Midterm Exam

CS131: Programming Languages

Monday, February 10, 2014

1.	5.
2.	6.
3.	7.
4.	

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 1:50pm.
- 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.*
- Relax!

1. (5 points each) Recall two of the different implementations from Homework 2 for a dictionary mapping keys of type `'a` to values of type `'b`:

   - an *association list* of type `('a * 'b) list`
   - a datatype defined as follows:
     `type ('a,'b) dict2 = Empty | Entry of 'a * 'b * ('a,'b) dict2`

   (a) Implement an OCaml function `d1tod2`, of type `('a * 'b) list -> ('a,'b) dict2`, which converts a given association list into an equivalent value as a `dict2`. For example, `d1tod2 [("hi",1);("bye",2)]` returns `Entry("hi", 1, Entry("bye", 2, Empty))`. **Don't define any helper functions or invoke any functions from the OCaml `List` module.**

```
let rec d1tod2 l =
  match l with
      [] -> Empty
    | (x,y)::xs -> Entry(x, y, d1tod2 xs)
```

   (b) Implement `d1tod2` again, but this time the entire body of the function should be a single call to `List.fold_right`. Recall the type of `List.fold_right`:
   `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

```
let d1tod2 l =
  List.fold_right (fun (k,v) rest -> Entry(k,v,rest)) l Empty
```

2. (5 points) Recall the third dictionary implementation from Homework 2: a function of type `('a -> 'b)`, which maps keys of type `'a` to values of type `'b` and raises the `Not_found` exception when given a key that is not in the dictionary. Implement a function `union3` of type `('a -> 'b) -> ('a -> 'b) -> ('a -> 'b)`, which takes two dictionaries in this representation and produces a new dictionary representing their union. The second argument to `union3` should *shadow* the first one: if the same key k maps to v_1 in the first dictionary and v_2 in the second dictionary, then k should map to v_2 in their union (similar to what `combine_envs` did in Homework 3).

```
let union3 d1 d2 =
  (function k ->
    try
      d2 k
    with
      Not_found -> d1 k)
```

3. (2 points each) Consider the `get1` function for looking up a key in an association list from Homework 2, of type `'a -> ('a * 'b) list -> 'b`:

```
let rec get1 k d =
  match d with
      [] -> raise Not_found
    | (k',v')::d' -> if k=k' then v' else get1 k d'
```

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

   i. whenever `get1` is given a key of some type `T` as an argument, both the keys and values in the given association list would also need to have type `T`

   ii. `get1` would be just as useful as it is currently

   iii. `get1` would have to take its two arguments as a pair rather than through currying

   iv. none of the above

   iv

(b) **Choose the single best answer.** What happens when this expression is entered into the OCaml interpreter (assuming `get1` has already been defined as above)?
`get1 "hi" [("bye", 34); ("hello", 2000)]`

   i. Static typechecking succeeds and gives the expression the type `int`.

   ii. Static typechecking succeeds but the expression's type is not determined until execution, when `'a` and `'b` are respectively instantiated with `string` and `int`.

   iii. Static typechecking fails with a type error.

   iv. Static typechecking fails with a `Not_found` exception.

   i

(c) **Choose the single best answer.** What happens when this expression is entered into the OCaml interpreter (assuming `get1` has already been defined as above)?
`get1 56 [("bye", 34); ("hello", 2000)]`

   i. Static typechecking succeeds but the expression's execution terminates with a `Not_found` exception.

   ii. Static typechecking succeeds but the expression's execution terminates with a dynamic type error.

   iii. Static typechecking fails with a type error.

   iv. Static typechecking fails with a `Not_found` exception.

   iii

(d) **Choose the single best answer.** OCaml doesn't support static overloading, so:

   i. the `get1` function can only be declared if the name `get1` is not already in scope

   ii. the `get1` function can only be declared if it has the same type as all previously declared functions named `get1`

   iii. the `get1` function can only be invoked with association lists that map strings to integers

   iv. none of the above

   iv

4. (2 points each) Let DOCaml be a dynamically typed version of OCaml, just like MOCaml from Homework 3 (but not limited to the MOCaml subset of OCaml).

