COMP 212 Spring 2015
Homework 03

1 Introduction

This homework will focus on writing functions on lists and proving properties of them. This homework is longer and harder than the previous two: start early!

Because the programming and written tasks are integrated, all problems are described in this handout, but there is a separate *handin* sheet for the written on the course web page.

2 Zippidy Doo Da

It's often convenient to take a pair of lists and make one list of pairs from it. For instance, if we have the lists

$$[5, 1, 2, 1] \qquad \text{and} \qquad [\text{“}a\text{”}, \text{“}b\text{”}]$$

we might be interested in the list

$$[(5, \text{“}a\text{”}), (1, \text{“}b\text{”})]$$

Task 2.1 (5%). Write the function

```
zip : int list * string list -> (int * string) list
```

that performs the transformation of pairing the n^{th} element from the first list with the n^{th} element of the second list. If your function is applied to a pair of lists of different length, the length of the returned list should be the minimum of the lengths of the argument lists. You should ensure that `zip` is a total function (but you do not need to formally prove this fact).

Task 2.2 (5%). Write the function

```
unzip : (int * string) list -> int list * string list
```

`unzip` does the opposite of `zip` in the sense that it takes a list of tuples and returns a tuple of lists, where the first list in the tuple is the list of first elements and the second list is the list of second elements. You should ensure that `unzip` is a total function (but you do not need to formally prove this fact).

Task 2.3 (10%). Prove Theorem 1.

Theorem 1. *For all* `l : (int * string) list`, *`zip(unzip l)`* $\cong$ *l*.

Be sure to use the template for a proof by *structural induction on lists*; see the Lecture 4 notes. In your proof, be sure to state when you are using valuability, and explain why the expressions in question are valuable. You may use totality of `zip` and `unzip` in your explanation but need to cite such uses carefully.

Task 2.4 (4%). Prove or disprove Theorem 2.

Theorem 2. *For all* `l1 : int list` *and* `l2 : string list`,

`unzip(zip (l1,l2))` $\cong$ *`(l1,l2)`*

3 Conway's Lost Cosmological Theorem

3.1 Definition

If l is any list of integers, the look-and-say list of s is obtained by reading off adjacent groups of identical elements in s. For example, the look-and-say list of

$$l = [2, 2, 2]$$

is

$$[3, 2]$$

because l is exactly "three twos.". Similarly, the look-and-say sequence of

$$l = [1, 2, 2]$$

is

$$[1, 1, 2, 2]$$

because l is exactly "one ones, then two twos."

We will use the term *run* to mean a maximal length sublist of a list with all equal elements. For example,

$$[1, 1, 1] \qquad \text{and} \qquad [5]$$

are both runs of the list

$$[1, 1, 1, 5, 2]$$

but

$$[1, 1] \qquad \text{and} \qquad [5, 2] \qquad \text{and} \qquad [1, 2]$$

are not: $[1, 1]$ is not maximal, $[5, 2]$ has unequal elements, and $[1, 2]$ is not a sublist.

You will now define a function `look_and_say` that computes the look-and-say sequence of its argument using a helper function and a new pattern of recursion.

3.2 Implementation

To help define the `look_and_say` function, you will write a helper function `lasHelp` with the following spec. `lasHelp` takes three arguments

- `l : int list`, the tail of the list
- `x : int`, the number found in the current run
- `acc : int`, the number of times the current number has already been seen in the run.

From these arguments, the `lasHelp` computes the pair (`tail`, `total`) where

- `tail : int list` is the tail of `l` following the last number equal to `x` at the front of the list
- `total : int` is the total length of the current run (*i.e.*, the sum of `acc` and the length of the run of numbers equal to `x` at the front of `l`).

For example,

$$\texttt{lasHelp}([1,2,3],4,1) \cong ([1,2,3],1)$$
$$\texttt{lasHelp}([2,2,6,3],2,2) \cong ([6,3],4)$$

Task 3.1 (10%). Write the function

```
lasHelp : int list * int * int -> int list * int
```

according to the given specification. Note that you can use the function `inteq` in `hw03.sml` to compare integers for equality. Now, write the function

```
look_and_say : int list -> int list
```

using this helper function.[1]

3.3 Cultural Aside

The title of this problem comes from a theorem about the sequence generated by repeated applications of the "look and say" operation. As `look_and_say` has type `int list -> int list`, the function can be applied to its own result. For example, if we start with the list of length one consisting of just the number `1`, we get the following first 6 elements of the sequence:

```
[1]
[1,1]
[2,1]
[1,2,1,1]
[1,1,1,2,2,1]
[3,1,2,2,1,1]
```

Conway's theorem states that any element of this sequence will "decay" (by repeated applications of `look_and_say`) into a "compound" made up of combinations of "primitive elements" (there are 92 of them, plus 2 infinite families) in 24 steps. If you are interested in this sequence, you may wish to consult [Conway(1987)] or other papers about the "look and say" operation.

