

Solutions for Sample Questions for Midterm (CS 421 Fall 2006)

On the actual midterm, you will have plenty of space to put your answers.
Some of these questions may be reused for the exam.

1. Given the following OCAML code:

```
let x = 3;;
let f y = x + y;;
let x = 5;;
let z = f 2;;
let x = "hi";;
```

What value will **z** have? Will the last declaration (**let x = "hi";**) cause a type error?
What is the value of **x** after this code has been executed?

Solution:

z is bound to 5
let x = "hi" will not cause a type error
x is bound to "hi"

2. What environment is in effect after each declaration in the code in Problem 1?

Solution:

```
let x = 3;;
{x → 3}
let f y = x + y;;
{f → <y → x+y, {x → 3}>, x → 3}
let x = 5;;
{x → 5} + {f → <y → x+y, {x → 3}>, x → 3} =
{x → 5, f → <y → x+y, {x → 3}>}
let z = f 2;;
{z → 5, x → 5, f → <y → x+y, {x → 3}>}
let x = "hi";;
{x → "hi"} + {z → 5, x → 5, f → <y → x+y, {x → 3}>} =
{x → "hi", z → 5, f → <y → x+y, {x → 3}>}
```

3. Given the following OCAML datatype:

```
type int_seq = Null | Snoc of (int_seq * int)
```

write a tail-recursive function in OCAML **all_pos : int_seq -> bool** that returns **true** if every integer in the input **int_seq** to which **all_pos** is applied is strictly greater than 0 and **false** otherwise. Thus **all_pos (Snoc(Snoc(Snoc(Null,3),5),7))** should return **true**, but **all_pos (Snoc(Null,~1))** and **all_pos (Snoc(Snoc(Null, 3),0))** should both return **false**.

Solution:

```
let rec all_pos s =
  (match s with Null -> true
   | Snoc(seq, x) -> if x <= 0 then false else all_pos seq);;
```

4. Write an OCAML function **pair_up** takes first a function, then an input list and returns a list of pairs of an element from input list (the second argument), paired with the result of applying the first argument to that element. What is the OCAML type of **pair_up**? What is the result of the following expressions:
- pair_up (fun x -> x + 3) [6;4;1];;**
 - pair_up ((fun x -> "Hi, ^x), ["John"; "Mary"; "Dana"]);;**
 - pair_up (fun x -> x *. 2.0);;**

Solution:

```
let rec pair_up f l =
  (match l with [] -> []
   | x :: xs -> (x, f x)::pair_up f xs)
alternately let pair_up f = map (fun x -> (x, f x))
```

pair_up : ('a -> 'b) -> 'a list -> ('a * 'b) list

- [(6, 9); (4,7); (1,4)];;
 - type error
 - A function of type float list -> (float * float) list which returns a list of pairs of an element from the input list paired with twice itself.
5. Write an Ocaml function **palindrome :string list -> unit** that first prints the strings in the list from left to right, followed by printing them right to left, recursing over the list only once. (Potential extra credit problem: Do this using each of **List.fold_right** and **List.fold_left** but no explicit use of **let rec**.)

Solution:

```
let rec palindrome l =
  match l with [] -> ()
  | s::ss -> (print_string s; palindrome ss; print_string s);;
```

```
let rec palindrome l =
  List.fold_right
  (fun s -> fun print_middle -> (fun () -> (print_string s; print_middle (); print_string s)))
  l
  (fun () -> ())
  ();
```

```
let palindrome l =
  List.fold_left
  (fun print_middle -> fun s -> (print_string s; fun () -> (print_middle (print_string s))))
  (fun () -> ())
  l
  ();
```

6. Using the rules provided in class, derive a valid type judgment for **let rec fact = fun n -> if n = 0 then 1 else let r = fact (n - 1) in n * r in fact;** (The rules will be provided for you on the exam, if this kind of question is asked.)

Solution: (I didn't just write the tree because it wouldn't fit.)

By the let_rec rule we have

$$\frac{\begin{array}{l} (1) \{ \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (\text{fun } n \rightarrow \text{if } n = 0 \text{ then } 1 \text{ else let } r = \text{fact } (n - 1) \text{ in } n * r) : \text{int} \rightarrow \text{int} \\ (2) \{ \text{fact} : \text{int} \rightarrow \text{int} \} \vdash \text{fact} : \text{int} \rightarrow \text{int} \end{array}}{\{ \} \vdash \text{let rec fact} = \text{fun } n \rightarrow \text{if } n = 0 \text{ then } 1 \text{ else let } r = \text{fact } (n - 1) \text{ in } n * r \text{ in fact} : \text{int} \rightarrow \text{int}}$$

