15-150 Fall 2013
Homework 03

Out: Wednesday, 11 September 2013
Due: Tuesday, 17 September 2013 at 23:59 EST

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!

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** -10.0**, your submission failed the check script; if it is** 0.0**, 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, 17 September 2013 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, 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 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 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

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

we might be interested in the list

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

Task 2.1 (5%). Write the function

```
zip : string list * int list -> (string * int) 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 lengths, 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).

Some terminology:
A function `f : t -> t'` is *total* iff the expression `f(x)` reduces to a value for all values `x : t`.

Task 2.2 (5%). Write the function

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

`unzip` does the opposite of `zip`, in the sense that it takes a list of pairs and returns a pair of lists, where the first list in the pair 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. You may assume this fact in your proofs but should cite it if used.

Task 2.3 (15%). Prove the following Theorem by induction on length of lists. 1.

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

Hint: Look at the way we reasoned about `eval decimal n = n` in Lecture 3 and 4 notes.

Solution 2.3

Proof. This proof will be by simple induction over length on l

Base Case for `[]`:
To Show: `zip (unzip []) ≅ []`.

`zip(unzip []) => zip([], [])`step upzip, clause 0`zip([],[]) => []` step zip, clause 0

We have shown that `zip (unzip []) = []`. This completes the base case.

Inductive Step for `x::xs`:
Inductive Hypothesis: For all xs where `length xs < length x::xs`, `zip(unzip(xs))` holds.

To show: `zip (unzip x::xs) = x::xs`. Because x is a value of type `string * int`, `x = (i,j)` for some string i and int j.

Thus, we can calculate as follows:

```
unzip (i,j)::xs = let val (l1,l2) = unzip xs in (i:
```

By the totality of `unzip`, we know that `unzip l` reduces to a value, `unzip l = (l1',l2')` for some two lists l_1' and l_2'. Using this, we continue evaluating the ML:

```
let val (l1,l2) = (l1',l2') in (i::l1,j::l2)(
```

We will now continue to evaluate `zip`

```
zip (i::l1',j::l2')(fn (x::xs, y::ys) => (x, y)::zip(xs,ys)) (i::l1', j::l2')(i, j)
```

Since we know (i, j) = x, we have shown what we set out to show.

□

```
Rubric, as decided by F2013
2.3:
  Base Case
    +1 stepping zip
    +1 stepping unzip
  Induction Step
    +2 IH
    +1 explicitly starting and ending with L,
         noting x::xs=L if inducting on length
    +1 observing x = (a, b) for some a:int, b:string
    +2 citing IH
    +2 citing totality of unzip (or unpack from equality rules and IH)
    +3 stepping
    +1 conclusion
    -1 for stating one type of induction and using the other
```

Task 2.4 (5%). Prove or disprove Theorem 2.

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

```
unzip(zip (l1,l2)) = (l1,l2)
```

Solution 2.4 This is false. Consider the case where `l1 = [1,2]` and `l2 = ["apple"]`. Then `unzip(zip(l1,l2))` evaluates as follows:

```
  unzip(zip([1,2],["apple"]))
≅ unzip(case ([1,2],["apple"]) of ...  (1::[2],"apple"::[]) => ...)
≅ unzip((1,"apple")::zip([2],[]))
≅ unzip((1,"apple")::[])
≅ ([1],["apple"])
```

`[1]` does not equal `[1,2]`. Therefore `unzip(zip(l1,l2))` does not always evaluate to the original lists `(l1,l2)`.

3 Look And Say

3.1 Definition

If l is any list of integers, the look-and-say list of l is obtained by reading off adjacent groups of identical elements in l. For example, the look-and-say list of $l = [2, 2, 2]$ would be read as "three twos". For our implementation of look-and-say, this would be represented as $[(3, 2)]$.

Similarly, the look-and-say list of

$$l = [1, 1, 2] \qquad \text{is} \quad [(2, 1), (1, 2)]$$

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

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 `lookSay` that computes the look-and-say sequence of its argument using a helper function and a new pattern of recursion.

3.2 Implementation

Before defining the `lookSay` function, you will write a helper function `runWith`. `runWith` will remove a "run" from the front of a list.

Task 3.1 (5%). Write the function `runWith` that satisfies the following spec:

