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:

Reference material

For your convenience, the following information about various useful functions from the List module is copied verbatim from the OCaml documentation:

> `val hd : 'a list -> 'a`
> Return the first element of the given list. Raise `Failure "hd"` if the list is empty.
>
> `val tl : 'a list -> 'a list`
> Return the given list without its first element. Raise `Failure "tl"` if the list is empty.
>
> `val map : ('a -> 'b) -> 'a list -> 'b list`
> `List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.
>
> `val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
> `List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.
>
> `val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
> `List.fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`. Not tail-recursive.
>
> `val filter : ('a -> bool) -> 'a list -> 'a list`
> `filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved.

You may use these where appropriate throughout the exam.

CS51 CHEAT SHEET

```
# (* ..................................... BASICS *)
# () ;;
 - : unit = ()

# 3 ;;
 - : int = 3

# 3.0 ;;
 - : float = 3.

# 'A' ;;
 - : char = A

# "xyz" ;;
 - : string = "xyz"

# false ;;
 - : bool = false

# 3 < 5 && true ;;
 - : bool = true

# Some 3 ;;
 - : int option = Some 3

# None ;;
 - : 'a option = None

# [3; 4] ;;
 - : int list = [3; 4]

# [] ;;
 - : 'a list = []

# (2, "xyz", 3.0) ;;
 - : int * string * float = (2, "xyz", 3.)

# fst (2,"abc") ;;
 - : int = 2

# snd (2, "abc") ;;
 - : string = "abc"

# "x" ^ "y" ^ "z" ;;
 - : string = "xyz"

# let x = 3 in
# if x<0 || x>0 then "nonzero" else "zero" ;;
 - : string = "nonzero"

# let e = exp 1. in
# let pi = 2.*.asin 1. in
# (e, pi) ;;
 - : float * float = (2.71828182846, 3.14159265359)

# (* ................... FUNCTIONS AND APPLICATION *)
# fun x -> x + 1 ;;
 - : int -> int = <fun>

# (fun x -> x + 1) 3 ;;
 - : int = 4

# fun x y -> x + y ;;
 - : int -> int -> int = <fun>

# fun (x, y) -> x + y ;;
 - : int * int -> int = <fun>

# fun () -> 4 ;;
 - : unit -> int = <fun>

# let uncurriedplus (x, y) = x + y in
# let curriedplus x y = x + y in
#   (uncurriedplus (1, 2), curriedplus 1 2) ;;
 - : int * int = (3, 3)

# (* ..................................... MATCHING *)
# let x = 3 in
#   match x with
#   | 0 -> "zero"
#   | 1 -> "one"
#   | _ -> "more than one" ;;
 - : string = "more than one"

# let (x, y) = (Some 111, 2999) in
#   match (x, y) with
#   | Some z, _ -> z + y
#   | None, _ -> y ;;
 - : int = 3110

# let rec sum (x : int list) : int =
#   match x with
#   | [] -> 0
#   | u :: t -> u + sum t ;;
 val sum : int list -> int = <fun>

# (* ..................................... RECORDS *)
# type rcrd = {foo : int; bar : string} ;;
 type rcrd = { foo : int; bar : string; }

# {foo = 3; bar = "xyz"} ;;
 - : rcrd = {foo = 3; bar = "xyz"}

# (* ..................................... LISTS *)
# 3 :: [4; 5] ;;
 - : int list = [3; 4; 5]

# [1;2;3] @ [4;5;6] ;;
 - : int list = [1; 2; 3; 4; 5; 6]

# List.length [8; 9; 10] ;;
 - : int = 3

# List.hd [3; 4] ;;
 - : int = 3

# List.tl [3 ;4] ;;
 - : int list = [4]

# List.tl [4] ;;
 - : int list = []

