Midterm Exam

CS131: Programming Languages

Monday, May 6, 2013

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.
- If you have any questions, please ask.
- The exam ends promptly at 3:50pm.

A bit of advice:

- Read questions carefully. Understand a question before you start writing. *Note: Some multiple-choice questions ask for a single answer, while others ask for all appropriate answers.*
- The questions are not necessarily in order of difficulty, so feel free to skip around.
- Relax!

1. (a) (5 points) Implement an OCaml function `count`, of type `('a -> bool) -> 'a list -> int`. The invocation `count p l` returns the number of elements in the list `l` that satisfy the predicate `p`. For example, `count (function x -> x > 3) [1;4;5;0]` returns `2`. Don't define any helper functions or invoke any functions from the OCaml `List` module.

```
let rec count p l =
  match l with
      [] -> 0
    | x::xs -> (if (p x) then 1 else 0) + (count p xs)
```

   (b) (2 points) **Choose the single best answer.** OCaml's parametric polymorphism makes it possible for:

   i. `count` to be passed lists of many different lengths as arguments
   ii. `count` to be passed predicates of many different types as arguments
   iii. `count` to return many different integer values, depending on the arguments
   iv. none of the above

   ii

   (c) (2 points) **Choose the single best answer.** Consider this OCaml expression: `count (function x -> x > 3)`

   i. The expression passes the static typechecker and has type `int list -> int`.
   ii. The expression passes the static typechecker and and has type `'a list -> int`.
   iii. The expression passes the static typechecker and has type `int list`.
   iv. The expression causes a static type error.

   i

2. (a) (5 points) Implement `count` from Problem 1 again, but this time the entire function body should consist of a single call to `List.fold_right`. Recall the type of `List.fold_right`: `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`.

```
let count p l =
  List.fold_right (fun x c -> if (p x) then c else c+1) l 0
```

(b) (5 points) Implement `count` one more time, but this time it should be implemented as a *tail-recursive* function. Do not use any functions from the `List` module.

```
let count p l =
  let rec helper l c =
    match l with
        [] -> c
      | x::xs -> helper xs (if (p x) then c else c+1)
  in helper l 0
```

(c) (2 points) **Circle the single best answer.** Why is making the `count` function tail recursive a desirable thing to do versus an implementation that is not tail recursive?

i. It allows the function to be statically typechecked.
ii. It allows the function to be statically scoped.
iii. It ensures the function will take constant time.
iv. It allows the function to use less stack space.

iv

3. (2 points each) Answer "true" or "false" to each statement below.

   (a) One reason that OCaml is considered *strongly typed* is that it does not require variables to have explicitly declared types.
   false

   (b) One reason that OCaml is considered *statically typed* is that it does not allow variables to be re-assigned after initialization.
   false

   (c) One reason that C is considered *weakly typed* is that it allows a pointer to be dereferenced after it is freed.
   true

   (d) A language that performs garbage collection at run time is considered *dynamically typed*.
   false

   (e) Static typechecking ensures that a program will never raise an exception at run time.
   false

   (f) If a program is rejected by a static typechecker, then the program definitely has an error.
   false

4. (2 points each) Answer "true" or "false" to each statement below.

   (a) *Static scoping* ensures that each variable usage can be bound to its associated declaration at compile time.

   true

   (b) *Static scoping* ensures that each variable's value never changes after initialization.

   false

   (c) Under *static scoping* each variable can be garbage collected as soon as it goes out of scope.

   false

   (d) Under *dynamic scoping* it is possible for two calls to the same function, with the same argument values, to return a different result.

   true

5. (5 points each) The built-in OCaml lists can only have a finite size, but sometimes infinite lists are useful. One way to represent an infinite list of elements of type `'a` is as a function:

   ```
   type 'a infList = (int -> 'a)
   ```

   Here `'a infList` is not a new type, but rather just a shorthand for the type `(int -> 'a)`. Given a list `l` of this type, the intent is that `(l i)` returns the value of the `i`th element of the list, starting from 1. For example, the infinite list `[1;2;3;...]` of positive integers can be represented by the function `function x -> x`.

   (a) Write a function `addToFront`, of type `'a -> 'a infList -> 'a infList`, which acts like the `::` operator but for infinite lists. That is, it takes an element and a list and returns a new list that is identical to the old one but with a new element on the front.

   ```
   let addToFront x l =
     function i ->
       match i with
           1 -> x
         | _ -> l(i-1)
   ```

   (b) Write a function `map`, of type `('a -> 'b) -> 'a infList -> 'b infList`, which acts like the `List.map` function but for infinite lists. That is, `map f l` produces a new list whose elements are the result of applying the function `f` to each element of the list `l`.

   ```
   let map f l = function i -> f(l i)
   ```

6. (4 points) OCaml's lists are *homogeneous*: all elements must have the same type. One consequence of this restriction is that lists cannot be nested within one another unless every element has exactly the same nesting depth. For example, `[[1;2]; [3]; [4]]` is a valid list in OCaml (of type `int list list`), but `[1; [[2;3];4]]` is not a valid list.

   Define a new OCaml type `deepList` that represents arbitrarily nested lists of integers. Then show how the list `[1; [[2;3];4]]` would be defined as a value of type `deepList`.

   One solution:

```
type deepList =
      Nil
    | Cons of int * deepList
    | DCons of deepList * deepList

Cons(1, DCons(DCons(Cons(2, Cons(3, Nil)), Cons(4, Nil)), Nil))
```

   Another solution:

```
type deepElem =
      Elem of int
    | Elems of deepElem list

type deepList = deepElem list

[Elem 1; Elems [Elems[Elem 2; Elem 3]; Elem 4]]
```

7. (3 points) Recall the module type for integer sets from class:

```
module type SET = sig
  type t
  val emptyset : t
  val contains: int -> t -> bool
  val addElem: int -> t -> t
end
```

Circle all answers that apply. Making the type `t` *abstract*, as is done above, provides which of the following benefits?

(a) It allows a module of type `SET` to choose its own definition of `t`.

(b) It allows code that uses a module of type `SET` to choose its own definition of `t`.

(c) It ensures that code that uses a module of type `SET` can never create a set containing duplicates, assuming the module is implemented properly.

(d) It allows code that uses a module of type `SET` to easily add new functions to the module.

a and c