CS51 **Spring 2024**

Code Review 2: HOFs, Polymorphism, Anomalous Conditions

1 Records

Definition 1.1 (Records)**. Records** are a *fixed* length sequence of elements composed of one or more elements each with a *unique label*.

- Unlike tuples, which use *order* to select elements, records select elements by *label*. You can think of records as a labeled form of tuples.
- Each of the elements in a record is a *field*. Unlike tuples, which are *built-in* datatype in `OCaml`, records are *user-defined*, so we must specify the *fields* and types of each field to use them using the type constructor.

Syntax 1.2 (Records)**.**
Here is how we would generally define a record type `typename`.

- Here is how we would generally define a record type `typename`. `type typename = { field1: type1; field2: type2; ... }`
- We construct records of type `typename` using the following syntax:

  `let rec_ex : typename = {field1 = value1; field2 = value2; ... } .`

Example 1.3 (Points)**.** In Lab 3, we use tuples and records to represent 2D-points.

We defined the type `point_pair` as follows:

```
type point_pair = int * int ;;
```

This definition allows us to represent points using a tuple of two integers.[1]

We can also use records to represent points. Here is how we defined the `point_record` type.

```
type point_record = {x: int; y : int}
```

Here, records of type `point_record` have two fields, `x` and `y`, both of type `int`.

We can construct a record of type `point_record`:

```
let r : point_record = {x = 5; y = 3} ;;
```

1.1 Record Decomposition

There are several ways to decompose records and select record fields. **Please read section 7.4 of the textbook and review lab 3 solutions** for details.

1. *Let Decomposition*. As with tuples, we can bind record field values to variables. `let { x = x1; y = y1 } = r` binds the `x` and `y` field value of `r` to `x1` and `y1`, respectively.

[1] Each point has an x-coordinate and a y-coordinate. However, with this definition, we would have to assume that the x-coordinate is always the first element in the tuple and the y-coordinate is always the second element. Conventionally, when we represent a point (x, y), we assume x represents the x-coordinate and y represents the y-coordinate. However, with points, we might be more concerned with what the x-coordinate is and what the y-coordinate is, so the order in which we list the coordinates doesn't really matter. Here is where records might be more practical.

2. *Dot Notation*. We can directly access field values by using the syntax `record.fieldname`. For example, we can access the `x-coordinate` of `r` by writing `r.x`.

3. *Field Punning* allows us to bind the value of a field to its own fieldname. `let {x; y} = r` binds the x-coordinate and y-coordinate to variables `x` and `y`.

Example 1.4 (Record Decomposition)**.** Suppose we want to define a function `dot_product_recd` that computes the dot product of two points encoded in the `point_recd` type.

Listing 1: Let Decomposition

```
let dot_product_recd (p1: point_recd) (p2 : point_recd) : int =
 let {x = x1, y = y1}, {x = x2, y = y2} = p1, p2 in
  x1 * x2 + y1 * y2 ;;
```

Listing 2: Dot Notation

```
let dot_product_recd (p1: point_recd) (p2 : point_recd) : int =
 p1.x * p2.x + p1.y * p2.y ;;
```

Listing 3: Field Punning and Header Decomposition

```
let dot_product_recd ({x; y} : point_recd) ({x = x2; y = y2} :
   point_recd) : int =
 x * x2 + y * y2 ;;
```

Note for Listing 3, that we can only use field punning once to not confuse the variable names `x` and `y`.[2] We also can decompose records and tuples in the header of the function, as shown there.

2 Polymorphism

Definition 2.1 (Polymorphism)**.** *Polymorphic* simply means to take on *many forms*. In the context of `OCaml`, we can define **polymorphic functions** which can operate on values of generic types.

For example, suppose we define the following function (as illustrated in Section 9.1 of the textbook):

```
let id x = x ;;
```

- We say the type of `id` is `'a -> 'a` (read "alpha to "alpha").
- The `id` function takes in a value of a type `'a` and returns a value of a different type. `'a` is a **type variable**, that specifies any type can be used.
- Type variables are prefixed by a quote mark, and conventionally read as a Greek letter. For instance, `'a`, `'b`, `'c`, and `'d` are type variables read as α ("alpha), β ("beta"), γ ("Gamma"), and δ ("Delta"), respectively.

