CS450: Progrmg Languages & Compilers
Fall 2014

Homework 3

University of Illinois, Urbana-Champaign,

Due: June 21, 2015, 5.00 pm. Submit PDF + code files on Moodle.

1 Theoretical Questions (40 points)

Objectives

The purpose of this HW is to test your understanding of

- The basic CSP transformation algorithm,
- How to use a formal mathematically recursive definition,
- test your understanding of the algebraic datatype system in OCaml.

What to Turn In

Your answers to the following questions are to be submitted electronically via the Moodle. You should type up your answers to each of the problems below and submit them in a file called `hw3.pdf`. You can use LaTeX or any other wordprocessor of your choice to type up the solutions but we can only grade PDF files. (You can even write your answers by hand legibly. You should then make a reasonable effort to make a readable scan of it, and submit those documents as PDF file; please don't submit raw image files.)

We shall only grade PDF submissions. We will ask you to resubmit solutions in any other file format.

Background and Definitions

Throughout this HW, we will be working with a (very) simple functional language. It is a fragment of MicroML (a frgament of SML, an OCaml-like language), and the seed of the language that we will be using in MPs throughout the rest of this semester. Using a mix of MicroML concrete syntax (expression constructs as you would type them in MicroML's top-level loop) and recursive mathematical functions, we will describe below the algorithm for CSP transformation for this fragment. You should compare this formal definition with the description given on examples in class.

The language fragment we will work with in this assignment is given by the following Context Free Grammar:

$$\begin{aligned} e \rightarrow\ & c \mid v \mid e \oplus e \\ & \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\ & \mid \texttt{fn}\ v\ \texttt{=>e} \mid e\ e \end{aligned}$$

The symbol e ranges recursively over all expressions in MicroML, c ranges over constants, v ranges over program variables, and $\oplus$ ranges over infixed binary primitive operations. The MicroML

construct `fn` v`=>` e corresponds the OCaml's construct `fun` v`->` e. This language will expand over the course of the semester.

Mathematically we represent CPS transformation by the functions $[[e]]_\kappa$, which calculates the CPS form of an expression e when passed the continuation κ. The symbol κ does not represent a programming language variable, but rather a complex expression describing the current continuation for e.

The defining equations of this function are given below. Recall that when transforming a function into CPS, it is necessary to expand the arguments to the function to include one that is for passing the continuation to it. We will use κ to represent a continuation that has already been calculated or given to us, and k, k_i, k' etc as the name of variables that can be assigned continuations. We will use v, v_i, v' etc for the ordinary variables in our program.

By v being fresh for an expression e, we mean that v needs to be some variable that is NOT in e. In MP5, you will implement a function for selecting one, but here you are free to choose a name as you please, subject to being different from all the other names that have already been used.

- The CPS transformation of a variable or constant expression just applies the continuation to the variable or constant, since during execution, when this point in the code is reached, both variables and constants are already fully evaluated (except for being looked up).

$$[[v]]_\kappa = \kappa\ v$$
$$[[c]]_\kappa = \kappa\ c$$

  Example:

$$[[\texttt{x}]]_{\texttt{(FUN y -> report y)}} = \texttt{(FUN y -> report y) x}$$

  This may be read as "load register `y` with the value for `x`, then do a procedure call to `report`".

- The CPS transformation for a binary operator mirrors its evaluation order. It first evaluates its first argument then its second before evaluating the binary operator applied to those two values. We create a new continuation that takes the result of the first argument, e_1, binds it to v_1 then evaluates the second argument, e_2, and binds that result to v_2. As a last step it applies the current continuation to the result of $v_1 \oplus v_2$. This is formalized in the following rule:

$$[[e_1 \oplus e_2]]_\kappa = [[e_1]]_{\texttt{FUN}\ v_1\ \texttt{->}\ [[e_2]]_{\texttt{FUN}\ v_2\ \texttt{->}\ \kappa\ (v_1 \oplus v_2)}} \quad \text{Where } \begin{array}{l} v_1 \text{ is fresh for } e_1,\ e_2, \text{ and } \kappa \\ v_2 \text{ is fresh for } e_1,\ e_2,\ \kappa, \text{ and } v_1 \end{array}$$

