CS 51: Introduction to Computer Programming II
First Midterm Examination
Spring, 2018

You have 90 minutes to complete this exam.

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.

The exam is in three sections comprised of 19 questions. Numbers in brackets like this ***[nnn points]*** *are the points (out of 86 total) allocated to the problem and may provide a very approximate recommendation for allocating time.*

Write the answers to all problems in the boxes provided. Write with a pen (or a very dark pencil) as we will be scanning your exams for grading. Write clearly, as we can and will only grade what we can unambiguously read. The exam packet is intentionally stapled in the lower left corner to facilitate the scanning process. Do not remove the staple or remove any pages from the exam packet. 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.

Many of the problems ask you to define something or write code to do something. Throughout the exam, when we ask you to define a value or function or type or class or module, we mean that you should provide a top-level OCaml definition written in well-formed, idiomatic OCaml using the appropriate OCaml definitional construct (`let`*,* `type`*,* `class`*,* `module`*, etc.). Your answers will be graded firstly on the well-formedness and correctness of the code, but in keeping with the course's goals, we may also secondarily consider the many other dimensions of code quality – including design and style issues – in evaluating your answers.*

To allow for anonymous grading of the exam, please write your name and ID number in the boxes below on this page ***[0.5 points]****, and your ID number (but not your name) in the box provided at the top of all subsequent odd-numbered pages* ***[0.5 points]****.*

By filling in your name and ID number below, you affirm your awareness of the standards of the Harvard College Honor Code.

Solution:

SOLUTION

1. TYPES AND TYPE INFERENCE

Problem 1. ***[12 points]*** *For each of the following OCaml function types define a function (with no explicit typing annotations, that is, no uses of the* `:` *operator) for which OCaml would infer that type. The expressions need not be practical or do anything useful; they need only have the requested type. Do not make use of anything from any library modules other than* `Pervasives`*. Provide your answers in the boxes provided below. (The first problem is done for you as an example.)*

(1) `bool -> unit`

(2) `'a option list -> bool`

Solution:

```
let f l =
  match l with
  | [Some x] -> true
  | _ -> false ;;
```

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

Solution:

```
let f l =
  match l with
    | [] -> None
    | hd :: _ -> Some hd ;;
```

(4) `(bool option -> bool) -> bool`

Solution:

```
let f g =
  g (Some true) = false ;;
```

(5) `'a -> 'a -> 'b`

Solution: There are two approaches to this problem.

(a) Raise an exception instead of returning:

```
let f x y =
  if x = y then failwith "true"
  else failwith "false" ;;
```

(b) Recur indefinitely to prevent a return type:

```
let rec f x y =
  if x = y then f x y
  else f x y ;;
```

or even more elegantly:

```
let rec f x y = f y x ;;
```

Problem 2. ***[12 points]*** *For each of the following function definitions, give a typing for the function that provides 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 f1 x =
  x +. 42. ;;
```

(2)
```
let f2 x y =
  x y y ;;
```

Solution:

```
f2 : ('a -> 'a -> 'b) -> 'a -> 'b
```

(3)
```
let f3 x y =
  x (y y) ;;
```
Solution: This definition does not type. Since `y` is applied as a function, its type must be of the form `'a -> 'b`. Since it is applied to `y` itself, `'a` must be identical to `'a -> 'b`. But there is no finite type satisfying that constraint. *A type can't be a subpart of itself.*

(4)
```
let rec f4 x =
  match x with
  | None
  | Some 0 -> None
  | Some y -> f4 (Some (y-1)) ;;
```
Solution:
```
f4 : int option -> 'a option
```

(5)
```
let f5 x y =
  if x then [x]
  else [not x; y] ;;
```

Solution:

```
f5 : bool -> bool -> bool list
```

2. Defining some functions

Problem 3. ***[4 points]*** *Define a function* `even` *that returns* `true` *if its integer argument is even, and false otherwise.*

Solution:

```
let even x =
  x mod 2 = 0 ;;
```

Problem 4. ***[2 points]*** *What is the type of* `even`*?*

Solution:

```
int -> bool
```

Problem 5. ***[3 points]*** *Define a function* `odd` *that returns* `true` *if its integer argument is odd, and false otherwise. You'll want to make use of the function* `even` *you just defined in Problem 3.*

Solution:

```
let odd x = not (even x) ;;
```

The OCaml documentation for the `List` module describes a function called `partition`:

```
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
```

`partition p l` returns a pair of lists `(l1, l2)`, where `l1` is the list of all the elements of `l` that satisfy the predicate `p`, and `l2` is the list of all the elements of `l` that do not satisfy `p`. The order of the elements in the input list is preserved.

For example, using the `even` function from Problem 3, the following should hold:

```
# partition even [1; 2; 3; 4; 5; 6; 7] ;;
- : int list * int list = ([2; 4; 6], [1; 3; 5; 7])
```

Problem 6. ***[6 points]*** *Give your own definition of the* `partition` *function, implemented directly without use of any library functions except for those in the* `Pervasives` *module.*

Solution:

