CS51 2024 Midterm 1 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.

Q3.1: The intended answer was `7`. The REPL verifies the answer.

```
# 6 * 7 ;;
- : int = 42
```

Q3.2. No such expression. (The `map` function always returns a list; `42` is not a list.)

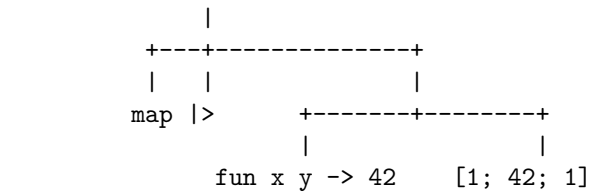
One approach that you might have considered is to try to find a way of throwing the `map` function (and even the `[1; 42; 1]` list) away – something along the lines of

```
|> (fun x y -> 42)
```

Sure enough, inserting this string into the blank, we get

```
# map |> (fun x y -> 42) [1; 42; 1] ;;
- : int = 42
```

But this approach doesn't satisfy the express requirements of the problem, which asks for "a single expression that is well-formed and well-typed in context". In the context of the expression evaluated above, the string `|> (fun x y -> 42)` is not a single expression. Rather, the full expression, including context, is parsed as indicated in the following abstract syntax tree:



There is no subtree that corresponds to `|> (fun x y -> 42)`. (Any subtree that includes both `|>` and `fun x y -> 42` also includes the other parts as well.)

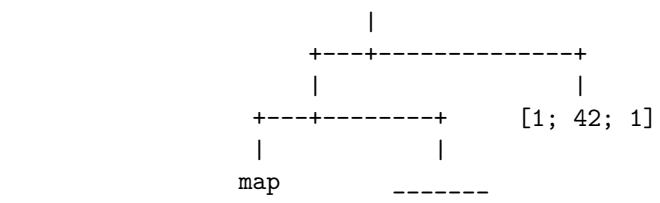
One way to tell the string is not being parsed as a single expression is by wrapping it in parentheses (thereby forcing interpretation as a single expression) and inserting it in the provided context:

```
# map (|> (fun x y -> 42)) [1; 42; 1] ;;
      ^   ^
Error: Syntax error: ')' expected, the highlighted '(' might be unmatched
```

A similar argument holds for this attested example:

```
(fun x -> x) [1;2;3] |> List.fold_left2 (fun acc el1 el2 -> acc) 42
```

In the end, for the blank to be filled with "a single expression that is well-formed and well-typed in context", the overall expression will need to be structured as



in which case the structure of the blank expression will need to be `int -> 'a` and the result will need to be of type `'a list` by virtue of the typing constraints from `map`. Since `42` is not an `'a list`, "No such expression" is the correct answer.

Q3.3: The intended answer was `y`:

```
# let x = 42 in
let y = 7 in
let y = x in
y ;;
Line 2, characters 4-5:
Warning 26 [unused-var]: unused variable y.
- : int = 42
```

The question states that the warning can be ignored.

Q3.4. Any function whose argument is consistent with `bool` and that returns any `int` will work, e.g., `fun _ -> 0`. The point of the question was to verify your ability to generate functions of a particular type.

```
# let x : bool -> int = fun _ -> 0 in 42 ;;
Line 1, characters 4-5:
Warning 26 [unused-var]: unused variable x.
- : int = 42
```

Q3.5. We want $x^2 - x = 42$. Solving this quadratic for x (just guessing works pretty well) gives -6 or 7, either of which is a correct answer.

```
let f x = x * x - x in f ~-6 ;;
- : int = 42
```

Q3.6. Any function that returns `40` on argument `42` will work here. For instance, `fun _ -> 40` or `fun x -> x - 2`.

```
# 42 |> fun _ -> 40 |> ((+) 2) ;;
- : int = 42
```

Q3.7. We need a pattern that extracts the `42` from `Some 42`. `Some x` is what's needed.

```
# let Some x = Some 42 in x ;;
Line 1, characters 0-25:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
None
Line 1, characters 0-25:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
```

```
Here is an example of a case that is not matched:
None
- : int = 42
```

There might be a concern that `Some x`, though a single expression, is not a *well-formed, well-typed* expression, and indeed it is not on its own:

```
# Some x ;;
Error: Unbound value x
```

But it is well-formed and well-typed *in context* as the problem specifies.

See Q3.2 for discussion of attempts at this problem that violate the "single expression" constraint, such as `x = 42 and _` or `x = 42 in let y`.

Q4.1. Once the function is defined in Q4.2, its type can be gleaned directly: `('a -> bool) -> 'a list -> 'a list`.

```
# findafter ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Q4.2. Here's a simple recursive definition. The requirement of an appropriate exception means that `Failure` (or perhaps `Not_found`) should be raised, and not, e.g., `Exit` or `Match_failure` (or even `Invalid_argument`, since it's the combination of the two arguments that leads to the problem; each argument separately is valid).

```
let rec findafter (condition : 'a -> bool)
                  (lst : 'a list)
                : 'a list =
  match lst with
  | [] -> failwith "findafter: not found"
  | hd :: tl ->
     if condition hd then lst else findafter condition tl ;;
```

Q4.3. The function can't return the empty list, as the first match case doesn't return a value and the second returns a nonempty list. So the shortest list that can be returned is of length 1.

Q4.4. The `mystery` function returns the empty list on the empty list, and on any non-empty list, it returns the list itself. In both cases, the original list is returned. So appropriate answers are

> It returns its list argument unchanged.
> It takes a list and returns the same list unchanged.
> It is the identity function on lists.

Minor infractions include not mentioning the limitation to lists.

> It is the identity function.
> It returns its argument unchanged.

Major infractions include poor explanations:

> It doesn't do anything.

or recapitulating the code as pseudocode:

> It returns the empty list if its argument is empty. Otherwise, it returns the tail of `x` that...

Q5.1. We just add a case for the uneven lists raising a `Failure` exception. Other exceptions are less appropriate (`Invalid_argument`) or inappropriate (`Exit`).