  Example:

$$\begin{aligned} &[[\texttt{x + 1}]]_{\texttt{(FUN w -> report w)}} \\ &= [[\texttt{x}]]_{\texttt{(FUN y -> }[[\texttt{1}]]_{\texttt{(FUN z -> (FUN w -> report w) (y + z))}}\texttt{)}} \\ &= [[\texttt{x}]]_{\texttt{(FUN y -> ((FUN z -> (FUN w -> report w) (y + z)) 1))}} \\ &= \texttt{(FUN y -> ((FUN z -> (FUN w -> report w) (y + z)) 1)) x} \end{aligned}$$

- Each CPS transformation should make explicit the order of evaluation of each subexpression. For if-then-else expressions, the first thing to be done is to evaluate the boolean guard. The

resulting boolean value needs to be passed to an if-then-else that will choose a branch. When the boolean value is true, we need to evaluate the transformed then-branch, which will pass its value to the final continuation for the if-then-else expression. Similarly, when the boolean value is false we need to evaluate the transformed else-branch, which will pass its value to the final continuation for the if-then-else expression. To accomplish this, we recursively CPS-transform the boolean guard e_1 with the continuation with a formal parameter v that is fresh for the then branch e_2, the else branch e_3 and the final continuation κ, where, based on the value of v, the continuation chooses either the CPS-transform of e_2 with the original continuation κ, or the CPS-transform of e_3, again with the original continuation κ.

$$[[\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]]\kappa = [[e_1]]_{\texttt{FUN } v \texttt{ -> IF } v \texttt{ THEN } [[e_2]]\kappa \texttt{ ELSE } [[e_3]]\kappa}$$

Where v is fresh for e_2, e_3, and κ

With $\texttt{FUN } v \texttt{ -> IF } v \texttt{ THEN } [[e_2]]\kappa \texttt{ ELSE } [[e_3]]\kappa$ we are creating a new continutation from our old. This is not a function at the level of expressions, but rather at the level of continuations, hence the use of a different, albeit related, syntax.

Example:

$$\begin{aligned}
&[[\texttt{if x > 0 then 2 else 3}]]_{\texttt{(FUN w -> report w)}} \\
&= [[\texttt{x > 0}]]_{\texttt{(FUN y -> IF y THEN } [[\texttt{2}]]_{\texttt{(FUN w -> report w) ELSE } [[\texttt{3}]]_{\texttt{(FUN w -> report w))}}} \\
&= \texttt{(FUN y -> IF y THEN } [[\texttt{2}]]_{\texttt{(FUN w -> report w) ELSE } [[\texttt{3}]]_{\texttt{(FUN w -> report w))}}}\texttt{ (x > 0)} \\
&= \texttt{(FUN y -> IF y THEN ((FUN w -> report w) 2) ELSE ((FUN w -> report w) 3)) (x > 0)}
\end{aligned}$$

- A function expression by itself does not get evaluated (well, it gets turned into a closure), so it needs to be handed to the continuation directly, except that, when it eventually gets applied, it will need to additionally take a continuation as another argument, and its body will need to have been transformed with respect to this additional argument. Therefore, we need to choose a new continuation variable k to be the formal parameter for passing a continuation into the function. Then, we need to transform the body with k as its continuation, and put it inside a continuation function with the same original formal parameter together with k. The original continuation κ is then applied to the result.

$$[[\texttt{fn } x \texttt{ => } e]]\kappa = \kappa\ (\texttt{FN } x\ k \texttt{ -> } [[e]]_k) \qquad \text{Where } k \text{ is new (fresh for } \kappa)$$

Notice that we are not yet evaluating anything, so $(\texttt{FN } x\ k \texttt{ -> } [[e]]_k)$ is a CPS function expression, not actually a closure.

Example:

$$\begin{aligned}
&[[\texttt{fn x => x + 1}]]_{\texttt{(FUN w -> report w)}} \\
&= \texttt{(FUN w -> report w) (FN x k -> (FUN y -> ((FUN z -> k (y + z)) 1)) x)}
\end{aligned}$$