```
runWith: int*int list -> int list * int list

runWith (x, L) = (L1, L2)
   where L = L1@L2
   and every element of L1 is equal to x
   and L2 does not begin with x.
```

Note that L2 is either [ ] or has the form y::R where x $\neq$ y.

For example,

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

Note that you can use the function `inteq` in `hw03.sml` to compare integers for equality.

Task 3.2 (10%). Now, write the function

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

using this helper function.[1]

Solution 3.2 See `hw03-sol.sml`

Task 3.3 (3%). Write the function

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

that will "flatten out" the list of pairs into a flat list of integers.

For example,

```
flatten([(1, 2)]) = [1,2]
flatten([(1, 2), (3, 4), (5, 6)]) = [1,2,3,4,5,6]
```

3.3 Cultural Aside

Repeated applications of our "look and say" function, followed by our "flatten" function, results in a special sequence.

Below are the first 5 steps in the instance of this sequence beginning with [1]:

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

This sequence is noted by Conway's theorem, which states that any element of this sequence will "decay" (by repeated applications of `lookSay` and `flatten`) 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 [**?**] or other papers about the "look and say" operation.

[1] *Hint:* The recursive call in the inductive case of `lookSay` will sometimes be on a list that is more than one element shorter.

4 Prefix-Sum

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 4.1 (5%). Implement the function

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

that computes the prefix-sum. You must use the `addToEach` function provided, which adds an integer to each element of a list, and your solution must be in $O(n^2)$ but *not* in $O(n)$. This implementation will be simple, but inefficient.

Solution 4.1 See `hw03-sol.sml`

Task 4.2 (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. Argue that your closed form does indeed indicate that $W_{\texttt{prefixSum}}(n)$ is $O(n^2)$.

You may use variables $k_0, k_1, \ldots$ for constants. You should assume that `addToEach` is a linear time function: `addToEach 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.

Solution 4.2

$$W_{\text{PS}}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 n + W_{\text{PS}}(n-1) & \text{if } s > 0 \end{cases}$$

The closed form of this recurrence is

$$W_{\text{PS}}(n) = k_2 n^2 + k_2 n + k_0$$

We have seen in lecture that this type of recurrence is in the set $O(n^2)$. Thus `prefixSum` runs in $O(n^2)$ time.

The above is a fine answer to a "what is the recurrence" question. However, if you are curious how we derived this recurrence from the code, here is an explanation: In the case for [], the code steps as follows:

```
prefixSum []
==> case [] of [] => [] | x :: xs => x :: add_to_list (prefixSum xs, x)
==> []
```

Therefore, $W_{\text{PS}}(0) = 2$, which is constant.

In the case for `x::xs`, we calculate as follows:

