CS51 2024 Midterm 2 Answer Key

This answer key provides possible answers to the midterm, with discussion of their basis and alternatives. It is not intended to be an exhaustive discussion of the questions. There may be other answers that would get full credit, and not all issues that might affect grading are discussed.

Q2.1: True. Tail-recursive functions, and all recursive functions, require the `rec` keyword. On the other hand, there's a usage of "tail-recursive" by which a non-recursive function whose computation involves calling recursive functions may be deemed tail-recursive if the embedded calls are all to tail-rcursive functions. In such a case, no `rec` keyword is needed since the function is not dirctly recursive. We therefore gave full credit for both possible answers.

Q2.2: True. Type inference reconstructs the types, so that type annotations are not required. They may be helpful, however, for the purpose of expressing the programmer's intention to facilitate finding bugs.

Q2.3: False. Constructors declared in variant types can have arguments, which are specified using `of`. A simple example is the `color` type from lab 5.

```
type color =
  | Simple of color_label
  | RGB of int * int * int ;;
```

Both constructors `Simple` and `RGB` take an argument.

Q2.4: False. The value in a lazy expression such as `lazy (1 + 1)` is computed at its first use, and then cached so is not reevaluated thereafter. At the time the value is declared, the value remains unevaluated, and indeed may never be evaluated if the expression is never forced.

Q2.5: False. Although looping expressions *are* used solely for their side effects, like all OCaml expressions, they do return a value, namely the `unit` value `()`.

Q2.6: True. The reference `ref 42` has type `int ref`, and like all expressions cannot change its type.

```
# let r = ref 42 ;;
val r : int ref = {contents = 42}
# r := 21 ;;
- : unit = ()
# r := true ;;
Error: This expression has type bool but an expression was expected of type
         int
```

Q2.7: False. OCaml doesn't provide a specific type for streams. Rather, streams can be implemented as user code, as done in the `NativeLazyStreams` module reproduced in Q4 of this exam.

Q2.8: False. The substitution yields the expression `37 + 5`, not `42`.

Q2.9: False. The free variables in the expression are `f`, `y`, and `z`, and the FV definition would reflect that.

Q3.1: We wrap the condition and body in unit functions.

```
let rev xs =
  let xs = ref xs in
  let accum = ref [] in
  (while_ (fun () -> !xs <> [])
     (fun () ->
        accum := (List.hd !xs) :: !accum;
        xs := List.tl !xs));
  !accum ;;
```

Some submissions named the delayed condition and body:

```
let rev xs =
  let xs = ref xs in
  let accum = ref [] in
  let condition = fun () -> !xs <> [] in
  let body = fun () ->
               (accum := (List.hd !xs) :: !accum;
                xs := List.tl !xs) in
  while_ condition body;
  !accum ;;
```

That works too.

Q3.2: We replace the iteration of the while loop with recursion. Note the forcing of the condition and body by applying to `()`.

```
let rec while_ (condition : unit -> bool) (body : unit -> unit) : unit =
  if condition () then
    (body ();
     while_ condition body) ;;
```

It's a bit inelegant to have to keep providing the same two arguments in each recursive call. An alternative is to define an auxiliary function that can refers to the arguments in the outer `while_` definition.

```
let while_ (condition : unit -> bool) (body : unit -> unit) : unit =
  let rec while' () =
    if condition () then
      (body ();
       while' ()) in
  while' () ;;
```

In these implementations, the `if` doesn't need an `else`, as the conditional returns a `unit`, although adding `else ()` would still be a correct, if less idiomatic, implementation.

Q3.3: Implementing `while_` in terms of `while` merely requires forcing the condition and body when needed.

```
let while_ condition body =
  while condition () do
    body ()
```

```
  done ;;
```

Q3.4: It is not possible to implement the delaying needed in the `while_` function using OCaml's `Lazy` module. Because delayed values built using `lazy` are memoized, they are not evaluated each time they are forced. Thus, the condition will be evaluated only and exactly once. If it evalutes to `true`, on each later evaluation it will return `true`, and the loop will never end. Similarly, the body will only be evaluated once, not each time through the loop, so the intended side effects from the subsequent evaluations will not be generated.