- The CPS transformation for application mirrors its evaluation order. In MicroML, we will uniformly use left-to-right evaluation. Therefore, to evaluate an application, first evaluate the function, e_1, to a closure, then evaluate e_2 to a value to which that closure is applied. We

create a new continuation that takes the result of e_1 and binds it to v_1, then evaluates e_2 and binds it to v_2. Finally, v_1 is applied to v_2 and, since the CPS transformation makes all functions take a continuation, it is also applied to the current continuation κ. This rule is formalized by:

$$[[e_1\ e_2]]_\kappa = [[e_1]]_{\texttt{(FUN } v_1 \texttt{ -> } [[e_2]]_{\texttt{(FUN } v_2 \texttt{ -> } v_1\ v_2\ \kappa\texttt{)}}\texttt{)}} \quad \text{Where } \begin{matrix} v_1 \text{ is fresh for } e_2 \text{ and } \kappa \\ v_2 \text{ is fresh for } v_1 \text{ and } \kappa \end{matrix}$$

Example:

$$\begin{aligned}
&[[\texttt{(fn x => x + 1) 2}]]_{\texttt{(FUN w -> report w)}} \\
&= [[\texttt{(fn x => x + 1)}]]_{\texttt{(FUN y -> } [[\texttt{2}]]_{\texttt{(FUN z -> y z (FUN w -> report w))}}\texttt{)}} \\
&= \texttt{(FUN y -> } [[\texttt{2}]]_{\texttt{(FUN z -> y z (FUN w -> report w))}}\texttt{)} \\
&\quad \texttt{(FN x k -> (FUN a -> ((FUN b -> k (a + b)) 1)) x)} \\
&= \texttt{(FUN y -> ((FUN z -> y z (FUN w -> report w)) 2))} \\
&\quad \texttt{(FN x k -> (FUN a -> ((FUN b -> k (a + b)) 1)) x)}
\end{aligned}$$

Problems

Problem 1: (30 points)

Compute the following CPS transformation. All parts should be transformed completely.

$$[[\texttt{fn f => fn x => if x > 0 then f x else f ((-1) * x)}]]_{\texttt{(FUN w -> report w)}}$$

Problem 2: (5 points)

A 2–3 tree is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. Nodes on the outside of the tree (leaf nodes) have no children and one or two data elements.

Give an OCaml datatype that can represent 2-3 trees with `int` keys and `string` data. Your datatype should be able to represent 2-3 trees in a clear fashion but does not need to enforce balance or ordering requirements (i.e., your representation may admit illegal trees of the right "shape").

Problem 3: (5 points)

A red-black tree is a kind of tree sometimes used to organize numerical data. It has two types of nodes, black nodes and red nodes. Red nodes always have one piece of data and two children, each of which is a black node. Black nodes may have either: 1) one piece of data and two children that are red nodes; 2) one piece of data and two children that are black nodes; or 3) no data and no children (i.e., a leaf node). (This isn't a precise description of red-black trees but suffices for this exercise.)

Write a pair of mutually recursive OCaml datatypes that represent red nodes and black nodes in red-black trees. The *data* should be able to have any type, that is, your type should be polymorphic in the kind of data stored in the tree.

2 Machine Problems (60 points)

Objectives and Background

The purpose of this MP is to test the student's ability to

- use higher order functions;
- understand continuation-passing style; and
- perform CPS transformations

Another purpose of MPs in general is to provide a framework to study for the exam. Several of the questions on the exam will appear similar to the MP problems.

What to Turn In

You should put code answering each of the problems below in a file called `mp3.ml`. A good way to start is to copy `mp3-skeleton.ml` to `mp3.ml` and edit the copy. Please read the Guide for Doing MPs.

Problems

Note: In the problems below, you do not have to begin your definitions in a manner identical to the sample code, which is present solely for guiding you better. However, you have to use the indicated name for your functions and values, and they will have to conform to any type information supplied, and have to yield the same results as any sample executions given, as well as satisfying the specification given in English.

Higher order functions

Problem 1: (4 point)

The objective of this problem is to write a function `minmax_list` that takes an `int list list` and returns an `int * int list` such that the i^{th} tuple contains the minimum and maximum values of the i^{th} list.

To perform this operation, write a function `minmax` using tail recursion (explicit or using `fold_left`) that takes a list of integers and returns a 2-tuple of (min,max). Empty list should map to (0,0). Use this function and `List.map` to obtain the said function `minmax_list`.

```
# let minmax lst = ...;;
val minmax : int list -> int * int = <fun>
# minmax [1;-2;4;5;-8];;
- : int * int = (-8, 5)
# let minmax_list lst = ...;;
val minmax_list : int list list -> (int * int) list = <fun>
# minmax_list [[1];[-1;2];[1;3;4];[]];;
- : (int * int) list = [(1, 1); (-1, 2); (1, 4); (0, 0)]
```

Problem 2: (4 point)

The objective of this problem is to write a function `cumlist` that takes `'a list` and returns `'a list list` such that i^{th} list is the sublist from the beginning to the element i using the library routine `List.fold_right`.