Type expressions that include type variables are known as **polymorphic types**. A type expression `'a * 'b -> 'c` would mean that `'a`, `'b`, and `'c` could be the same or different types.

[2]Style Note: Although Field Punning makes the function definition slightly more concise, it might not be best to use it here since our function takes in two records, and we have an asymmetry by only using field punning to deconstruct the first record. It would be more idiomatic to deconstruct the first record in the same manner in which we deconstructed the second record.

Remark 2.2 (Type Variable Naming)**.** . We can name type variables just like normal variables, and their naming does not need to follow the convention with Greek letters. For instance, we could specify the type of the `id` expression as `'in -> 'in`.

3 Higher-Order Functions

Definition 3.1 (Higher-Order Function)**.** Remember that in `OCaml`, **functions are values**. Since functions map values to other values, functions can also take in other functions as inputs or return functions.

A **higher-order-function (HOF)** is a function that does at least one of the following:

1. Takes one or more functions as arguments
2. Returns a function as its output

Example 3.2 (Apply)**.** A function that takes in a function `f` and applies it to an argument `x` is an example of a HOF. The type of this function is `('a -> 'b) -> 'a -> 'b`.

Listing 4: Apply HOF

```
let apply (f: 'a -> 'b) (x : 'a) : 'b =
 f x ;;
```

3.1 Currying

Definition 3.3 (Curried Function)**.** A **curried function** takes its arguments *one at a time*.

- The function really only has one argument, and when it's applied to that argument, it returns another function which takes in the next argument.
- For example, `let add x y = x + y` is a curried function. As discussed in Code Review 1, `let add x y = x + y` is *syntactic sugar* for `let add x = fun y -> x + y`.

Definition 3.4 (Uncurried Function)**.** An **uncurried function** is a function that takes in its arguments *all at once*.

- The uncurried version of `add` is `let add (x, y) = x + y`. Here we take in our two arguments `(x, y)` as a tuple.[3]

Definition 3.5 (Partial Application)**. Partial application** is the applying of a *curried* function to only *some* of its arguments, resulting in a function that takes in the remaining arguments.[4]

- In the `add` example, we could directly compute the addition of two integers by evaluating something like `add 5 2`. We could also leverage *partial application* to return a function that adds an integer to the number `5` by calling `add 5`.
- Observe that partial application isn't possible with the uncurried `add` function, which takes in its arguments as a tuple `(x, y)`, so the only way to use this function is to apply it on both of its arguments at the same time.

[3] Many students get confused about the distinction between curried and uncurried functions because it is said that all functions in `OCaml` are curried. In the example, the "uncurried version" of add can be thought of as a "curried function", interpreting the tuple `(x, y)` as its only argument. To avoid confusion, it's best to just think about the distinctions as illustrated in the definition above. Thinking about how *partial application* is used can also help understand the distinction.

[4] Definition taken directly from the course textbook

4 List HOFs

`map`, `fold`, and `filter` are common HOF functions that operate on lists and are found in many functional programming languages. The `OCaml List` module defines these functions.

4.1 Map

Definition 4.1 (Map)**.** The `map` abstraction takes in a list ℓ and function `f` and applies `f` to every element in the list ℓ.

In `OCaml`, `List.map` is a function of type `('a -> 'b) -> 'a list -> 'b list`.

Here's how `map` could internally be defined:

Listing 5: Map

```
let rec map (f : 'a -> 'b) (lst : 'a list) : 'b list =
 match lst with
 | [] -> []
 | h :: t -> f h :: map f t ;;
```

Example 4.2 (Map Example)**.** Here's how we might use `map` to define a function `floats_of_ints` that converts a list of integers to a list of floats.

Listing 6: floats_of_ints

```
let floats_of_ints (lst : int list) : float list =
 List.map float_of_int lst  ;;

// This works, but since List.map is a curried function, we can use
   partial application.

let float_of_ints : int list -> float list =
 List.map float_of_int ;;
```

4.2 Filter

Definition 4.3 (Filter)**.** The `filter` abstraction takes in a list ℓ and predicate p and returns a list ℓ' containing elements in ℓ that satisfy the predicate p.

- In `OCaml`, `List.filter` is a function of type `('a -> bool) -> 'a list -> 'a list`.
- IThe predicate is a function of type `'a -> bool` that returns `true` if the element in the input list should be included in the output list, otherwise `false` if not.

