Recursion fairy practice
Supplementary notes for lecture 2, CS 421, Spring 2013

Prof. Kamin

January 17, 2013

The whole, entire idea of the recursion fairy — and people who regularly write recursive functions agree that this is the only approach to writing them — is to *assume* the recursive call will give you the correct result and figure out how to get the correct result starting from there. Don't try to work through the mechanics of the recursive calls, or think about what's on the stack. Unfortunately, it can be difficult at first to follow this discipline — the urge to try to puzzle out the function in *operational* terms is too strong.

So what I'm going to do in these notes is start by asking you a set of very simple questions. Then I'll ask you to write some recursive functions. Hopefully you'll see how answering those simple questions at the start makes the recursive functions really easy.

Simple questions

1. Suppose r is a list consisting of $n-1$ copies of a value x. Give an expression, in r and x, for a list of n copies of x.

2. Let B be a list of booleans and X a list the same length as B. Further, let Y be the list containing "X compressed by B" — that is, Y contains the elements of X corresponding to the places where B has `true`. (E.g. if B = `[true; false; false; true]` and X = `[1; 2; 3; 4]`, then Y = `[1; 4]`.) Then give the expression for Z, which is $x::X$ compressed by $b::B$ (b is a boolean value and x is some other value). The expression you give must express Z in terms of b, x, and Y.

3. Let (X, Y) be a pair of lists that contain alternate elements of a list A — X contains the first, third, fifth, etc., elements of A, and Y contains the second, fourth, etc. Give an expression for the pair of lists containing the alternate elements of the list $a::b::A$ (for any a and b). The expression you give should express the result in terms of X, Y, a, and b.

4. This one is similar to the last. You're given (X, Y) as above. This time you need to give an expression for the pair of lists containing the alternate elements of the list $a::A$, in terms of X, Y, and a. (For example, if A = `[1; 2; 3; 4; 5]` — so X = `[1; 3; 5]` and Y = `[2; 4]` — and $a = 7$, then $a::A$ = `[7; 1; 2; 3; 4; 5]`, so the expression you give should have value (`[7; 2; 4]`, `[1; 3; 5]`).)

5. Suppose A is a list of integers and p is an integer. Further suppose that X contains all the elements of A less than or equal to p, and Y contains all the others. Finally, suppose i is an integer, and let $A' = i::A$. Give an expression for the pair of lists containing the elements of A' that are less than or equal to p and greater than p; it should express this pair in terms of X, Y, p, and i.

6. Suppose X and Y are two lists of integers of equal length, and let A be a list of booleans of the same length; furthermore, an element of A is true if and only if the corresponding elements of X and Y are equal. (E.g. if X = `[1; 3; 5; 4; 2]` and Y = `[3; 3; 1; 3; 2]`, then A = `[false; true; false; false; true]`. Let x and y be two integers. Give the expression for the list that has the same relation to x`::`X and y`::`Y that A has to X and Y. It should express this new list solely in terms of x, y, and A.

7. Suppose A is a list that bears the following relation to lists X and Y and value x: A is the result of concatening the *reverse* of list X to the list x`::`Y. (E.g. if X were `[2; 3]`, Y were `[4; 5]`, and x were 1, then A would be `[3; 2; 1; 4; 5]`.) Give an expression for the list that is the result of concatenating the reverse of x`::`X to the list Y. It should express this new list solely in terms of x and A.

8. As noted in lecture 2, the "run-length encoding" of a list L is a list E of pairs `[(`a`,`x`); (`b`, `y`); ...]`, where a, b, ... are integers and x, y, ... are elements of L, such that L consists of a copies of x followed by b copies of y, etc. (E.g. if L = `['a'; 'a'; 'b'; 'c'; 'c'; 'c'; 'a']`, then E is `[(2, 'a'); (1, 'b'); (3, 'c'); (1, 'a')]`.) We have several questions here:

   (a) Given such an E and L, give an expression for the encoding of x`::`L, in terms of x and E, *on the assumption that x is different from the head of L.*

   (b) Given such an E and L, give an expression for the encoding of x`::`L, in terms of x and E, *on the assumption that x is the same as the head of L.* You can use operations `hd` and `tl` on E.