To perform this operation, write an auxillary function `cumlist_step` that you can pass to `List.fold_right`. You can write this function in any manner you see fit.

```
# let cumlist_step <args> = ...;;
val cumlist_step : <type> = <fun>
# let cumlist lst = List.fold_right cumlist_step lst <base_case>;;
val cumlist : 'a list -> 'a list list = <fun>
# cumlist [1; 2; 3];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]]
```

Problem 3: (4 point)

The objective of this problem is to write a function `revsplit`, that, when applied to a test function `f`, and a list `lst`, returns a pair of lists. The first list of the pair should contain every element `x` of `lst` for which `(f x)` is true; and the second list contains every element for which `(f x)` is false. The order of the elements in the returned lists should be the reverse of the order in the original list.

To perform this operation, write an auxillary function `revsplit_step` that you can pass to `List.fold_left`.

```
# let revsplit_step f <args> = ...;;
val revsplit_step : <type> = <fun>
# let revsplit f lst = List.fold_left (revsplit_step f) <base_case> lst;;
val revsplit : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
# revsplit (fun x -> x < 0) [-1; 4; -2; 3; 0; 1];;
- : int list * int list = ([-2; -1], [1; 0; 3; 4])
```

Continuation-passing style

These exercises are designed to give you a feel for continuation-passing style. A function that is written in continuation-passing style does not return once it has finished computing. Instead, it calls another function (the continuation) with the result of the computation. Here is a small example:

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

# let inck i k = k (i+1)
val inck : int -> (int -> 'a) -> 'a = <fun>
```

The `inck` function takes an integer and a continuation. After adding 1 to the integer, it passes the result to its continuation.

```
# inck 3 rep_int;;
Result: 4
- : unit = ()
# inck 3 inck rep_int;;
Result: 5
- : unit = ()
```

In line 1, `inck` increments 3 to be 4, and then passes the 4 to `rep_int`. In line 4, the first `inck` adds 1 to 3, and passes the resulting 4 to the second `inck`, which then adds 1 to 4, and passes the resulting 5 to `rep_int`.

Transforming Primitive Operations

Primitive operations are "transformed" into functions that take the arguments of the original operation and a continuation, and apply the continuation to the result of applying the primitive operation on its arguments.

In the helper module `Mp3common`, we have given you a testing continuation and a few low-level functions in continuation-passing style. These are as follows:

```
val rep_int : int -> unit = <fun>
val rep_bool : bool -> unit = <fun>
val rep_float : float -> unit = <fun>
val addk : int -> int -> (int -> 'a) -> 'a = <fun>
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
val mulk : int -> int -> (int -> 'a) -> 'a = <fun>
val divk : int -> int -> (int -> 'a) -> 'a = <fun>
val modk : int -> int -> (int -> 'a) -> 'a = <fun>
val float_addk : float -> float -> (float -> 'a) -> 'a = <fun>
val float_subk : float -> float -> (float -> 'a) -> 'a = <fun>
val float_mulk : float -> float -> (float -> 'a) -> 'a = <fun>
val float_divk : float -> float -> (float -> 'a) -> 'a = <fun>
val pairk : 'a -> 'b -> ('a * 'b -> 'c) -> 'c = <fun>
val catk : string -> string -> (string -> 'a) -> 'a = <fun>
val consk : 'a -> 'a list -> ('a list -> 'b) -> 'b = <fun>
```

Problem 4: (8 points)

Write the following low-level functions in continuation-passing style. A description of what each function should do follows:

- `andk` performs boolean 'and' of the two bolean values following the 'and' rules of evaluation;
- `ork` performs boolean 'or' of the boolean values following the 'or' rules of evaluation;
- `landk` performs bitwise 'and' of the two integers;
- `lork` performs bitwise 'or' of the two integers;
- `lxork` performs bitwise 'xor' of the two integers;
- `expk` computes e raised to the float provided;
- `logk` computes log of the float provided to base e;
- `powk` raises the first float to the second float (exponentiation);

Note that all these operations have primitive functions.

