15-150 Spring 2012
Lab 3

February 1, 2012

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

Take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are encouraged 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! When writing tests for functions that can raise exceptions, you currently only need to write tests for the cases that don't raise exceptions.

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 Evens

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,

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

You should use the function `evenP` that we provided from last lab to determine if a number is even.

3 Fibonacci

3.1 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 (n : int) : int =
    case n of
       ~1 => 0
     | 0 => 1
     | 1 => 1
     | _ => 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.

Like `evenP` in lab last week, this function has *three* useful cases—zero one and $2 + n$ (we won't count the negative one case). The new thing about this function is that it makes recursive calls not just on $n - 2$ but also on $n - 1$.

Because of these two recursive calls, the recurrence for the work looks like this:

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

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

$$\sum_{i=1}^{n} 2^i$$

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.

3.2 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 computaton 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)`.

3.2.1 Programming

Task 3.1 Implement a function

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

such that for all nats n, `fastfib` $n \cong$ (`fib`$(n-1)$, `fib` n)

3.2.2 Analysis

Task 3.2 Write a recurrence for the work of `fastfib`,
$W_{\texttt{fastfib}}$.

Task 3.3 Compute and informally justify the closed form for your recurrence.

Task 3.4 Give a tight big-O bound for this closed form.

3.2.3 Proof

Task 3.5 Prove that your code meets the spec, which is to say:

Theorem 1. *For all natural numbers* `n`, `fastfib n` $\cong$ `(fib (n - 1) , fib n)`

Use the template on the following page. Have your TA check your work once you finish.

Theorem 2. *For all natural numbers* `n`, `fastfib n` $\cong$ `(fib (n - 1) , fib n)`

The proof is by induction on m.

- **Case for** 0

  To show:

  Proof:

- **Case for** $1 + k$

  Inductive hypothesis:

  To show:

  Proof:

Have a TA check your code, analysis, and proof for `fastfib` before proceeding.

4 Merging Two Lists

Task 4.1 Write a function

```
merge : int list * int list -> int list
```

that merges two sorted lists into one sorted list. You should assume that your input lists are sorted in increasing order, and the list you return should also be in increasing order.

Task 4.2 Write a recurrence relation for the work of `merge`, in terms of the lengths of `l1` and `l2`. What is the O of this recurrence?

Task 4.3 Prove the following correctness theorem about `merge`:

Theorem 3. *For all lists of integers `l1` and `l2`, if `l1` and `l2` are both sorted in increasing order, then `merge (l1, l2)` is sorted in increasing order.*

Hint: In your proof, you will need a lemma about how the contents of `merge(l1,l2)` relates to `l1` and `l2`. You should state this lemma, and convince yourself it is true, but you don't need to prove it formally.