CS 51: Introduction to Computer Programming II
First Midterm Examination
Spring, 2017

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. We have provided a separate handout with reference material – a cheat sheet on OCaml – which you need not turn in.

The exam is in three sections comprised of 14 questions. Numbers in brackets like this ***[nnn points]*** *are the points (out of 90 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 clearly, as we can and will only grade what we can unambiguously read. Write with a dark pencil or pen as we will be scanning your exams for grading. 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 idiomatic OCaml using the appropriate OCaml definitional construct (`let`*,* `type`*,* `class`*,* `module`*, etc.).*

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 pages* ***[0.5 points]****.*

Your name:
SOLUTION
Your Harvard ID number:

1. 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`

```
let f b = if b then () else () ;;
```

(2) `int -> int -> int option`

```
let f x y =
  Some (x + y) ;;
```

(3) `(int -> int) -> int option`

```
let f g =
  Some (1 + g 3) ;;
```

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

```
let f x g = g x ;;
```

or

```
let f = ( |> ) ;;
```

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

```
let f xs ys =
  match xs, ys with
  | (Some x) :: _, (Some y) :: _ -> x, [y]
  | _, _ -> failwith "" ;;
```

Problem 2. ***[10 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. ;;
```

```
f1 :  float -> float
```

(2)
```
let f2 x y =
  match x with
  | (w, z) -> if w then y z else w ;;
```

```
f2 : bool * 'a -> ('a -> bool) -> bool
```

(3)
```
let rec f3 x =
  match x with
  | [] -> f3
  | h :: t -> raise Exit ;;
```

No type exists for `f3`. Assume that the type of `f3` is some instantiation of the function type `'a -> 'b`. Since the first match clause returns `f3`, the result type `'b` of `f3` must be the entire type `'a -> 'b` of `f3` itself. But a type can't contain itself as a subpart. So no type for `f3` exists.

(4)
```
let f4 x =
  if x then (x, true)
  else (true, not x) ;;

f4 : bool -> bool * bool
```

Problem 3. ***[4 points]*** *Provide a more succinct definition of the function* `f4` *from Problem 2(4), which defines* `f4` *to have the same type and behavior.*

Taking advantage of the fact that `f4` always returns the same value:

```
let f4 (b : bool) = true, true
```

Note that the typing of `b` is required to force the function type to be `bool -> bool * bool` instead of `'a -> bool * bool`.

2. Walking trees

Problem 4. ***[8 points]*** *The following code was intended to define a data type* `'a tree` *for a kind of polymorphic binary trees with values stored at some of the leaves and with other leaves left empty, plus a function* `sum_tree : int tree -> int` *that returns the sum of all the integers stored in an integer tree. However, it contains errors that will generate error messages and warnings in several places. Identify as many such errors and warnings as there are (but no more), giving line numbers for each and explaining what each problem is as specifically as you can.*

```
1  let 'a tree =
2    | Empty
3    | Leaf of 'a
4    | Node of (tree, tree) ;;
5
6  let sum_tree (t : int tree) =
7    match t with
8    | Leaf x -> x
9    | Node (l, r) -> (sum_tree l) + (sum_tree r) ;;
```

1. The type definition should be introduced by let instead of type.
4. The comma (a value constructor) should be replaced by * (the appropriate type constructor.
4. The recursive references to the type being defined need to mention the argument type 'a (twice).
6. The definition is intended to be recursive, but the rec keyword is missing after the let.

7–8. There is no pattern for Empty trees, so the code would generate a nonexhaustive match warning.

Problem 5. ***[6 points]*** *Write the correct type definition for the* `'a tree` *data type from the previous problem. Make sure it is consistent with how the type is used in the* `walk` *function below.*

```
type 'a tree =
  | Empty
  | Leaf of 'a
  | Node of ('a tree * 'a tree)
```

Consider the following function named `walk` that "walks" over a tree and applies a binary function (the argument `f`) to the children of a node in the tree – a kind of "fold" operation for trees. At the leaves, it returns whatever value is stored there, and for empty nodes it returns a default value (the argument `default`).