Here's how filter might internally be implemented:

Listing 7: Filter

```
let rec filter (pred : 'a -> bool) (lst : 'a list) : 'a list =
 match lst with
 | [] -> []
 | h :: t ->
    if pred h then h :: filter pred t
    else filter pred t ;;
```

Example 4.4 (Map Example)**.** We might use `filter` to define a function `positive_pairs` that that takes in a list of `point_pairs` and returns a list of pairs in which both coordinates are positive.

Listing 8: positive_pairs

```
let positive_pairs : point_pair list -> point_pair list =
 List.filter (fun (x, y) -> x > 0 && y > 0) ;;
```

4.3 Fold

Definition 4.5 (Fold)**.** In computer science, the `fold` abstraction refers to a family of higher-order functions that *combine* elements in a data structure by iteratively or recursively processing the elements and accumulating a result.

The `OCaml List Module` defines `List.fold_left` and `List.fold_right`, which allows us to fold over lists.

For simplicity, we'll start with `fold_left`.

`fold_left` is a function of type `('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc`.

Here are the parameters of `fold_left`:

- The first argument is of type *'acc* and represents the **initial value** of the *accumulator*. In general, an *accumulator* is used to keep track of the result during an iteration or computation. In the context of lists, the accumulator is updated and passed along as `fold` processes each element of the list.
- The second argument, of type, `'a list`, represents the list that is being folded over.
- The third argument, of type `('acc -> 'a -> 'acc)`, is a function f that has two arguments: an element in the list and the current accumulator. The function returns the updated accumulator value as we apply f on each list element.
- The **return value** is the final value of the accumulator after we have gone through the entire list.

`fold_right` is defined similarly, just the order of the arguments are different. We will discuss the semantic differences between `fold_left` and `fold_right`.

Example 4.6 (Fold Example: Sum)**.** Suppose we want to define a function that returns the sum of elements in a list.

Listing 9: sum

```
let sum : int list -> int =
 List.fold_left (+) 0 ;;
```

Here, our accumulator is an integer that tracks the running sum as we process elements in the list. We pass in the function `(+)`, which will add each element to the current accumulator value.

4.3.1 Fold Left vs Fold Right

- Semantically, `fold_left` processes elements in the list from **left to right** while `fold_right` does so from **right to left**.

- The `OCaml` language documentation also mentions that `fold_left` is **tail-recursive** while `fold_right` is not. To illustrate this distinction, let's look at how `fold_left` and `fold_right` might be implemented.

Listing 10: fold_left

```
let rec fold_left
    (f : 'acc -> 'a -> 'acc) (init : 'acc)
    (lst : 'a list) : 'acc =
 match lst with
 | [] -> init
 | h :: t -> fold_left f (f init h) t;;
```

Listing 11: fold_right

```
let rec fold_right
    (f : 'a -> 'acc -> 'acc) (lst : 'a list)
    (init : 'acc) : 'acc =
 match lst with
 | [] -> init
 | h :: t -> f h (fold_right f t init) ;;
```

Definition 4.7 (Tail-Recursion)**.** A function is **tail-recursive** if the last computation that is executed in the function is a **recursive call**.

- We can see that `fold_left` is tail-recursive: in the recursive step, the last expression we compute is a recursive call to `fold_left`.
- `fold_right` is **not** tail-recursive: we first compute the result of `fold_right f t init`, and our last computation is applying the function `f` to the head of the list and the result of the recursive call.

Tail-recursive functions are generally more efficient than non-tail recursive functions. We'll discuss this more later in the course.

Example 4.8 (Reverse)**.** Suppose we want to define a function `reverse` that reverses the elements in a list.

Here's how we would define the function using `fold_left`.

Listing 12: Reverse

```
let reverse_left : 'a list -> 'a list =
 List.fold_left (fun acc x -> x :: acc) [] ;;
```

Now, suppose we define try to define `reverse` the same way, except using `fold_right`.

