CS 3110 Fall 2016
Sample problems for Prelim 1 based on Spring 2015

Solutions

Please note that the actual Spring 2015 Prelim 1 was longer than this selection of problems. We have attempted to remove any problems that were about topics not covered this semester, but it's possible we missed some.

1. True–False [20 pts]

   Label the following statements with either "true" (T) or "false" (F). Correct answers receive two points, blank or omitted answers receive 1 point, and incorrect answers receive zero points.

   (a) `List.fold_left f acc l` and `List.fold_right f l acc` always return the same value, but the former is tail recursive and the latter is not.

   False

   (b) `map` can be implemented with `fold_left` but not with `fold_right`.

   False

   (c) "The pair `(n,d)` represents the rational number n/d" is a representation invariant for rational numbers.

   False

   (d) Anonymous functions are expressions in OCaml.

   True

   (e) The following expression is well-typed: `3110 + failwith "ZARDOZ"`

   True

2. Types and values [15 pts] For each of the following expressions, write the type of the expression and the value to which it evaluates. Or, if the expression would not compile, indicate this by writing "would not compile" and give a brief explanation of why. If it evaluates to a function, use the substitution model to provide your answer.

   (a) [3 pts] `let (x,y) = (3,5) in (float_of_int x) *. (float_of_int y)`

   Answer:

   Type: `float`*; Value:* `15.`

   (b) [3 pts] `List.fold_left (fun a x -> a::x)`
   (see Appendix for type of `List.fold_left`)

   Answer:

   Would not compile: not well-typed.

(c) [3 pts]

```
let u f (x,y) = f x y in
let f x y = 3 in
u f
```

Answer:

Type: `'a * 'b -> int`; *Value:* `fun (x,y) -> 3`

(d) [3 pts]

```
(fun x -> match x with
    |    []    -> 0
    | []::[]   -> 1
    | [[[]]]   -> 2
    |    _     -> 3
) [[]]
```

u: ('a -> 'b -> 'c) -> 'a * 'b -> 'c
f: 'a -> 'b -> int
u f is of type 'a * 'b -> int

'c gets filled in

let f x - let f = fun x

Answer:

Type: `int`; *Value:* `1`

(e) [3 pts] Assume the following definitions:

```
module type T = sig val x : int end
module X : T  = struct
  let x = 7
  let y = 3
end
```

Now answer the question above for this expression:

```
X.y
```

Answer:

Would not compile: unbound value.

3. Functions on lists [20 pts]

(a) [14 pts] Consider writing a function `count : int -> int list -> int`, such that `count n lst` is the number of elements of `lst` that are strictly greater than `n`. For example, `count 5 [4;5;6]` should return `1`.

Implement this function in four ways:

i. `count_rec`: recursively, without using any `List` module functions;

ii. `count_fold`: using `List.fold_left` or `List.fold_right`, but no other `List` module functions nor the `rec` keyword;

iii. `count_lib`: using any combination of `List` module functions, but excluding any `fold` functions and the `rec` keyword; and

iv. `count_tr`: tail recursively, using any functions or keywords you wish, including any of the previous three implementations.

The Appendix provides the names and types of many `List` module functions.

Answer:

```
let (>?) m n = if m>n then 1 else 0

let rec count_rec (n:int) (ts: int list) : int =
  match ts with
  | [] -> 0
  | m::ts' -> (m>?n) + (count_rec n ts')

let count_fold (n:int) (ts: int list) : int =
  List.fold_left (fun acc m -> acc + (m>?n)) 0 ts
  (* Also acceptable:
   * List.fold_right (fun m acc -> acc + (m>?n)) ts 0
   *)

let count_lib (n:int) (ts: int list) : int =
  List.length(List.filter (fun m -> m > n) ts)

let count_tr = count_fold
(* if count_fold is implemented with fold_left, but not fold_right *)

(* Also acceptable:
let count_tr (n:int) (ts: int list) : int =
  let rec helper (ts': int list) (acc:int) : int =
    match ts' with
    | [] -> acc
    | m::ts'' -> helper ts'' (acc + (m>?n))
  in
  helper ts 0
*)
```

(b) [6 pts] Write a function `cart : 'a list -> 'b list -> ('a * 'b) list` that computes the Cartesian product of two lists. That is, `cart l1 l2` should return a list `l3`, which contains the pair `(x,y)` iff `x` is an element of `l1` and `y` is an element of `l2`. For example, `cart [3110;7] [2;13]` could return `[(3110,2); (3110,13); (7,2); (7;13)]`. The order of pairs in `l3` is unspecified. You may assume that `l1` does not contain any duplicate elements, and likewise for `l2`. For full credit, your solution should be tail recursive and run in $O(|\texttt{l1}| \cdot |\texttt{l2}|)$ time, where $|\ell|$ denotes the length of list ℓ.

