# 15-150 Fall 2011
# Lab 3

September 14, 2011

The goal for the third lab is to make you more comfortable writing functions in SML that operate on lists and proving them correct.

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 Getting Started

Please download the `lab03.sml` file from the course Web site or copy it from the lab afs directory

`/afs/andrew.cmu.edu/course/15/150/asgn/lab/03/`

This file contains some of the functions that were discussed in the last two lectures.

FIXME: directions on running SML in emacs

## 2 Functions on Lists of Integers

### 2.1 Appending Lists

The "cons" function `::` adds one new element to a list. What if you want to *append* a whole list onto the front of another? Appending a list `l1` to another list `l2` evaluates to a list that contains all of the elements of `l1` in the same order order, followed by all of the elements of `l2`, also in the same order. For example,

```
append([1,2,3], [5,13,5]) ==> [1,2,3,5,13,5]
```

A simple implementation of `append` takes elements off of `l1` one at a time, consing them onto the result of appending the rest of the list to `l2`.

**Task 2.1** Write the function

```
append : int list * int list -> int list
```

that behaves according to the specification given above.

**Task 2.2** Write a recurrence relation for the work of `append`, in terms of the lengths of `l1` and `l2`. What is the $O$ of this recurrence?

For future reference, note that append is built in to the basis library of SML in the form of the `@` operator. The expression `l1 @ l2` appends `l1` to `l2`.

## 2.2 Partitioning a List

One of the most important parts of an implementation of QuickSort is the `partition` function that partitions a list around a pivot element. Partitioning a list `L` around an element `p` returns two lists: one containing all of the elements of `l` less than or equal to `p`, and one containing all of the elements of `l` greater than `p`. Moreover, these elements should be in the same order as in `l`.

**Task 2.3** Write the function

```
partition : int * int list -> int list * int list
```

such that

**Theorem 1.** *For all `p : int` and `l : int list`, `partition (p, L)` evaluates to `(less, greater)` such that*

1. *`less` contains all and only the elements of `l` that are less than or equal to `p`, and*
2. *`greater` contains all and only the elements of `l` that are greater than `p`, and*
3. *both `less` and `greater` are sublists of `l`*

Now that you have implemented `partition`, it is time to prove that it satisfies the specification.

**Task 2.4** Write a recurrence relation for the work of `partition`, in terms of the length of $l$. What is the $O$ of this recurrence?

**Task 2.5** Prove that your implementation of `partition` satisfies the above specification. This is a good opportunity to try out *proof-directed debugging*: if you can't do the proof, think about whether the reason is a bug in your code. Follow the template on the next page.

**Case for `[]`**
To show:

**Case for `x::xs`**
Inductive hypothesis:

To show:

## 2.3 Merging Two Lists

**Task 2.6** 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 2.7** 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 2.8** Prove the following correctness theorem about `merge`:

**Theorem 2.** *For all lists of integers `l1` and `l2`, if `l1` and `l2` are both sorted in increasing order, then the list `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.

**Case for `[]`**
To show:

**Case for `x::xs`**
Inductive hypothesis:

To show: