

CS 421 – Spring, 2008
More examples of recursive functions on lists

This note gives several more examples of recursive functions on lists. First, a correction to the notes and an answer to a question asked in class:

Correction: I stated the scope rules for `let` slightly incorrectly in class. If `e` is the expression “`let x = e1 in e2`”, the scope of `e1` (the set of variables visible in `e1`) is the same as the scope of `e`, while the scope of `e2` is that same scope with `x` added. This is just what I said in class. However, if `e` is “`let f x = e1 in e2`”, the rules are a little different: the scope of `e1` is the scope of `e` plus `x`, while the scope of `e2` is the scope of `e` plus `f`. In “`let rec f x = e1 in e2`”, the scope of `e1` includes both `x` and `f`.

Question: One student asked if there is a way to have a sequence of local definitions, all in the same scope. To do this, you need to use “`and`”:

```
let x = e1
and y = e2
in e
```

Here, `e1` and `e2` are evaluated in the same scope. This means that `x` cannot be used in `e2`, or rather, if used, it refers to an `x` defined in the surrounding scope. For example:

```
let x = 3
in let x = 4
and y = x
in x+y
```

evaluates to 7. For defining mutually recursive functions, as I mentioned in class, you also use “`and`”.

Some recursive functions on lists: These are provided for self-study. Try to do them yourself before reading the solution. (By the way, some of the solutions use the `hd` and `tl` operations; remember that to use those, you need to enter “`open List;`” first.)

1. `copy n x` returns a list consisting of `n` copies of `x`. The recursive call `copy (n-1) x` will return a list with `n-1` copies, so we can create a list of `n` copies by consing `x` to the front. The recursive call makes sense only if `n > 0`; if it is 0, `copy n x` is the empty list:

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

2. `rotate n lis`, where `n` is a list, gives the result of rotating the list to the right `n` times. E.g. `rotate 2 [1;2;3;4;5;6] = [5;6;1;2;3;4]`. We can assume the recursive call `rotate (n-1) lis` will rotate the list `n-1` times. How do we get the final result? By rotating it one more time. So we have to write a function to rotate the list once. We could do that recursively, but instead I’ll accomplish this using `reverse`:

```

let rec rotate n lis =
  if lis = [] then []
  else let rec rotatel lis = let (x::xs) = reverse lis
                              in x::reverse xs
        in if n = 0 then lis else rotate (n-1) (rotatel lis)

```

3. `compress bs xs` has as argument two lists of the same length, the first a list of Booleans and the second a list of ints. It returns a list consisting of all the elements of `xs` in positions where the corresponding element of `bs` is true:

```

compress [true; false; true; true; false] [1;2;3;4;5] = [1;3;4]

```

This is a simple recursion on the tail: recursively calculating `compress (tl bs) (tl xs)`, we just need to cons the head of `xs` to the result if the head of `bs` is true:

```

let rec compress bs xs =
  if bs = [] then []
  else let c = compress (tl bs) (tl xs)
        in if (hd bs) then (hd xs)::c else c

```

4. `split lis` returns a pair of two lists, the first containing every other element of `lis`, starting with the first, and the second containing the remaining elements, all in the same order as in `lis`:

```

split [1;2;3;4;5;6] = ([1;3;5], [2;4;6])

```

We'll give two solutions.

(A) The solution involving recursion on `(tl lis)` is a little harder, so we'll start with a solution in which we recurse on `(tl (tl lis))`. In the above example, this call returns `([3;5], [4;6])`, and we get the solution by consing `(hd lis)` to the first list and `(hd (tl lis))` to the second list:

```

let rec split lis = match lis with
  [] -> ([], [])
  | [x] -> ([x], [])
  | (x::y::xs) -> let (lis1, lis2) = split xs
                  in (x::lis1, y::lis2)

```

(B) We can solve the problem with recursion on `(tl lis)`, but we need to be careful. `split (tl lis)` is guaranteed to return `lis` split in half. You would think the recursive case would just be:

```

let (lis1, lis2) = split (tl lis)
in (hd lis :: lis1, lis2)

```

But this doesn't work at all – in fact, with this definition, `split lis` would just return `(lis, [])`. Why didn't the recursion fairy work? It did really – but remember that the bargain works both ways: the recursion fairy will return the correct result from the

recursive call, but we have to return the correct result from this call. We didn't do that. Remember, `split` is specified like this:

```
split [x1; x2; x3; ...] = ([x1; x3; ...], [x2; x4; ...])
```

The recursive call on the tail can be assumed to return, correctly,

```
split [x2; x3; ...] = ([x2; x4; ...], [x3; x5; ...])
```

That is, it returns the first, third, etc. elements of its argument, and then the other elements, just as specified. If we just cons `x1` onto the first list, we get `([x1; x2; x4; ...], [x3; x5; ...])`, which is not right. The solution should be clear: We need to cons `x1` onto the second list, and reverse the order of the tuple. Here's the complete definition:

```
let rec split lis = if lis = [] then ([], [])
                    else let (lis1, lis2) = split (tl lis)
                          in (hd lis :: lis2, lis1)
```

5. `partition lis pivot` returns a pair of lists `(lis1, lis2)`, where `lis1` contains all the elements of `lis` that are less than `pivot`, and `lis2` contains all the other elements (all in the same order as in `lis`):

```
partition [3;7;2;9;12;5] 6 = ([3;2;5], [7;9;12])
```

This is a straightforward recursion on the tail of `lis`: If `(lis1, lis2)` is the result of the call `partition (tl lis) pivot`, then to get `partition lis pivot` we just need to cons `(hd lis)` on to either `lis1` or `lis2`, depending on whether it is less than `pivot`:

```
let rec partition lis pivot =
  if lis = [] then ([], [])
  else let (lis1, lis2) = partition (tl lis) pivot
        in if hd lis < pivot then (hd lis :: lis1, lis2)
           else (lis1, hd lis :: lis2)
```

6. `qsort lis` sorts `lis` using the quicksort algorithm. That is, it takes `hd lis` as the pivot, partitions `(tl lis)` around it, then recursively sorts the two parts and concatenates the results. As usual, the recursion fairy says that `qsort` applied to any sublist of `lis` will return the correct result:

```
let rec qsort lis =
  if lis = [] then []
  else let (lis1, lis2) = partition (tl lis) (hd lis)
        in (qsort lis1) @ [hd lis] @ (qsort lis2)
```