Midterm Exam

CS131: Programming Languages

Wednesday, May 3, 2017

1.	2.
3.	4.
5.	

Name:_____

ID:_____

Rules of the game:

- **Write your name and ID number above.**
- The exam is closed-book and closed-notes.
- Please write your answers directly on the exam. Do not turn in anything else.
- The exam ends promptly at 3:50pm.
- Read questions carefully. Understand a question before you start writing.
- Relax!

1. (5 points each)

   (a) Implement a function `find` of type `('a -> bool) -> 'a list -> 'a option`, where `find p l` returns the first element in the list `l` that satisfies the predicate `p`. The result type is an `option` in order to have something to return in the case when no element in the list satisfies the predicate. Recall the `option` type's definition:
   `type 'a option = None | Some of 'a`
   For example, `find (function x -> x > 2) [1;2;3;4]` returns `Some 3`.
   Implement this function recursively; do not write any helper functions or use any functions from the `List` module.

```
let rec find p l =
  match l with
    [] -> None
  | h::t -> if p h then Some h else (find p t)
```

   (b) Now implement `find` again, but this time **without using recursion**. **Instead, the entire function body should be a single call to `List.fold_right`.** Recall the type of `List.fold_right`: `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.

```
let find p l =
  List.fold_right (fun x curr -> if p x then Some x else curr) l None
```

2. (3 points each) Consider the `find` function from the previous problem, of type `('a -> bool) -> 'a list -> 'a option`.

   (a) **Choose the single best answer.** If OCaml did not support parametric polymorphism:

      i. `find` could not be passed another function as an argument.
      ii. `find` could not be implemented using currying.
      iii. `find` could not be passed lists of different lengths on different calls.
      iv. none of the above

      ANSWER: iv

   (b) **Choose the single best answer.** Consider this OCaml expression:
   `find (function x -> x > 2) ["hi"; "there"]`

      i. The expression incurs an error at compile time.
      ii. The expression incurs an error at run time.
      iii. The expression is determined to have type `string option` at compile time.
      iv. The expression is determined to have type `string option` at run time.
      v. None of the above.

      ANSWER: i

   (c) **Choose the single best answer.** OCaml does not support static overloading of functions. As a consequence:

      i. `find` cannot be passed a list that contains both `int`s and `float`s
      ii. `find` cannot be passed lists of different types on different calls
      iii. it is an error to declare `find` if there already exists another function of that name
      iv. none of the above

      ANSWER: iv

3. We've seen two ways to define a two-argument function in OCaml: the arguments can either be supplied as a tuple, or they can be supplied separately through currying. For example, a function having two integer inputs that returns an integer could be implemented to have either the type `int * int -> int` or the type `int -> int -> int`. In different circumstances, one or the other form of function may be more convenient. It turns out that a function defined in either form can be converted to the other.

   (a) (4 points) Define a function `curry` of type `(('a * 'b) -> 'c) -> ('a -> 'b -> 'c)` that converts an uncurried function of two arguments into a curried function of two arguments. For example, `curry (function (x,y) -> x+y)` returns a function `f` of type `int -> int -> int` such that `f e1 e2` returns the sum of `e1` and `e2`, for any integer arguments `e1` and `e2`.

   ```
   let curry f = (fun x y -> f(x,y))
   ```

   (b) (4 points) Define a function `uncurry` of type `('a -> 'b -> 'c) -> (('a * 'b) -> 'c)` that converts a curried function of two arguments into an uncurried function of two arguments.

   ```
   let uncurry f = (fun (x,y) -> f x y)
   ```

   (c) (2 points) In OCaml `(fun x y -> y+x+y) 2 3` returns the value `8`. Suppose OCaml used dynamic scoping instead of static scoping. Then if `(fun x y -> y+x+y) 2 3` were allowed to execute in a fresh invocation of the OCaml interpreter, it would:

   i. still return the value `8`
   ii. incur an error when trying to look up the value of `x`
   iii. incur an error when trying to look up the value of `y`
   iv. incur an error when trying to perform an addition
   v. none of the above

   ii

4. (2 points each) For each property of OCaml below, say whether it is a consequence of OCaml being statically typed (write "static"), strongly typed (write "strong"), both (write "both"), or neither (write "neither").

   (a) OCaml treats functions as first-class values.

   ANSWER: neither

   (b) OCaml does not support static overloading.

   ANSWER: neither

   (c) OCaml never allows a primitive operation to be invoked with arguments of the wrong types.

   ANSWER: strong

   (d) OCaml never allows a user-defined function to be invoked with arguments of the wrong types.

   ANSWER: both

   (e) OCaml rejects some programs at compile time that would not incur any errors if allowed to execute.

   ANSWER: static

5. (4 points) Consider a variant of `find` from Problem #1 called `rfind` such that `rfind p l` returns the *last* element of the list `l` that satisfies the predicate `p`. Can `rfind` be implemented as a single call to `List.fold_right`, with no other function calls (e.g., reversing the list is not allowed)? If so, implement it; if not, write NO.

```
let rfind p l =
  List.fold_right
    (fun x curr ->
      match curr with
        None -> if p x then Some x else None
      |  _ -> curr)
  l None
```