Structure of Programming Languages, Assignment 5
Due: Friday of Week 10

Ensure you understand the course policies for assignments.

This homework is required for all. You may optionally work in pairs.

Continuation Passing Style

The problems below are all trivial or straightforward recursive problems. You must first write each of the functions in this part of the assignment in direct style (according to the problem specification), then transform the function definition into continuation-passing style. Several examples of direct and CPS style were given in lecture (e.g., Factorial). In the first section, you will familiarize yourself with the basics of CPS transformations: returning values, creating continuations, and linearizing computations.

1. Write the following low-level functions in continuation-passing style. You do not have to write direct style functions for this problem. A description of what each function should do follows:

   - `addk` adds two integers
     `val addk : int -> int -> (int -> a) -> a = <fun>`
   - `subk` subtracts one integer from another
     `val subk : int -> int -> (int -> a) -> a = <fun>`
   - `timesk` multiplies two integers
     `val timesk : int -> int -> (int -> a) -> a = <fun>`
   - `plusk` adds two floats
     `val plusk : float -> float -> (float -> a) -> a = <fun>`
   - `take_awayk` subtracts one float from another
     `val take_awayk : float -> float -> (float -> a) -> a = <fun>`
   - m̂ultk multiplies two floats
     `val multk : float -> float -> (float -> a) -> a = <fun>`
   - `catk` concatenates two strings; `val catk : string -> string -> (string -> a) -> a = <fun>`
   - `consk` creates a new list by adding an element onto the front of a list
     `val consk : a -> a list -> (a list -> b) -> b = <fun>`
   - `lessk` determines if one argument is less than another
     `val lessk : a -> a -> (bool -> b) -> b = <fun>`
   - `eqk` tests if two arguments are equal
     `val eqk : a -> a -> (bool -> b) -> b = <fun>`

   You can use the following report function to test your functions, e.g., `addk 3 4 report`.

```
let report x =
print_string "Result: ";
print_int x;
print_newline();;
val report : int -> unit = <fun>
```

2. Nesting continuations We wish to add three numbers, but `addk` itself only adds two numbers. We could define add3k as follows:

```
et add3k a b c k =
     addk a b (fun ab -> addk ab c k);;
val add3k : int -> int -> int -> (int -> a) -> a = <fun>
# add3k 1 2 3 report;;
 Result: 6
 - : unit = ()
```

On line 2 we give the first call to `addk` a function that saves the sum of `a` and `b` in the variable `ab`. Then this function adds `ab` to `c` and passes its result to the continuation `k`.

Using `multk` and `plusk` as helper functions, write a function `abcdk`, which takes four float arguments a b c d and "returns" $(a + (b \times c)) + (d \times a)$. You may only use the `multk` and `plusk` operators to do the arithmetic. The order of evaluation of operations must be as indicated by the parentheses in the given formula. Where there are ambiguities, evaluate the expression on the right first.

```
# let abcdk a b c d k = ...
val abcdk : float -> float -> float -> float -> (float -> a) -> a = <fun>
# abcdk 2.0 3.0 4.0 5.0 (fun y -> report (int_of_float y));;
Result: 24
 - : unit = ()
```

3. Transforming recursive functions. Consider the factorial example from class.

```
# let rec factorial n =
if n = 0 then 1 else n * factorial (n - 1);;
val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
```

We can rewrite this making each step of the computation explicit:

```
# let rec factoriale n =
       let b = n = 0 in
           if b then 1
           else let s = n - 1 in
                let m = factoriale s in
                n * m;;
val factoriale : int -> int = <fun>
# factoriale 5;;
- : int = 120
```

To put the function into full CPS, we must make factorial take an additional argument, a continuation, to which the result of the factorial function should be passed. When the recursive call is made to factorial, instead of it returning a result to build the next higher factorial, it needs to take a continuation for building that next value from its result. In addition, each intermediate computation must be converted so that it also takes a continuation. We use the functions defined in Problem 1 and the code becomes:

```
# let rec factorialk n k =
       eqk n 0
       (fun b -> if b then k 1
                 else subk n 1
                     (fun s -> factorialk s
                               (fun m  -> timesk n m k)));;
# factorialk 5 report;;
Result: 120
- : unit = ()
```

Note that to make a recursive call, we needed to build an intermediate continuation capturing all the work that must be done after the recursive call returns and before we can return the final result. If `m` is the result of the recursive call in direct style (without continuations), then we need to build a continuation to:

- take the recursive value: `m`
- build to the final result: `n * m`
- pass it to the final continuation `k`

This is an extension of the "nested continuation" method.

