CS 51: Introduction to Computer Programming II
Midterm Examination
Spring, 2016

This is an open-book exam: You are free to use books and notes in preparing your solutions for this exam. However, no electronic devices of any kind may be used.

Numbers in parentheses are the points (out of 85 total) allocated to the problem and may provide an approximate recommendation for allocating time.

Write the answers to all problems in the boxes provided. Anything written outside of the provided boxes will not be graded. If additional room is needed for an answer, make a note in the box provided and write the remainder of your answer in one of the boxes at the back of the examination.

Please write your name and ID number in the boxes below, but nowhere else on the exam to maintain anonymity of grading.

Your name:
Your Harvard ID number:

1. Type inference

Problem 1. ***[12 points]*** *For each of the following ML types construct an expression (with no explicit typing annotations) for which ML would infer that type. Keep in mind that the expressions need not be practical or do anything useful; they need only have the requested type. Provide your answers in the boxes provided below. (The first problem is done for you as an example.)*

(1) `bool`

```
true
```

(2) `bool * bool -> bool`

```
let f (x, y) =
  if x && y then x else y in
f ;;
```

(3) `'a list -> bool list`

```
let rec f xs =
  match xs with
  | [] -> []
  | h :: t -> true :: (f t) in
f ;;
```

(4) `('a * 'b -> 'a) -> 'a -> 'b -> 'a`

```
let f g a b =
  if g (a, b) = a then a else a in
f ;;
```

(5) `int * 'a * 'b -> 'a list -> 'b list`

```
let f (i, a, b) alst =
  if i = 0 && (List.hd alst) = a then [b] else [] in f ;;
```

Problem 2. ***[12 points]*** *For each of the following definitions of a function* `f`*, give its most general type (as would be inferred by OCaml) or explain briefly why no type exists for the function. (The first problem is done for you as an example.)*

(1)
```
let f x =
  x +. 42. ;;
```

```
f :  float -> float
```

(2)
```
let f g x =
  g (x + 1) ;;
```

```
f : (int -> 'a) -> int -> 'a
```

(3)
```
let f x =
  match x with
  | [] -> x
  | h :: t -> h
```

No typing is possible. Since `h` is the head of `x`, if the type of `h` is `'a` then the type of `x` is `'a list`. But by the match, both `x` and `h` must be of the same type. Thus `'a list = 'a`, which is not possible.

(4)
```
let rec f x a =
  match x with
  | [] -> a
  | h :: t -> h (f t a) ;;
```

```
f : ('a -> 'a) list -> 'a -> 'a
```

2. Functional programming

Problem 3. *[7 points] We've seen that* `List.map` *can be implemented using* only *a call to* `List.fold_right`*, as follows:*

```
let map f lst =
  List.fold_right (fun elt acc -> (f elt) :: acc) lst [] ;;
```

Provide an OCaml definition of the `fold_right` *function using only a call to* `List.map`*, or give a concise argument for why it's not possible to do so.*

The type of `fold_right` makes clear that the output may be of any type. But `map` always returns a list. So a single call to `map` cannot generate the range of answers that `fold_right` can.

The composition of two single-argument functions `f` and `g` is the function that first applies `f` to its argument and then `g` to the result. For example, consider the following two functions `double` and `square`:

```
# let double x = 2 * x ;;
val double : int -> int = <fun>
# let square x = x * x ;;
val square : int -> int = <fun>
```

The composition of `double` and `square` is a function that first doubles and then squares its argument. That function would map 1 to the square of the double of 1, that is, 4, and would map 2 to the square of the double of 2, that is, 16.

Problem 4. *[5 points] Provide an OCaml definition for a higher-order function* `compose` *that takes two argument functions and returns their composition. The function should have the following behavior:*

```
# let f = compose double square ;;
val f : int -> int = <fun>
# f 1 ;;
- : int = 4
# f 2 ;;
- : int = 16
# let strdouble = compose int_of_string double ;;
val strdouble : string -> int = <fun>
# strdouble "42" ;;
- : int = 84
```

```
let compose f g x = g (f x) ;;
```

Problem 5. *[4 points] What is the type of* `compose`*?*

```
compose :  ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

Problem 6. *[5 points] How would you define a function* `second` *using just a single call to* `compose` *that extracts the second element of a list. Functions from the* `List` *module may be useful. (You needn't worry about what the function does on lists shorter than length 2.)*