let oneE e lst = List.fold_left(fun acc el -> ([e]@[el])::acc) [] lst;;
val oneE : 'a -> 'a list -> 'a list list = <fun> —(08:40:04)—< command 96
>——{ counter: 0 }—utop # let pairs lst1 lst2 = List.fold_
(fun acc el -> acc @ (oneE el lst1)) [] lst2;;
val pairs : 'a list -> 'a list -> 'a list list = <fun>

Answer:

```
(* omitted in these solutions *)
```

4. Datatypes and Folding [30 pts]

HTML (HyperText Markup Language) is used to create web pages. Here is a grammar for a simple subset of HTML text, in which `string` represents an OCaml string:

```
url ::= string://string/string  (* scheme, host, path *)

text ::= string                  (* plain text *)
       | <a href="url">text</a> (* anchor text *)
       | <br/>                   (* break *)
       | text1 text2 ... textn  (* nested content *)
```

A URL, for example, could be `http://3110.com/zardoz.html`. The scheme is `http`, the host is `3110.com`, and the path is `zardoz.html`. Here is an example of some HTML text according to this grammar:

```
<a href="http://3110.com/zardoz.html>3110 is fun</a>
```

(a) [5 pts] Define two OCaml types to represent `url` and `text`.

Answer:

Need {} around url type because record

```
type url = scheme : string; host : string; path : string

type text =
  | PlainText of string
  | Anchor of url * text
  | Break
  | Content of text list
```

(b) [2 pts] Write an expression, using the types you defined, that represents the following HTML:

```
<br/>
<a href="http://3110.com/zardoz.html">3110 is fun</a>
<br/>
```

Answer:

```
let ex = Content [Break;
                  Anchor(scheme="http";
                         host="3110.com";
                         path="zardoz.html",
                        PlainText "3110 is fun");
                  Break]
```

(c) [4 pts] Write a function `count_breaks : text -> int` that counts the number of breaks in HTML text.

Answer:

```
let rec count_breaks = function
  | PlainText _ -> 0
  | Anchor (u,t) -> count_breaks t
  | Break -> 1
  | Content ts -> begin
      match ts with
      | [] -> 0
      | t::ts' -> (count_breaks t) + (count_breaks (Content ts'))
    end
```

ts' is a list, but the function requires Text abstraction

5. Modular programming [25 pts] Define a type `'a t` to be *list-like* if the following conditions hold:

- There is a value `empty` of type `'a t`.
- There is a `cons` operation that takes an element of type `'a` and a list-like value of type `'a t` and returns a list-like value of type `'a t`.
- There is a `decons` operation that takes a list-like value of type `'a t` and
  - returns `None` if the list-like value is `empty`, or
  - returns `Some (x,xs)` if the list-like value is the `cons` of an element `x` and a list-like value `xs`.

You may use any function from the OCaml `List` module. The Appendix contains a list of function names and their types.

(a) [6 pts] Write a module type `ListLike` that encodes the above specification. You do not need to write any specification comments.

Answer:

```
module type ListLike = sig
  type 'a t
  val empty  : 'a t
  val cons   : 'a -> 'a t -> 'a t
  val decons : 'a t -> ('a * 'a t) option
end
```

(b) [6 pts] Write a module `ListImpl` that uses `list` to implement the `ListLike` interface.

Answer:

```
module ListImpl : ListLike = struct
  type 'a t = 'a list
  let cons a lst = a::lst
  let decons lst = match lst with
    | []    -> None
    | h::tl -> Some (h,tl)
  let empty = []
end
```

(c) [5 pts] Write a functor that takes a `ListLike` module and produces a module containing a function `map`. That function should behave like `List.map` but generalized from `list` to `ListLike` values.

Answer:

```
module Extend (L : ListLike) = struct
  let map f lst = match L.decons lst with
    | None       -> L.empty
    | Some (h,t) -> L.cons (f h) (map f t)
end
```

(d) [8 pts] An `'a list` of length `k` can be encoded as a function `f` of type `int -> 'a option`, where `f n` returns one of the following values:

- `Some x` if `x` is the value at position `n` of the list. The first element of the list is at position 0.
- `None` if `n` is not a valid position in the list.

Use this encoding to write a module `FunImpl` that implements the `ListLike` interface with the type `'a t = int -> 'a option`.

Answer:

```
(* omitted in these solutions *)
```

```
module FunImpl:ListLike = struct
    type 'a t = int -> 'a option
    let empty = fun _ -> None
    let cons ele lst = (fun ind ->
        if ind = 0 then Some ind else
        lst (ind-1))
    let decons lst =
        match lst 0 with
        | None -> None
        | Some v -> Some (v, fun ind -> lst(ind+1))
```