```
# let andk a b k = ...;;
val andk : bool -> bool -> (bool -> 'a) -> 'a = <fun>
# let ork a b k = ...;;
val ork : bool -> bool -> (bool -> 'a) -> 'a = <fun>
```

```
# let landk a b k = ...;;
val landk : int -> int -> (int -> 'a) -> 'a = <fun>
# let lork a b k = ...;;
val lork : int -> int -> (int -> 'a) -> 'a = <fun>
# let lxork a b k = ...;;
val lxork : int -> int -> (int -> 'a) -> 'a = <fun>
# let expk a k = ...;;
val expk : float -> (float -> 'a) -> 'a = <fun>
# let logk a k = ...;;
val logk : float -> (float -> 'a) -> 'a = <fun>
# let powk a b k = ...;;
val powk : float -> float -> (float -> 'a) -> 'a = <fun>
```

Nesting Continuations

```
# let 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 = ()
```

We needed to add three numbers together, but `addk` itself only adds two numbers. 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.

Problem 5: (8 points)

Implement integer modulo arithmetic operations in continuation-passing style as described below.

- `modaddk a b n k` computes `(a + b) mod n`;
- `modsubk a b n k` computes `(a - b) mod n`;
- `modmulk a b n k` computes `(a * b) mod n`;
- `modeqk a b n k` tells if `a` and `b` are congruent in mod n;

you can only use operations in CPS that are defined in Mp3common.

```
# let modaddk a b n k = ...;;
val modaddk : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# modaddk 5 7 10 rep_int;;
Result: 2
- : unit = ()

# let modsubk a b n k =...;;
val modsubk : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# modsubk 12 4 7 rep_int;;
Result: 1
- : unit = ()

# let modmulk a b n k = ...;;
val modmulk : int -> int -> int -> (int -> 'a) -> 'a = <fun>
# modmulk 3 4 8 rep_int;;
Result: 4
- : unit = ()
```

```
# let modeqk a b n k = ...;;
val modeqk : int -> int -> int -> (bool -> 'a) -> 'a = <fun>
# modeqk 6 11 5 rep_bool;;
Result: true
- : unit = ()
```

Transforming recursive functions

How do we write recursive programs in CPS? Consider the following recursive function:

```
# 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 computation explicit as follows:

```
# 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
```

Now, 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. Thus 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 rep_int;;
Result: 120
- : unit = ()
```

Notice 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 it to the final result: `n * m`
- pass it to the final continuation `k`

Notice that this is an extension of the "nested continuation" method.

Problem 6: (6 points)

a. **(2 pts)**

Write a function `power`, which takes two integers a, n; and computes a^n by multiplying a to itself n times. It should return 1 if $n \leq 0$.

```
# let rec power a n = ...;;
val power : int -> int -> int = <fun>
# power 2 3;;
- : int = 8
```

b. **(4 pts)**

Write the function `powerk` which is the CPS transformation of the code you wrote in part a.

```
# let rec powerk a n k = ...;;
val powerk : int -> int -> (int -> 'a) -> 'a = <fun>
# powerk 2 3 rep_int;;
Result: 8
- : unit = ()
```

Problem 7: (6 points)

a. **(2 pts)**

Write the function `dup_alt` that takes a list `l` and creates a new list that duplicates every alternate element starting from the second.

```
# let rec dup_alt l = ...;;
val dup_alt : 'a list -> 'a list = <fun>
# dup_alt [1; 2; 3; 4];;
- : int list = [1; 2; 2; 3; 4; 4]
```

b. **(4 pts)**

Write the function `dup_altk` that is the CPS transformation of the code you wrote in part a.

```
# let rec dup_altk l k = ...;;
val dup_altk : 'a list -> ('a list -> 'b) -> 'b = <fun>
# dup_altk [1; 2; 3; 4] dup_alt;;
- : int list = [1; 2; 2; 2; 3; 3; 4; 4; 4]
```

Problem 8: (10 points)

a. **(3 pts)**

Write a function `rev_map` which takes a function f (of type `'a -> 'b`) and a list l (of type `'a list`). If the list l has the form $[a_1; a_2; \ldots; a_n]$, then `rev_map` applies f on a_n, then on a_{n-1}, then ..., then on a_1 and then builds the list $[f(a_1); \ldots; f(a_n)]$ with the results returned by f. The function f may have side-effects (e.g. `report`). There must be no use of list library functions.