```
let rec partition test lst =
  match lst with
  | [] -> [], []
  | hd :: tl ->
     let yeses, nos = partition test tl in
     if test hd then hd :: yeses, nos
     else yeses, hd :: nos ;;
```

FIGURE 1. Alexander Calder, "Blue and yellow among reds", 1964.

3. MOBILES

The artist Alexander Calder (1898–1976) is well known for his distinctive *mobiles*, sculptures with different shaped objects hung from a cascade of connecting metal bars. An example is shown in Figure 1.

His mobiles are made with varying shapes at the ends of the connectors – circles, ovals, fins. The exquisite balance of the mobiles depends on the weights of the various components. In the next sections of the exam, you will model the structure of mobiles as binary trees to allow determining if a Calder-like mobile design is balanced or not.

Let's start with the objects at the ends of the connectors. For our purposes, the important properties of an object will be its shape and its weight (in arbitrary units; you can interpret them as pounds).

Problem 7. ***[2 points]*** *Define a* `weight` *type consisting of a single floating point weight.*

Solution:

```
type weight = float ;;
```

Problem 8. ***[3 points]*** *Define a* `shape` *type, a variant type that allows for three different shapes: circles, ovals, and fins. (No information beyond the type of shape need be provided for.)*

Solution:

```
type shape = Circle | Oval | Fin ;;
```

Problem 9. ***[3 points]*** *Define an* `obj` *type that will be used to store information about the objects at the ends of the connectors, in particular, their weight and their shape.*

Solution:

```
type obj = { weight : weight; shape : shape }
```

Problem 10. ***[2 points]*** *To demonstrate the use of the types you've just defined, define a value* `shape1 : obj` *that represents an oval of weight 8.*

Solution:

```
let shape1 =
  { shape = Oval;
    weight = 8.0 } ;;
```

Note the floating point weight.

4. A BINARY TREE SIGNATURE AND FUNCTOR

A mobile can be modeled as a kind of binary tree, where the leaves of the tree, representing the objects, are elements of type `obj`, and the internal nodes, representing the connectors, have a weight, and each internal node (connector) connects two submobiles. Rather than directly writing code for a `mobile` type, though, we'll digress to build a more general binary tree module, and then model mobiles using that.

An appropriate signature `BINTREE` for a simple binary tree module might be the following:

```
module type BINTREE =
  sig
    type leaft  (* the type for the leaves of the tree *)
    type nodet  (* the type for the internal nodes of the tree
  *)
    type tree   (* the type for the trees themselves *)

    val make_leaf : leaft -> tree
    val make_node : nodet -> tree -> tree -> tree

    val walk : (leaft -> 'a) -> (nodet -> 'a -> 'a -> 'a) -> tree
  -> 'a
  end ;;
```

This module signature specifies separate types for the leaves of trees and the internal nodes of trees, along with a type for the trees themselves; functions for constructing leaf and node trees; and a single function to "walk" the tree. (We'll come back to the `walk` function later.)

In addition to the signature for binary tree modules, we would need a way of generating implementations of modules satisfying the `BINTREE` signature, which we'll do with a functor `MakeBintree`. The `MakeBintree` functor takes an argument module of type `BINTREE_ARG` that packages up the particular types for the leaves and nodes, that is, the types to use for `leaft` and `nodet`. The following module signature will work:

```
module type BINTREE_ARG =
  sig
    type leaft
    type nodet
  end ;;
```

Problem 11. ***[6 points]*** *Define the header of a functor named* `MakeBintree` *to generate modules satisfying the* `BINTREE` *signature by filling in the template below, keeping in mind the need for users of the functor-generated modules to access appropriate aspects of the generated trees.*

```
module [                                                    ] =
  struct

    (* ... the implementation would go here,
           but you don't have to worry about that ...  *)

  end ;;
```

Solution:

```
module MakeBintree (Element : BINTREE_ARG)
       : (BINTREE with
            type leaft = Element.leaft and
            type nodet = Element.nodet) =
  struct
    ...
  end ;;
```

Using this functor, you can now generate a `Mobile` module, which has `obj`s at the leaves and `weight`s at the interior nodes.

Problem 12. ***[5 points]*** *Define a module* `Mobile` *using the functor* `MakeBintree`.

Solution:

```
module Mobile = MakeBintree (struct
                               type leaft = obj
                               type nodet = weight
                             end) ;;
```

An alternative is to explicitly define the argument value:

```
module MobileArg =
  struct
    type leaft = obj
    type nodet = weight
  end ;;

module Mobile = MakeBintree (MobileArg) ;;
```

If a module type is given to the argument module, however, there need to be sharing constraints. So the following is bad:

