15-150 Spring 2012
Lab 10

March 28, 2012

1 Introduction

This lab is divided into two parts. First, you will follow given directions to make sure that you can run the Homework 8 visualization. Second, you will experiement with the module system. In lecture, we discussed the module system as a way to clearly mark and enforce abstraction boundaries. In this lab, you will use modules and abstract types from the perspective of both a client and an implementer.

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`).

1.2 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 a purpose and tests.

2 Running the Visualizer

If you haven't run the n-body visualizer already, now is the time to try it. In the code directory for Homework 7, open an SML REPL and enter the following commands:

```
- CM.make "sources-real.cm";
[autoloading]
...
val it = true : bool
- Simulation.runPairwise Solars.two_body (Plane.s_fromInt 86400) 365
  "transcript.txt";
```

This produces a file `transcript.txt` in that directory. Then go to

```
http://www.cs.cmu.edu/~15150/visualizer/
```

and upload `transcript.txt`. You will see a brief animation of the earth revolving around the sun. Please ask for help if you are unable to see this animation.

When you have finished implementing Barnes-Hut, you can run the visualizer with the output from your algorithm by running `Simulation.runBH` instead of `Simulation.runPairwise`. The arguments are the same as for `runPairwise`.

3 Dictionaries

A *dictionary* is a datastructure that acts as a finite map from *keys* and to *values*. We represent a dictionary by a type

```
('k, 'v) dict
```

`dict` is a type constructor that takes two type arguments (unlike e.g. `'a list`, which takes only one). The first, `'k`, represents the type of keys, whereas the second, `'v` represents the type of values. For example, an `(int,string) dict` maps integers to strings.

There are many possible implementations of dictionaries, including using lists, trees, and functions. Since there are so many different ways to implement dictionaries, it would be nice if we could have an abstract interface to them that is the same regardless of the underlying implementation. This is where the module system comes in.

In `LabDict.sig`, we have provided the following signature for you to implement for dictionaries (see Figure 1).

The type and values in it have the following specifications:

- `('k, 'v) dict` is an abstract type representing the type of the dictionary. Note that it is parametrized over two different types—`'k`, the type of keys, and `'v`, the type of values.

- `empty` is a dictionary that contains no mappings.

- `insert` is a function that takes a comparison function for keys[1], a dictionary, and a key-value pair and returns the dictionary with the mapping added. If the key is already in the dictionary, the new value supersedes the old one.

- `lookup` is a function that takes a comparison function, a dictionary, and a key, and returns `SOME v` if that key maps to the value `v` in the dictionary, or `NONE` if there is no mapping from the key.

We have called this signature `LABDICT` because it is a version of dictionaries that is small enough for you to implement in lab; a real dictionary library would provide more operations.

[1]You may feel a bit uncomfortable adding the comparison function as an argument to each of these functions, instead wondering why we cannot just abstract over it somehow. If so, good! We will go over this on Thursday.

```
signature LABDICT =
sig

  (* We model a dictionary as a set of key-value pairs written k ~ v:
     (k1 ~ v1, k2 ~ v2, ...) *)
  type ('k, 'v) dict

  (* the empty mapping *)
  val empty : ('k, 'v) dict

  (* insert cmp (k1 ~ v1, ..., kn ~ vn) (k,v)
        == (k1 ~ v1, ..., ki ~ v,...) if cmp(k,ki) ==> EQUAL for some ki
     or == (k1 ~ v1, ..., kn ~ vn, k ~ v) otherwise
     *)
  val insert : ('k * 'k -> order) -> ('k, 'v) dict -> ('k * 'v) -> ('k, 'v) dict

  (* lookup cmp (k1 ~ v1,...,kn ~ vn) k == SOME vi
                                            if cmp(k,ki) ==> EQUAL for some ki
                                         == NONE otherwise
     *)
  val lookup : ('k * 'k -> order) -> ('k, 'v) dict -> 'k -> 'v option

end
```

Figure 1: Dictionary Signature

3.1 Implementation: Binary Search Trees

First, you will implement dictionaries using a binary search tree (BST). Recall the discussion of binary search trees from when we implemented mergesort on trees: the key invariant is that, for every `Node(l,x,r)`, everything in `l` is less than or equal to `x`, and everything in `r` is greater than or equal to `x`.

