Midterm Exam

CS131: Programming Languages

Thursday, October 30, 2014

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 1:50pm.
- Read questions carefully. Understand a question before you start writing.
- Relax!

1. (5 points each) Recall one of the 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`. In this problem you will implement two versions of an OCaml function `joindicts`, of type
`('a * 'b) list -> ('b * 'c) list -> ('a * 'c) list`
which *joins* two dictionaries to produce a new dictionary. A call `joindicts d1 d2` should return a dictionary containing all pairs (k, w) such that (k, v) is in `d1` and (v, w) is in `d2`. For example:

```
# joindicts [(1,"hi"); (2,"bye")] [("bye", false); ("hello", true)];;
- : (int * bool) list = [(2, false)]
```

You may assume that each argument dictionary contains at most one binding for each key. You may also assume the existence of the function `get1` that you wrote in Homework 2, whose type is `'a -> ('a * 'b) list -> 'b option`, which looks up a key in an association list and returns either `None` (when the key is not in the dictionary) or `Some v` (when the key is mapped to `v` in the dictionary).

(a) Implement `joindicts` using explicit recursion. **Don't use any functions from the `List` module.**

```
let rec joindicts al1 al2 =
  match al1 with
    [] -> []
  | (k,v)::rest ->
      match (get1 v al2) with
        None -> joindicts rest al2
      | Some v' -> (k,v')::(join rest al2)
```

(b) Recall the `fold_right` function from the `List` module, of type
`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
Implement `joindicts` again, but this time the entire body of the function should be a single call to `fold_right`. **Don't use any other functions from the `List` module.**

```
let joindicts al1 al2 =
  List.fold_right
    (fun (k,v) joined ->
      match (get1 v al2) with
        None -> joined
      | Some v' -> (k,v')::joined)
    al1 []
```

2. (3 points each)

   (a) Name one advantage of static typechecking over dynamic typechecking, and provide OCaml code that illustrates the point.

   It detects type errors early, at compile time instead of run time. For example, a function containing something bad like `1+true` will not typecheck, while in a dynamically typed language the error will only be caught when the function is executed.

   (b) Name one advantage of dynamic typechecking over static typechecking, and provide OCaml code that illustrates the point.

   Dynamic typechecking does not have to be conservative while static typechecking does. For example, OCaml does not allow heterogenous lists like `[1; true]` since it can't give such a list a type, while a dynamically typed language allows this.

3. In class we saw how to define a datatype for binary trees in OCaml. This problem will explore n-ary trees, where each internal node can have any number of children rather than exactly two.

   (a) (3 points) Define a type `tree` for n-ary trees. You should assume that leaf nodes do not hold any data and that internal nodes each contain an integer as the data.

   ```
   type tree = Leaf | Node of int * tree list
   ```

   (b) (5 points) Define a function `incTree` of type `int -> tree -> tree` such that a call `incTree n t` returns a tree identical to `t` but with each internal node's data value incremented by `n`. You may use functions from the `List` module if that is useful.

   ```
   let rec incTree n t =
     match t with
       Leaf -> Leaf
     | Node(d,children) -> Node(d+n, List.map (incTree n) children)
   ```

4. (3 points each)

   (a) Which of these things are implied by the fact that OCaml is *strongly typed*?
   Circle all answers that apply.

      i. Each of a program's expressions is given a type at compile time.
      ii. A program cannot access memory that it did not allocate.
      iii. A program can make use of parametric polymorphism.
      iv. A program cannot terminate with an exception at run time.

   ii

   (b) Which of these things are implied by the fact that OCaml supports *parametric polymorphism*?
   Circle all answers that apply.

      i. A single function can be passed arguments of different types.
      ii. A function call cannot be completely typechecked until run time.
      iii. Two functions can have the same name.
      iv. Functions can have other functions as arguments.

   i

   (c) Which of these things are implied by the fact that OCaml supports *static scoping*?
   Circle all answers that apply.

      i. A variable's value can never change after initialization.
      ii. Each variable usage can be bound to its associated declaration at compile time.
      iii. Currying properly preserves the behavior of passing multiple arguments to a function.
      iv. All memory can be allocated at compile time.

ii and iii

5. (5 points each) Here is one way to represent *lazy* lists of integers in OCaml, which produce their elements on demand rather than all at once:

```
type lazylist = Nil | Cons of int * (unit -> lazylist)
```

This definition is *almost* the standard definition of lists as a datatype that we saw in class, except that each `Cons` node contains a function that produces the next node in the list, rather than directly containing the next node. We can correspondingly define functions to access the head and tail of a (non-empty) lazy list:

```
let head (Cons(x,f)) = x
```

```
let tail (Cons(x,f)) = f()
```

Here when we ask for the tail we execute the function stored in the datatype, producing a new `lazylist`.

Here's an example of the lazy list representing `[1;2]`:

```
# let l = Cons(1, (function () -> Cons(2, (function () -> Nil))));;
val l : lazylist = Cons (1, <fun>)
# head l;;
- : int = 1
# head (tail l);;
- : int = 2
# tail (tail l));;
- : lazylist = Nil
```

(a) One benefit of lazy lists is that they can be used to define lists of infinite length. Define a function `intsFrom` of type `int -> lazylist` such that `(intsFrom n)` produces the infinite list of successive integers starting from `n`. For example, `(intsFrom 1)` returns the infinite list of positive integers.

```
let rec intsFrom n = Cons(n, (function () -> intsFrom(n+1)))
```

(b) Write a function `lazymap`, of type `(int -> int) -> lazylist -> lazylist`, which acts like the `List.map` function but for lazy lists. That is, `lazymap f l` produces a new lazy list whose elements are the result of applying the function `f` to each element of the lazy list `l`. For example, `lazymap (function x -> x*2) (intsFrom 1)` produces the infinite list of positive even numbers.

```
let rec lazymap f l =
  match l with
    Nil -> Nil
  | Cons(x,tFun) -> Cons(f x, (function () -> lazymap f (tFun ())))
```