Listing 13: Reverse (Incorrect)

```
let reverse_right (lst : 'a list) : 'a list =
 List.fold_right (fun x acc -> x :: acc) lst [] ;;
```

If we test the behavior of the functions in the `REPL`:

```
utop # reverse_left [1; 2; 3; 4; 5] ;;
- : int list = [5; 4; 3; 2; 1]
utop # reverse_right [1; 2; 3; 4; 5] ;;
- : int list = [1; 2; 3; 4; 5]
```

We see that the definition of `reverse_left` correct, while the definition of `reverse_right` is incorrect. Here, the fact that `fold_left` combines elements from left to right and `fold_right` combines elements from right to left leads to different behaviors.

5 Anomalous Conditions

Definition 5.1 (Anomaly)**.**

- The course textbook defines an **anomaly** as a condition that a function can't handle.
- The function can either return a value that indicates the anomaly or call a *handler*, a function that handles the anomaly.
- For example, suppose we want to define a function that finds the maximum element in a list. An anomaly is the empty list, as the maximum is undefined for an empty list.
- Two ways to deal with anomalies are **options** and **exceptions**.

5.1 Options

Definition 5.2 (Options)**. Option types** are a **structured data type**.

- We can create an option type from a base type, using the `option` type constructor, in the same way we create a list type using the `list` type constructor.
- For example, a value can have type `'a option` where `'a` is the base type.

There are two value constructors for the option type values:

1. `None`: Indicates the presence of an anomalous condition.
2. `Some v`: Prefixes a value `v` of the base type. For example, `Some 5` is a value that has type `int option`.

Example 5.3 (Maximum of List with Options)**.** In lab 4, we defined a function `max_list_opt` that returns the maximum of a list using option types.

Listing 14: max_list_opt

```
let max_list_opt (lst : 'a list) : 'a option =
 match lst with
 | [] -> None
 | head :: tail ->
    match max_list_opt tail with
     | None -> Some head
     | Some max_tail -> Some (max head max_tail)
```

5.1.1 Option Poisoning

- There can be a problem with using option types to handle anomalies. For example, if we would like to do further computations with the `max_list_opt` function, we can't use the result of the function directly, we would have to carefully extract the value from the option type.
- For instance, consider if we just wanted to add the value 5 to the maximum of the list `[1; 2; 3; 4; 5]`. We couldn't simply do `(max_list_opt [1; 2; 3; 4; 5]) + 5`; this would result in a type error since `max_list_opt` would return a value of type `int option`, not `int`.

Instead, we would have to do something like this:

```
match (max_list_opt [1; 2; 3; 4; 5]) with
| None -> None
| Some v -> Some (v + 5)
```

- The course textbook refers to this phenomenon as **option poisoning**: "the introduction of an option type in an embedded computation requires verbose extraction of values and reinjecting them into an option type as the computation continues".
- **Option poisoning** makes the use of option types **less appropriate for when the occurrence of an anomalous condition is rare**. For these situations, **exceptions** are often more suitable (also see **Section 10.2.1** of the textbook for more discussion).

5.2 Exceptions

Definition 5.4 (Exceptions)**.**

- In a function like `max_list_opt`, instead of changing the return type of the function (returning a value of type `'a option` instead of `'a`), we can **raise an exception**.
- **When an exception is raised, the execution of the function stops**, which makes sense because if an anomaly occurs, there is no appropriate value to return and therefore the function shouldn't return any value.
- Like functions, exceptions are **first-class values** of the type `exn`. Like lists and options, **exceptions have value constructors**.
- To **raise an exception**, we can use the built-in `raise` function of type `exn -> 'a`.

Example 5.5 (Maximum of a List with Exceptions)**.** Suppose we want to redefine a function to find the maximum of a list with exceptions rather than options.[5]

Listing 15: max_list

```
let max_list (lst : 'a list) : 'a =
 match lst with
 | [] -> raise
    (Invalid_argument "maximum of empty list is undefined")
 | [elt] -> elt
 | h :: t -> max h (max t) ;;
```

[5] In fact, using an exception would be more appropriate, as *the passing of an empty list is a rare occurrence*.

Here, we use the pre-defined `Invalid_argument` value constructor that takes in a string and constructs an exception.

5.3 Defining Exceptions

In addition to using predefined exceptions using built-in value constructors such as `Exit`, `Failure`, and `Invalid_argument`, we can also declare new exceptions by defining new exception value constructors.

As illustrated in **Section 10.3.3** in the textbook, we can define a new exception by using the syntax `exception <name> of <type>`.

Example 5.6 (Declaring Exceptions)**.** Here's how we would declare some new exceptions.

