CS 3110 Spring 2015	**Soft deadline: 4/16**
Problem Set 4	**Hard deadline: 4/18**
Version 0 (last modified April 6, 2015)	

Overview: This problem set revolves around a simple interface: lists that support random access. You will implement this interface using a variety of data structures, and you will analyze your code in a variety of ways.

Objectives:

- Implementing data structures with interesting invariants
- Testing and specification
- Using modules and functors to structure code
- Formal proofs of correctness
- Asymptotic worst-case and amortized complexity analysis
- Performance testing

Additional reading:

- Real World OCaml, Chapter 9: Functors.
- Lectures 10–12, 15-17; Recitations 15–17.
- Map module documentation
- QCheck.Arbitrary API

Getting help: When you need help, there are many resources available to you. The CS 3110 Piazza site is a great place to ask questions. Course staff and other students are very active and will typically respond within a couple hours. Consulting hours are a great place to ask about anything in this assignment and future assignments, as well as other questions you may have about OCaml or setting up your environment.

Part 1: The ListLike interface (40 points)

For the first part of the assignment, you will work with the ListLike interface, defined in ListLike.mli. The ListLike module contains a few interfaces and functors:

ListLike.CORE defines the core list operations like `cons`, `lookup`, and `update` that every implementor must provide.

Most of these functions are familiar, but `decons` is unusual. `decons` deconstructs a non-empty list into the head and tail, or returns `None` if the list is empty. `decons` makes pattern-matching on ListLike objects easy:

```
match decons l with
  | None          -> (* empty case *)
  | Some (h, tl) -> (* h::tl case *)
```

ListLike.EXTENSION contains useful functions like `map` and `fold` that users of the module can use. The `ListLike.Extend` functor provides implementations of these functions for any module that implements `ListLike.CORE`.

Implementations of the `CORE` interface can include `Extend(Core)` to get the `EXTENSION` functions "for free". For example, your `ListImpl` module need only define the `ListLike.CORE` functions; the `include Extend(Core)` function at the bottom of `listLike.ml` will automatically include your definitions of `map`, `fold`, and so on:

```
utop # module M = ListImpl;;
module M : sig
    module Core

    (* implemented *)
    val equals : 'a t -> 'a t -> bool
    val empty  : 'a t
    val cons   : 'a -> 'a t -> 'a t
    val decons : 'a t -> ('a * 'a t) option
    val lookup : 'a t -> int -> 'a option
    val update : 'a t -> int -> 'a -> 'a t option
    val length : 'a t -> int

    (* inherited *)
    val to_list    : 'a t -> 'a t
    val of_list    : 'a t -> 'a t
    val map        : ('a -> 'b) -> 'a t -> 'b t
    val fold_left  : ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a
    val fold_right : ('b -> 'a -> 'a) -> 'b t -> 'a -> 'a
end
```

ListLike.S combines the `ListLike.CORE` and `ListLike.EXTENSION` module types, and is the interface that *users* of ListLike implementations will typically rely on (just as modules with a "main type" typically call that type "`Module.t`", modules with a "main interface" often call that interface "`Module.S`".

Combining these interfaces makes it easy to create `.mlis` for modules that simply implement the interface. Note, for example, that the file `mapImpl.mli` only contains the line `include ListLike.S`. Modules that wish to expose their types (such as `ListImpl`) can do so by writing `include ListLike.S with type t := <the concrete type>` (see `listImpl.mli` for an example).

ListLike.Spec will contain generic properties that any implementation of the `ListLike.CORE` interface must satisfy. Each property should have a corresponding function that checks the property. For example, the `equals` function should be reflexive (all lists should equal themselves) and symmetric (if `l1` equals `l2` then `l2` should equal `l1`). These properties are checked by the `equals_self` and `equals_symmetry` examples that we have provided:

```
let equals_self l =
  C.equals l l

let equals_symmetric (l1,l2) =
  if C.equals l1 l2 then C.equals l2 l1 else test_passes
```

The `ListLike.Spec` functor should also contain randomized tests for each of the properties. These are designed to be used with the `TEST_MODULE` command; for example the `ListImpl` implementation contains the line

```
TEST_MODULE "ListImpl spec tests" = ListLike.Spec(Core)
```

When run with `cs3110 test listImpl.ml`, the `TEST_MODULE` will cause all of the `TEST`s in `ListLike.Spec(Core)` to be run.

To complete part 1 of this assignment, complete the following tasks. These tasks can all be completed independently and in any order; in fact we find it useful to split the implementation of `ListLike.Spec` and `ListImpl.Core` between different partners; developing the tests and implementation separately can help to shake out misunderstandings about the spec.

The rest of the problem set builds on this work, so we recommend finishing this part early.

Exercise 1: Write ListLike.Spec properties.