We can test this by replacing the function delaying in the definition above.

```
let while_ condition body =
  while (sLazy.force condition) do
    Lazy.force body
  done ;;

let rev xs =
  let xs = ref xs in
  let accum = ref [] in
  (while_ (lazy (!xs <> []))
    (lazy
      (accum := (List.hd !xs) :: !accum;
       xs := List.tl !xs)));
  !accum ;;
```

Using `rev` with this `while_` definition, we end up in an infinite loop that requires an ungraceful exit.

```
# rev [1; 2; 3; 4] ;;
^CInterrupted.
#
```

Q4.1: There are multiple approaches that will work. Here is perhaps the simplest.

```
let rec stream_of x =
  lazy (Cons (x, stream_of x)) ;;
```

Another thought is to make use of `ones`, and multiply each element by `x`:

```
let rec ones = lazy (Cons (1, ones)) ;;
let stream_of x =
  smap (( * ) x) ones ;;
```

but this is not polymorphic.

Q4.2: Here, we start with 0 and then add `n` to the `every_nth` stream.

```
let rec every_nth (n : int) : int stream =
  lazy (Cons (0, smap ((+) n) (every_nth n))) ;;
```

The mapping approach that was insufficient for Q4.1 works here, since the solution is inherently monomorphic. We can multiply the `nats` stream by the argument `n`.

```
let rec nats =
  lazy (Cons (0, smap succ nats)) ;;
let every_nth (n : int) : int stream =
  smap (( * ) n) nats ;;
```

Q5.1:

```
let listarr = [ref 0; ref 5; ref 10] ;;
```

Q5.2:

```
let update (listarr : 'a ref list) (index : int) (new_value : 'a) : unit =
  (List.nth listarr index) := new_value ;;
```

We don't need to raise the exceptions, because, thankfully, `List.nth` does that for us, as depicted in the next problem (Q5.3).

Q5.3: The interpreter tells us the type of the function, namely, `'a list -> int -> 'a`:

```
# let rec nth lst index =
  if index < 0 then raise (Invalid_argument "nth")
  else
    match lst with
    | [] -> raise (Failure "nth")
    | hd :: tl -> if index = 0 then hd else nth tl (pred index) ;;
val nth : 'a list -> int -> 'a = <fun>
```

Q5.4: The recursion through the list will extend in the worst case to the very end of the list. The function is thus linear, $O(n)$.

Q5.5: For long enough sequences, implementing them with arrays will provide for more time-efficient indexing than implementing them with reference lists.

Q6.1: All of the information can be inherited from the `library_item_type` except for `get_runtime`, which we add to the class type.

```
class type dvd_type =
  object
    inherit library_item_type
    method get_runtime : int
  end ;;
```

Q6.2: Again, almost everything can be inherited from the `library_item` class. We need to add `get_runtime` and override `get_details`, which can augment the `get_details` result from the superclass.

```
class dvd (title : string) (id : string) (runtime : int) : dvd_type =
  object
    inherit library_item title id as super
    val runtime : int = runtime
    method get_runtime : int = runtime
    method! get_details : string =
      "DVD " ^ super#get_details ^ ", Runtime: " ^ (string_of_int runtime) ^ " minutes"
  end;;
```

Q6.3: Only one line needs to be added to the `library_item_type` class type to specify the `print_details` method.

```
class type library_item_type =
  object
    method get_title : string
    method get_id : string
    method get_details : string
    method print_details : unit
  end ;;
```

Q6.4: Only one line needs to be added to the `library_item` class to implement the `print_details` method, which just uses `get_details` to generate the details to be printed.

```
class library_item (title : string) (id : string) : library_item_type =
  object (this)
    val title : string = title
    val id : string = id
    method get_title : string = title
    method get_id : string = id
    method get_details : string = "Title: " ^ title ^ ", ID: " ^ id
    method print_details : unit = print_endline (this#get_details)
  end ;;
```

Q6.5: If the `library_item_type` code is modified as above, no changes are needed to the `dvd_type` code. It will continue to inherit the needed methods, including `print_details`.

Q6.6: If the `library_item` code is modified as above, no changes are needed to the `dvd` code. It will continue to inherit the needed methods, including `print_details`. That's the payoff of inheritance.

Q7.1:

```
let x = ref 42 in x
```

Q7.2:

```
let x = ref (fun y -> y) in x
```

Q7.3:

```
let f = fun y -> x + y in
let x = 42 in
x
```

Q7.4:

```
let x = ref (let x = 42 in fun y -> x + y) in x
```

or

```
let x = 42 in let x = ref (fun y -> x + y) in x
```