# List.map ;;
 - : ('a -> 'b) -> 'a list -> 'b list = <fun>

# List.map (fun x -> x + 100) [2;3;4] ;;
 - : int list = [102; 103; 104]

# List.map (fun x -> x = 3) [2;3;4] ;;
 - : bool list = [false; true; false]

# List.filter ;;
 - : ('a -> bool) -> 'a list -> 'a list = <fun>

# List.filter (fun x -> x < 4) [4;3;9;6;1;0;5] ;;
 - : int list = [3; 1; 0]

# List.fold_right ;;
 - : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# List.fold_right (^) ["a";"b";"c"] "x" ;;
 - : string = "abcx"

# List.fold_left ;;
 - : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```
# List.fold_left (^) "x" ["a";"b";"c"] ;;
 - : string = "xabc"

# List.find ;;
 - : ('a -> bool) -> 'a list -> 'a = <fun>

# List.find (fun x -> x > 10) [1;5;10;13;19] ;;
 - : int = 13

# List.find (fun x -> x > 10) [1;5;10] ;;
 Exception: Not_found.

# List.rev [8; 9; 10] ;;
 - : int list = [10; 9; 8]

# List.nth [8; 9; 10] 2 ;;
 - : int = 10

# List.nth [8; 9; 10] 3 ;;
 Exception: (Failure nth).

# (* ......................... VARIANT DATA TYPES *)
# type 'a option =
#   | Some of 'a
#   | None  ;;
 type 'a option = Some of 'a | None

# type 'a stack =
#   | Empty
#   | Top of ('a * 'a stack) ;;
 type 'a stack = Empty | Top of ('a * 'a stack)

# Top (3, Empty) ;;
 - : int stack = Top (3, Empty)

# (* ................................. EXCEPTIONS *)
# Not_found ;;
 - : exn = Not_found

# raise Not_found ;;
 Exception: Not_found.

# raise (Failure "error") ;;
 Exception: (Failure error).

# failwith "error" ;;
 Exception: (Failure error).

# try
#   Some (List.find (fun x -> x > 10) [1;5;10])
# with
#   | Not_found -> None
#   | e -> raise e ;;
 - : int option = None

# (* ..................... MODULES AND SIGNATURES *)
# module type STACK =
#   sig
#     type elt
#     type stack
#     exception Empty of string
#     val empty : unit -> stack
#     val push : stack * elt -> stack
#     val pop : stack -> elt * stack
#     val isEmpty : stack -> bool
#   end ;;
 module type STACK =
   sig
     type elt
     type stack
     exception Empty of string
     val empty : unit -> stack
     val push : stack * elt -> stack
     val pop : stack -> elt * stack
     val isEmpty : stack -> bool
   end

# module MakeStack(Arg: sig type t end)
#        : (STACK with type elt = Arg.t) =
#   struct
#     type elt = Arg.t
#     type stack = elt list
#     exception Empty of string
#     let empty () = []
#     let push (s,x) = x::s
#     let pop s = match s with
#     | x :: t -> (x, t)
#     | [] -> raise (Empty "empty")
#     let isEmpty = fun x -> x = []
#   end ;;
 module MakeStack :
   functor (Arg : sig type t end) ->
     sig
       type elt = Arg.t
       type stack
       exception Empty of string
       val empty : unit -> stack
       val push : stack * elt -> stack
       val pop : stack -> elt * stack
       val isEmpty : stack -> bool
     end

# module IntStack = MakeStack(struct type t = int end) ;;
 module IntStack :
   sig
     type elt = int
     type stack
     exception Empty of string
     val empty : unit -> stack
     val push : stack * elt -> stack
     val pop : stack -> elt * stack
     val isEmpty : stack -> bool
   end

# IntStack.push (IntStack.empty () , 3) ;;
 - : IntStack.stack = <abstr>
```