(a) The provided implementation of `ListLike.Spec` only tests the two example properties of described above. List all of the properties that you think the functions in the `ListLike.CORE` module should satisfy. For each property, create a corresponding function in the `ListLike.Spec` module. Be sure to document these functions in the `listLike.mli` (note that for this project you *should* change `listLike.mli`, though your code should still work if you replace your submitted `listLike.mli` with the release `listLike.mli`).

For full credit, your specs should be:

- Strong enough: broken implementations should fail to satisfy at least one property
- Not too strong: all correct implementations should satisfy all properties
- Concise: avoid redundant properties so that it is easy to read all of them.

A good way to start is to think about how the functions should interact with each other: what can you say about `decons (cons x l)` for example?

(b) For each of the properties tests you have implemented, add a `TEST` to `ListLike.Spec` that uses QCheck to test the property on random instances.

QCheck is a library that takes a specification function and generates many random values to pass to that function. You will interface with the QCheck library by using the function `assert_qcheck`.

The `assert_qcheck` function adapts the QCheck library to work with the cs3110 tools. It has the following type:

```
val assert_qcheck : 'a QCheck.Arbitrary.t -> ('a -> bool) -> unit
```

An `'a QCheck.Arbitrary.t` is essentially a function that generates random values of type `'a`; `assert_qcheck` uses this function to generate many values and passes them to the property given by the second argument.

Within the `ListLike.Spec(C)` implementation we have provided an "arbitrary" `C.t`. That is, we have provided a the value

```
val arb_listlike : (char C.t) Arbitrary.t
```

This allows you to test properties that depend on a single list-like value:

```
TEST_UNIT "equals_self test" =
  assert_qcheck arb_listlike equals_self
```

The `QCheck.Arbitrary` module provides functions for easily combining arbitrary values to create new arbitrary values. For example, we have provided an arbitrary pair of lists:

```
let arb_listlike_pair : (char C.t * char C.t) Arbitrary.t =
  Arbitrary.(pair arb_listlike arb_listlike)
```

`arb_listlike_pair` can be used to test properties that depend on pairs of lists:

```
TEST_UNIT "equals_symmetric test" =
  assert_qcheck arb_listlike_pair equals_symmetric
```

Exercise 2: Implement ListImpl.Core.

In the file `listImpl.ml`, implement the `ListLike.CORE` functions using a plain OCaml list as a the underlying data structure. Note that `lookup` and `update` may be inefficient!

Exercise 3: Implement MapImpl.Core.

In the file `mapImpl.ml`, implement the `ListLike.CORE` functions using the `Map` module from the standard library. The `Map` module uses a balanced binary search tree to map from keys to values; you can use an integer as the key to the map, allowing you to easily find or update an element at a given offset.

Be sure to write a `rep_ok` function in `mapImpl.ml` to document and test your representation invariant. You should also clearly specify your abstraction function.

Exercise 4: Implement ListLike.Extend.

In the file `listLike.ml`, implement the `Extend` functor to provide default implementations of the `EXTENSION` interface based on the `CORE` operations.

Part 2: The Bits implementation (30 points)

In this part, you will implement a functional data structure that provides fast random access to a list-like data structure.

The motivating idea behind this data structure is that a naïve implementation of lists closely mirrors a unary implementation of the natural numbers:

```
type nat    = Zero | Succ of nat
type 'a list = Nil | Cons of 'a * 'a list
```

Taking the tail of a list is analogous to subtracting one from a number

```
let pred = function
| Zero -> failwith "Error: pred"
| Succ n -> n
let tail = function
| Nil -> failwith "Error: tl"
| Cons (x,xs) -> xs
```

while appending two lists is analogous to adding two numbers

```
let rec add = function
| Zero,m -> m
| Succ n, m -> Succ (add (n,m))
let rec append = function
| [],ys -> ys
| Cons(x,xs) ,ys -> Cons(x,(append (xs,ys)))
```

This analogy suggests that we can think of a list as a natural number that carries some extra information. Container abstractions designed with this analogy in mind are called numerical representations.

In this exercise we will use a numerical representation for lists that is based on a binary representation of numbers. In the binary representation of a natural number, a one in the kth position stands for 2^k. You can think of this as representing 2^k applications of the Succ constructor. Similarly, in the binary representation of a random access list, a "one" in the kth position contains 2^k elements; in a sense it represents 2^k applications of the Cons constructor.

