1 Introduction

The goal for this lab is to make you more familiar with higher-order functions in SML.

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

Update your clone of the git repository to get the files for this weeks lab as usual by running

    git pull

from the top level directory (probably named 15150).

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.

Survey

Please fill out the following survey:

https://www.surveymonkey.com/s/wheredotreescomefrom
2 Higher-order Functions on Lists

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

\[
\text{listmap} : (\alpha \rightarrow \beta) \times \alpha \text{ list} \rightarrow \beta \text{ list}
\]

listmap \((f, L)\) applies \(f\) to each element of \(L\), returning a list of the results; that is, listmap \((f, [v_1, \ldots, v_n])\) computes \([f v_1, \ldots, f v_n]\)

2.1 Filter

Consider the following two functions:

\[
\begin{align*}
\text{fun evens} \ (l : \text{int list}) : \text{int list} &= \text{case} \ l \ \text{of} \\
& \quad [] \Rightarrow [] \\
& \quad | \ x :: \ xs \Rightarrow (\text{case} \ \text{evenP} \ x \ \text{of} \\
& \quad \quad \text{true} \Rightarrow x :: \text{evens} \ xs \\
& \quad \quad \text{false} \Rightarrow \text{evens} \ xs )
\end{align*}
\]

\[
\begin{align*}
\text{fun allLessThan} \ (\text{pivot} : \text{int}, l : \text{int list}) : \text{int list} &= \text{case} \ l \ \text{of} \\
& \quad [] \Rightarrow [] \\
& \quad | \ x :: \ xs \Rightarrow (\text{case} \ x < \text{pivot} \ \text{of} \\
& \quad \quad \text{true} \Rightarrow x :: \text{allLessThan} \ (\text{pivot}, \ xs) \\
& \quad \quad \text{false} \Rightarrow \text{allLessThan} \ (\text{pivot}, \ xs) )
\end{align*}
\]

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

Task 2.1 Define a function

\[
\text{fun filter} \ (p : \alpha \rightarrow \text{bool}, l : \alpha \text{ list}) : \alpha \text{ list} = ...
\]

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.

Task 2.3 On Homework 4, we hadn’t introduced higher-order functions yet, so for quicksort_l (quicksorting lists) we had you define a first-order but less-general variant of filter. Rewrite quicksort_l to use the filter function you defined above.
2.2 All

Play the same game with these two:

```ml
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 2.4** 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

\[
\text{treemap} : ('a -> 'b) * 'a tree -> 'b tree
\]
such that \text{treemap} \ (f, t) \ computes \ a \ tree \ whose \ elements \ are \ given \ by \ applying \ f \ to \ the \ elements \ in \ t.

Using \text{treemap}, write the function

\[
\text{treemult} : \text{int} * \text{int tree} -> \text{int tree}
\]
such that \text{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

\[
\text{treeall} : ('a -> \text{bool}) * 'a tree -> \text{bool}
\]
such that \text{treeall} \ (p, T) \ evaluates \ to \ \text{true} \ if \ p \ x \ evaluates \ to \ \text{true} \ for \ each \ element \ x \ of \ T, \ and \ evaluates \ to \ \text{false} \ otherwise. \ Using \text{treeall}, \ write \ the \ function

\[
\text{nattree} : \text{int tree} -> \text{bool}
\]
such that \text{nattree} \ T \ evaluates \ to \ \text{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:

\[
\text{fun sum (t : int tree)} : \text{int} = \\
\text{case t of} \\
| \text{Empty} => 0 \\
| \text{Leaf x} => x \\
| \text{Node(t1,t2)} => \text{(sum t1) + (sum t2)}
\]

\[
\text{fun max (t : int tree)} : \text{int} = \\
\text{case t of} \\
| \text{Empty} => 0 \\
| \text{Leaf x} => x \\
| \text{Node(t1,t2)} => \text{Int.max((max t1), (max t2))}
\]
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

```plaintext
  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

```plaintext
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

```plaintext
  (* 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"
```