(2) is valid by the variable rule:

$$\frac{}{(2) \{ \text{fact} : \text{int} \rightarrow \text{int} \} \vdash \text{fact} : \text{int} \rightarrow \text{int}}$$

By the fun rule we have

$$\frac{(3) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (\text{if } n = 0 \text{ then } 1 \text{ else let } r = \text{fact } (n - 1) \text{ in } n * r) : \text{int}}{(1) \{ \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (\text{fun } n \rightarrow \text{if } n = 0 \text{ then } 1 \text{ else let } r = \text{fact } (n - 1) \text{ in } n * r) : \text{int} \rightarrow \text{int}}$$

By the if_then_else rule we have

$$\frac{\begin{array}{l} (4) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (n = 0) : \text{bool} \quad (5) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash 1 : \text{int} \\ (6) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (\text{let } r = \text{fact } (n - 1) \text{ in } n * r) : \text{int} \end{array}}{(3) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (\text{if } n = 0 \text{ then } 1 \text{ else let } r = \text{fact } (n - 1) \text{ in } n * r) : \text{int}}$$

(5) is valid by the rule for constants. By the rule for binary relations we have

$$\frac{\begin{array}{l} (7) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash n : \text{int} \quad (8) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash 0 : \text{int} \end{array}}{(4) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (n = 0) : \text{bool}}$$

(7) is valid by the rule for variables. (8) is valid by the rule for constants.

By the rule for let, we have

$$\frac{\begin{array}{l} (9) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash \text{fact } (n - 1) : \text{int} \quad (10) \{ r : \text{int}, n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (n * r) : \text{int} \end{array}}{(6) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (\text{let } r = \text{fact } (n - 1) \text{ in } n * r) : \text{int}}$$

By the rule for applications we have

$$\frac{\begin{array}{l} (11) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash \text{fact} : \text{int} \rightarrow \text{int} \quad (12) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (n - 1) : \text{int} \end{array}}{(9) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash \text{fact } (n - 1) : \text{int}}$$

(11) is valid by the variable rule. By the rule for binary operations, we have

$$\frac{\begin{array}{l} (13) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash n : \text{int} \quad (14) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash 1 : \text{int} \end{array}}{(12) \{ n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (n - 1) : \text{int}}$$

(13) is valid by the variable rule. (14) is valid by the constant rule. Thus (12) is valid. Thus (9) is valid. We have (10) left. By the rule for binary operations we have:

$$\frac{\begin{array}{l} (15) \{ r : \text{int}, n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash n : \text{int} \quad (16) \{ r : \text{int}, n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash r : \text{int} \end{array}}{(10) \{ r : \text{int}, n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \} \vdash (n * r) : \text{int}}$$

(15) and (16) are both valid by the variable rule. Hence (10) is valid. Hence (6) is valid. Hence (3) is valid, and hence (1) is valid

7. Write the definition of an OCAML variant type **reg_exp** to express abstract syntax trees for regular expressions over a base character set of booleans. Thus, a boolean is a **reg_exp**, epsilon is a **reg_exp**, the concatenation of two **reg_exp**'s is a **reg_exp**, the "choice" of two **reg_exp**'s is a **reg_exp**, and the Kleene star of a **reg_exp** is a **reg_exp**.

Solution:

```
type reg_exp =
  Epsilon
| Var of bool
| Choice of (reg_exp * reg_exp)
| Concat of (reg_exp * reg_exp)
| Kleene_star of reg_exp
| Paren of reg_exp (* I would accept it with his case missing *)
```

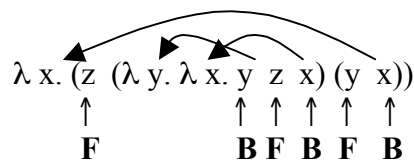
8. Give a (most general) unifier for the following unification instance. Capital letters denote variables of unification. Show your work by listing the operation performed in each step of the unification and the result of that step.

$$\{X = f(g(x), W), h(y) = Y, f(Z, x) = f(Y, W)\}$$

Solution: 11:39 – 11:44

$\{X = f(g(x), W), h(y) = Y, f(Z, x) = f(Y, W)\}$
 $\rightarrow \{h(y) = Y, f(Z, x) = f(Y, W)\}$ with $\{X = f(g(x), W)\}$ by eliminate
 $\rightarrow \{Y = h(y), f(Z, x) = f(Y, W)\}$ with $\{X = f(g(x), W)\}$ by orient
 $\rightarrow \{f(Z, x) = f(h(y), W)\}$ with $\{X = f(g(x), W), Y = h(y)\}$ by eliminate
 $\rightarrow \{Z = h(y), x = W\}$ with $\{X = f(g(x), W), Y = h(y)\}$ by decompose
 $\rightarrow \{x = W\}$ with $\{X = f(g(x), W), Y = h(y), Z = h(y)\}$ by eliminate
 $\rightarrow \{W = x\}$ with $\{X = f(g(x), W), Y = h(y), Z = h(y)\}$ by orient
 \rightarrow answer: $\{X = f(g(x), x), Y = h(y), Z = h(y), W = x\}$ by eliminate

9. In the λ -expression below, write under each arrow whether the indicated variable occurrence is free or bound (write F or B). Also, for each bound occurrence, draw an arrow back to the abstraction that binds it.



