15-150 Summer 2016
Homework 03

Out: Saturday, 21 May 2016
Due: Tuesday, 24 May 2016 at 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

Submissions will be handled through Autolab, at

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

In preparation for submission, your `hw/03` directory should contain a file named exactly `hw03.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/03` directory (that contains a `code` folder and a file `hw03.pdf`). 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.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the "check" section of your latest handin on the "Handin History" page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

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 Tuesday, 24 May 2016 at 23:59 EST. Remember that you do not have late days.

1.4 Methodology

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

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

For example, for the factorial function:

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

fun fact (n : int) : int =
  case n of
    0 => 1
  | _ => n * fact(n-1)

(* Tests: *)

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

1.5 Style

In this and future homeworks, we will also grade your code for style. If your code for a task does not adhere to the style guidelines on the course website, you will temporarily receive a score of 0 for that task. You may then fix the problems with your code style and show it to a TA within two weeks of getting your graded assignment back. If you do this, you will get the number of points you would get if your original submission had proper style. If you do not do this, you will keep the grade of 0 for that task.

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

$$[\text{“}\textit{apples}\text{”}, \text{“}\textit{milk}\text{”}, \text{“}\textit{banana}\text{”}, \text{“}\textit{potato}\text{”}] \qquad \text{and} \qquad [5, 1, 2, 1]$$

we might be interested in the list

$$[(\text{“}\textit{apples}\text{”}, 5), (\text{“}\textit{milk}\text{”}, 1), (\text{“}\textit{banana}\text{”}, 2), (\text{“}\textit{potato}\text{”}, 1)]$$

In `hw03.sml` we have provided you with two functions, `zip` and `unzip`. These functions implement this idea of combining two lists. Here are some examples using `zip` and `unzip`:

```
zip (["apples", "milk"],[5, 1, 2, 1]) ≅ [("apples", 5), ("milk",1)]
unzip [("apples", 5), ("milk",1)] ≅ (["apples", "milk"], [5, 1])
```

Here are the types for the `zip` and `unzip` functions:

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

In your tasks below you may assume that `zip` and `unzip` are total functions.

In these proofs, you will at some point need to reason about a `val` binding that pattern-matches on (takes apart) a pair (i.e. one that looks like `let val (x,y) = e1 in e2 end`). To do this rigorously, you should use the following two equivalence rules and cite them when you use them:

$$\begin{aligned} \texttt{let val (x,y) = e1 in e2} &\cong [\texttt{fst e1}/\texttt{x}][\texttt{snd e1}/\texttt{y}]\ \texttt{e2} && \text{[Rule 1, applies when } e_1 \text{ valuable]} \\ (\texttt{fst e, snd e}) &\cong \texttt{e} && \text{[Rule 2, applies always]} \end{aligned}$$

The first rule says that to evaluate a `val` declaration where `e1` has a pair type ($A \times B$ for some A and B), we can replace x and y with expressions that compute the first and second element of `e1`, called `fst e1` and `snd e2`. As an example usage of this rule, we can compute

```
let val (a,b) = f x in (a + 1 + b * a) end
==
fst (f x) + 1 + snd (f x) * fst (f x)
```

Since the first rule only holds true when `e1` is valuable, you need to ensure that `e1` is actually valuable any time you use this rule. The second rule says that if we have the first and second element of some pair e, then pair them up, that gives us back the original pair e, for example:

$$(\texttt{fst}(1,2), \texttt{snd}(1,2)) \cong (1,2)$$

In these rules, `fst` and `snd` are defined as

```
fun fst(x,y) = x
fun snd(x,y) = y
```

but you should not need to know the implementation of `fst` and `snd` in your proof — Rules 1 and 2 should be sufficient.

Task 2.1 (14 pts). Prove or disprove Theorem 1.

Theorem 1. *For all values* `l : (string * int) list`*,*

$$\texttt{zip(unzip l)} \cong \texttt{l}.$$

Task 2.2 (5 pts). Prove or disprove Theorem 2.

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

$$\texttt{unzip(zip (l1,l2))} \cong \texttt{(l1,l2)}$$

3 Heads or Tails

Task 3.1 (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 3.2 (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 3.3 (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]
```

4 Look and Say

4.1 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."

4.2 Implementation

Task 4.1 (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!

The elements of a list can have any type, as long as they are all the same type. This extends to lists of lists as well! For instance, consider the following examples of values with type `string list list`:

```
val a : string list list =
[
 ["fun","ctions"],
 ["are"],
 ["v", "a", "l", "", "u", "e", "s"]
]

val b : string list list =
["fun", "ctions"] :: ["are"] :: ["v", "a", "l", "", "u", "e", "s"] :: []
```

In this case, a and be would represent the same value. Notice that we can use the cons operator (`::`) to insert lists into a list the same way we did with integers.

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 4.2 (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.

5 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 5.1 (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 5.2 (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 5.3 (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 5.4 (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.

6 With a List-le Help From My Friends

It is often useful to determine if a list is a subset of another list. If `L` is any list, we say that `S` is a *subset* of `L` if and only if all the elements of `S` appear in `L`, counting multiplicities. This means, for example, if some `x` appears 3 times in `S` then it must occur at least 3 times in `L`. For example: `[12,4,3,4]` is a subset of the list `[4,3,24,5,4,12]`. Some other examples:

```
subset([12,4,3],[12,5,6,3,4]) ≅ true
subset([12,4,3,3],[12,5,6,7,4]) ≅ false
subset([12,4,3,3],[12,5,6,7,4,3]) ≅ false
subset([],[]) ≅ true
```

Task 6.1 (6 pts). Implement a function

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

that returns `true` if the first list is a subset of the second and returns `false` otherwise. Hint: you may want to use a helper function.

Another useful list concept is that of a subsequence. A list `X` of the form `[`$x_1, \ldots, x_n$`]` is considered to be a *subsequence* of a list `Y` of the form `[`$y_1, \ldots, y_m$`]` if and only if for every x_k in `X` there exists some y_{i_k} in `Y` such that $x_k = y_{i_k}$ with $i_1 < i_2 < \cdots < i_n$. That is, `Y` contains every element of `X`, the same number of times, in the same order (but they don't have to be consecutive). Moreover, the empty list is a subsequence of every list.

Task 6.2 (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. 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 6.3 (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
```

7 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

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

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.

7.1 Basic solution

Task 7.1 (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]

7.2 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 7.2 (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)` $\cong$ (`true`, `U`) where `U` is a subset of `M` which sums to `n`.

If no such subset exists, `subset_sum_cert (M, n)` $\cong$ (`false`, []). [2]

Task 7.3 (Extra Credit).The P = NP problem, one of the biggest open problems in computer science, asks whether there are polynomial-time algorithms for *all* of the problems in NP. Right now, there are problems in NP, such as subset sum, for which only exponential-time algorithms are known. However, it is known that subset sum is *NP-complete*, which means that if you could solve it in polynomial time, then you could solve all problems in NP in polynomial time, so P = NP. So, for extra credit, several million dollars, and a PhD, define a function that solves the subset sum problem and has polynomial time work.

[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.