15-150 Fall 2011
Lab 5

September 28, 2011

1 Introduction

The goal for this lab is to make you more familiar with higher-order functions in SML, in addition to reinforcing the use of some very common ones. You will write several functions for performing generic operations on lists and trees, in addition to a function for mapping over a data structure that you define.

Please take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are encouraged to collaborate with your classmates and to ask the TAs for help.

1.1 Getting Started

Please download the `lab05.sml` file from the course Web site or copy it from the lab afs directory

`/afs/andrew.cmu.edu/course/15/150/asgn/lab/05/`

This file contains some of the datatype definitions and functions that were discussed in the last couple lectures.

1.2 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have a purpose and tests.

2 Higher-order Functions on Lists

Recall `map` from lecture, which we will call `listmap` here:

```
listmap : ('a -> 'b) * 'a list -> 'b list
```

`listmap (f, L)` applies `f` to each element of `L`, returning a list of the results; that is, `listmap` (`f`, $[\texttt{v}_1, ..., \texttt{v}_n]$) computes $[\texttt{f v}_1, ..., \texttt{f v}_n]$

2.1 Filter

Consider the following two functions:

```
fun evens (l : int list) : int list =
    case l of
        [] => []
      | x :: xs => ( case evenP x of
                       true => x :: evens xs
                     | false => evens xs )

fun allLessThan (pivot : int, l : int list) : int list =
    case l of
        [] => []
      | x :: xs => ( case x < pivot of
                       true => x :: allLessThan (pivot, xs)
                     | false => allLessThan (pivot, xs) )
```

The pattern here is "keep all the elements of the list that satisfy some predicate."

Task 2.1 Define a function

```
fun filter (p : 'a -> bool, l : 'a list) : 'a list = ...
```

that abstracts over this pattern. The function `p` represents the predicate.

Task 2.2 Define `evens` and `allLessThan` by calling `filter` with the appropriate predicate.

2.2 All

Play the same game with these two:

```
fun allPos (l : int list) : bool =
    case l of
        [] => true
      | x :: xs => (x > 0) andalso allPos xs

fun allOfLength (len : int, l : 'a list list) : bool =
    case l of
        [] => true
      | x :: xs => (inteq(List.length x, len)) andalso allOfLength(len, xs)
```

Task 2.3 Write a higher-order function `all` that can be used to define `allPos` and `allOfLength`, and then define these two functions in terms of it.

Have the TAs check your code proof before proceeding!

3 Higher-order Functions on Trees

Last week, we used trees that had data at each node. An alternative is to use trees where there is data only at the leaves:

An `'a tree` is either

1. empty
2. a leaf with value `x:'a`
3. a node with two subtrees

and that's it!

Task 3.1 Define a datatype `'a tree` representing such trees. Don't forget to fill in the constructors you use in the definition of `treeFromList` in the support code!

Have the TAs check your datatype definition before proceeding!

For many of the higher-order list functions previously discussed, it is possible to define corresponding functions that operate over trees instead.

3.1 Map

Task 3.2 Write the function

```
treemap : ('a -> 'b) * 'a tree -> 'b tree
```

such that `treemap (f,t)` computes a tree whose elements are given by applying `f` to the elements in `t`.

Using `treemap`, write the function

```
treemult : int * int tree -> int tree
```

such that `treemult (c, T)` evaluates to the tree `T'` where each node in `T'` contains the element in that node of `T` multiplied by `c`.

3.2 All

Task 3.3 Write the function

```
treeall : ('a -> bool) * 'a tree -> bool
```

such that `treeall (p, T)` evaluates to `true` if `p x` evaluates to true for each element `x` of `T`, and evaluates to `false` otherwise. Using `treeall`, write the function

```
nattree : int tree -> bool
```

such that `nattree T` evaluates to `true` if all of the elements of `T` are natural numbers (that is, greater than or equal to zero).

3.3 Reduce

Assuming the constructors for `tree` are named `Empty`, `Leaf`, and `Node`, here are two functions:

```
fun sum (t : int tree) : int =
   case t of
        Empty => 0
      | Leaf x => x
      | Node(t1,t2) => (sum t1) + (sum t2)
fun max (t : int tree) : int =
   case t of
        Empty => 0
      | Leaf x => x
      | Node(t1,t2) => Int.max((max t1), (max t2))
```

Though we should perhaps be using options instead, we will define the `sum` and `max` of an empty tree to be `0`.

The general pattern here is called `reduce`, which takes a binary operator of type `'a * 'a -> 'a` to apply at each node, and a value of type `'a` for the empty tree, and computes an `'a` from an `'a tree`.

Task 3.4 Write the function

```
treereduce : ('a * 'a -> 'a) * 'a * 'a tree -> 'a
```

that implements the operation of reduction on trees. Using `treereduce`, rewrite the above functions.

Have the TAs check your functions before proceeding!

4 Map/reduce Puzzles

We have provided

```
lines : string -> string tree
words : string -> string tree
```

lines divides a string into lines (delimited by the newline character). **words** divides a string into words (delimited by spaces or newlines).

Task 4.1 Define functions

```
(* computes the number of words in a document *)
fun wordcount (s : string) : int = ...
(* computes the number of words in the longest line in a document *)
fun longestline (s : string) : int = ...
```

These functions should not be defined recursively.

For example, given the string

```
for life's not a paragraph
And death i think is no parenthesis
```

`wordcount` should return 12, and `longestline` should return `7`. Note that you can type in this document using `\n` for newlines:

```
"for life's not a paragraph\nAnd death i think is no parenthesis\n"
```

5 Options

On Homework 3, we had you write a function

```
subset_sum_cert : int list * int -> bool * int list
```

Here is a solution:

```
fun subset_sum_cert (l : int list, s : int) : bool * int list =
    case l of
         nil => (inteq (s, 0), nil)
       | n1::l' => (case subset_sum_cert (l', s - n1) of
                         (true, l1) => (true, n1 :: l1)
                       | (false, _) => subset_sum_cert (l', s))
```

This function is kind of gross, because it evaluates to `(true, U)` if `U` was a solution to the subset sum problem and `(false, nil)` if there is no such solution. This is a somewhat ugly way to handle the situation, since the list in the second case is completely superfluous—it was only there because the type required it. It is also possible for invalid results to occur this way—returning `(false, [1])`, for example.

A better way to address this problem is to use the option type:

Task 5.1 Implement the function

```
subset_sum_opt : int list * int -> int list option
```

that return `SOME U` if there is some submultiset `U` that sums to the target, or `NONE` otherwise.