   (a) Provide an expression that incurs a static type error in OCaml and signals a dynamic type error during execution in DOCaml. Say "none" if no such expression exists.

   `1 + false`

   (b) Provide an expression that incurs a static type error in OCaml but executes without signaling any error in DOCaml. Say "none" if no such expression exists.

   `if true then 0 else false`

   (c) Provide an expression that typechecks successfully in OCaml but signals a dynamic type error during execution in DOCaml. Say "none" if no such expression exists.

   none

   (d) Consider the following declaration:

   ```
   let f = (function x -> x + true)
   ```

   Circle all answers that apply.

   i. Static typechecking of the declaration incurs a type error in OCaml.
   ii. Static typechecking of the declaration succeeds in OCaml, but OCaml will signal a type error during static typechecking of any call to the function `f`.
   iii. DOCaml signals a dynamic type error when executing the declaration.
   iv. DOCaml evaluates the declaration without error, but it will signal a dynamic type error during the execution of any call to the function `f`.

   i and iv

5. (3 points each)

(a) Provide an OCaml expression that evaluates to `0` but would evaluate to `1` if OCaml used dynamic scoping. Say "none" if no such expression exists. You may define as many helper declarations before the expression as you need.

```
let x = 0
let f() = x
let x = 1
f()
```

(b) Provide an OCaml expression that evaluates to `0` but would have a run-time variable lookup failure if OCaml used dynamic scoping. Say "none" if no such expression exists. You may define as many helper declarations before the expression as you need.

```
let f x y = x + y
let g = f 0
g 0
```

(c) Provide an OCaml expression that incurs a static type error due to an unbound variable but would execute without error and evaluate to 0 if OCaml used dynamic scoping. Say "none" if no such expression exists. You may define as many helper declarations before the expression as you need.

```
let f() = x
let x = 0
f()
```

6. Recall the standard recursive implementation of concatenation for OCaml lists:

```
let rec concat l1 l2 =
  match l1 with
    [] -> l2
  | x::xs -> x::(concat xs l2)
```

(a) (5 points) Provide another implementation of `concat` that is also explicitly recursive, but this time it should be *tail recursive*. You may define a helper function to do most of the work, as usual for tail recursive functions.

A few acceptable solutions:

```
let concat l1 l2 =
  let rec helper l acc =
    match l with
      [] -> acc
    | x::xs -> helper xs (x::acc)
  in helper (List.rev l1) l2

let concat l1 l2 =
  let rec helper l acc =
    match l with
      [] -> acc
    | x::xs -> helper xs (acc@[x])
  in helper l2 l1

let concat l1 l2 =
  let rec helper l k =
    match l with
      [] -> k l2
    | x::xs -> helper xs (function res -> k(x::res))
  in helper l1 (function x -> x)
```

(b) (2 points) **Circle the best answer.** Every tail recursive function in OCaml is guaranteed to:

i. take constant time
ii. use linear space for its recursive calls
iii. use constant stack space for its recursive calls
iv. none of the above

iii

7. (3 points) Recall the `Env` module from Homework 3, which declares the type `'a env` for environments as follows:

```
type 'a env = (string * 'a) list
```

The `Env` module is explicitly annotated to have the module type `ENV`, defined as follows:

```
module type ENV = sig
  type 'a env
  exception NotBound
  val empty_env: unit -> 'a env
  val add_binding:  string -> 'a -> 'a env -> 'a env
  val combine_envs: 'a env -> 'a env -> 'a env
  val lookup: string -> 'a env -> 'a
end
```

Circle all answers that apply. Suppose the `Env` module were not declared to have type `ENV` but rather had no explicitly declared type. This change would enable which of the following abilities?

(a) A client of the `Env` module can now pass the empty list `[]` as the argument to `lookup` without incurring a static type error.

(b) A client of the `Env` module can now instantiate `'a` with whatever type they want.

(c) A client of the `Env` module can now pass the value `Empty` from the `dict2` type (see Problem 1 for its definition) as the argument to `lookup` without incurring a static type error.

(d) A client of the `Env` module can now change the definition of the type `'a env` without having to modify the `Env` module.

a