   (c) Give E and L again, give an expression for the encoding of x`::`L, in terms of x and E, that works whether x is the same as the first element of L or not. Here you will need to give a conditional expression.

Recursive functions

These functions correspond to the questions above, with each question showing the use of the recursion fairy. In some cases, we tell you what the recursive call is, and then you just copy the answer above, *mutatis mutandis*, and provide the base case.

1. Define `copy`, where `copy` n x returns a list containing n copies of x. The recursive call is `copy` $n-1$ x.

```
copy 3 'a';;
- : char list = ['a'; 'a'; 'a']
```

2. Define `compress`, where `compress` B X returns X compressed by B. You can assume B and X are of the same length. The recursive call is `compress (tl` B`) (tl` X`)`.

```
compress [true; false; false; true] [1; 2; 3; 4];;
- : int list = [1; 4]
```

3. Define `alternates`, where `alternates` A returns a pair (X, Y), containing the alternate elements of A, as above. The recursive call is `alternates (tl (tl` A`))`. You may not assume that A has length at least two, so you will have to handle the empty list and one–element list as base cases.

```
alternates [1; 2; 3; 4; 5];;
- : int list * int list = ([1; 3; 5], [2; 4])
```

4. Define `alternates` again, but this time the recursive call is `alternates (tl` A`)`. You have to handle only the empty list as a base case.

5. Define `partition`, where `partition` A p returns a pair of lists (X, Y), with X containing all the elements of A less than or equal to p and Y containing all the others. The recursive call is on the tail of A.

```
partition [2; 6; 3; 9; 5; 7] 5;;
- : int list * int list = ([2; 3], [6; 9; 5; 7])
```

6. Define `pairwiseequal`, where `pairwiseequal` X Y returns a boolean list where each element is true iff the corresponding elements of X and Y are equal. You may assume X and Y are of equal length. You'll need to figure out the recursive call yourself.

```
pairwiseequal [1; 2; 3; 4; 5] [5; 4; 3; 2; 1];;
- : bool list = [false; false; true; false; false]
```

7. Define `revappend`, where `revappend` X Y returns the result of concatenating the *reverse* of X to Y. You'll need to figure out the recursive call yourself. (This function is interesting for this reason: You can make the definition `let reverse x = revappend x []`, and this definition of `reverse` is much more efficient than the definition we gave in class. In fact, this definition runs in time linear in the length of `x`, while the one we gave in class, which repeatedly calls `append`, is quadratic.)

```
revappend [1; 2; 3] [4; 5; 6];;
- : int list = [3; 2; 1; 4; 5; 6]
```

8. Define `runencode`, where `runencode` L returns the run-length encoding of L. The recursive call here is on the tail of L. (Only the last of the expressions you wrote above for run-length encoding is relevant here.)

```
runencode ['a'; 'a'; 'b'; 'c'; 'c'; 'c'; 'a'];;
- : (int * char) list = [(2, 'a'); (1, 'b'); (3, 'c'); (1, 'a')]
```

Solutions

```
let rec copy n x = if n=0 then [] else x :: copy (n-1) x;;

let rec compress b x = match b with
          [] -> []
        | true::bt -> hd x :: compress bt (tl x)
        | false::bt -> compress bt (tl x) ;;

let rec alternates a = match a with
          [] -> ([], [])
        | [h] -> ([h], [])
        | h::ht::tt -> let (x,y) = alternates tt in (h::x, ht::y) ;;

let rec alternates a = match a with
          [] -> ([], [])
        | h::t -> let (x,y) = alternates t in (h::y, x) ;;

let rec partition a p = match a with
          [] -> ([], [])
        | h::t -> let (x, y) = partition t p
                 in if h<p then (h::x, y) else (x, h::y) ;;

let rec pairwiseequal x y = match (x,y) with
          ([], []) -> []
        | (hx::tx, hy::ty) -> (hx=hy) :: pairwiseequal tx ty ;;

let rec revappend x y = if x=[] then y
                        else revappend (tl x) (hd x :: y) ;;

let rec runencode l = match l with
          [] -> []
        | [x] -> [(1,x)]
        | h::t -> let (n,x)::y = runencode t
                  in if h=x then (n+1,x)::y
                            else (1,h)::(n,x)::y ;;
```