15-150 Summer 2017
Homework 03

Out: Saturday, 27 May 2017
Due: Wednesday, 31 May 2017, 23:59 EST

1 Introduction

This assignment will focus on writing functions on lists and proving properties of them.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Code submissions will be handled through Autolab, at

`https://autolab.andrew.cmu.edu`

Written submissions will be handled through Gradescope, at

`https://gradescope.com`

To submit your code, run `make` from the `hw/03` directory (that contains a `code` folder. This should produce a file `hw03.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw03.tar` file via the "Handin your work" link.

To submit your written solutions (which *must* be in PDF form), log in to Gradescope and submit your PDF to the assignment **Homework 03**. If you are unable to do this, make a Piazza post or find a TA to help.

The Autolab handin script does some basic checks on your submission: making sure file names are correct; making sure that no files are missing; making sure your code compiles cleanly; making sure your functions have the right types. In addition, the script tests each function that you submit individually using public tests. These public tests are in no way comprehensive, and will in most cases consist of a simple base case test. In order to receive full points, you must pass private tests that will not be run until after the submission deadline has passed, so you will still have to write your own tests to ensure correctness.

The script indicates a correct submission via a score on the Autolab website. If all files are present, your submission will receive a score of “0.0” in the “files” category. If your code compiles, it will receive a score of “0.0” in the “compile” category. If files are missing or the code does not compile, you will receive a highly negative score for the corresponding category. For each function, you will receive a score that represents the number of public tests passed for that function. If you do not receive full points for the public tests for a particular function, you will receive a score of zero for that function.

Your `hw03.sml` file must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Wednesday, 31 May 2017, 23:59 EST. Remember that you ***do not get grace days this semester***, so we will not be accepting late submissions without a documented, university-approved excuse.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

2 take and bake

Consider the following ML function definitions:

```
fun take (i : int, L : int list) : int list * int list =
if i = 0 then ([], L) else
   case L of
     []  => ([], [])
   | x::R => let val (A, B) = take (i - 1, R) in (x :: A, B) end

fun bake (i : int, L : int list): int list * int list =
if i = 0 then ([], L) else
   case L of
     []  => ([], [])
   | x::R => let val (A, B) = bake (i - 1, L) in (x :: A, B) end
```

(The only difference, apart from the names of the functions, is the list used in the recursive call. Although the two definitions are syntactically very similar, the two functions behave very differently.)

Task 2.1 (6 pts). Fill in the following specifications for these two functions, in each case giving a detailed and accurate description of the function's applicative behavior when applied to a pair `(i, L)` that satisfies the assumptions.

```
(* take : int * int list -> int list *  int list  *)
(* REQUIRES  0 <= i <= length L                   *)
(* ENSURES   take (i, L) = a pair of lists (A, B)
                                   such that ...  *)

(* bake : int * int list -> int list *  int list  *)
(* REQUIRES  0 <= i <= length L                   *)
(* ENSURES   bake (i, L) = a pair of lists (A, B)
                                   such that ...  *)
```

Task 2.2 (14 pts). Prove that the `take` function satisfies the specification from Task 2.1. Say clearly what proof method you use, and explain your proof steps with enough detail to show that you know how to write an accurate proof. You may cite the following lemma(s) in your proof:

Lemma 1.
For all `a : int`, `B : int list`, and `C : int list`, `(a :: B) @ C = a :: (B @ C)`.

3 working hard or hardly working

Recall the `take` function from before:

```
fun take (i : int, L : int list) : int list * int list =
if i = 0 then ([], L) else
   case L of
     []  => ([], [])
   | x::R => let val (A, B) = take (i - 1, R) in (x :: A, B) end
```

Task 3.1 (4 pts). Write an ML function `take_work : int * int list -> int` such that for all non-negative integers `i` and integer lists `L`, `take_work (i, L)` evaluates to the total number of `::` operations used to evaluate `take (i, L)`. Note that the `::` operation is used only when building lists; when pattern-matching, we are merely examining the structure of the input and so we don't actually `cons` any element to any list.

Your code should fit in the following template:

```
fun take_work (i : int, L : int list) : int =
if i = 0 then (* FILL IN *) else
   case L of
     []   => (* FILL IN *)
   | x::R => (* FILL IN *)
```

Task 3.2 (3 pts). What are the values of the following expressions?

```
   take_work (1, [1,2,3,4,5,6,7,8,9])
   take_work (2, [1,2,3,4,5,6,7,8,9])
   take_work (42, [1,2,3,4,5,6,7,8,9])
```

Task 3.3 (3 pts). Now that we've seen a few examples of `take_work` on various inputs `i` and `L`, please give a closed-form expression for the value of `take_work (i, L)` and *briefly* justify your answer (which is to say, no formal proof is needed).

4 Heads or Tails

Task 4.1 (3 pts). Write an ML function `heads : int * int list -> int` such that for all integers `x` and integer lists `L`, `heads (x, L)` evaluates to the number of occurrences of `x` at the front of `L`. For example,

```
heads (1, [1,1,2,1,3]) ==>* 2
heads (2, [1,1,2,1,3]) ==>* 0
```

Task 4.2 (3 pts). Write an ML function `tails : int * int list -> int list` such that for all integers `x` and integer lists `L`, `tails (x, L)` evaluates to the list obtained from `L` by deleting initial occurrences of `x`, if any. For example,

```
tails (1, [1,1,2,1,3]) ==>* [2,1,3]
tails (2, [1,1,2,1,3]) ==>* [1,1,2,1,3]
```

Task 4.3 (3 pts). Write an ML function `filterInt : int * int list -> int list` such that for all integers `x` and integer lists `L`, `filterInt (x, L)` evaluates to the list obtained from `L` by deleting all occurrences of `x` in `L`. For example,

```
filterInt (1, [1,1,2,1,3]) ==>* [2,3]
filterInt (2, [1,1,2,1,3]) ==>* [1,1,1,3]
filterInt (3, [1,5,1,5,0]) ==>* [1,5,1,5,0]
```

5 Look and Say

5.1 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 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'll adopt the convention that the look-and-say sequence of `[]` is `[]`.

