CS 3110 Fall 2014 **Due at 11:59 PM, 10/02/14**
Problem Set 3
Version 3 (last modified September 26, 2014)

Revision log

- [09/26/14] Clarified Problem 1, Exercise 1(d). Corrected range of latitudes in Problem 2, Exercise 2(a). Corrected `t` to `N.t` in Problem 3, Exercise 4. Clarified requirements of karma solution in Problem 3, Exercise 4. Clarified that alien symbols must be non-negative in Problem 3, Exercise 5.
- [09/23/14] Removed extra parentheses around arguments to `Node` constructor in Problem 2, Exercise 1. (The supplied code in `ps3.zip` was correct.) Corrected two minor English typos.

Objectives

- Use the environment model to explain how programs evaluate.
- Write programs that use OCaml modules and functors.
- Use abstraction and encapsulation to implement data structures.
- Learn about a new data structure, the quadtree.

Recommended reading

The following materials should be helpful in completing this assignment:

- Course readings: lectures 7, 8, 9, and 10; recitations 7, 8, and 9
- The CS 3110 style guide
- The OCaml tutorial
- Real World OCaml, Chapters 4 and 9

What we supply

We provide an archive `ps3.zip` that you should download from CMS. It provides interfaces, some template code for modules, and a test input file for one exercise.

What to turn in

Submit these files in a single `ps3.zip` on CMS. There should not be any directory structure inside your zip.

- A file `ps3written.pdf` containing your solutions to the written problems of this problem set, which are identified below as "[written]". This file should also contains any comments you have about the problem set, as described at the end of this writeup. And it should contain any information about the karma problem, if you choose to solve it.
- Files containing your solutions and unit tests for the coding exercises of this problem set, which are identified below as "[code]". These files should have the same names as the files we distribute in `ps3.zip`.
- A commit log `ps3log.txt` documenting your activity on the repository.

Partners and source control

You are required to work with a partner for this problem set. Each partner is responsible for understanding all parts of the assignment. You need not use the same partner(s) as in previous problem sets.

You are required to use `git`, a version control system, to work with your partner. Your repository must be private. We expect the `git` log you submit to show evidence of work over a period of time, not just a single commit at the end.

Grading issues

Compilation errors: All code you submit must compile. If your submission does not compile, we will notify you immediately. You will have 48 hours after the due date to supply us with a patch. If you do not submit a patch, or if your patched code does not compile, you will receive an automatic zero.

Naming: We use automated grading, so it is crucial that you name your functions and order their arguments according to the problem set instructions, and that you place the functions in the correct files. Incorrectly named functions will be treated as compilation errors.

Code style: Refer to the CS 3110 style guide and lecture notes. Ugly code that yet functionally correct will nonetheless be penalized. Take extra time to think and find elegant solutions.

Late submissions: Carefully review the course policy on submission and late assignments. Verify before the deadline on CMS that you have submitted the correct version.

Function Specification and Testing

Complete each of the coding exercises below by following these instructions:

1. Write a function with the appropriate name and type.
2. Write a specification comment above the definition of the function that documents a concise and accurate description of the function's *precondition* and *postcondition*. Also document a brief description of the function (one or two sentences) and/or a brief description of each argument (a couple of words), if your pre- and postconditions do not already address those descriptions.
3. Write unit tests that demonstrate the function's correctness. If the function is named `f`, then these tests should be named `f_test1`, `f_test2`, ...`f_testn`. How many unit tests should you write? As many as necessary to make you confident that your solution is correct. Your tests should be in a separate file, as described above.

No Imperative Features

Imperative features—such as `ref`'s, the `Array` module, and `mutable` fields—are not permitted in your solutions to this problem set. You have not seen these features in lecture or recitation, so we doubt you'll be tempted.

Problem 1: Semantics (50 points)

Exercise 1.

[written] The type-checking rules for `if` expressions require the *guard* of the expression to have type `bool`. (In the expression `if e0 then e1 else e2`, the guard is `e0`—i.e., the expression between `if` and `then`.) In C, however, guards are required to have type `int`; the `then` branch is executed if the guard is nonzero, otherwise the `else` branch is executed. In C, for example,

```
if (1) {
  printf("1\n");
} else {
  printf("0\n");
}
```

will print `1`.

Let's modify OCaml `if` expressions such that guards have type `int` rather than `bool`.

(a) Give a new type-checking rule for OCaml `if` expressions that requires guards to have type `int`. Your rule should otherwise be the same as the standard OCaml rule. Give the entire rule, not just a fragment of it.

