Solutions for the CS51 first midterm of 2021

Q1: Box should be checked.

Q2.1:

```
let add_three : int -> int = (+) 3
```

or

```
let add_three (n : int) : int = n + 3
```

Q2.2:

```
let rec power (base : int) (expt : int) : int =
  if expt = 0 then 1
  else base * power base (pred expt) ;;
```

Q2.3:

```
let power_uncurried (base, expt : int * int) : int =
  power base expt
```

or

```
let rec power_uncurried (base, expt : int * int) : int =
  if expt = 0 then 1
  else base * power_uncurried (base, pred expt) ;;
```

Q3.1-5:

We asked just for the types of the expressions, but here we give some discussion of how we got the answers. There was no need to provide this explanation, and in fact, doing so would likely be graded down for "excessive verbiage".

Notice that the definition of the function `f` involves a match that incorporates new local variables `f` and `x` that are different from the `f` and `x` in the header line.

a. We start with the type of the local variable `f` in line 3. It's the first argument of the `Base` constructor, hence of type `'a -> 'a`. It is applied to `a` in the value of that match case, so `a` must be of type `a`, and `f a` is also of type `'a`. Thus the answer for (a) is `'a`.

b. Since `f` is, again, the first argument of the `Base` constructor, it must be of type `'a -> 'a`.

c. We now turn to the variable `f` being defined in line `1`. It is applied to `r` and then `g a` in line 5, and `r` is the second argument of the `Combine` constructor, hence of type `'a combine`. Thus `f` must be of type `'a combine -> ...`. Since `g` is the first argument of the `Combine` constructor, it is of type `'a -> 'a`, and `g a` must be of type 'a. So `f` must be of type `'a combine -> 'a -> 'a`.

d. Since `f` is of type `'a combine -> 'a -> 'a` and `r` is of type `'a combine`, `f r` is of type `'a -> 'a`.

e. As argued above, `g a` is of type `'a`.

Q4.1:

```
[42]
```

Q4.2: No such expression

Q4.3:

```
fun _ -> 42
```

Q4.4:

```
{first = 6; second = true}
```

or

```
{first = ~-7; second = true}
```

Q5.1:

```
let tower (lst : int list) : int =
  List.fold_right power lst 1
```

Q5.2:

```
let find (p: int -> bool) (lst: int list) : int =
  match List.filter p lst with
  | [] -> raise Not_found
  | first :: _ -> first ;;
```

Q6.1:

```
type circuit =
  | Single of float
  | Series of circuit * circuit
  | Parallel of circuit * circuit ;;
```

Q6.2:

```
let circ_a : circuit = Single 3. ;;
```

Q6.3:

```
let circ_c : circuit =
  Parallel (Single 2., Single 4.) ;;
```

Q6.4:

```
let circ_d : circuit =
  Series (Parallel (Single 4., circ_c),
          Single 1.) ;;
```

Q6.5:

```
let rec resistance (circ : circuit) : float =
  match circ with
  | Single v -> v
  | Series (c1, c2) -> resistance c1 +. resistance c2
  | Parallel (c1, c2) ->
```

```
      1. /. (1. /. (resistance c1)
             +. 1. /. (resistance c2)) ;;
```

Q7.1:

```
SEQUENCE with type t = int
```

Q7.2:

```
int
```

Q7.3:

```
sequence_from 0 length
```

Q7.4:

```
SEQUENCE with type t = float
```

Q7.5:

```
float
```

Q7.6:

```
(from /. 2.)
```

Q7.7

```
sequence_from 1. length
```

Q7.8:

```
module type ELEMENT =
  sig
    type t
    val initial : t
    val next : t -> t
  end ;;
```

Q7.9:

```
ELEMENT
```

Q7.10:

```
SEQUENCE with type t = Element.t
```

Q7.11:

```
Element.t
```

Q7.12:

```
from :: sequence_from (Element.next from) (length - 1)
```

Q7.13:

```
sequence_from Element.initial length
```

Q7.14:

```
type t = int
let initial = 0
let next = succ
```

Q7.15:

```
module Diminishing =
  Sequence (struct
              type t = float
              let initial = 1.
              let next x = x /. 2.
            end)
```