15-150 Fall 2013
Lab 09

23 October 2013

1 Introduction

This lab will help you become familiar with SML's module system. In lecture, we discussed the module system as a way to clearly mark and enforce abstraction boundaries. We will see several implementations of modules ascribing to a simple signature, and we will get comfortable working with them.

1.1 CM

In this lab, because we're dealing with lots of files containing structures, compilation 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.

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 and Signatures

Recall from lecture, a *signature* is an interface specification that usually lists some types and values (that might use those types). In this task, you will write two implementations of the `INTSET` signature that can be found in `intset.sig`, and is given below:

The components of the signature have the following specifications:

- `set` is the type of the set of elements of type `int`.
- `empty` is a set that contains no elements.
- `find` is a function that takes a set and an element, and returns `true` if that element is in the set, or `false` if the element is not in the set.
- `insert` is a function that takes a set and an element and returns the set with the element added.
- `delete` is a function that takes a set and an element and returns the set with the element removed.
- `union` is a function that takes two sets X and Y and evaluates to the set $X \cup Y$, i.e a set that contains all the elements in X and Y that results from performing a mathematical union of the two sets.
- `intersection` is a function that takes two sets X and Y and evaluates to the set $X \cap Y$, i.e. a set that contains the elements in both X and Y that results from performing a mathematical intersection of the two sets.
- `difference` is a function that takes two sets X and Y as input and evaluates to the set $X \backslash Y$, i.e. a set that contains all elements in X and not in Y, that results from performing the mathematical difference of the two sets.

There are many ways to implement the functionality of sets. We will implement sets in two different ways in this lab. For the first we will use lists to represent sets. For the second we will use binary search trees to represent sets.

You should think about what invariants you will want to have on your internal representation before starting to program; there are a few ways to do this. With that said, we suggest you do not allow for duplicates in these two implementations of sets.

Task 2.1 Implement the structure `ListSets` ascribing to `INTSET` found in `ListSets.sml`. Keep in mind that all functions implemented will want to maintain the invariants for your internal representation of sets.

You should use higher order functions, rather than recursion, for your solutions.

Recall our definition of an int tree:

```
datatype int tree = Empty
                  | Node of int tree * int * int tree
```

Also you may recall from previous classes that a Binary Search Tree has the following Ordering Invariant:

At any node with key k in a binary search tree, all keys of the elements in the left subtree are strictly less than k, while all keys of the elements in the right subtree are strictly greater than k.

Task 2.2 Implement the structure TreeSets ascribing to `INTSET` found in `TreeSets.sml`. Keep in mind that all functions implemented must account for the fact that the internal representation follows the ordering invariant of a Binary Search Tree and the other invariants for your internal representation of sets. Also you do not have to implement the function `delete` for this set implementation.

3 Find With Exceptions

The general form of exception handling is

```
e handle p1 => e1 | ... | pn => en
```

which has type `T` if `e : T`, and given the bindings introduced in each pattern $\texttt{p}_i$ (which matches against an exception packet, of type `exn`), each of the $\texttt{e}_i$ branches has type `T`.

As a reminder, if in the expression `e handle p1 => e2 | ... | pn => en`, `e` evaluates to a value (without raising an exception), then the whole expression evaluates to that value. If `e` raises an exception, then each of the patterns ($\texttt{p}_i$) are tried in succession, and the entire expression steps to the right side ($\texttt{e}_i$) of the first successfully matching branch (with the appropriate bindings introduced, just like in a case expression). If no pattern matches, the exception remains unhandled.

Task 3.1 Implement a function

```
find : ('a -> bool) -> 'a tree -> 'a
```

such that `find p T` evaluates to `x` where `x` is the first element in `T` (when using an inorder traversal of a tree) where `p x` evaluates to true, and raises an exception `NotFound` otherwise. Make sure to write tests for your code. A few of them should *handle* the exception and return somethin else i fan exception is raised (this handling will also be found in your function).

When you write tests (and in your function) for code that you're expecting to raise an exceptoin, you should wrap it in a handler that should unambiguously indicate whether the exception was raised or not. For example, if `e` was an expression that was supposed to raise an exception `Ex`, we could test the behavior of `e` with

```
val true = (let val _ = e in false end) handle Ex => true
```

That is, if `e` evaluates without raising an exception, the test will evaluate to `false` (which will fail to compile); the only way that we can compile this code successfully is if `e` raises the exception `Ex`.

Have the TAs check your code and tests before continuing!