```
# let rec rev_map f l = ...;;
val rev_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# rev_map (fun x -> print_int x; x + 1) [1;2;3;4;5];;
54321- : int list = [2; 3; 4; 5; 6]
```

b. **(7 pts)**

Write the function `rev_mapk` that is the CPS transformation of the code you wrote in part a. You must assume that the function f is also transformed in continuation-passing style, that is, the type of f is not `'a -> 'b`, but `'a -> ('b -> 'c) -> 'c`.

```
# let rec rev_mapk f l k = ...;;
val rev_mapk :
('a -> ('b -> 'c) -> 'c) -> 'a list -> ('b list -> 'c) -> 'c = <fun>
# let print_intk i k = k (print_int i);;
val print_intk : int -> (unit -> 'a) -> 'a = <fun>
# rev_mapk (fun x -> fun k -> print_intk x (fun t -> inck x k))
[1;2;3;4;5] (fun x -> x);;
54321- : int list = [2; 3; 4; 5; 6]
```

Problem 9: (10 points)

a. **(3 pts)**

Write a function `partition` which takes a list `l` (of type `'a list`), and a predicate `p` (of type `'a -> bool`), and returns a pair of lists (l_1, l_2) where l_1 contains all the elements in l satisfying p, and l_2 contains all the elements in l not satisfying p. The order of the elements in l_1 and l_2 is the order in l. There must be no use of list library functions.

```
# let rec partition l p = ...;;
val partition : 'a list -> ('a -> bool) -> 'a list * 'a list = <fun>
# partition [1;2;3;4;5] (fun x -> x >= 3);;
- : int list * int list = ([3; 4; 5], [1; 2])
```

b. **(7 pts)**

Write a function `partitionk` which is the CPS transformation of the code you wrote in part a. You must assume that the predicate p is also transformed in continuation-passing style, that is, its type is not `'a -> bool`, but `'a -> (bool -> 'b) -> 'b`.

```
# let rec let rec partitionk l p k = ...;;
val partitionk :
'a list -> ('a -> (bool -> 'b) -> 'b) -> ('a list * 'a list -> 'b)
-> 'b = <fun>
# partitionk [1;2;3;4;5] (fun x -> fun k -> geqk x 3 k) (fun x -> x);;
- : int list * int list = ([3; 4; 5], [1; 2])
```

3 Guide for Doing MPs

1. Setup a directory for CS 421 and a subdirectory for HWs/MPs.

2. Create a subsubdirectory corresponding to this HW within them.

3. Retrieve the directory for this MP posted on Moodle and all its contents. Get into that directory. (If we revise an assignment, you will need to repeat this to obtain the revision.)

4. To make sure you have all the necessary pieces, start by executing make. This will create the grader executable. Run the executable (`>$ ./grader`). Examine the failing test cases for places where errors were produced by your code. At this point, everything should compile, but the score will be 0.

5. Read and understand the problem for the handout on which you wish to begin working. (Usually, working from top to bottom makes most sense.) There is a tests file in this directory. This is an important file containing the an incomplete set of test cases; you'll want to add more cases to test your code more thoroughly. Reread the problem from the handout, examining any sample output given. Open the tests file in the mpX directory. Find the test cases given for that problem. Add your own test cases by following the same pattern as of the existing test cases. Try to get a good coverage of your function's behaviour. You should even try to have enough cases to guarantee that you will catch any errors. (This is not always possible, but a desirable goal.) And yes, test cases should be written even before starting the implementation of your function. This is a good software development practice.

6. If necessary, reread the statement of the problem once more. Place your code for the solution in mpX.ml (or mpX.mll or mpX.mly as specified by the assignment instructions) replacing the stub found there for it. Implement your function. Try to do this in a step-wise fashion. When you think you have a solution (or enough of a part of one to compile and be worth testing), save you work and execute make and the ./grader again. Examine the passing and failing test cases again. Each failure is an instance where your code failed to give the right output for the given input, and you will need to examine your code to figure out why. When you are finished making a round of corrections, run make, followed by ./grader again. Continue until you find no more errors. Consider submitting your partial result so that you will at least get credit for what you have accomplished so far, in case something happens to interfere with your completing the rest of the assignment.

7. When your code no longer generates any errors for the problem on which you were working, return to steps 3) and 4) to proceed with the next problem you wish to solve, until there are no more problems to be solved.

8. When you have finished all problems (or given up and left the problem with the stub version originally provided), you will need to submit your file along with your PDF submission on Moodle.