# 15-150 Fall 2013
# Lab 10

30 October 2013

## 1 Introduction

This lab is meant for you to delve further into the modular system, get experience with functors which are functions on modules, and get familiar with the invariants of red-black trees.

### 1.1 CM

In this lab, because we're dealing with lots of files containing structures, everything will be orchestrated by CM (SML's Compilation Manager). This means you'll be editing the `sources.cm` file as needed, and running `CM.make "sources.cm"` to load your code.

One quirk of `CM` is that it will give you warnings if you have code that exists outside of a structure. It will evaluate that code and fail to compile if it doesn't type check, but it will never introduce any bindings from it into the environment. To avoid this annoying behavior, it's best to just put everything inside structures—even if they don't ascribe to a signature.

**Remember to add the files you implement the lab in (in this case `dictset.sml`, `rbtree.sml`, `setofsets.sml` and `pairset.sml`) to your `sources.cm` file!**

### 1.2 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`).

### 1.3 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have REQUIRES and ENSURES clauses and tests.

## 2 Sets With Functors

Recall from lecture, a *functor* is analogous to a function that takes in a module as an argument and returns another module. In lab last week, you implemented sets in two different ways as structures that ascribed to the SET signature. In those implementations, you were told to assume that the set contained integers and so elements of the set could be compared by `op=`.

An interesting problem to think about is how you would compare elements in a set if you did not know what their type will be. This problem can be solved by the useful idea of packaging up a type with a function that checks equality. An example of this is the EQUAL signature that can be seen in `equal.sig`.

This is an example of a type class, which is an important use of signatures where you describe a type equipped with a (probably non-exhaustive) collection of operations on it.

When the type of the element being stored is not known, a set can be implemented as a functor that accepts a structure that ascribes to the signature `EQUAL` and returns a structure that ascribes to the signature `SET`. However, we're going to take a different approach - implementing them as dictionaries where the keys are the elements of the sets and all map to unit.

A dictionary is implemented as a functor that takes in a structure ascribing to signature `EQUAL` and returns a structure that ascribes to signature `DICT`. A set will then be a functor whose argument will be a structure that ascribes to signature `DICT` and returns an implementation of the `SET` signature. The implementation of a dictionary is given to you in `dict.sml` which you should know about and use later in the lab but for the purpose of creating your set you only need to refer to `dict.sig`.

**Task 2.1** Implement the functor `DictSet` (found in `dictset.sml`) that takes as argument a structure ascribing to `DICT` and returns an implementation of the `SET` signature.

# 3 Red-Black Trees

Now we take a short break from functors to discuss red-black trees. In lecture you saw an implementation of red-black trees using functors to allow any key type you want so long as you create a structure ascribing to `ORDERED` to represent that key type. We will quickly recap the structural invariants of red-black trees here:

1. Nodes can be only red or black.
2. The tree is sorted according to the `compare` function.
3. No red node has a red node as a child.
4. Any path from a given node to `Empty` contains the same number of black nodes.
5. `Empty` is considered to be black.

In the file `rbtree.sml` we have provided some minimal code to have working red-black trees. To allow any key type, we would have to implement a functor, but for simplicity of testing we have allowed you to assume that the type of the key is `int`. Also note that the first invariant is implicit in how we have defined our red-black trees.

**Task 3.1** Implement the function `isRBT` in `rbtree.sml` such that when given a valid binary tree `isRBT` returns `true` if and only if the input is a red-black tree satisfying all of the invariants listed above. Note that you may not assume anything about the type of the key in the implementation, but you may assume the tree is correctly ordered.

# 4 Fun With Sets

Now that you have implemented sets that work for a generic type, you can perform some interesting set operations. However, some of these operations would require you to implement additional types of sets. In this problem we're going to construct those types of sets, but not actually use them to implement any new set operations.

One example of this is the power set (which is an example of a set containing sets). As you know, the power set of a set $S$, $\mathcal{P}(S)$, is the set of all possible subsets of the set $S$. However, to take the power set of a set, you need to be able to represent sets of sets.

**Task 4.1** Implement a set of sets as a functor `SetOfSets` (can be found in `setofsets.sml`) that takes in a structure $S$ with signature `SET` and creates another structure with signature `SET` that represents a set of sets (where each inner set uses the set implementation $S$).

**Task 4.2** Test your implementation of `SetOfSets` by passing sets of different types to it as arguments.

Another interesting type of set is the Cartesian Product set. The cartesian product of two sets $A$ and $B$ denoted $A \times B$ is the set of all $(a, b)$ ordered pairs such that $a \in A$ and $b \in B$. To take a cartesian product you need to be able to represent a set of pairs, where each pair has one element from $A$ and one from $B$.

**Task 4.3** Implement sets of pairs as a functor `PairSet` (can be found in `pairset.sml`) that takes in two structures $E1, E2$ with signature `EQUAL` (implementing equality for types `t1` and `t2` respectively) and creates another structure with signature `SET` that represents sets of pairs with type `(t1 * t2)`.

Note that SML has a syntax for writing functors with multiple arguments (so we could have made the two arguments `E1` and `E2`), but in our experience a lot of people find this syntax confusing, so we make all functors take one argument. In particular, `PairSet` takes a structure of signature `PAIREQUAL` containing two structures `E1` and `E2`, each ascribing to `EQUAL`. This signature is defined in `equal.sig`.

**Task 4.4** Test your implementation of `PairSet` by passing sets of different types to it as arguments. This will require making some structures of signature `PAIREQUAL`.