15-150 Fall 2015
Homework 03

Out: Wednesday, 16 September 2015
Due: Tuesday, 22 September 2015 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.cs.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 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.

All the code that you want to have graded for this assignment should be contained in `hw03.sml`, 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, 22 September 2015 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

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

For example, for the factorial function presented in lecture:

```
(*  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 6 = fact 3
val 720 = fact 6
```

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

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

we might be interested in the list

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

In `hw03.sml` we have provided you with function `zip`. This function implement the idea of combining two lists. Let `zip` be the function defined by:

```
fun zip ([ ], [ ]) = [ ]
|   zip (x::L, y::R) = (x, y) :: zip (L, R)
```

Let `length` be the usual function for computing the length of a list:

```
fun length [ ] = 0
|   length (a::A) = 1 + length A
```

ML has a built-in infix operator `@` for appending two lists of the same type.

Task 2.1 (2 pts). What is the type of `zip`?

Task 2.2 (2 pts). The ML runtime system complains when you make this function definition. What does it say, and what do you think it means?

Task 2.3 (2 pts). Which of the following specifications does this function satisfy? There is no need for you to prove anything here, but your answer must contain enough detail for us to see that you understand what it means to say that a function does or does not satisfy these specifications.

```
(* Specification 1 *)
(* REQUIRES  true *)
(* ENSURES
    zip(L, R) = a list built by pairing each item of L with an item of R *)

(* Specification 2 *)
(* REQUIRES  length(L) = length(R)  *)
(* ENSURES
    zip(L, R) = a list built by pairing each item of L with an item of R *)

(* Specification 3 *)
(* REQUIRES  length(L) = length(R)  *)
(* ENSURES
    zip(L, R) = a list A such that length(A) = length(L) = length(R) *)
```

Task 2.4 (10 pts). Prove that for all lists $\texttt{A}_1, \texttt{B}_1, \texttt{A}_2, \texttt{B}_2$ of the same type, if

$$\texttt{length}(\texttt{A}_1) = \texttt{length}(\texttt{A}_2) \text{ and } \texttt{length}(\texttt{B}_1) = \texttt{length}(\texttt{B}_2),$$

then

$$\texttt{zip}(\texttt{A}_1\texttt{@}\texttt{B}_1, \texttt{A}_2\texttt{@}\texttt{B}_2) = \texttt{zip}(\texttt{A}_1, \texttt{A}_2) \texttt{ @ } \texttt{zip}(\texttt{B}_1, \texttt{B}_2).$$

(Think carefully about what this says! Your answer should show that you understand what equations like this mean.)

You can use without proof the facts that for all suitably typed values `x`, `A` and `B`,

```
length(A@B) = length(A) + length(B)
```

and

```
x :: (A@B) = (x :: A) @ B.
```

You can either answer using equations and equational reasoning, or by translating the property to be proven into evaluational form and using `=>*`. Your answer must show consistent and accurate use of notation and terminology. If you appeal to any basic equations or facts about equality of evaluation, state them.

3 Counting

List values are built using the "cons" operation (written as infix `::`). For example the value `1::2::3::nil` is built using 3 cons operations. We use list notation like $[1, 2, 3]$ as an abbreviation for 1::2::3::`nil`, so we can also say that $[1, 2, 3]$ is a list value built with 3 cons operations.

In this question, you can assume without proof the facts that:

- When $\texttt{L}_1$ and $\texttt{L}_2$ are list values of the same type, with lengths n_1 and n_2, respectively, $\texttt{L}_1\texttt{@}\texttt{L}_2$ evaluates to a list value of length $n_1 + n_2$ and performs n_1 cons operations. (It conses the items of $\texttt{L}_1$ onto $\texttt{L}_2$.)
- When $\texttt{L}_1$ and $\texttt{L}_2$ are list values of the same length n, $\texttt{zip}(\texttt{L}_1, \texttt{L}_2)$ evaluates to a list of length n and performs n cons operations.

Suppose $\texttt{A}_1, \texttt{A}_2$ are list values of the same type, each with length n, and $\texttt{B}_1, \texttt{B}_2$ are list values of the same type, each with length m.

Task 3.1 (4 pts). How many cons operations are performed in the evaluation of

$$\texttt{zip}(\texttt{A}_1\texttt{@}\texttt{B}_1, \texttt{A}_2\texttt{@}\texttt{B}_2)?$$

Task 3.2 (4 pts). How many cons operations are performed in the evaluation of

$$\texttt{zip}(\texttt{A}_1, \texttt{A}_2) \texttt{ @ } \texttt{zip}(\texttt{B}_1, \texttt{B}_2)?$$

Note that here we are counting "exactly", not asymptotically! There's no big-O in this question, and your answer should also avoid big-O.

4 Hits and Strips

Task 4.1 (5 pts). Write an ML function `hits : int * int list -> int` such that for all integers `x` and integer lists `L`, `hits (x, L)` evaluates to the number of occurrences of `x` at the front of `L`. For example,

```
hits (1, [1,1,2,1,3]) = 2
hits (2, [1,1,2,1,3]) = 0
```

Task 4.2 (5 pts). Write an ML function `strip : int * int list -> int list` such that for all integers `x` and integer lists `L`, `strip (x, L)` evaluates to the list obtained from `L` by deleting initial occurrences of `x`, if any. For example,

```
strip (1, [1,1,2,1,3]) = [2,1,3]
strip (2, [1,1,2,1,3]) = [1,1,2,1,3]
```

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

$$[(2, 3)]$$

because l is exactly "two thrice.". Similarly, the look-and-say sequence of

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

is

$$[(1, 1), (2, 2)]$$

because l is exactly "one once, then two twice."

