Midterm Exam

CS131: Programming Languages

Tuesday, May 8, 2012

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 5:50pm.

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 skip around.
- Relax!

1. (5 points) Implement an OCaml function `hasOddLength`, of type `'a list -> bool`. The function should return `true` if the given list has an odd number of elements and return `false` otherwise. Don't define any helper functions or invoke any functions from the OCaml `List` module.

```
let rec hasOddLength l =
  match l with
    [] -> false
  | _::rest -> not (hasOddLength rest)
```

Alternatively:

```
let rec hasOddLength l =
  match l with
    [] -> false
  | [_] -> true
  | _::_::rest -> hasOddLength rest
```

2. (2 points each) Circle the **single best** answer.

   (a) Parametric polymorphism in OCaml allows programmers to:

      i. define multiple functions of the same name
      ii. define one function with multiple names
      iii. define one function that can be passed lists of different lengths on different invocations
      iv. define one function that can be passed different types of arguments on different invocations

   iv

   (b) OCaml does not support function overloading. 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. some function calls must be typechecked at run time
      iv. i and ii above
      v. none of the above

   v

   (c) Consider the OCaml *identity* function `id` of type `'a -> 'a`, defined as `let id x = x;;`. For the function call `(id [1;2;3])`

      i. `'a` is determined to be `int` at compile time
      ii. `'a` is determined to be `int list` at compile time
      iii. `'a` is determined to be `int` at run time
      iv. `'a` is determined to be `int list` at run time
      v. `'a` can be anything so it is never determined

   ii

   (d) Consider the OCaml expression `"hi"::(id [1;2;3])`

      i. The expression fails to typecheck at compile time.
      ii. The expression typechecks at compile time but raises an exception at run time.
      iii. The expression typechecks at compile time and executes successfully.

   i

3. (5 points each)

   (a) Implement `hasOddLength` from Problem 1 again, but this time using a single call to `List.fold_right` instead of using explicit recursion.

   ```
   let hasOddLength l = List.fold_right (fun _ res -> not res) l false
   ```

   (b) A set of items of some type `T` can be represented by its *characteristic function*, which is just a function of type `T -> bool`. For example, the set of positive integers can be represented by the characteristic function `(function x -> x > 0)`. Write a function `union`, of type `('a -> bool) -> ('a -> bool) -> ('a -> bool)`, which takes two sets represented as characteristic functions and returns a new characteristic function for the set representing their union.

   ```
   let union s1 s2 =
     function x -> (s1 x) || (s2 x)
   ```

4. (a) (2 points each) For each property below, say whether it is a property of static typechecking only (write "static"), a property of dynamic typechecking only (write "dynamic"), a property of both (write "both"), or a property of neither (write "neither"):

   i. it detects bugs without running the program **: static**

   ii. it only signals an error when the program really has a bug **: dynamic**

   iii. it determines a type for each expression in the source program **: static**

   iv. it ensures the program will never raise an exception at run time **: neither**

   (b) (2 points) Circle the **single best** answer. C is considered *weakly* typed because:

   i. it does not support parametric polymorphism

   ii. it does not prevent array bounds violations

   iii. it performs some typechecking at run time

   iv. it requires each variable to have an explicitly declared type

   ii

   (c) (2 points) Circle the **single best** answer. OCaml is considered *statically* typed because:

   i. each program expression is given a type at compile time

   ii. it prevents array bounds violations

   iii. the value of a variable never changes after initialization

   iv. it does not require variables to have explicitly declared types

   i

5. (2 points each)

   Assume the following OCaml declarations have been entered in this order into the OCaml interpreter:

```
let n = 3;;
let f x = x - n;;
let n = [3];;
```

   Give the value of each expression below, or say "static error" if it would cause a static error or "dynamic error" if it would cause a run-time error.

   (a) `f 7`
   4

   (b) `f n`
   static error

   (c) `f x`
   static error

6. (2 points) **Circle all answers that apply.** Which of these are properties of static scoping?

   (a) Each variable usage can be bound to its associated declaration at compile time.

   (b) Each variable's value never changes after initialization.

   (c) Each variable can be garbage collected as soon as a new variable of the same name shadows it in the environment.

   (d) A new variable declaration cannot change the behavior of functions defined before that declaration.

   a and d

7. (5 points each) (Continues on the next page) Here's a simple signature for modules that implement a button which can toggle between off and on:

```
module type BUTTON = sig
  type t
  val init : t
  val toggle : t -> t
  val isOn : t -> bool
end
```

The value `init` is a button initialized to the "off" position. The function `toggle` toggles the button. The function `isOn` returns a boolean indicating whether or not the button is currently on.

(a) Complete the following implementation of the `BUTTON` signature, in which the type `t` is implemented with a user-defined type:

```
module Button : BUTTON = struct
  type t = Off | On
  (* provide implementations of init, toggle, and isOn *)



  let init = Off

  let toggle b =
    match b with
      Off -> On
    | On -> Off

  let isOn b = (b=On)







end
```

(b) Complete the following implementation of the BUTTON signature, in which the type t is now just a synonym for bool:

```
module Button : BUTTON = struct
  type t = bool
  (* provide implementations of init, toggle, and isOn *)


  let init = false

  let toggle b = (not b)

  let isOn b = b















end
```

8. (2 points) **Circle all answers that apply.** Hiding the definition of the type `t` in the `BUTTON` signature above provides which of the following benefits for modules that implement that signature?

   (a) It allows the definition of `t` to be later changed without breaking clients of the module

   (b) It ensures that a button's state can only be changed by calling `toggle`.

   (c) It allows clients to manipulate the button more efficiently

   (d) It allows the `isOn` function to be later removed without breaking clients of the module

   a and b