```
module MobileArg : BINTREE_ARG =
  struct
    type leaft = obj
    type nodet = weight
  end ;;

module Mobile = MakeBintree (MobileArg) ;;
```

It should be

```
module MobileArg : (BINTREE_ARG with type leaft = obj
                                 and type nodet = weight) =
  struct
    type leaft = obj
    type nodet = weight
  end ;;

module Mobile = MakeBintree (MobileArg) ;;
```

Problem 13. ***[2 points]*** *You've just used the* `MakeBintree` *functor without ever seeing its implementation. Why is this possible?*

Solution: The only aspects pertinent to the *use* of a module are manifest in the *signature*. The users of a module aren't able to take advantage of any aspects of the implementation other than those shown in the signature.

(This is why enforcing module signatures is a great way of satisfying the edict of compartmentalization.)

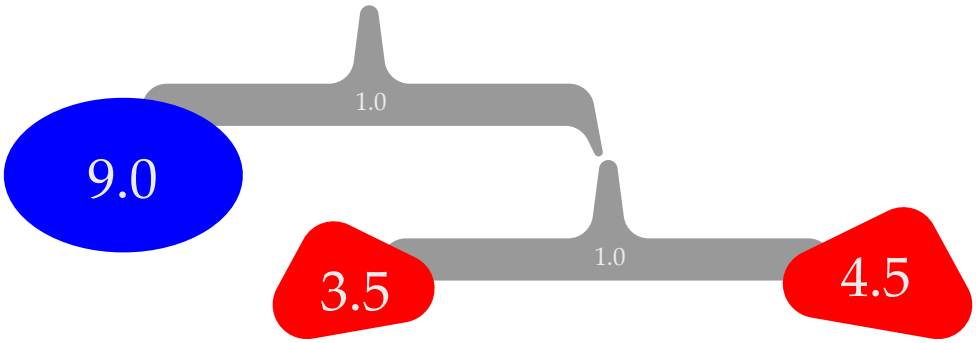

FIGURE 2. A simple Calder-style mobile. The depicted mobile has two connectors and three objects (an oval and two fins). The connectors each weigh 1.0, and the objects' weights are as given in the figure.

You can now build a representation of a mobile using the functions that the `Mobile` module makes available.

Problem 14. ***[4 points]*** *Define a value* `mobile1` *of type* `Mobile.tree` *that represents a mobile structured as the one in Figure 2.*

Solution:

```
let mobile1 =
  let open Mobile in
  make_node 1.0
            (make_leaf {shape = Oval; weight = 9.0})
            (make_node 1.0
                       (make_leaf {shape = Fin; weight = 3.5})
                       (make_leaf {shape = Fin; weight = 4.5}))
  ;;
```

The `walk` function, of type `(leaft -> 'a) -> (nodet -> 'a -> 'a -> 'a) -> tree -> 'a`, is of special interest, since it is the sole method for performing computations over these binary trees. The function is a kind of fold that works over trees instead of lists. It takes two functions – one for leaves and one for nodes – and applies these functions to a tree to generate a single value. The leaf function takes a `leaft` and returns some value of type `'a`. The node function takes a `nodet` and the two `'a` values recursively returned by walking its two subtrees and computes the value for the node itself.

For example, we can use `walk` to define a function `size` that counts how many objects there are in a mobile. The function uses the fact that leaves are of size 1 and the size of a non-leaf is the sum of the sizes of its subtrees.

```
let size mobile =
  Mobile.walk
    (fun _leaf -> 1)
    (fun _node left_size right_size -> left_size + right_size)
    mobile ;;
```

Problem 15. ***[3 points]*** *What is the type of* `size`*?*

Solution:

```
Mobile.tree -> int
```

Problem 16. ***[2 points]*** *Use the fact that the* `walk` *function is curried to give a slightly more concise definition for* `size`*.*

Solution:

```
let size =
  Mobile.walk
    (fun _leaf -> 1)
    (fun _node left_size right_size -> left_size + right_size)
  ;;
```

Problem 17. ***[5 points]*** *Use the* `walk` *function to implement a function* `shape_count : shape -> Mobile.tree -> int` *that takes a shape and a mobile (in that order), and returns the number of objects in the mobile that have that particular shape.*

Solution:

```
let shape_count (s : shape) =
  Mobile.walk (fun leaf -> if leaf.shape = s then 1 else 0)
              (fun _node l r -> l + r) ;;
```

A mobile will said to be *balanced* if every connector has the property that the total weight of all components (that is, objects and connectors) of its left submobile is the same as the total weight of all components of its right submobile. (In actuality, we'd have to worry about other things like the relative lengths of the arms of the connectors, but we'll ignore all that.)

Problem 18. ***[2 points]*** *Is the mobile shown in Figure 2 balanced? Why or why not?*

> **Solution:** No. While the upper internal node connects two submobiles of the same weight (9.), the lower internal node connects two submobiles of different weights (3.5 and 4.5).

Problem 19. ***[7 points]*** *Implement a function* `balance : Mobile.tree -> weight option` *that takes a mobile, and returns* `None` *if the argument mobile is not balanced, and* `Some w` *if the mobile is balanced, where* `w` *is the total weight of the mobile.*

> **Solution:**
>
> ```
> let balanced =
>   Mobile.walk (fun leaf -> Some leaf.weight)
>               (fun node l r ->
>                match l, r with
>                | Some wt1, Some wt2 ->
>                    if wt1 = wt2 then Some (node +. wt1 +. wt2)
>                    else None
>                | _, _ -> None) ;;
> ```

End of exam.

Total points: 86

EXTRA SPACE FOR ANSWERS