(b) To use the "Boolean" comparison operators with our new `if` expressions, they now need to return integers. Let's consider the less-than operator `<`. It should now be a curried function that returns an integer. Write down its new type (and just its type—nothing else).

(c) Give a new evaluation rule for `if` expressions, using the environment model. Your rule should result in the same semantics as that demonstrated above for C. Give the entire rule.

(d) OCaml `if` expressions can be understood as syntactic sugar for `match` expressions. Show how to *desugar* our new kind of integer-guarded `if` expressions. That is, how could a compiler rewrite `if e0 then e1 else e2` as a `match` expression, given your new evaluation rule from the previous part? Your answer may not use `if` expressions. Your answer might involve some bad style, but that's okay—we're talking here about compiler-generated code, not code that humans write.

Exercise 2.

[written] Using the environment model, show how to determine the value of the following expression under lexical scope.

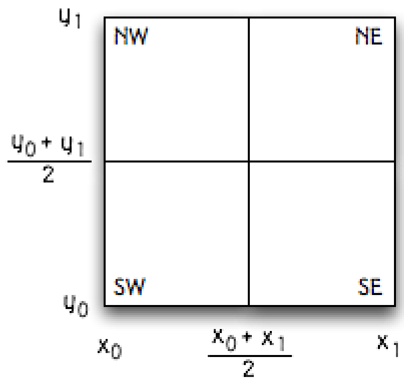
```
let x = 1 in
let f = fun y ->
  (let x = y+1 in
   fun z -> x+y+z) in
let x = 3 in
let g = f 4 in
let y = 5 in
let z = g 6 in
z
```

Show your work, as in the recitation on the environment model—don't just give the final value.

Problem 2: Quadtrees (100 points)

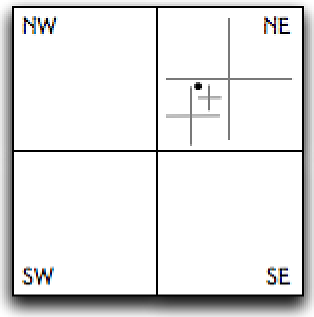
[code] A *quadtree* is a data structure that provides a sparse representation of 2D space. Quadtrees are used heavily in Graphics, Computational Geometry, and scientific simulations. Here, we'll use quadtrees to implement geographic search; specifically, finding cities that are within specified latitude and longitude ranges.

For this problem, you will implement a variation of quadtrees in which a leaf node represents a (possibly empty) set of objects, and a non-leaf node represents a rectangular region of space parameterized by four values x_0, x_1, y_0, and y_1:



The node depicted above represents the region of space between x coordinates x_0 and x_1, and between y coordinates y_0 and y_1. The region is partitioned into four equally-sized quadrants. Each of those quadrants is a subtree. This recursive division of space into quadrants is used to avoid wasting memory on representation of empty space.

To find an object near point (x, y), a quadtree is traversed starting from the root, walking down the appropriate sequence of child nodes that contain the point until a leaf node is reached. The set of objects at that leaf can then be examined. The figure below depicts the quadtree nodes visited during the search for the black dot:



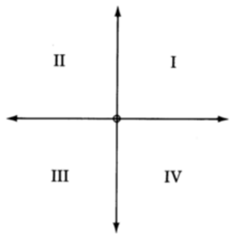
We provide two files for you to finish implementing: `quadtree.ml` and `city_search.ml`. We provide interface files, which specify the functions that you need to implement. We also provide a parser (described below) for reading information about cities from a CSV file. Note that compiling that parser requires passing the `-l str` flag to `cs3110`.

Exercise 1: Implement Quadtrees.

The type of quadtrees is given in `quadtree.ml`:

```
type coord = float * float
type region = coord * coord
type 'a quadtree =
    Node of region * 'a quadtree * 'a quadtree
                   * 'a quadtree * 'a quadtree
  | Leaf of region * ((coord * 'a) list)
```

A point or *coordinate* `coord` is a pair of an x coordinate followed by a y coordinate. The first `coord` of a `region` represents the lower-left coordinate of that region of space, and the second `coord` represents the upper-right coordinate. A `region` is always defined by these two coordinates. The first quadtree in a `Node` is the north-eastern quadrant (I) of the region, the second is the north-western quadrant (II), the third is the south-western quadrant (III), and the fourth is the south-eastern quadrant (IV):



Usually, only one object is present in the list at a `Leaf`. If inserting an object into a `Leaf` would make the list have length 2 or longer, the `Leaf` becomes a `Node`, and the `Leaf`'s objects are distributed into the `Node`. However, to avoid the quadtree becoming too finely divided, a `Leaf` is never separated in this way if doing so would cause the size of the region to become too small. In particular, if the region already has a diagonal length (from its lower-left coordinate to the upper-right) that is less than a constant `min_diagonal`, then the `Leaf` remains a leaf, and its list simply grows in length with each new object added at that `Leaf`. The order of objects in the list is unspecified. The `min_diagonal` constant is provided for you in `quadtree.ml`; please do not change its value.