5.2 Implementation

Task 5.1 (12 pts). Write the function

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

according to the given specification.

HINT: You can use `heads` and `tails` from the previous question to help you find a recursive way to solve this problem.

If you invent your own helper functions, be sure to give clear specifications for them!

Since the look-and-say list is itself a list of integers, we can repeatedly apply look-and-say to generate a fun pattern:

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

Task 5.2 (8 pts). Write the function

```
look_say_table : (int list * int) -> int list list
```

such that `look_say_table (L, n)` evaluates to a list of length $n+1$ of repeated look-and-say lists, *starting with* L. For example, `look_say_table ([1], 5)` should evaluate to the example above.

6 Started From the Bottom

The prefix-sum of a list `l` is a list `s` where the i^{th} index 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]
```

Note that the first element of list is regarded as position 0.

Task 6.1 (5 pts). Implement the function

```
prefixSum : int list -> int list
```

that computes the prefix-sum. You must use the `addToEach` function, which adds an integer to each element of a list. You may NOT use any other helper functions for this task.

Task 6.2 (1 pts). Given a list, `L`, of length n, give an asymptotic bound for the work of `prefixSum L`? You do not need to show your steps for this task.

Task 6.3 (10 pts). Write the `prefixSumHelp` function that uses an additional argument to compute the prefix sum operation. You must determine what the additional argument should be. Once you have defined `prefixSumHelp`, use it to define the function

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

that computes the prefix sum.

Task 6.4 (1 pts). Given a list, `L`, of length n, give an asymptotic bound for the work of `prefixSumFast L`? Once again you do not need to show your steps for this task.

7 With a List-le Help From My Friends

Another useful list concept is that of a subsequence. A list `X` of the form `[`$x_1, \dots, x_n$`]` is considered to be a *subsequence* of a list `Y` if and only if `Y` is of the form `[`$y_1, \dots, y_m$`]` and for every x_k in `X` there exists some y_{i_k} in `Y` such that $x_k = y_{i_k}$ with $i_1 < i_2 < \cdots < i_n$. Moreover, the empty list is a subsequence of every list.

Task 7.1 (6 pts). Implement the function

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

that returns `true` if the first list is a subsequence of the second and returns `false` otherwise. For example:

```
subsequence ([2,4],[1,2,3,4]) = true
subsequence ([1,2],[1,3,2]) = true
subsequence ([3,1],[1,3,2]) = false
subsequence ([],[5,3,4]) = true
```

Finally, we will determine if a list is a subrun of another list. This is useful in some lossless compression algorithms. A list `S` is considered to be a subrun of the list `L` if for some `X : int list, Y : int list, L = X @ S @ Y`

Task 7.2 (6 pts). Implement the function

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

that returns `true` if the first list is a subrun of the second and returns `false` otherwise.

Some examples:

```
subrun ([2,3],[1,2,3,4]) = true
subrun ([2,4],[1,2,3,4,5]) = false
subrun ([2,3,4],[1,2,3,4,5]) = true
subrun ([],[1,2,3,4]) = true
```

8 Sum Nights

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* in this problem. (This is consistent with our earlier definition of subset for lists.)

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 exist any subset U of M such that the sum of the elements in U is exactly n?

Consider the subset sum problem given by

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

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 subset of M because 2 occurs only once in M but twice in U_2.

Representation You'll implement two solutions to the subset sum problem. In both, 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.

8.1 Basic solution

Task 8.1 (10 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.[1]

8.2 NP-completeness and certificates

Subset sum is an interesting problem because it is *NP-complete*. NP-completeness has to do with the time-complexity of algorithms, and is covered in more detail in courses like 15-251, but here's the basic idea:

- A problem is in P if there is a polynomial-time algorithm for it—that is, an algorithm whose work is in $O(n)$, or $O(n^2)$, or $O(n^{14})$, etc.
- A problem is in NP if an affirmative answer can be *verified* in polynomial time.

Subset sum is in NP. Suppose that you're presented with a multiset M, another multiset U, and an integer n. You can easily *check* that the sum of U is actually n and that U is a subset of M in polynomial time. This is exactly what the definition of NP requires.

This means we can write an implementation of subset sum which produces a *certificate* on affirmative instances of the problem—an easily-checked witness that the computed answer is correct. Negative instances of the problem—when there is no subset that sums to n—are not so easily checked.

You will now prove that `subsetSum` is in NP by implementing a certificate-generating version.

Task 8.2 (7 pts). Write the function

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

such that for all values `M:int list` and `n:int`, if `M` has a subset that sums to `n`, `subset_sum_cert (M, n) = (true,U)` where `U` is a subset of `M` which sums to `n`.

If no such subset exists, `subset_sum_cert (M, n) = (false,nil)`. [2]

[1] *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 elegant lines.

[2] You'll note that the empty list returned when a qualifying subset does not exist is superfluous; soon, we'll cover a better way to handle these kinds of situations, called `option` types.