In this problem, you are asked to first write a function in direct style and then manually transform the code into continuation-passing style. When writing functions in continuation-passing style, all uses of functions need to take a continuation as an argument. All uses of primitive operations (e.g. +, -, *, ¿, =) should use the corresponding functions defined in Problem 1. If you need to make use of primitive operations not covered in Problem 1, you should include a definition of the corresponding version that takes a continuation as an additional argument, as in Problem 1.

For each problem in this section, **first write the function as described in direct style**. Then, write the function again in continuation-passing style; append `k` to the name of the second definition. For example, if a problem asks you to write a function `splat`, then you should define `splat` in direct style and `splatk` in continuation-passing style.

(a) Factorial range

i. Write the function `fact_range`, which takes an integer n, multiplies it to all integers less than n down to m (inclusive), and returns the result. If $n < m$, then return 1.

```
# let rec fact_range n m = ...;;
val fact_range : int -> int -> int = <fun>
# fact_range 5 1;;
- : int = 120
```

ii. Write the function `fact_rangek : int -> int -> (int -> a) -> a` that is the CPS transformation of `fact_range` defined in part a.i.

```
# let rec fact_rangek n m k = ...;;
val fact_rangek : int -> int -> (int -> a) -> a = <fun>
# fact_rangek 7 5 report;;
Result: 210
- : unit = ()
```

4. Apply function to list.

(a) Write the function `app_all : (a -> b) list -> a -> b list` that takes a list of functions `flst` and a value `x` and creates the list made by applying each function in `flst` to `x`. The functions should be applied in the order in which they occur in the list, and the results list should be the order corresponding to the function list. You should assume OCaml order of evaluation.

```
# let rec app_all flst x = ...;;
val app_all : (a -> b) list -> a -> b list = <fun>
# app_all [((+) 1); (fun x -> x * x); (fun x -> 13)] 5;;
- : int list = [6; 25; 13]
```

(b) Write the function `app_allk : (a -> (b -> c) -> c) list -> a -> (b list -> c) -> c` that is the CPS transformation of the code you wrote in part i. Your definition of `app_allk` must assume the functions in the input list will also be in continuation-passing style; that is, the type of the list is not `(a -> b) list`, but `(a -> (b -> c) -> c) list`.

```
# let rec app_allk flstk x k = ...;
val app_allk : (a -> (b -> c) -> c) list -> a -> (b list -> c) -> c = <fun>
# app_allk [(addk 1); (fun x -> timesk x x); (fun x -> (fun k -> k 13))]
5 (fun x -> x) ;;
- : int list = [6; 25; 13]
```

5. Using continuations to alter control flow. As we have seen in the previous sections, continuations allow us a way of explicitly stating the order of events, and in particular, what happens next. We can use this ability to increase our flexibility over the control of the flow of execution (referred to as control flow). If we build and keep at our access several different continuations, then we have the ability to choose among them which one to use in moving forward, thereby altering our flow of execution. You are all familiar with using an if-then-else as a control flow construct to enable the program to dynamically choose between two different execution paths going forward.

   Another useful control flow construct is that of raising and handling exceptions. In class, we gave an example of how we can use continuations to abandon the current execution path and roll back to an earlier point to continue with a different path of execution from that point. This method involves keeping track of two continuations at the same time: a primary one that handles "normal control flow, and one that remembers the point to roll back to when an exceptional case turns up. As in regular continuation-passing style, the primary continuation should be continuously updated; however, the exception continuation remains the same. The exception continuation is then passed the control flow (by being called) when an exceptional state comes up, and the primary continuation is used otherwise.

   Write the function sum `wholesk` that takes a list of integers, a regular continuation, and an exception continuation, and sums all the numbers in the list. If any of the integers is negative, call the exception continuation and pass the offending value to it. Your definition must be in continuation-passing style, and must follow the same restrictions about calling primitives as the previous sections problems. (For this problem, you will receive no points for the direct style definition of sum wholes, though writing it will be helpful for you to convert it to continuation-passing style.)

```
# let rec sum_wholesk l k xk = ...;;
val sum_wholesk : int list -> (int -> a) -> (int -> a) -> a

# sum_wholesk
[0; -1; 2; 3]
report
(fun i ->
  print_string ("Error: " ^  (string_of_int i) ^  " is not a whole number");
  print_newline());;
Error: -1 is not a whole number
- : unit = ()
```

Turn in: OCaml code in file `hw5/hw5.ml` in your bitbucket repository.