```
let rec walk f default (t : 'a tree) =
  match t with
  | Empty -> default
  | Leaf x -> x
  | Node (l, r) -> f (walk f default l) (walk f default r) ;;
```

Problem 6. ***[4 points]*** *What type would OCaml infer for the* `walk` *function as defined above? Write the type as a single OCaml type expression.*

```
walk : ('a -> 'a -> 'a) -> 'a -> 'a tree -> 'a
```

Problem 7. ***[6 points]*** *Notice that each time* `walk` *is recursively called, it passes along the same first two arguments. Write a version of* `walk` *that uses a local function to avoid this redundancy.*

```
let walk f default (t : 'a tree) =
  let rec walk' t =
    match t with
    | Empty -> default
    | Leaf x -> x
    | Node (l, r) -> f (walk' l) (walk' r) in
  walk' t ;;
```

or this slightly less attractive alternative (which introduces a level of function application indirection and doesn't take advantage of the lexical scoping):

```
let rec walk f default (t : 'a tree) =
  let walk' = walk f default in
  match t with
  | Empty -> default
  | Leaf x -> x
  | Node (l, r) -> f (walk' l) (walk' r) ;;
```

At least it uses the partal application of `walk` in the definition of `walk'`.

Problem 8. ***[4 points]*** *Use the* `walk` *function in writing a definition for a function* `sum_tree : int tree -> int` *that sums up the values stored at all of the leaves of an* `int tree`*.*

```
let sum_tree = walk (+) 0 ;;
```

The parentheses are required here.

Problem 9. ***[4 points]*** *Use the* `walk` *function in writing a definition for a function* `max_tree : int tree -> int` *that returns the maximum value stored at the leaves of a tree. You'll want to think carefully about what integer* `max_tree` *should return for the* `Empty` *tree that makes your code as simple as possible.*

```
let max_tree = walk max min_int ;;
```

Functions returning an `option` type in an attempt to deal with the `Empty` case are inconsistent with the problem spec, which specifies a return type of `int`.

3. An ADT for intervals

A good candidate for an abstract data type is the interval. Abstractly speaking, an interval is a region between two points, where all that is required of points is that we be able to compare them as an ordering (so that we have a well-defined notion of "between"). That is, points ought to obey the following signature, which may look familiar, as you've seen it in other contexts:

```
module type COMPARABLE =
  sig
    type t
    type order = Less | Equal | Greater
    val compare : t -> t -> order
  end ;;
```

Natural operations over intervals are the construction of an interval between two points, the extraction of the endpoints of an interval, taking the union of two intervals (the smallest interval containing both), and determining the relation between two intervals. Here is a signature that provides for this functionality.

```
module type INTERVAL =
  sig
    type point
    type interval
    type relation = Disjoint | Overlaps | Contains
    (* Returns the interval between two points *)
    val interval : point -> point -> interval
    (* Returns the endpoints of an interval as a pair
       with the first point less than the second. *)
    val endpoints : interval -> point * point
    (* Returns the union of two intervals *)
    val union : interval -> interval -> interval
    (* Returns the relation holding between two intervals *)
    val relation : interval -> interval -> relation
  end ;;
```

The possible relations between two intervals are depicted in Figure 1. (For the interval arithmetic cognoscenti, we've left out many details, such as whether intervals are open or closed; more fine-grained relations; and many other useful operations on intervals. These issues are beyond the scope of this problem.)

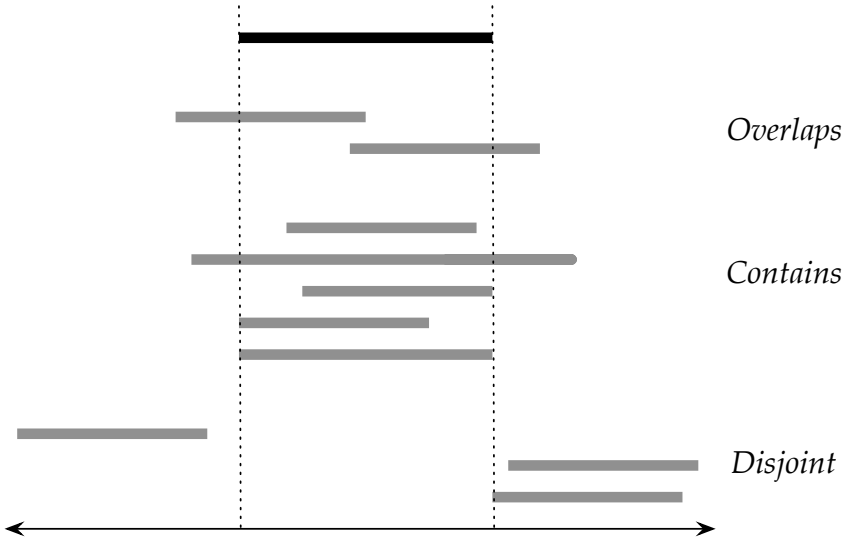

FIGURE 1. A diagrammatic depiction of the possible relations holding between two intervals. In the diagram, the gray intervals in the three groups below the black interval are in the "overlaps" (top 2), "contains" (next 5), and "disjoint" (bottom 3) relations, respectively, with the black interval at top. The vertical dotted lines depict the endpoints of the black interval.

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