Task 5.1 (15 pts). Write an ML function `look_and_say : int list -> (int * int) list` such that for all integer lists `L`, `look_and_say L` evaluates to a list of pairs of integers, where each pair consists of an integer from `L` paired with the number of consecutive occurrences of that integer in the list. For example,

```
look_and_say [1,1,2,2,2,1] = [(1,2),(2,3),(1,1)]
```

because the list `[1,1,2,2,2,1]` is "1, 2 times, 2, 3 times, 1, 1 time". Note that each consecutive "run" of 1 contributes a pair to the look-and-say list.

Your function should have the property that every pair has form `(i,j)` where `j` > 0, and no consecutive pairs have the same `i` component. (For example, lists `[(1,~2),(2,2)]` and `[(1,1),(1,2),(2,3)]` are invalid.) A list of pairs of integers that has this property is called a "look-and-say list".

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

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

Task 5.2 (10 pts). ML has a type `string`, whose values are strings, for example `"foo"` and `"'s "`. The built-in infix operator `^` is used to concatenate strings. For example, the ML expression

```
"foo" ^ "bar"
```

has type `string` and its value is the string `"foobar"`.

ML also has a built-in function `Int.toString : int -> string` for producing a string representation of integer values. For example,

```
Int.toString (21 + 21) = "42".
```

Write an ML function

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

so that when `L` is look-and-say list, `display L` evaluates to a string that "describes" what the list "says" using a simple descriptive format. For example,

```
display [(1,2),(2,3),(1,1)] = "2 1's 3 2's 1 1's"
```

Don't try to be clever by dropping the substring `"'s"` when the number of occurrences is 1. It's OK for your function to have

```
display [(1,1),(2,2)] = "1 1's 2 2's"
```

rather than the more colloquial `"1 1 2 2's"`.

Don't worry about cases where the look-and-say list has silly entries like `(1, ~12)` (meaning `~12` occurrences of 1) or consecutive entries like `(1,2)` immediately followed by `(1,3)` – there's no need to combine them into a single pair `(1,5)`. It's OK to have

```
display [(1,~12),(1,2),(1,3)] = "~12 1's 2 1's 3 1's".
```

Note: you can use the ML function `print : string -> unit` to examine the results produced by your function.

6 Prefix Sums

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 elements of `l`. (Note that the first element of list is regarded as position 0.) For example,

```
prefix_sum [] = [0]
prefix_sum [1,2,3] = [0,1,3,6]
prefix_sum [5,3,1] = [0,5,8,9]
```

We give you the definitions for `prefix_sums` and its helper function `add_to_each`.

```
fun add_to_each (x:int, [ ]:int list) : int list = [ ]
|   add_to_each (x, y::L) = (x+y) :: add_to_each (x+y, L)

fun prefix_sums [ ] = [0]
|   prefix_sums (x::L) = 0 :: add_to_each(x, prefix_sums L)
```

Task 6.1 (2 pts). Find recurrence relations for

$$W_{\texttt{add\_to\_each(n)}}$$
$$W_{\texttt{prefix\_sums(n)}}$$

i.e. the work to evaluate `add_to_each(x, L)` when `x` is an integer value and `L` is an integer list of length `n`, and the work to evaluate `prefix_sums L` when `L` is an integer list of length `n`.

Task 6.2 (4 pts). What are the big-O estimates of the solutions of these recurrence relations?

Now we give you

```
fun prefix_sums_helper ([ ], a) = [a]
|   prefix_sums_helper (x::L, a) = a :: prefix_sums_helper (L, x+a)

fun fast_prefix_sums L = prefix_sums_helper(L, 0)
```

where `fast_prefix_sums L`=`prefix_sums L` for all valid `L`. Function `prefix_sums_helper` takes an additional argument of type `int` that keeps track of sum of integers it's seen as it goes down the list.

Task 6.3 (5 pts). Show that the work to evaluate `fast_prefix_sums L`, when `L` is a list value of length `n`, is in `O(n)`.

Task 6.4 (5 pts). Note that for all $n \geq 0$ and all integers $x_1, \ldots, x_n$,

$$\texttt{prefix\_sums}[x_1, \ldots, x_n] = [0, y_1, \ldots, y_n]$$

where $y_1 = x_1, y_2 = x_1 + x_2, \ldots, y_n = x_1 + \cdots + x_n$.

Use `prefix_sums` and `zip` to write an ML function

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

such that for all $n \geq 0$ and all integers $x_1, \ldots, x_n$,

$$\texttt{pair\_with\_prefix\_sum}\ [x_1, \ldots, x_n] = [(x_1, y_1), \ldots, (x_n, y_n)]$$

where $y_1 = x_1, y_2 = x_1 + x_2, \ldots, y_n = x_1 + \cdots + x_n$.

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.

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 one 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 (15 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]

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.