[1] *Hint:* The recursive call in the inductive case of `look_and_say` will sometimes be on a list that is more than one element shorter. This corresponds to the notion of well-founded recursion discussed in lecture.

4 Prefix-Sum

The prefix-sum of a list `l` is a list `s` where the i^{th} element of `s` is the sum of the first $i+1$ elements of `l`. For example,

```
prefixSum [] ≅ []
prefixSum [1,2,3] ≅ [1,3,6]
prefixSum [5,3,1] ≅ [5,8,9]
```

Here is a simple implementation of this function:

```
fun prefixSum (l : int list) : int list =
    case l of
      [] => []
    | x::xs => x :: add_to_each (prefixSum xs, x)
```

The function `add_to_each` is the "raise salaries" function from lecture; it adds `x` to each element of `prefixSum xs`.

Task 4.1 (5%). Write a recurrence for the work of `prefixSum`, $W_{\texttt{prefixSum}}(n)$, where n is the length of the input list. Give a closed form for this recurrence. Give a big-O bound for $W_{\texttt{prefixSum}}(n)$.

You may use variables $k_0, k_1, \ldots$ for constants. You should assume that `add_to_each` is a linear time function: `add_to_each l` evaluates to a value in kn steps where n is the length of `l` and k is some constant; your recurrence should involve the constant k.

In order to compute the prefix sum operation faster, we will use the technique of adding an additional argument: *harder problems can be easier.*

To do this, you should write a `prefixSumHelp` function that uses an additional argument to compute the prefix sum operation in time that is asymptotically better than the above function. You must determine what the additional argument should be.

Task 4.2 (4%). Give a precise mathematical specification for your `prefixSumHelp` function, which describes the role of the extra argument, by relating `prefixSumHelp` to `prefixSum`. You do not need to prove the relationship, but use it to help you write the code.

Task 4.3 (10%). Write the function. Once you have defined `prefixSumHelp`, use it to define the function

```
prefixSumFast : int list -> int list
```

that computes the prefix sum.

Task 4.4 (5%). Write recurrences

- $W_{\texttt{prefixSumHelp}}(n)$, for the work of `prefixSumHelp`, and
- $W_{\texttt{prefixSumFast}}(n)$, for the work of `prefixSumFast`$(n)$,

in terms of the length n of the input list. Give a closed form for both recurrences, and then give a big-O bound for both.

5 Subset sum

A *multiset* is a slight generalization of a set where elements can appear more than once. A *submultiset* of a multiset M is a multiset, all of whose elements are elements of M. To avoid too many awkward sentences, we will use the term *subset* to mean *submultiset*.

It follows from the definition that if U is a sub(multi)set of M, and some element x appears in U k times, then x appears in M at least k times. If M is any finite multiset of integers, the sum of M is

$$\sum_{x \in M} x$$

With these definitions, the multiset subset sum problem is answering the following question.

> Let M be a finite multiset of integers and n a target value. Does there exists any sub(multi)set U of M such that the sum of U is exactly n?

Consider the subset sum problem given by

$$M = \{1, 2, 1, -6, 10\} \qquad\qquad n = 4$$

The answer is "yes" because there exists a subset of M that sums to 4, specifically

$$U_1 = \{1, 1, 2\}$$

It's also yes because

$$U_1 = \{-6, 10\}$$

sums to 4 and is a subset of M. However,

$$U_3 = \{2, 2\}$$

is not a witness to the solution to this instance. While U_3 sums to 4 and each of its elements occurs in M, it is not a submultiset of M because 2 occurs only once in M but twice in U_2.

We represent multisets of integers as SML values of type `int list`, where the integers may be negative. You should think of these lists as just an enumeration of the elements of a particular multiset. The order that the elements appear in the list is not important.

Task 5.1 (12%). Write the function

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

that returns `true` if and only if the input list has a subset that sums to the target number. As a convention, the empty list `[]` has a sum of 0. Start from the following useful fact: each element of the set is in the subset, or it isn't.[2]

References

[Conway(1987)] J. Conway. The weird and wonderful chemistry of audioactive decay. In T. Cover and B. Gopinath, editors, *Open Problems in Communication and Computation*, pages 173–188. Springer-Verlag, 1987.

[2] *Hint:* It's easy to produce correct and unnecessarily complicated functions to compute subset sums. It's almost certain that your solution will have $O(2^n)$ work, so don't try to optimize your code too much. There is a very clean way to write this in a few (5-10ish) elegant lines.