```
module MakeInterval (Point :  COMPARABLE)
                    :  (INTERVAL with type point = Point.t)
  struct
    (* ...  the implementation would go here ...  *)
  end ;;
```

Problem 11. ***[8 points]*** *An appropriate module satisfying* `COMPARABLE` *for the purpose of generating* discrete *time intervals would be one where the type is* `int`*, with an appropriate comparison function. Define a module named* `DiscreteTime` *satisfying* `COMPARABLE` *where the type is* `int`*. Make sure the type is accessible outside the module.*

```
module DiscreteTime : (COMPARABLE with type t = int) =
  struct
    type t = int
    type order = Less | Equal | Greater
    let compare x y = if x < y then Less
                      else if x = y then Equal
                      else Greater
  end ;;
```

Problem 12. *[3 points] Now use the functor* `MakeInterval` *to define a module* `DiscreteTimeInterval` *that provides interval functionality over discrete times as defined by the module* `DiscreteTime` *above.*

```
module DiscreteTimeInterval =
  MakeInterval (DiscreteTime) ;;
```

Problem 13. *[6 points] The intersection of two intervals is only well-defined if the intervals are not disjoint. Assume that the* `DiscreteTimeInterval` *module has been opened (as we've already done for you in the box below), allowing you to make use of everything in its signature. Now, define a function* `intersection : interval -> interval -> interval option` *that takes two intervals and returns* `None` *if they are disjoint and otherwise returns their intersection (embedded appropriately in the option type).*

```
open DiscreteTimeInterval ;;

let intersection i j =
  if relation i j = Disjoint then None
  else let (x, y), (x', y') = endpoints i, endpoints j in
       Some (interval (max x x') (min y y')) ;;
```

Problem 14. ***[8 points]*** *Provide three different unit tests that would be useful in testing the correctness of the* `DiscreteTimeInterval` *module. We've provided some context in which your tests should appear. You can use* `CS51.verify` *or* `assert` *as you prefer.*

There are myriad solutions here. The idea is just to establish a few intervals and then test that you can recover some endpoints or relations.

```
open CS51 ;;
let test () =
  let open DiscreteTimeInterval in

  let i1 = interval 1 3 in
  let i2 = interval 2 6 in
  let i3 = interval 0 7 in
  let i4 = interval 4 5 in
  verify (relation i1 i4 = Disjoint) "disjoint\n";
  verify (relation i1 i2 = Overlaps) "overlaps\n";
  verify (relation i1 i3 = Contains) "contains\n";
  verify (relation (union i1 i2) i4 = Contains) "unioncontains\n";
  let i23 = intersection i1 i2 in
  verify (let Some e23 = i23 in endpoints e23 = (2, 3)) "intersection";;

  Printf.printf "tests completed\textbackslash n" ;;
```

End of exam

Total points: 90

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”

Reference this area with “See box 5”

Reference this area with “See box 6”