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`

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

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

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

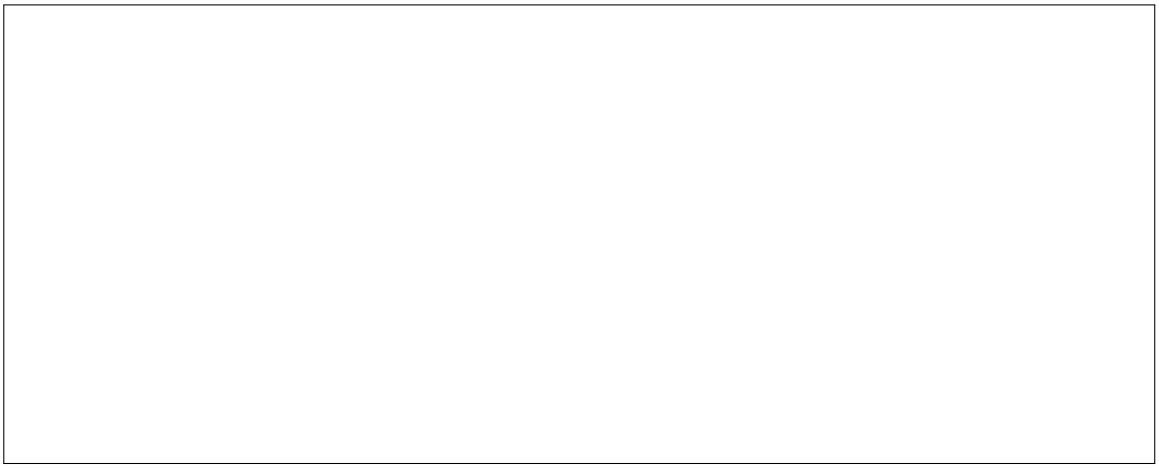
Problem 2. ***[12 points]*** *For each of the following definitions of a function* `f`*, give its most general type (as would be inferred by OCaml) or explain briefly why no type exists for the function. (The first problem is done for you as an example.)*