```
prefixSum (n::l)
==> case n::l of nil => nil | x :: xs => x :: add_to_list (prefixSum xs, x)
==> x :: add_to_list (prefixSum xs, x)
==> x :: add_to_list (l1, n)
==> l2
```

As it takes $W_{\text{PS}}(n-1)$ steps to evaluate `prefixSum l` and it takes $k(n-1)$ steps to evaluate `add_to_list`, it follows that $W_{\text{PS}}(n) = 2 + k(n-1) + W_{\text{PS}}(n-1)$.

It is sufficient to replace the numbers with constants, k_0 and k_1, yielding the above recurrence.

To find the closed form of the recurrence, we can visualize it by expanding the recurrence as in lecture. By doing so we get

$W_{\text{PS}}(n) = k_1 n + k_1(n-1) + k_1(n-2) + ... + k_0$

This is equivalent to $\sum_{i=0}^{n} k_1 i + k_0$. We can expand this using properties of aritmetic and sums to get

$$\begin{aligned}\sum_{i=0}^{n} k_1 i + k_0 &= k_1 \frac{n(n+1)}{2} + k_0 \\ &= \frac{k_1}{2} n^2 + \frac{k_1}{2} n + k_0 \\ &= k_2 n^2 + k_2 n + k_0\end{aligned}$$

This closed form is $O(n^2)$, as it is clearly quadratic, and the n^2 term will dominate for large n.

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

Task 4.3 (10%). Write the `prefixSumHelp` function that uses an additional argument to compute the prefix sum operation in linear time. 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.

Solution 4.3 See `hw03-sol.sml`

Task 4.4 (5%). Write a recurrence for the work of `prefixSumFast`, $W_{\texttt{prefixSumFast}}(n)$, where n is the length of the input list. Give a closed form for this recurrence. Argue that your closed form does indeed indicate that `prefixSumFast` is in $O(n)$.

Solution 4.4 Let $W_{\text{PSH}}(n)$ be the time for `prefixSumHelp` on a list of length n, and $W_{\text{PSF}}(n)$ be the time for `prefixSumFast` on a list of length n.

$$W_{\text{PSH}}(n) = \begin{cases} k_0 & \text{if } n = 0 \\ k_1 + W_{\text{PSH}}(n-1) & \text{if } n > 0 \end{cases}$$

$$W_{\text{PSF}}(n) = k_2 + W_{\text{PSH}}(n)$$

The closed form of this recurrence is

$$W_{\text{PSF}}(n) = k_3 + k_1 n$$

Clearly this closed form is $O(n)$, as the dominating term is a factor of n for large values of n. Thus `prefixSumFast` must run in $O(n)$ time.

5 Sublist

When programming with lists, we often need to work with a segment of a larger list. For example, one might need to access only the last three elements of a list or only the middle element. Any such segment is called a *sublist*.

More formally: if `L` is any list, we say that `S` is a sublist of `L` starting at i if and only if there exist `l1` and `l2` such that

$$\texttt{l1@S@l2} = \texttt{L}$$

and

$$\texttt{length l1} = \texttt{i}$$

For example, $[1, 2]$ is a sublist of $[1, 2, 3]$ starting at 0 because

$$[\,]@[1,2]@[3] = [1,2,3] \qquad \text{and} \qquad \texttt{length}\ [\,] = 0$$

Task 5.1 (3%). The spec for a function sublist that computes sublists as defined above will have the form:

> For all `l:int list`, `i:int`, `k:int`, if _____ then there exists an `S` such that `S` is the sublist of `l` starting at `i`, and
>
> $$\texttt{length S} = \texttt{k} \qquad \text{and} \qquad \texttt{sublist}(\texttt{i},\texttt{k},\texttt{l}) = \texttt{S}$$

The blank is called the *preconditions*, and represents assumptions about the input. Fill in the blank to complete this spec correctly. Note that this formal spec should very closely resemble any REQUIRES and ENSURES clauses on your function.

Solution 5.1 There are three preconditions that must hold for `sublist` to work properly.

- The index i must be non-negative, meaning `0 <= i`.
- The length of the sublist k must be non-negative, meaning `0 <= k`.
- The sublist must be contained within the list `L`. That is, `i + k <= length L`

Task 5.2 (7%). Implement a function

```
sublist : int * int * int list -> int list
```

that meets the spec you gave above.

Because the spec has the form of an implication, in the body of `sublist` you should assume that whatever preconditions you required in Task 5.1 are met: if they are not, your function can do anything you want and still meet its spec!

Note that the definition above implies that we index lists from zero, so

```
sublist (0, 3, [1,2,3,4]) = [1,2,3]
```

Solution 5.2 See `hw03-sol.sml`

The spec that you completed above is good because it closely mirrors the abstract notion of a sublist, but bad because it's very stringent: any code calling sublist must ensure that the assumptions about the input hold or else it will fail. Since the exact mode of failure is not documented in the type or in the spec, this can produce behaviour that's very hard to debug.

Sometimes, the caller will be able to prove that these assumptions hold because of other specification-level information. Other times, the information available at compile-time will not be enough to ensure that these assumptions are met. In these circumstances, you can use a run-time check to bridge the gap, which is something we will be able to implement once we see exceptions.

6 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 subset U of M such that the sum of U is exactly n?

Consider the subset sum problem given by

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

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 three solutions to the subset sum problem. In all three, 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.

6.1 Basic solution

Task 6.1 (10%). Write the function

```
subsetSum : 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]

Solution 6.1 See `hw03-sol.sml`

6.2 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 6.2 (7%). Write the function

```
subsetSumCert : 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`, `subsetSumCert (M, n) =` (`true`, `U`) where `U` is a subset of `M` which sums to `n`.

If no such subset exists, `subsetSumCert (M, n) =` (`false`, `nil`). [3]

Solution 6.2 See `hw03-sol.sml`

Task 6.3 (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.

[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.

[3] 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.

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 in SML a function that solves the subset sum problem and has polynomial time work.