Solution:

10. Evaluate the following λ -expression to $\alpha\beta$ -normal form, if one exists or explain why one does not exist. Show all work.

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

Solution:

$$\begin{aligned} & (\lambda x. \lambda y. \lambda z. y z x) (\lambda x. x x) (\lambda x. \lambda y. y x) (\lambda x. x) \\ \rightarrow & (\lambda y. \lambda z. y z (\lambda x. x x)) (\lambda x. \lambda y. y x) (\lambda x. x) \\ \rightarrow & (\lambda z. (\lambda x. \lambda y. y x) z (\lambda x. x x)) (\lambda x. x) \\ \rightarrow & (\lambda z. (\lambda y. y z) (\lambda x. x x)) (\lambda x. x) \text{ (* Mistake here before *)} \\ \rightarrow & (\lambda z. (\lambda x. x x) z) (\lambda x. x) \\ \rightarrow & (\lambda z. z z) (\lambda x. x) \\ \rightarrow & (\lambda x. x) \end{aligned}$$

11. Evaluate the following λ -expression as far as possible using each of lazy evaluation and eager evaluation. If either evaluation fails to terminate, write “Diverges” and a brief explanation why the evaluation fails to terminate.

$$(\lambda y. \lambda x. y x x) (\lambda x. \lambda y. x x) ((\lambda x. x (\lambda y. y)) (\lambda x. x))$$

Solution:

Lazy Evaluation: $(\lambda y. \lambda x. y x x) (\lambda x. \lambda y. x x) ((\lambda x. x (\lambda y. y)) (\lambda x. x))$

$$\begin{aligned} \rightarrow & (\lambda x. (\lambda x. \lambda y. x x) x x) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) \\ \rightarrow & (\lambda x. \lambda y. x x) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) \\ \rightarrow & (\lambda y. ((\lambda x. x (\lambda y. y)) (\lambda x. x)) ((\lambda x. x (\lambda y. y)) (\lambda x. x))) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) \text{ (*left out*)} \\ \rightarrow & ((\lambda x. x (\lambda y. y)) (\lambda x. x)) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) \\ \rightarrow & ((\lambda x. x) (\lambda y. y)) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) \\ \rightarrow & (\lambda y. y) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) \\ \rightarrow & (\lambda x. x (\lambda y. y)) (\lambda x. x) \\ \rightarrow & (\lambda x. x) (\lambda y. y) \\ \rightarrow & (\lambda y. y) \end{aligned}$$

Eager Evaluation:

$$\begin{aligned} & (\lambda y. \lambda x. y x x) (\lambda x. \lambda y. x x) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) \\ \rightarrow & (\lambda x. (\lambda x. \lambda y. x x) x x) ((\lambda x. x (\lambda y. y)) (\lambda x. x)) \\ \rightarrow & (\lambda x. (\lambda x. \lambda y. x x) x x) ((\lambda x. x) (\lambda y. y)) \\ \rightarrow & (\lambda x. (\lambda x. \lambda y. x x) x x) (\lambda y. y) \\ \rightarrow & (\lambda x. \lambda y. x x) (\lambda y. y) (\lambda y. y) \\ \rightarrow & (\lambda y. ((\lambda y. y) (\lambda y. y))) (\lambda y. y) \\ \rightarrow & (\lambda y. y) (\lambda y. y) \\ \rightarrow & (\lambda y. y) \end{aligned}$$

12. Represent the constructors for the following OCAML type in the lambda calculus

type answer = yes | no | maybe

Write the lambda term that returns the representation of **yes** when applied to the representation of **no**, the representation of **no** when applied to the representation of **yes**, and the representation of **maybe** when applied to the representation of **maybe**.

Solution:

yes $\rightarrow \lambda x. \lambda y. \lambda z. x$ no $\rightarrow \lambda x. \lambda y. \lambda z. y$ maybe $\rightarrow \lambda x. \lambda y. \lambda z. z$

The flipping function:

$\lambda a. a (\lambda x. \lambda y. \lambda z. y) (\lambda x. \lambda y. \lambda z. x) (\lambda x. \lambda y. \lambda z. z)$