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

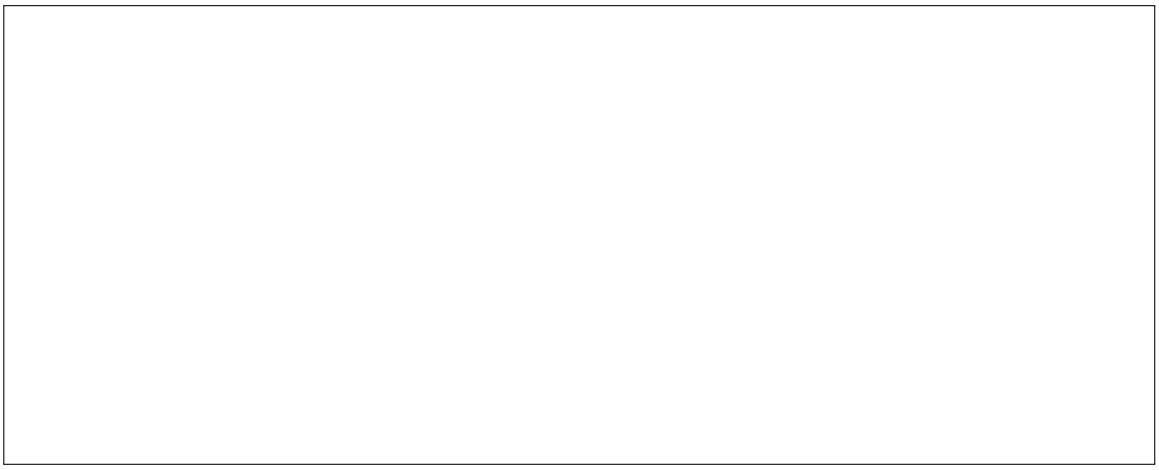
```
f :  float -> float
```

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

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



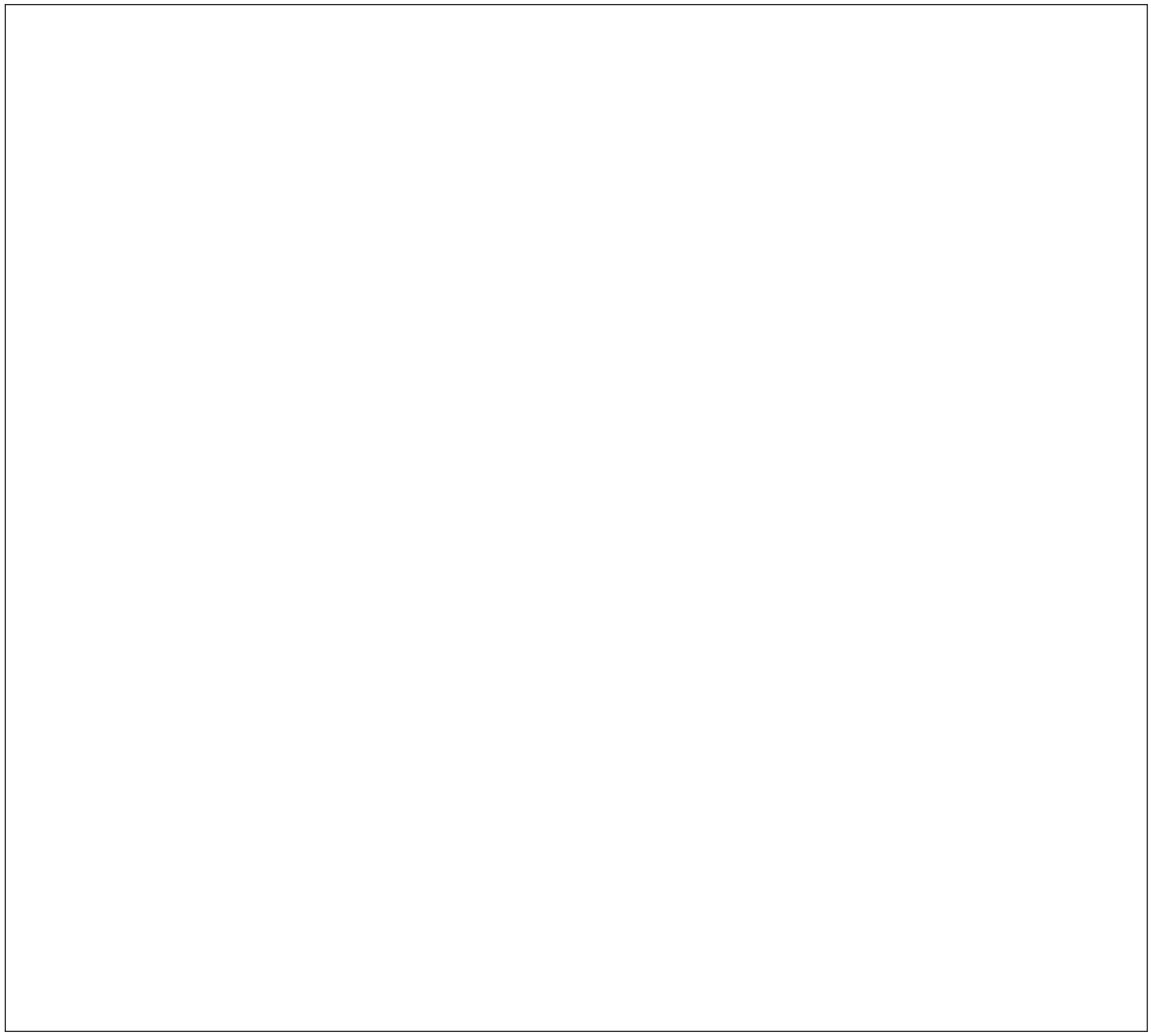
(4)
```
let rec f x a =
  match x with
  | [] -> a
  | h :: t -> h (f t a) ;;
```



