = True_exp | False_exp | ...
```

where the constructor `If` is for the abstract syntax of an `if_then_else_` construct. Also assume you have a type `mem` of memory associating values to identifiers, where values are just intergers (`int`). Further assume you are given a function `eval_bool: (mem * bool_exp) -> bool` for evaluating boolean expressions. Write the OCaml code for the clause of `eval_comm:(mem * comm) -> mem` that implements the following natural semantics rules for the evaluation of `if_then_else_` commands:

$$\frac{\langle m, b\rangle \Downarrow \mathsf{true} \quad \langle m, C_1\rangle \Downarrow m'}{\langle m, \texttt{if } b \texttt{ then } C_1 \texttt{ else } C_2\rangle \Downarrow m'} \qquad \frac{\langle m, b\rangle \Downarrow \mathsf{false} \quad \langle m, C_2\rangle \Downarrow m''}{\langle m, \texttt{if } b \texttt{ then } C_1 \texttt{ else } C_2\rangle \Downarrow m''}$$

Solution:
```
let rec eval_comm (mem, comm) =
    match comm with . . .
 | If (bexp,c1,c2) ->
   (match eval_bool (mem, bexp) with true ->  eval_comm (mem, c1)
    | false -> eval_com (mem, c2))
. . .
```

11. Using the natural semanitics rules given in class, give a proof that, starting with a memory that maps `x` to 5 and `y` to 3, `if x = y then z := x else z := x + y` evaluates to a memory where `x` maps to 5, `y` maps to 3. and `z` maps to 8.

Solution: Let $m = \{\mathtt{x} \mapsto 5; \mathtt{y} \mapsto 3\}$.

$$\dfrac{\dfrac{\dfrac{}{\langle m, \mathtt{x}\rangle \Downarrow 5} \quad \dfrac{}{\langle m, \mathtt{y}\rangle \Downarrow 3} \quad (5 = 3) = \mathtt{false}}{\langle \{\mathtt{x} \mapsto 5; \mathtt{y} \mapsto 3\}, \mathtt{x\ =\ y}\rangle \Downarrow \mathtt{false}} \quad \dfrac{\dfrac{}{\langle m, \mathtt{x}\rangle \Downarrow 5} \quad \dfrac{}{\langle m, \mathtt{y}\rangle \Downarrow 3} \quad 5 + 3 = 8}{\langle \{\mathtt{x} \mapsto 5; \mathtt{y} \mapsto 3\}, \mathtt{z\ :=\ x\ +\ y}\rangle \Downarrow \langle \{\mathtt{x} \mapsto 5; \mathtt{y} \mapsto 3; \mathtt{z} \mapsto 8\}}}{\langle \{\mathtt{x} \mapsto 5; \mathtt{y} \mapsto 3\}, \mathtt{if\ x\ =\ y\ then\ z\ :=\ x\ else\ z\ :=\ x\ +\ y}\rangle \Downarrow \langle \{\mathtt{x} \mapsto 5; \mathtt{y} \mapsto 3; \mathtt{z} \mapsto 8\}}$$

12. Prove that $\lambda x.x(\lambda z.zxz)$ is α-equivalent $\lambda z.z(\lambda x.xzx)$. You should label every use of α-conversion and congruence.

Solution: By α-conversion

$$\lambda x.x(\lambda z.zxz) \xrightarrow{\alpha} \lambda y.y(\lambda z.zyz).$$

Because α-conversion implies α-equivalence, we have

$$\lambda x.x(\lambda z.zxz) \stackrel{\alpha}{\sim} \lambda y.y(\lambda z.zyz).$$

By α-conversion

$$\lambda z.zyz \xrightarrow{\alpha} \lambda x.xyx$$

and thus

$$\lambda z.zyz \stackrel{\alpha}{\sim} \lambda x.xyx$$

. By congruence for application, we have

$$y(\lambda z.zyz) \stackrel{\alpha}{\sim} y(\lambda x.xyx),$$

and by congreunce for abstraction, we have

$$\lambda y.y(\lambda z.zyz) \stackrel{\alpha}{\sim} \lambda y.y(\lambda x.xyx).$$

By transitivity, we then have

$$\lambda x.x(\lambda z.zxz) \stackrel{\alpha}{\sim} \lambda y.y(\lambda x.xyx).$$

By α-conversion,

$$\lambda y.y(\lambda x.xyx)z \xrightarrow{\alpha} \lambda z.z(\lambda x.xzx)$$

Again, because α-conversion implies α-equivalence, we have

$$\lambda y.y(\lambda x.xyx) \stackrel{\alpha}{\sim} \lambda z.z(\lambda x.xzx)$$

and by transitivity, we have

$$\lambda x.x(\lambda z.zxz) \stackrel{\alpha}{\sim} \lambda z.z(\lambda x.xzx)$$

as was to be shown.

