15-150 Spring 2016
Lab 3

27 January 2016

The goal for the third lab is to make you more comfortable writing functions that operate on lists, and doing asymptotic analysis and proofs.

Take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. Today in lab we encourage you to collaborate with your classmates and to ask the TAs for help.

Remember to follow the methodology for writing functions—specifications and tests are part of your code!

1 Introduction

1.1 Getting Started

Update your clone of the `git` repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

2 List Operations

Task 2.1 Write a function

```
evens : int list -> int list
```

that filters out all odd elements of a list without changing the order. For example,

$$\texttt{evens}[0,0,4] = [0,0,4]$$
$$\texttt{evens}[] = []$$
$$\texttt{evens}[0,0,4,9,3,2] = [0,0,4,2]$$

You should use the function `evenP` that we provided at the top of your `lab03.sml` file to determine if a number is even.

Task 2.2 That's one rad function!

```
coolSort: (int * string) list -> (int option) list
```

Write a function of the above type that takes in a list of tuples L and returns a list where the element at index i is SOME(s) if there exists and element (i,s) in L and NONE otherwise.

Task 2.3 What is the work of this function? What would the work be if we returned a data type that had constant time access?

3 Proving Termination

Consider the function `foo : int list -> int list` given by

```
fun foo (L: int list) : int list =
    case L of
        [ ] => [ ]
     |  (x::R) => x :: rev(foo R)
```

where `rev` is a given function of type `int list -> int list` such that for all integer lists `L`, `rev(L)` evaluates to the reverse of list `L`.

Task 3.1 Prove the following theorem by induction on the length of `L`

Theorem: For all values `L : int list`, `foo(L)` terminates.

You may use the following lemmas as facts in your proof, but be sure to cite them.

Lemma 1: If `L : int list` and `L` is a value, then `L = [ ]` if `length(L) = 0` and `L = x::R` for some `x : int, R : int list` if `length(L) > 0`

Lemma 2: For all values `L : int list`, `rev L` terminates and `length(rev L) = length(L)`.

You may assume `length(L)` is defined by

```
    fun length([ ]) = 0
      | length(x::R) = 1 + length(R)
```

Theorem:

Proof: By __________________ (method) on ___________________ (variable/type)

Base Case (   ):

NTS (Need to Show):

Inductive Step (   ):

IH (Inductive Hypothesis):

NTS (Need to Show):

Have the TAs check your work in the previous section before proceeding.

4 Fibonacci

4.1 Simple Fibonacci

Name that integer sequence:

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

That's right; it's Fibonacci!

Here's the obvious way to implement it:

```
fun fib (0 : int) : int = 1
  | fib (1) = 1
  | fib (n) = fib (n - 1) + fib (n - 2)
```

We're going to add the harmless but slightly strange base case defining the negative first element of the sequence to the definition as well; you'll see why later in lab, but just go with it for now.

We can use `fib`'s recursion pattern (that is, the different cases it covers based on its input values, as well as the ways we make recursive calls within `fib`) to think critically about its performance. One of our strongest tools to do this is *writing recurrences* (recurrences let us model the flow of data from one call to a function to another, or to a base case).

Recurrences for functions of two cases on natural numbers are usually of the form:

$$\begin{aligned} &W_{\texttt{fn}}(0) = k_0 \\ &W_{\texttt{fn}}(n) = k_1 + W_{\texttt{fn}}(<\text{ new size }>) \text{ for non-zero } n \end{aligned}$$

That is, we write a case in our recurrence relation for each case of the function (the base cases and the recursive cases). Each of the base cases does some trivial *constant* amount of work, so we denote this using a constant (usually, something like k_i for some previously-unused value i). In our recursive case, we might do some amount of constant work (which we again denote by the term k_1), or even some amount of work dependent on the value of n (which we denote using the variable n multiplied by some amount of work), but we additionally do more work in the form of a *recursive call.* We show this by re-referencing the recurrence relation $W_{\texttt{fn}}$, but with an argument representative of the relative size of the new argument to `fn`. For example, if every recursive call decreases n by 1, our value for <new size> might be $n - 1$.

Task 4.1 Write a recurrence relation for the work of `fib` from above. We've written the cases for you; all you need to do is fill in the right-hand-side of each equation. Note: we left out the case for `~1`; don't worry about it!

$W_{\texttt{fib}}(0) =$
$W_{\texttt{fib}}(1) =$
$W_{\texttt{fib}}(n) =$

This is not so helpful, since it says that the time to compute the n^{th} Fibonacci n is proportional to the n^{th} Fibonnaci number!

However, if we can get an *upper bound* for this recurrence as follows:

$$
\begin{aligned}
&W_{\texttt{fib}}(0) = k_0\\
&W_{\texttt{fib}}(1) = k_1\\
&W_{\texttt{fib}}(n) \leq k_2 + 2W_{\texttt{fib}}(n-1)\text{for non-zero } n
\end{aligned}
$$

Because $W_{\texttt{fib}}(n)$ is *monotonically increasing* (it's never smaller on bigger inputs), we can pretend that it's two recursive calls on $n-1$.

If you write it out, you can see that the closed form of this recurrence is

$$W_{\texttt{fib}}(n) = k_0 + k_1 + k_2 * (2^{n+1} - 1)$$

To see this, you can write the recursion out as a tree. `fib` does `k2` work at each recursive call, so we can label each node with `k2`. Each node has two children, because each call makes two recursive calls.

```
      k2
   k2      k2
 k2 k2   k2 k2
...
```

The k_2 is uniform, so factor it out

```
      1          1
   1      1      2
 1   1  1   1    4
...
```

We want to count the number of nodes in this tree. The total has the form 1+2+4+8+16+.... The reason is that the tree has n levels, because the recurrence recurs on $n-1$, and the i^th level has 2^i work. Thus, the total amount of work is

$$\sum_{i=0}^{n} 2^i$$

If you look it up, the closed form of this sum is $2^{n+1} - 1$ (cf. how many binary numbers are there with n bits).

Once you've written out the closed form, it's clear that this recurrence is $O(2^n)$, just by forgetting the constants. The "O" means "order". Roughly speaking, it measures the most significant (as in largest) term of a function as inputs to the function get large. We will talk more about it in lecture on Thursday.

4.2 Fast Fibonacci

In this problem, you will show that you can compute Fibonacci more efficiently. The key insight is that one of the recursive calls can be reused each time:

```
To compute     We need
fib n          fib (n-1) and fib (n-2)
fib (n-1)      fib (n-2) and fib (n-3)
fib (n-2)      fib (n-3) and fib (n-4)
```

So we really don't need two recursive calls, if we reuse the same computation of `fib n` the two times we use it. To implement this, you must *generalize* the problem so that we compute both `fib n` and `fib (n-1)`.

4.2.1 Programming

Task 4.2 Implement a function

```
  fastfib : int -> int * int
```

such that for all natural numbers n, `fastfib` $n = (\texttt{fib}(n-1), \texttt{fib}\ n)$

4.2.2 Analysis

Task 4.3 Write a recurrence for the work of `fastfib`, $W_{\texttt{fastfib}}$.

Task 4.4 Compute and informally justify the closed form for your recurrence.

Task 4.5 Give a tight big-O bound for this closed form.

Have a TA check your code and analysis for `fastfib`.