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 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
```

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

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.)*

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.

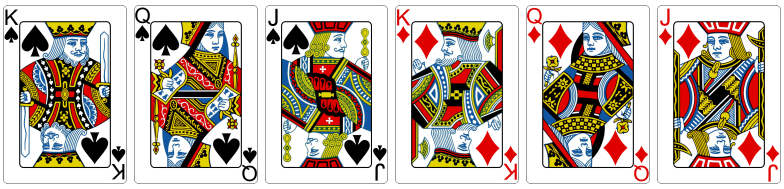


Figure 1. The cards of mini-poker, depicted in order from best to worst.

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.*

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?*

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

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.)*

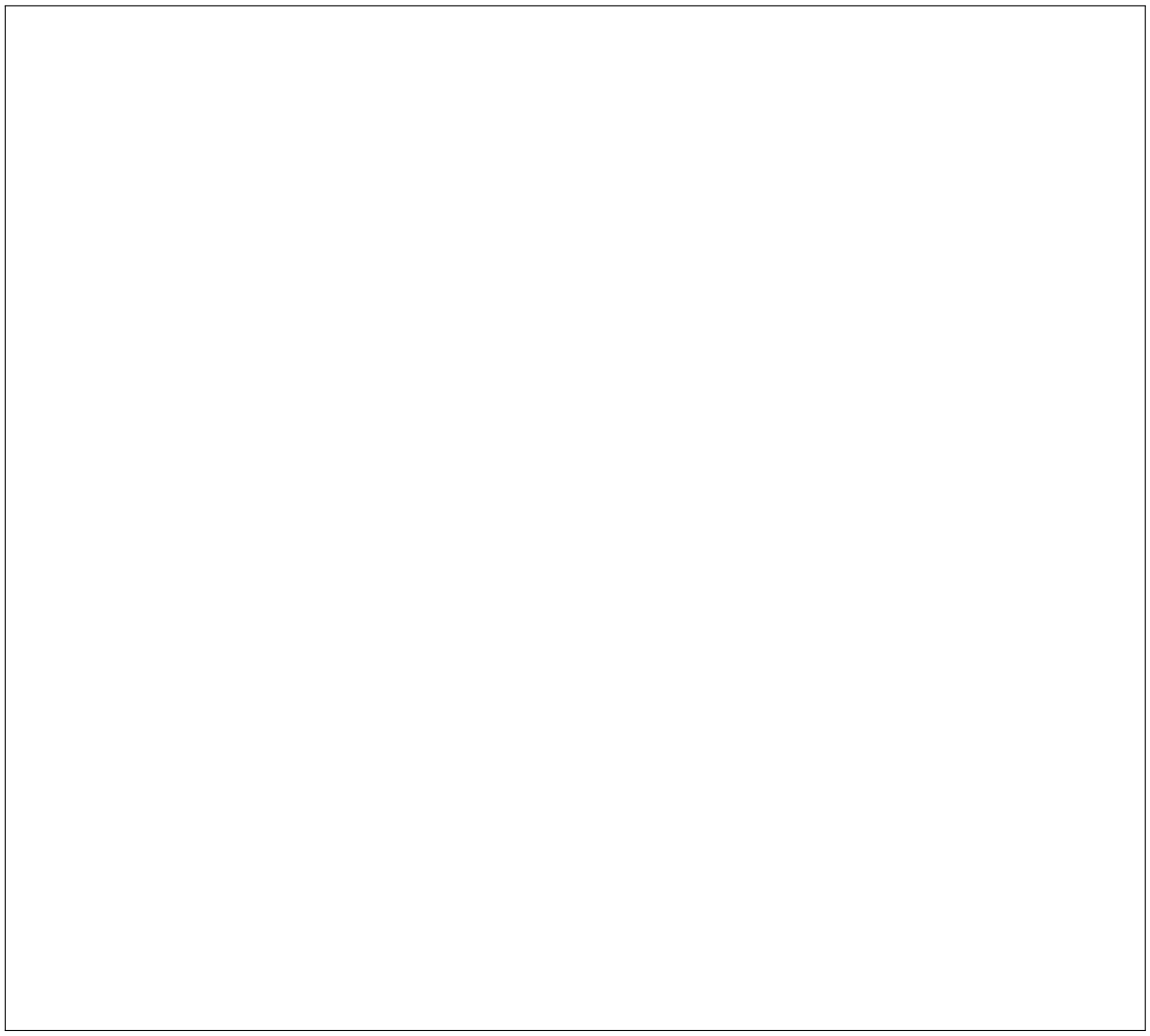
```
module                                                    =
  struct
    (* ...  the implementation would go here ...  *)
  end ;;
```

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`*.*

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.*



Total points: 85

Extra space for answers

Reference this area with “See box 1”

Reference this area with “See box 2”

Reference this area with “See box 3”

Reference this area with “See box 4”