13. Reduce the following expression: $(\lambda x\lambda y.yz)((\lambda x.xxx)(\lambda x.xx))$

   a. Assuming Call by Name (Lazy Evaluation)

Solution: With Call by Name (Lazy Evaluation):
$(\lambda x.\lambda y.yz)((\lambda x.xxx)(\lambda x.xx)) - \beta \rightarrow (\lambda y.yz)$

   b. Assuming Call by Value (Eager Evaluation)

Solution: With Call by Value (Eager Evaluation):
$(\lambda x.\lambda y.yz)((\lambda x.xxx)(\lambda x.xx)) - \beta \rightarrow$
$(\lambda x.\lambda y.yz)((\lambda x.xx)(\lambda x.xx)(\lambda x.xx)) - \beta \rightarrow$
$(\lambda x.\lambda y.yz)((\lambda x.xx)(\lambda x.xx)(\lambda x.xx)) - \beta \rightarrow$
... (the expression doesn't terminate)

   c. To full $\alpha\beta$-normal form.

Solution: Since lazy evaluation yielded an $\alpha\beta$-normal form. we may use its reduction: $(\lambda x.\lambda y.yz)((\lambda x.xxx)(\lambda x.xx)) - \beta \rightarrow (\lambda y.yz)$

14. Give a proof in Floyd-Hoare logic of the partial correctness assertion:

$$\{\texttt{True}\}\ \texttt{y := w; if x = y then z := x else z := y}\ \{\texttt{z = w}\}$$

Solution: Because this proof tree is rather wide, we shall break it up into pieces.
Let $Tree1 =$

$$\dfrac{\texttt{((y = w) \& (x = y))} \Rightarrow \texttt{(x = w)} \qquad \dfrac{}{\{\texttt{x = w}\}\ \texttt{z := x}\ \{\texttt{z = w}\}}\,\textsf{A}}{\{\texttt{(y = w) \& (x = y)}\}\ \texttt{z := x}\ \{\texttt{z = w}\}}\,\textsf{PS}$$

Let $Tree2 =$

$$\dfrac{\{\texttt{(y = w) \& (x} \neq \texttt{y)}\} \Rightarrow \texttt{(y = w)} \qquad \dfrac{}{\{\texttt{y = w}\}\ \texttt{z := y}\ \{\texttt{z = w}\}}\,\textsf{A}}{\{\texttt{(y = w) \& (x} \neq \texttt{y)}\}\ \texttt{z := y}\ \{\texttt{z = w}\}}\,\textsf{PS}$$

Then the main proof tree is

$$\dfrac{\dfrac{\texttt{True} \Rightarrow \texttt{(w = w)} \qquad \dfrac{}{\{\texttt{w = w}\}\ \texttt{y := w}\ \{\texttt{y = w}\}}\,\textsf{A}}{\{\texttt{True}\}\ \texttt{y := w}\ \{\texttt{y = w}\}}\,\textsf{PS} \qquad \dfrac{Tree1 \qquad Tree2}{\{\texttt{y = w}\}\ \texttt{if x = y then z := x else z := y}\ \{\texttt{z = w}\}}\,\textsf{ITE}}{\{\texttt{True}\}\ \texttt{y := w; if x = y then z := x else z := y}\ \{\texttt{z = w}\}}\,\textsf{Seq}$$

15. What should the Floyd-Hoare logic rule for `repeat` C `until` B be? The code causes C to be executed, and then, if B is true it completes, and otherwise it does `repeat` C `until` B again.

Solution:

$$\frac{\{Q \vee (P \wedge (\neg B))\}\ C\ \{P\}}{\{Q\}\ \texttt{repeat}\ C\ \texttt{until}\ B\ \{P \wedge B\}}$$

But I would accept the weaker

$$\frac{\{P\}\ C\ \{P\}}{\{P\}\texttt{repeat}\ C\ \texttt{until}\ B\ \{P \wedge B\}}$$