Listing 16: Exceptions

```
exception Timeout ;;
exception UnboundVariable of string ;;
exception HttpError of int ;;
```

Note that we declare a value constructor `Timeout` that takes in no arguments while `UnboundVariable` and `HttpError` are value constructors that take in one argument.

5.4 Handling Exceptions

By default, when an exception is raised, the execution of the program stops. However, we might want to define what happens when an exception is raised. In `OCaml`, we can define code that *handles exceptions* by using the `try with` construct.

Syntax 5.7 (try-with)**.**

```
try
  e1
with
  ex1 -> e2
  ex2 -> e3
....
```

Here, we evaluate `e1`. If no exception is raised, `e1` terminates normally. Otherwise, we *catch* the exception that is raised and execute the expressions as defined by the match statement after the `with` keyword.

Example 5.8 (Division by Zero)**.** Here's how we can use exception handling to write a function `div_opt` that divides a number by another number.

Listing 17: div_opt

```
let div_opt (x : int) (y : int) : int option =
 try
    Some (x / y)
 with
    Division_by_zero -> None ;;
```

5.5 Options vs Exceptions

When should we use an exception vs an option? As **Section 10.4 of the textbook** explains, this is a design decision, so there is not always a right answer. However, the following are good principles to keep in mind:

1. When an anomaly is a **rare occurrence**, you should use an exception, as we saw with `max_list`. When the anomaly is a **frequent, standard part of computation**, we should use an option.
2. When an anomaly only occurs in a **small, localized part of the code**, we should use an **option**. If the anomaly is **ubiquitous** and **could occur an any part of the code**, we should use an exception.

Read the discussion in Section 10.4 of the textbook for further detail.

5.6 Makefiles

Makefiles allow us organize code compilation and tell us how to compile and link a program when using the command `make`.

To create a `makefile`, in the directory with our source files, we create a file named `makefile` or `Makefile`. Suppose we consider the `makefile` from Problem Set 1:

Listing 18: Makefile Example

```
all: ps1 ps1_tests

ps1: ps1.ml
        ocamlbuild -use-ocamlfind ps1.byte

ps1_tests: ps1_tests.ml
        ocamlbuild -use-ocamlfind ps1_tests.byte

clean:
        rm -rf _build *.byte
```

`makefile`s contain **rules**, which typically follow this syntax:

Listing 19: Rule Syntax

```
target: prerequisites
    recipe
```

The `target` is the keyword that we invoke with `make`, by running `make target`. The **prerequisites** are input files that the rule depends on, and are typically source files that we will compile such as `ps1.ml` and `ps1_tests.ml` in the `makefile` from Problem Set 1.

The file component of the rule, the **recipe**, are the instructions executed upon running `make target`. For example, running `make ps1` or `make ps1_tests` create executable files from the `ps1.ml` and `ps1_tests` source files.

After creating these executables, we can then run our code by running `./ps1.byte` and `./ps1_tests.byte`, respectively.

However, for two of the rules under Listing 22, no **prerequisites** are specified. The first rule `all: ps1 ps1_tests`, which is invoked by `make all`, runs both the `ps1` and `ps1_tests` rules. `make clean` will execute the **recipe**, which removes all files with the extension **.byte** from the directory.

6 Practice Problems

Problem 6.1 (Type Inference)**.** 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.[6]

1.
```
let f x =
  let a, b = x in
  b (a 5) ;;
```

2.
```
let rec f x a =
  match x with
 | [] -> a
 | h :: t -> h (f t a) ;;
```

3.
```
let rec f x =
  match x with
  | None
  | Some 0 -> None
  | Some y -> f (Some (y - 1)) ;;
```

4.
```
let f x y =
 if x then [x]
else [not x; y] ;;
```

Problem 6.2 (HOF Exercises)**.** Implement the following functions.

1. Write a function `remove_duplicates` that takes in a list and returns the duplicates from the list. Write a non-recursive solution. You may use `List.sort`. You may also assume that you have the function `to_run_length` (from problem set 1).