```
let second = compose List.tl List.hd ;;
```

3. Data types

The game of mini-poker is played with just six playing cards: You use only the face cards (king, queen, jack) of the two suits spades and diamonds. There is a ranking on the cards: Any spade is better than any diamond, and within a suit, the cards from best to worst are king, queen, jack.

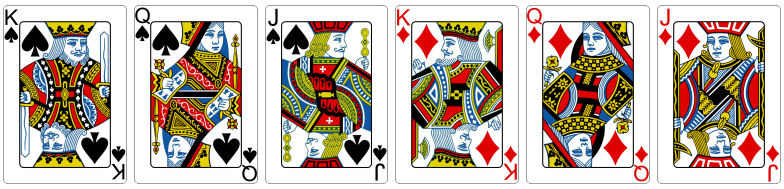

Figure 1. The cards of mini-poker, depicted in order from best to worst.

In this two-player game, each player picks a single card at random, and the player with the better card wins.

For the record, it's a terrible game.

Problem 7. ***[8 points]*** *Provide appropriate type definitions to represent the cards used in the game. It should contain structured information about the suit and value of the cards.*

```
type suit = Spades | Diamonds ;;
type value = King | Queen | Jack ;;
type card = {suit: suit; value: value} ;;
```

Note that the field names and type names can be identical, since they are in different namespaces. An alternate solution would be to use a pair instead of a record.

I have carefully ordered the constructors from better to worse and ordered the record components from higher to lower order so that comparisons on the data values will accord with the "better" relation, as seen below.

Problem 8. ***[5 points]*** *What is an appropriate type for a function* `better` *that determines which of two cards is "better" in the context of mini-poker, returning* `true` *if and only if the first card is better than the second?*

```
better : card -> card -> bool
```

Problem 9. ***[8 points]*** *Provide a definition of the function* `better`*.*

This works if you carefully order the constructors and fields from best to worst and higher order (suit) before lower-order (value).

```
let better card1 card2 =
  card1 < card2 ;;
```

To not rely on the ad hoc polymorphism of <, we need something like this:

```
let better {suit = suit1; value = value1}
           {suit = suit2; value = value2} =
  let val v = match v with
              | King -> 3
              | Queen -> 2
              | Jack -> 1 in
  if suit1 = suit2 then (val value1) > (val value2)
  else suit1 = Spades
```

though this is hacky since it doesn't generalize well to adding more suits. Of course, a separate map of suits to an int suit value would solve that problem, but at a certain point, you're better off taking advantage of the implicit mapping to comparable values for variant types and records already built in to <.

4. Modules and abstract data types

We define here a signature for modules that deal with images and their manipulation.

```
module type IMAGING =
  sig
    (* types for images, which are composed of pixels *)
    type image
    type pixel
    (* an image size is a pair of ints giving number of rows and
       columns *)
    type size = int * int
    (* converting between integers and pixels *)
    val to_pixel : int -> pixel
    val from_pixel : pixel -> int
    (* apply an image filter, a function over pixels, to every pixel
       in an image *)
    val filter : (pixel -> pixel) -> image -> image
    (* apply an image filter to two images, combining the images pixel
       by pixel *)
    val filter2 : (pixel -> pixel -> pixel) -> image -> image -> image
    (* return a "constant" image of the specified size where every
       pixel has the same value *)
    val const : pixel -> size -> image
    (* display the image in a graphics window *)
    val depict : image -> unit
  end ;;
```

The pixels that make up an image are specified by the following signature:

```
module type PIXEL =
  sig
    type t
    val to_pixel : int -> t
    val from_pixel : t -> int
  end
```

Problem 10. ***[6 points]*** *Fill in the box to complete the skeleton of a functor named* `MakeImaging` *for generating implementations of the* `IMAGING` *signature based on modules satisfying the* `PIXEL` *signature. (We've purposefully left the implementation out.)*

```
module  MakeImaging (P : PIXEL) : IMAGING with pixel = P.t  =
  struct
    (* ...  the implementation would go here ...  *)
  end ;;
```

Here is a module implementing the `PIXEL` signature for integer pixels.

```
module IntPixel : (PIXEL with type t = int) =
  struct
    type t = int
    let to_pixel x = x
    let from_pixel x = x
  end ;;
```

Problem 11. ***[6 points]*** *Write code that uses the* `IntPixel` *module to define an imaging module called* `IntImaging`*.*

```
module IntImaging = MakeImaging(IntPixel) ;;
```

Optionally, signature specifications can be added, so long as appropriate sharing constraints are provided.

Problem 12. ***[7 points]*** *Give code to use the* `IntImaging` *module that you just defined to display a 100 by 100 pixel image where all of the pixels have the constant integer value 5000.*

```
let open IntImaging in
  depict (const (to_pixel 5000) (100, 100)) ;;
```

Total points: 85