To implement dictionaries, we will store both a key and a value at each node, using the following datatype:

```
datatype ('k, 'v) tree =
    Leaf
  | Node of ('k, 'v) tree * ('k * 'v) * ('k, 'v) tree
```

In `Node(l,(k,v),r)`, `k` is the key and `v` is the value. Every key in `l` should be less than or equal to `k`, and every key in `r` should be greater than `k`. That is, the keys satisfy the BST invariant; the values are just along for the ride.

Task 3.1 Create a file `TreeDict.sml` in which you implement a structure `TreeDict` matching the signature `LABDICT`. You should use the datatype above as the internal representation of a dictionary. Make sure you ascribe the signature LABDICT to make the type `dict` abstract!

To test your implementation, you can run the command

```
- CM.make "sources.cm";
```

from the REPL. Note that your code will not compile until you make `TreeDict.sml` and put a module in it.

To create tests for your code inside the SML file, you can do them normally as we have been doing all semester. You should put them inside the `TreeDict` structure.

To test your code from the REPL, you will need to refer to functions inside your `TreeDict` structure as components of the module. (i.e. as `TreeDict.<function_name>` where `<function_name>` is the name of the function you want to run). Recall that you can only refer to functions that have been defined in the signature.

Some notes about the compilation manager: If you compile your code using `CM.make`, the compilation manager will compile all of the files specified in the `.cm` file. One way to work is to use `CM.make` every time you want to compile.

However, this has the disadvantage that none of the project loads if there is a compilation problem anywhere, which can make debugging harder—you can't use the REPL to play around. An alternative is to `CM.make` when you first start working on a module, assuming your initial state is one where the `CM.make` succeeds (this is generally true for the support code we hand out). Then you can reload the file containing the module you are currently working on with `use` after you make updates to it. Note that this will shadow the modules in that file, and thus **not** update the modules "downstream". However, it is useful if you are using emacs, and like to use the emacs command to load the current buffer: you can load one module repeatedly and use the REPL to test it. It's also useful if your current implementation of the module has a bug that causes later files in the `.cm` file to fail to compile. But when when you are done working on the module, you will want to run `CM.make` again to reload all the downstream modules, so that they refer to your new implementation.

Have the TAs check your code before continuing!

3.2 Client: Polynomials

A multi-variable polynomial is an expression built out of $+$ and $*$ and some constants and variables. For example, the polynomial

$$x^2 + y^2 - 1$$

is 0 for all points on a circle of radius 1.

We can represent multi-variable polynomials by the following datatype:

```
datatype poly =
    Var of var
  | K of real
  | Plus of poly * poly
  | Times of poly * poly
```

where `var` is some type representing variables. For example, we can take

```
type var = string
```

and then write $x^2 + y^2 - 1$ as

```
Plus(Times (Var "x", Var "x"),
     Plus(Times(Var "y" , Var "y"),
          K (~1.0)))
```

`Plus` represents $+$; `Times` represents $*$; `Var` represents variables; `K` represents constants.

You can evaluate a multi-variate polynomial on a particular input, given bindings for each variable occuring in the polynomial (e.g., both x and y in the example above). We will represent these bindings as a dictionary mapping variables to `real`s.

Task 3.2 In the file `poly.sml`, write the function

```
evaluate : (var, real) Dict.dict -> poly -> real
```

that takes a dictionary and a polynomial and evaluates to the value of that polynomial under the bindings given by the dictionary. If a variable occuring in the polynomial does not appear in the dictionary, `evaluate` should raise the exception `Unbound`.

To load your code, you will once again want to run `CM.make`. To test your functions at the REPL, you must refer to them as components of the module, e.g. `Poly.Times`, `Poly.K`. For example, to test the evaluate function you can do something along the lines of:

```
- val D = TreeDict.insert(String.compare)(TreeDict.empty)("x",2.0);
val D = Node (Leaf,("x",2.0),Leaf) : (string,real) TreeDict.dict
- val D' = TreeDict.insert(String.compare) D ("y",3.0);
val D' = Node (Leaf,("x",2.0),Node (Leaf,(#,#),Leaf))
- Poly.evaluate D' (Poly.Plus(Poly.Var "x", Poly.Var "y"));
val it = 5.0 : real
```

Task 3.3 BONUS TASK: Define infix operators `++`, `**`, and `^^`, so that $x^2 + 2x + 1$ can be written

```
(Var "x") ^^ 2 ++ K 2 ** (Var "x") ++ K 1
```

If you want to get really cutesey, define functions ` and `` so that you can write

```
`"x"^^2 ++ ``2 ** `"x" ++ ``1
```