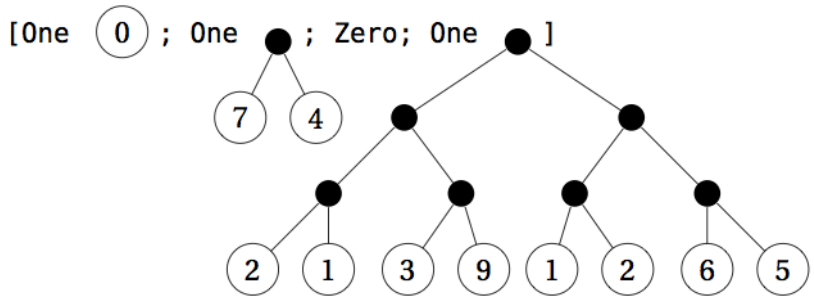
Of course, the "ones" in the random access list must actually store the 2^k entries in the list. Since there are exactly 2^k entries corresponding to the kth digit, we can store the entries in a completely balanced binary tree. We will use the following representation:

```
type 'a tree    = Leaf of 'a | Node of 'a tree * 'a tree
type 'a tbit    = Zero       | One of 'a tree
type 'a bitlist = 'a tbit list
```

For example, suppose that we wish to store the list `[0; 7; 4; 2; 1; 3; 9; 1; 2; 6; 5]`. This list contains 11 elements and the binary representation of 11 is `0b1011`, so the ralist representation should take the form `[One _; One _; Zero; One _]`. The digit in the

zeroth place corresponds to the first 2^0 elements (`[0]`). The bit in the first place corresponds to the 2^1 elements `[7; 4]`. The bit in the second place is Zero, so it contributes no elements to the total; the digit in the third place corresponds to the 2^3 elements `[2; 1; 3; 9; 1; 2; 6; 5]`.

We can draw this representation as follows:



To complete this part of the problem set you need to complete the following task:

Exercise 5: Implement bitsImpl.ml.

In the file `bitsImpl.ml`, implement the `CORE` functions using the binary representation of lists described above.

Also implement the function `rep_ok` function to check the representation invariant of an `'a bitlist`. In particular we need to check if the `'a tree` at each position is a perfectly balanced binary tree of the correct size and that the `'a bitlist` does not have trailing `Zero`s.

Part 3: Written analysis (40 points)

In this part of the assignment, you will prove a few properties of your `BitsImpl` implementation. See the companion files `compendium.pdf` and `blackbook.pdf` for examples of proofs.

Exercise 6: BitsImpl.cons produces well-formed lists.

In the file `cons_correct.pdf` or `cons_correct.txt`, prove that if `l` is a valid `bitlist`, then `cons x l` is also a valid bitlist.

If your proof is inductive, be sure to clearly indicate what your inductive hypothesis is and where you are using it. Also be sure to check that it applies to the variables you are applying it to!

Exercise 7: BitsImpl.lookup runs in O(log n) time.

In the file `lookup_runtime.pdf` or `lookup_runtime.txt`, prove that the worst-case running time of your implementation of `BitsImpl.lookup l` is $O(\log n)$ where n is `BitsImpl.length l`.

Hint: Your implementation of `lookup i l` will probably proceed in two phases: first you must find the tree `t` containing the `i`th entry, then you will locate the `i`th entry in `t`. Prove (inductively) that each of these two operations can execute in $O(\log \mathrm{n})$ time, and then use that fact to conclude that `lookup i l` itself completes in $O(\log \mathrm{n})$ steps.

Exercise 8: BitsImpl.cons runs in amortized O(1) time.

In the file `cons_runtime.pdf` or `cons_runtime.txt`, prove that the amortized running time of multiple `cons` operations is $O(1)$ per operation. You may use any of the methods discussed in lecture. Here are some hints for each approach:

- For the aggregate method, try counting the number of times each position in the bit list changes from a `One` to a `Zero`. For example, the first bit in the list resets on every other cons insertion, while the second bit in the list resets on every fourth cons.

  You can also reason about the time taken for an insertion that resets the kth bit. Multiply these together and then sum them to find the total running time,

- For the banker's method or potential methods, note that having lots of `One`s in the list is what causes a particular operation to take a long time. Try associating accounts with each `One` in the bit list, or using the number of `One`s as the potential function.

Exercise 9: Performance measurement.

As you know, asymptotic complexity is an approximation of actual running time. To measure the real-world performance of your code, design and run tests that run various operations on your three `ListLike` implementations with lists of varying sizes.

In the file `performance.pdf` or `performance.txt`, report on the results of these measurements. Be sure to include:

(a) Instructions for running your tests and information about the environment that you ran your experiments in

(b) A plot (or plots) of run-time versus input size

(c) Comparison of your results with the theoretical bounds shown above

(d) Discussion of any surprises found in the data

To measure wall-clock running time, you can call the `Sys.time` function both before and after calling your function; the difference between the two times will tell you how long your function took to execute.

It will often be the case that the running time for your functions will be comparable to the resolution of the system clock. To get reliable data, we recommend measuring the running time for multiple executions and dividing by the number of executions.

Running time can also depend on other variables such as the state of the cache, the garbage collector, or background processes that may be running on the system. We recommend collecting statistics from multiple executions and reporting combined statistics (e.g. by providing a scatter plot or error bars).

Important note: be sure you disable your `rep_ok` function so that it doesn't skew your results!