Midterm Exam

CS131: Programming Languages

Wednesday, April 27, 2016

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.
- Obey our usual OCaml style rules.
- Except where explicitly disallowed, you can write any number of helper functions that you need.
- The exam ends promptly at 3:50pm.
- Read questions carefully. Understand a question before you start writing.
- Relax!

1. (5 points each) Recall that in Homework 1 you used a list containing no duplicates to represent a set.

   (a) Implement a function `diff` of type `'a list -> 'a list -> 'a list`, which produces the *set difference* of two given sets. In particular `(diff s1 s2)` returns a new set containing all elements of `s1` that are not in `s2`. For example, `diff [3;1;4] [2;4]` returns `[1;3]` (the order of elements in the returned set is irrelevant). **Implement this function recursively; do not write any helper functions or use any functions from the `List` module. You may assume the existence of a function `member` of type `'a -> 'a list -> bool` that determines whether an item is a member of a given set.**

```
let rec diff s1 s2 =
  match s1 with
    [] -> []
    | h::t -> (if member h s2 then [] else [h]) @ (diff t s2)
```

   (b) Now implement a function `cprod` of type `'a list -> 'b list -> ('a * 'b) list`, which produces the *Cartesian product* of two given sets. In particular `cprod s1 s2` returns a new set containing all pairs `(x,y)` where `x` is in `s1` and `y` is in `s2`. For example, `cprod [1;3;4] [2;4]` returns `[(1,2);(3,2);(4,2);(1,4);(3,4);(4,4)]` (the order of elements in the returned set is irrelevant). **Implement this function without using recursion. Instead, make good use of higher-order functions from the `List` module.**

```
let cartesianProduct s1 s2 =
  List.fold_right (fun x rest -> (List.map (fun y -> (x,y)) s2)@rest) s1 []
```

2. (3 points each) Consider the `cprod` function from the previous problem, of type `'a list -> 'b list -> ('a * 'b) list`.

   (a) **Choose the single best answer.** Consider a version of OCaml that lacks parametric polymorphism but is otherwise unchanged. Then there would be no type that could be given to `cprod` that would allow:

   i. `cprod` to be called with two argument lists of different lengths.
   ii. `cprod` to be called with two argument lists of different types.
   iii. `cprod` to return a list of pairs of integers.
   iv. none of the above

   ANSWER: iv

   (b) **Choose the single best answer.** Consider this OCaml expression:
   `cprod [1;3] ["hello"]`

   i. The expression is determined to have type `('a * 'b) list` at compile time.
   ii. The expression is determined to have type `(int * string) list` at compile time.
   iii. The expression is determined to have type `(int * string) list` at run time.
   iv. The expression causes a static type error.
   v. None of the above.

   ANSWER: ii

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

   i. two modules cannot define functions of the same name
   ii. a function cannot be passed different types of arguments on different invocations
   iii. it is an error to declare a function of the same name as an existing function
   iv. some function calls must be typechecked at run time
   v. none of the above

   ANSWER: v

3. (5 points each) Instead of representing a set with a list, we can represent a set of elements of type `T` by its *characteristic function*, which is just a function of type `T -> bool`, where the elements of the set are exactly those that cause the function to return `true`. For example, the set of positive integers can be represented by the characteristic function `(function x -> x > 0)`.

   (a) Implement a function `add`, of type `'a -> ('a -> bool) -> ('a -> bool)`, which takes an element `x` as well as a set `s` represented as characteristic function and returns a new characteristic function for the set `s` $\cup$ {x}. Just as with the implementation for lists from Homework 1, the resulting set should not have duplicates; in other words, if `x` is already in the set `s` then the result should be `s` itself.

   ```
   let add x s =
     if s x
     then s
     else (fun y -> y=x || (s y))
   ```

   (b) Implement a function `cprod`, of type
   `('a -> bool) -> ('b -> bool) -> (('a * 'b) -> bool)`
   which takes two sets represented as characteristic functions and returns a new characteristic function for the set representing their Cartesian product.

   ```
   let cprod s1 s2 =
      fun (x,y) -> (s1 x) && (s2 y)
   ```

4. We've seen two ways to signal errors in OCaml: by raising an exception and by using the `option` type. Recall the definition of the latter:
`type 'a option = None | Some of 'a`

   (a) (2 points) Name one advantage of signaling an error in OCaml by raising an exception over doing so by returning `None`.

   Exceptions allow you to separate the code for handling an error from the code for normal execution.

   (b) (2 points) Name one advantage of signaling an error in OCaml by returning `None` over doing so by raising an exception.

   Using the `option` type requires the caller to pattern match on the result, thereby encouraging the caller to check for the error case (and the typechecker warns you if you forget to check for that case).

   (c) (3 points) In different circumstances, each of the above forms of signaling errors might be the right one to use. Fortunately, it is easy to convert a function that uses one form of error signaling into a function that uses the other form. Implement a function `opt2exn` of type `('a -> 'b option) -> ('a -> 'b)`, which takes a function that uses the `option` type to signal errors and returns an equivalent function that instead raises an exception named `Error` (which you can assume to have already been declared) to signal errors.

```
let opt2exn f =
fun x ->
   match f x with
   | None -> raise Error
   | Some v -> v
```

   (d) (3 points) Now implement a function `exn2opt` of type `('a -> 'b) -> ('a -> 'b option)`, which takes a function that raises an exception named `Error` (which you can assume to have already been declared) to signal errors and returns an equivalent function that instead uses the `option` type to signal errors.

```
let exn2opt f =
    fun x ->
      try
        Some (f x)
      with Error -> None
```

5. (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 does not allow the values of variables to be updated after they have been initialized.

       ANSWER: neither

   (b) OCaml determines a type for each program expression at compile time.

       ANSWER: static

   (c) OCaml prevents out-of-bounds access to lists and other data structures.

       ANSWER: strong

   (d) OCaml automatically infers the types of expressions without needing type annotations.

       ANSWER: neither

   (e) OCaml guarantees that a function’s body will never invoke an operation with arguments of the wrong types, for all possible invocations of the function.

       ANSWER: both