COMP 212 Fall 2024
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. You should hand in your `hw03.sml` and `hw03-written.pdf` files by uploading them to your Google Drive handin folder.

You must write purposes and tests for all functions on this assignment (and on all future assignments!).

2 Zip/Unzip

Lists can contain elements of any type, not just numbers. For example, `["a","b"] : string list` and `[(1,"a"),(2,"b")] : (int * string) list`.

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

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

“zipping” this together gives the list

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

where each element of one list is paired with the corresponding element of the other (if both lists have that many elements).

Task 2.1 (10 pts). 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 smaller 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 (10 pts). 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 pts). 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 5 notes.

You may assume that for all inputs `l1,l2,l`, `zip(l1,l2)` and `unzip l` are valuable (are equal to values). In particular, this means that

- for any `l1 : int list` and `l2 : string list`, there is a list `l12 : (int * string) list` such that `zip(l1,l2)` $\cong$ `l12`.
- for any `l : (int * string) list`, there are lists l_i `: int list` and l_s `: string list` such that `unzip l` $\cong$ `(`l_i`,`l_s`)`.

You will need to use the latter of these facts to step through the code in your proof: if you get stuck stepping, you can replace `unzip l` with its value, of shape `(`l_i`,`l_s`)` for some lists l_i and l_s.

Task 2.4 (5 pts). 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 contiguous 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 contiguous 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 two inputs

- `l` : `int list`, a list
- `y` : `int`, the number you are looking for in a run

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

- `tail` : `int list` is the tail of `l` after any and all numbers equal to `y` at the front of the list have been removed
- `total` : `int` is the total length of the run of numbers equal to `y` at the front of `l`.

For example,

$$\texttt{lasHelp}([1,2,3],4) \cong ([1,2,3],0)$$
$$\texttt{lasHelp}([2,2,6,3],2) \cong ([6,3],2)$$

Task 3.1 (20 pts). Write the function

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

according to the given specification. Note that you can write `x = y` to compare integers `x` and `y` for equality (the result has type `bool`). 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 is like recurring on `n-d` in the `divmod` problem from last week.

4 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 4.1 (15 pts). 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]

[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.

5 NON-COLLABORATIVE PROBLEM: Inverse of Adjacent Numbers

Remember that non-collaborative problems are to be done independently. You are not allowed to communicate with anyone about the problems, except to ask the instructor or TAs clarification questions (not hints). Additionally, you are not allowed to search for help on the specific problem from any sources besides the course materials. You are free to ask clarification questions about the concepts involved — in this case, recursion, induction, the SML type `real`, which represents floating point numbers — numbers with a decimal point represented using a fixed (roughly 64) number of bits (0 or 1) of information.

Task 5.1 (5 pts). Implement a function
`inverse_adjacent : int -> real`, where `inverse_adjacent(n)` computes the sum

$$\frac{1}{1\times 2}+\frac{1}{2\times 3}+\frac{1}{3\times 4}+\ldots+\frac{1}{n\times(n+1)}$$

That is, the function computes the "the inverse of (the product of) adjacent numbers".

You should write examples in a comment but you do not need to make a `test_inverse_adjacent` function — instead you can test by running the function interactively in SMLNJ using the `show` function in the handout code, which prints a floating point number to 20 decimals, and comparing the answer to the expected output by hand.

Hints:

- You can use the function `real : int -> real` to convert an integer to a floating point number.
- Floating point constants must be written e.g. `1.0` and `2.0` even when they are whole numbers — you can't write `1` as a float.

Task 5.2 (5 pts). It turns out that this sum has a simple alternative description:

Theorem 3. *For natural number values* `n`*,*

$$\textit{inverse\_adjacent(n)} = \frac{n}{n+1}$$

Prove this by induction on `n`. For this part of the problem, you should assume that the floating point numbers represented by the SML type `real` are mathematical real numbers, so you can use the usual properties of addition/multiplication/division.

Task 5.3 (1 pts). Compare `show(inverse_adjacent 200)` with `show(200.0/201.0)`. Explain why what you see does not match the theorem you proved in the previous task.

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.