```
let rec zip (x : 'a list) (y : 'b list) : ('a * 'b) list =
  match x, y with
  | [], [] -> []
  | [], _ | _, [] -> failwith "zip: unequal lengths"
  | xhd :: xtl, yhd :: ytl -> (xhd, yhd) :: (zip xtl ytl) ;;
```

Q5.2. We just extend the recursive cases to add defaults once the empty tail is reached.

```
let rec zip_default default lst1 lst2 =
  match lst1, lst2 with
  | [], [] -> []
  | [], hd :: tl -> (default, hd) :: (zip_default default [] tl)
  | hd :: tl, [] -> (hd, default) :: (zip_default default tl [])
  | hd1 :: tl1, hd2 :: tl2 -> (hd1, hd2) :: (zip_default default tl1
  tl2)
```

Q5.3. We're looking for the following four steps in the argument: the default might be added to either list; thus the default must have the same type as each lists' elements; thus both lists must have the same type; `zip` has no such constraint. For example,

> In `zip_default`, since the same default value is used regardless of which list argument is shorter, both lists must have element types the same as the default value. In the `zip` function, no such restriction exists.

Q5.4.

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

Q5.5. This problem appeared on problem set 1 in a monomorphic version. The answer here should just relax the typing constraint to allow full polymorphism.

```
let rec unzip (lst : ('a * 'b) list) : 'a list * 'b list =
  match lst with
  | [] -> [], []
  | (left, right) :: tail ->
      let ltail, rtail = unzip tail in
      left :: ltail, right :: rtail ;;
```

It turns out that the `List` module already has this function as `split`, so the following was also possible (!):

```
let unzip = split ;;
```

Q6.1. The idea we're looking for is the existence of a prior, important type also named `unit`, that is, the type of the unit value `()`, and a desire to avoid losing access to it. An ideal answer would be

> Because it would shadow the existing `unit` type.

Q6.2.

```
type measure =
  | Count of int * ingredient
  | Measure of float * unit_ * ingredient ;;
```

The use of unabstracted types, using `string` instead of `ingredient`, is stylistically problematic.

Q6.3. Implementation is mostly an exercise in tedium to get the conversions straight.

```
let rec ounces (ingredients : measure list) : float =
  let ouncify measure =
    match measure with
    | Count _ -> 0.
    | Measure (amt, un, _) ->
       match un with
       | Teaspoon -> amt /. 6.
       | Tablespoon -> amt /. 2.
       | Cup -> amt *. 8.
       | Ounce -> amt in
  ingredients
  |> map ouncify
  |> fold_left (+.) 0. ;;
```

Q7.1. Any `int list` that contains an integer other than 0 or 1 suffices, for instance `[2]`.

Q7.2. The simple approach is to make use of `zip_default` and `unzip`, as the hint urges.

```
let lengthen_bits bits1 bits2 =
  unzip (zip_default 0 bits1 bits2)
```

We don't require typings in the header for this problem and for Q7.3 (though it's fine to provide them) because it is assumed that the explicit module type for the module should provide those typings. (See Q7.4.) For the same reason, the `Bits` module definition we provided does not contain typings.

Q7.3. The obvious `not_` function implementation is

```
let not_ bits =
  map (fun x -> 1 - x) bits
```

which can be simplified by partial application

```
let not_ =
  map (fun x -> 1 - x)
```

or even the too-clever-by-half

```
let not_ =
  map ((-) 1) ;;
```

Q7.4. The contents of the module type `BITS` should contain the type (without implementation), and typed values for each function *except for* ***`lengthen_bits`***, which is explicitly described as "an auxiliary function that isn't of use outside the module."

```
type t
val bits_of_int : int -> t
val int_of_bits : t -> int
val xor_ : t -> t -> t
val and_ : t -> t -> t
val or_ : t -> t -> t
val not_ : t -> t
val serialize : t -> string
```

Typical errors would be things like

```
type t = int list
```

or

```
val bits_of_int : int -> int list
```

or any mention of `int lists` actually.

Q7.5. The module just needs to be "typed" as satisfying `BITS`

```
module Bits : BITS =
```

No use of sharing constraints is desirable, or even possible. Adding a `with` that directly exposes the implementation of the type `t`, e.g.,

```
module Bits : (BITS with type t = int list) =
```

violates the whole point of constraining the `Bits` module with the module signature, which is to hide the implementation type from users of the module, that is, to generate an *abstract* data type.

Q7.6. Here was our intended answer:

> No. Since the type `Bits.t` is not exposed by the module signature, the only way to create a value of type `Bits.t` is via the functions in the module, all of which preserve the invariant.

However, it turns out that the inadequate implementation we provided of `Bits.bits_of_int` inadvertently allows for breaking the invariant. (Oops.)

We can see that by using the unconstrained `Bits` module:

```
# Bits.bits_of_int ~-1 ;;
- : int list = (::) (-1, [])
```

So we give full credit also for a "Yes" *when accompanied by this argument.*