Finish `quadtree.ml`, including adding specification comments and writing unit tests:

a. First, write a specification comment above the type `'a quadtree`, documenting your understanding of how the type represents quadtrees. Then, implement the following functions.

b. `new_tree : region -> 'a quadtree` : Initialize a new quadtree that will contain points within the given region.

c. `insert : 'a quadtree -> coord -> 'a -> 'a quadtree` : Insert an object at a given coordinate. If the coordinate is outside the region represented by the tree, raise `OutOfBounds`.

d. `fold_quad : ('a -> (coord * 'b) -> 'a) -> 'a -> 'b quadtree -> 'a` : Fold the function argument over the quadtree, starting with the accumulator argument of type `'a`.

Apply the function argument to every object in the quadtree. *Hint:* Think about how `List.fold_left` is implemented, then generalize from lists to trees.

e. `fold_region : ('a -> (coord * 'b) -> 'a) -> 'a -> 'b quadtree -> region -> 'a`: Fold the function argument over the quadtree, but applying it only to those objects that are within the region argument. *Hint:* The region might or might not be entirely contained within some subtree, so think carefully about your implementation.

Exercise 2: Implement City Search.

Implement these functions in `city_search.ml`, including adding specification comments and writing unit tests:

a. `load_city_data : string -> string quadtree` : Load all the cities from the file named by the `string` argument, store them in a quadtree, and return that quadtree.

   The file format is CSV. Each line in the file is a city. The format of a line is

```
Latitude, Longitude, Name
```

   We have supplied a very small example file `ithaca.csv` in the `ps3.zip` that you downloaded. The functionality for reading the file has already been implemented for you in the provided `Parser` module, so you don't need to implement any I/O yourself. Note that city coordinates are given in terms of latitude and longitude. Latitude ranges from `-90.0` to `90.0`, and longitude ranges from `-180.0` to `180.0`.

b. `city_search : string quadtree -> region -> string list` : Return all of the cities within a given region, specified by latitude and longitude.

Problem 3: The Natural Numbers (100 points)

[code] In this problem, we explore the capabilities for abstraction provided by OCaml modules and functors by way of a fundamental mathematical construction: the natural numbers, $\mathbb{N} = \{0, 1, \ldots\}$. There are many operations we could define on the natural numbers; here we'll consider only addition, multiplication, equality, and the less-than ordering. Two natural numbers, 0 and 1, are particularly interesting because of the way they behave with addition and multiplication. Here is an OCaml signature `NATN` representing those aspects of the natural numbers:

```
module type NATN = sig
  type t

  val zero: t
  val one: t
  val ( + ): t -> t -> t
  val ( * ): t -> t -> t
  val ( === ): t -> t -> bool
  val ( < ): t -> t -> bool

  exception Unrepresentable

  val int_of_nat: t -> int
  val nat_of_int: int -> t
end
```

The final two values in `NATN` are used to convert between natural numbers and the primitive `int` type. Not all natural numbers are representable with `int` (consider 2^{128}), and not all values of type `int` are representable as natural numbers (consider `-1`). So the two conversion functions must sometimes raise the `Unrepresentable` exception.

Abstract type `t` represents the type of natural numbers. Implementations of `NATN` are free to choose different instantiations `t`. The exercises below explore some of those implementations.

Exercise 1: Specification.

Write specification comments for each of the values (i.e., those entries introduced with keyword `val`) contained in the `NATN` signature. Your comments should, as usual, include postconditions as well as any necessary preconditions. You should also document the following properties:

- associativity of addition and multiplication
- commutativity of addition and multiplication
- zero and one are identity elements for addition and multiplication, respectively
- distributivity of multiplication over addition

Choose appropriate places to document those properties, and write down the meaning of each property in terms of the values of `NATN`. For example, to document associativity of addition, you might include at least the following in your specification comment for `( + )`:

```
(* + is associative:  (a+b)+c === a+(b+c) *)
val ( + ): t -> t -> t
```

Exercise 2: Implementation with `int`.

Write a module `IntNat` that implements signature `NATN`. The representation type `t` should be OCaml's primitive `int` type.

```
module IntNat: NATN = struct
  type t = int
  ...
end
```

A representation issue: For full credit, be careful to correctly implement addition and multiplication. Your implementation will sometimes need to raise `Unrepresentable`. Revisit and, if necessary, update your specification comments in `NATN` to account for this issue.

