0.5 What is CPS?

0.5.1 Tail Calls

In functional programming, we very frequently have functions which call other functions. For example, the function `foo` below makes two calls to other functions: a recursive call to itself and a call to some other function `bar`:

```
fun bar (x,y) = x * y
fun foo 0 = 1
  | foo n = bar(n,foo(n-1))
```

We make the distinction between *tail-calls* and non-tail-calls, based on whether the result of the recursive call is modified or not. If the evaluation of a function `f` makes a call to a function `g`, we say that it is a **tail call** if `f` does not modify or examine the result of evaluating `g`.

In the example above, the recursive call of `foo` to itself is *not* a tail-call, because the evaluation of `foo(n)` will take the result of calling `foo(n-1)` and ***then*** do something with that value (namely pass it as an argument to the function `bar`). By contrast, when `foo` calls `bar`, it *is* considered a tail-call, because `foo(n)` does not modify the value returned by the call to `bar`.

We say that a function is **tail recursive** if every recursive call it makes is a tail-call.

0.5.2 Continuations

It is a useful (and perhaps surprising) fact that *every* SML function can be written in tail recursive form. In order to achieve this, we make use of *continuations*.

Continuations are a way to require *at a type level* that our function is tail recursive. In this class, we implement continuations as additional function arguments. For example,

```
(*  fact : int -> (int -> 'a) -> 'a
    REQUIRES: n>=0
        ENSURES: fact n s ==> s(n!)
 *)
fun fact 0 s = s 1
  | fact n s = fact (n-1) (fn res => s(n*res))
```

First of all, convince yourself that this function meets its spec, and additionally is tail recursive. Furthermore, notice that the *type* forces it to be tail-recursive: suppose the return type of `s` is `t`. If `fact` makes a recursive call to itself, then the result of that recursive call will also have type `t`. But `t` could be anything, so there's no way for us to modify or case on the output of this recursive call.

So, for the purposes of this class (and in particular for this homework), think of continuations as functions which we pass in as arguments and which get applied to the result of our computation. The function `s` in the above example is a continuation, because it is passed in as an argument to `fact`, and the value of `fact n s` is the function `s` applied to `n!`. Most of the continuations you'll see in this class will have polymorphic return type.

0.5.3 CPS

For the purposes of this homework, we'll say that a function `f` is in ***continuation passing style*** if:

- `f` takes (at least one) continuation as an argument
- If `f` makes a call to a function `g` which itself has continuation(s), then this call is a tail call
- `f` only calls its continuation(s) in a tail call (e.g. `f` does not call its continuation and then try to do something with the result)

In particular, this requires that `f` is tail recursive: if `f` made a recursive call to itself which was not a tail-call, then `f` would violate the second condition above.

0.5.4 Guidelines

You **may** do the following in a CPS function:

- Case on the value of the input
- Use `let..in..end` expressions
- Write recursive functions

You **may not** do the following in a CPS function:

- Manipulate, case on, or otherwise use the result of recursive calls (this would break the tail-call requirement)
- Manipulate, case on, or otherwise use the result of a call to a CPS helper function
- Manipulate, case on, or otherwise use the result of a call to the continuation.

In the appendix, we provide some examples of code which "appears" to be written in CPS but which actually violates the specification given above.

A Appendix: Is this CPS?

Below, we implement some examples in both a "pseudo-CPS" style and in CPS. Note that all of the "bad" examples would earn 0 points on this homework (because they are not correct CPS). Please understand the distinction and ask us if you have any questions.

A.1 fact

We want to implement the factorial function in CPS. The following is **not** CPS:

```
fun fact_pseudoCPS 0 s = s 1
  | fact_pseudoCPS n s =
      let
        val res = fact_pseudoCPS (n-1) (fn x => x)
      in
        s(n*res)
      end
```

This is not CPS because it makes a recursive call to itself, and then does arithmetic with the result. Therefore, the recursive call is not a tail call, so this is not CPS. Instead, we should do something like the following.

```
fun factCPS 0 s = s 1
  | factCPS n s = factCPS (n-1) (fn res => s(n*res))
```

A.2 size

We want a function which calculates the size (number of nodes) of a tree. The following is **not** CPS:

```
fun size_pseudoCPS Empty s = s 0
  | size_pseudoCPS (Node(L,x,R)) s = 1 + (size_pseudoCPS L s) + (size_pseudoCPS R s)
```

This would not typecheck if the return type of `s` is not `int`, and also it is not CPS because we perform arithmetic on the result of the recursive call. Instead, we should do something like:

```
fun sizeCPS Empty s = s 0
  | sizeCPS (Node(L,x,R)) s =
      sizeCPS L (fn sizeL => sizeCPS R (fn sizeR => s(1+sizeL+sizeR)))
```

A.3 search

We want a function which searches through a tree for an element which satisfies a given predicate. The following is **not** CPS:

```
fun search' p Empty s k = k ()
  | search' p (Node(L,x,R)) s k =
      if p x then s x else
        let
          fun fail () = NONE
          val (searchLeft, searchRight) = (search p L SOME fail, search p R SOME fail)
        in
          (case (searchLeft, searchRight) of
            (SOME x, _) => s x
          | (NONE, SOME y) => s y
          | (NONE, NONE) => k ()  )
        end
```

This is not CPS because it examines the results of recursive calls and cases on them. Instead, we could do:

```
fun search p Empty s k = k ()
  | search p (Node(L,x,R)) s k =
        if p x then s x else search p L s (fn ()=> search p R s k)
```

This is CPS. Note that we make a non-tail-call to the predicate `p`. This is okay (it's still CPS) because `p` doesn't take continuations so we don't have to call it in a tail call. Note that it's also okay to write search as:

```
fun search p Empty s k = k ()
  | search p (Node(L,x,R)) s k = if p(x) then s x else
      let
        fun failure () = search p R s k
      in
        search L s failure
      end
```

This is still CPS because the call to `search L` is a tail-call, and `failure` is not called except as a tail-call if `search L` fails.