2. Write a function `tranpose` that takes a list of lists that represents a matrix (each list corresponds to a row) and returns the transpose of the matrix. For instance, the list `[[1;2;3]; [4;5;6]]` represents the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

   The **tranpose** of the matrix is defined to be the matrix such that the rows and columns are flipped. For instance, the transpose of the matrix above is:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

3. Write a polymorphic function `matrix_mult` that takes in a function to add elements two elements, a function to multiply elements, an initial value used to add and multiply elements (i.e. think of this as the zero element), and two matrices. The result should be the matrix formed by multiplying the two matrices together.

[6] Exercise 63 in the textbook provides more exercises

Given an $m \times n$ matrix A and an $n \times p$ matrix B, the matrix product $C = AB$ is an $m \times p$ matrix. If the two matrices that are being multiplied don't follow these dimensions, then they can't be multiplied together. For instance, a 3×5 can be multiplied with a 5×4 matrix. However, a 3×2 cannot be multiplied with a 3×4. In other words, the number of columns in A must be equal to the number of rows in B.

When we multiply two matrix, C_{ij} (the entry in the *ith* row and *jth* column) is defined to be the **dot product** of the *ith* row in A and *jth* column in B. So, suppose

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

Then, C is

$$C = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

To see how C is calculated, look at the element in the **first row** and **first column** of C, 58. To obtain 58, we take the dot product of the first row of A (`[1 2 3]` and the first column of B (`7 9 11`), which gives us:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 7 \\ 9 \\ 11 \end{bmatrix} = 1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 = 7 + 18 + 33 = 58$$

For this example, here's how our function should operate in OCaml.

Listing 20: Matrix Multiplication Example

```
(* Define A and B as lists of list *)
let a = [[1;2;3]; [4;5;6]] ;;
let b = [[7;8]; [9;10]; [11;12]] ;;

matrix_multiply (+) ( * ) 0 a b ;;

(*
matrix_multiply first takes in a function to add elements of the
   matrix. Since A and B are integer matrices, we can use the built
   -in OCaml add operator as an infix operator. Similarily, for the
    second argument, which is a function to multiply elements, we
   can use the multiply operator. The zero element of the set of
   integers is 0. Then we pass in A and B.
*)

(*
The result should C should be:

C = [[58; 64]; [139; 154]]
*)
```

(a) Create two functions: one function that multiplies integer matrices and another function that multiplies float matrices.

Problem 6.3 (Repeated Application)**.** Define a function `repeat f x n` that applies `f` to `x` exactly `n` times.

Problem 6.4 (Function Composition)**.** Define a function `compose : ('a -> 'a) list -> ('a -> 'a)` that takes a list of functions and returns the composition of those functions. For example, suppose we had a list of functions $[f; g; h]$. `compose` should return the function $f \circ g \circ h$ or $\ell(x) = f(g(h(x)))$.

Problem 6.5 (Option and Exception Conversion)**.** In this problem, the goal is to convert between functions that return option types and functions that raise exceptions.

1. Write a function `opt_to_ext : ('a -> 'b option) -> exn -> ('a -> 'b)` that takes in a function `f` of type `'a -> 'b option` and an exception. If the input function returns None for an argument, the output function should raise the exception passed into `opt_to_ext`. If the input function returns `Some v` on an argument, the output function should return `v` for that argument.
2. Write a function `ext_to_opt : ('a -> 'b) -> ('a -> 'b option)` that converts a function that (potentially) raises an exception to a function that returns a value having an option type. If the input function raises any exception on an input, the output function should return None. If the input function returns a value `v` for an input, the output function should return `Some v`.
3. In lab 4, we wrote the following functions:

Listing 21: Maximum of List

```
let rec max_list_opt (lst : int list) : int option =
  match lst with
  | [] -> None
  | head :: tail ->
     match (max_list_opt tail) with
     | None -> Some head
     | Some max_tail -> Some (max head max_tail) ;;

let rec max_list (lst : int list) : int =
  match lst with
  | [] -> raise (Invalid_argument "max_list: empty list")
  | [elt] -> elt
  | head :: tail -> max head (max_list tail) ;;
```

Write 4 unit tests that adequately test the behavior of `opt_to_ext` and `ext_to_opt` using the functions above. Specifically, make sure that:

(a) `opt_to_ext` returns a function that raises the appropriate exception.

(b) `ext_to_opt` returns a function that returns `None` on the proper input.

(c) The functions returned from `opt_to_ext` and `ext_to_opt` exhibit proper behavior when a non-`None` value or an exception isn't raised.