Comp302: Programming Languages and Paradigms

Follow-up on Midterm Review Session

1 Syntactic Sugar

Some were confused by the syntax I used during the review session so here is a reminder that the following are equivalent:

```
let rec botof k = match k with
  | Bottom b -> b
  | Onion r -> botof r
  | Tomato r -> botof r
  | Lamb r -> botof r
```

=

```
let rec botof = function
  | Bottom b -> b
  | Onion r -> botof r
  | Tomato r -> botof r
  | Lamb r -> botof r
```

The `function` keyword just waits for some inputs and directly pattern matches on it. This is useful syntax to know, but you can always get away with just writing `match with` if you find it confusing.

2 Tail-recursion

For writing tail-recursive function that are not straight-forwardly tail-recursive, we often use an *accumulator* to store up some construction (or computation) that will be unrolled (or executed) later. However, just sticking an accumulator into a function is not always sufficient. For example:

```
let tr_tlwo k =
  let rec f acc keb = match keb with
    | Bottom b -> acc
    | Onion k -> f (Onion (acc)) k
    | Lamb k -> f (Onion (Lamb (acc))) k
    | Tomato k -> f (Tomato (acc)) k
  in
  f (Bottom b) k *)
```

The above definition has two problems: 1) it doesn't know what bottom to start the accumulator with and 2) it returns a kebab upside down (first topping last; last topping first). This is because kebabs are built from the bottom up but they read (destroyed) from the top down.

The solution is to go through the kebab twice: once (with the helper function `f`) transfering the kebab on an arbitrary bottom (here we use an int) and twice (with `rev`) to construct the kebab on the right bottom and to put it right side up again:

```
(* val tr_tlwo : 'a kebab -> 'a kebab = <fun> *)
  let tr_tlwo k =
  let rec rev k = function
    | Bottom _ -> k       (* Q: Which k is this? Why? *)
    | Onion acc' -> rev (Onion k) acc'
    | Lamb acc' -> rev (Lamb k) acc'
    | Tomato acc' -> rev (Tomato k) acc'
  in
```

```
let rec f acc = function
  | Bottom b -> rev (Bottom b) acc
  | Onion k -> f (Onion (acc)) k
  | Lamb k -> f (Lamb (Onion (acc))) k
  | Tomato k -> f (Tomato (acc)) k
in
f (Bottom 0) k
```

This is not the only (or most elegant) way to make `tlwo` tail-recursive. For example, notice that the functions `f` and `rev` are very similar in structure, so one could imagine combining them and using a argument to navigate between the specifics of the two.

3 Options

Here is a concrete example of where options are useful.

Intruction: Take an `rod kebab` and return the `Skewer` at the bottom of a kebab
NOTE: The kebab in question may not have a `Skewer` at the bottom (it could also have a `Dagger` or a `Sword`)

Solution:

```
(* val return_skew : rod kebab -> rod option = <fun> *)
let rec return_skew = function
  | Onion k -> return_skew k
  | Lamb k -> return_skew k
  | Tomato k -> return_skew k
  | Bottom Skewer -> Some Skewer
  | Bottom _ -> None
```