Hint: The OCaml manual [section 1.4, Records and Variants] suggests code similar to the following to check for overflow:

```
type sign = Positive | Negative
let sign_int (n:int) : sign =
  if n >= 0 then Positive else Negative
let sum_overflows (i1:int) (i2:int) : bool =
  sign_int i1 = sign_int i2 && sign_int(i1 + i2) <> sign_int i1
```

Here are two examples:

```
# sum_overflows max_int 0;;
- : bool = false
# sum_overflows max_int 1;;
- : bool = true
```

Exercise 3: Implementation with unary lists.

Write a module `ListNat` that implements signature `NATN`. The representation type `t` should be `int list`:

```
module ListNat: NATN = struct
  (* The list [a1; ...; an] represents the
   * natural number n.  That is, the list lst represents
   * length(lst). The empty list represents 0. The values of
   * the list elements are irrelevant. *)
  type t = int list
  ...
end
```

You should include the above specification comment for `t` in your code. Such comments are good practice, and we will discuss them in future lectures.

Here are some examples of natural numbers represented with unary lists:

```
let three = [1; 1; 1]
let four = [10; 8; 1; -1]
let ten = [10; 10; 10; 10; 10; 10; 10; 10; 10; 10]
```

A representation issue: The behavior of your implementation may be undefined if representation of a natural number would require more memory than is available in the virtual machine. But for full credit, be careful that your implementation does not cause stack overflows. Recall that tail-recursive functions use constant stack space. Revisit and, if necessary, update your specification comments in NATN to account for this issue.

Exercise 4: Implementation-independent conversion functions.

Each of our implementations so far has provided its own conversion functions based on knowledge of the representation type t. We could actually write conversion functions that are completely agnostic about that type.

Complete the functor NatConvertFn:

```
module NatConvertFn(N: NATN) = struct
  let int_of_nat(n: N.t): int = ...
  let nat_of_int(n: int): N.t = ...
end
```

Add specification comments to the two conversion functions. The comments you wrote while completing the previous exercises should be helpful.

Karma: The simplest solution involves using the conversion functions of N. We will give full credit for that solution. An alternative solution involves using (+) or NATN.(+) to add one successively until reaching the desired int or NATN. That solution runs in $O(n)$ time, where n is the magnitude of the number being converted.

But for karma (see the course syllabus for a definition of karma), your functions should be asymptotically more efficient than the $O(n)$ solution identified above, and should not call the conversion functions of N. If you opt to implement an efficient solution, tell us about it in your written solutions file, and document the efficiency of it in your code. Hint: For nat_of_int, think about a binary representation of the naturals. For int_of_nat, think about finding an interval in which the NATN lies.

Exercise 5: Aliens.

It is the year 3110 CE, and humankind has transcended its earthly shackles and conquered the vast frontier of space. Humans have established a vast intergalactic commercial empire and the decimal number system is standard throughout most of the Universe. However, humans still want to interact with civilizations who have not yet adopted human numbers. So humanity has called upon the cleverest computer scientists to design a system to make trading with these civilizations possible.

To make the interface between humans and the other civilizations as simple as possible, we ask an alien civilization for a mapping from each of their own symbols to a human (indeed, OCaml) int. That int must be non-negative. We also require the civilization to tell us which symbol represents 0, and which symbol represents 1.

```
module type AlienMapping = sig
  type aliensym

  val int_of_aliensym: aliensym -> int
  val one: aliensym
  val zero: aliensym
end
```

For example, a civilization might tell us that ♣ = 0, ♢ = 1, and `int_of_aliensym(♠)`= 10. Of course, we require that `int_of_aliensym` maps `one` and `zero` to 1 and 0.

Given any `AlienMapping`, we can produce a `NATN` that humanity can use. As the representation type, we choose `aliensym list`, and we interpret such a list as the sum of the `int`s it represents. Continuing our example, the list `[♢; ♠; ♣; ♠; ♢]` represents the natural number 22, because $1 + 10 + 0 + 10 + 1 = 22$. Implement `AlienNatFn`, a functor that maps an `AlienMapping` to a `NATN`.

```
module AlienNatFn (M: AlienMapping): NATN = struct
  type t = M.aliensym list

  ...
 end
```

A representation issue: As with `ListNat`, be careful that your implementation does not cause stack overflows. Recall that tail-recursive functions use constant stack space.

Problem 4 (0 points)

[written,ungraded] In your file of written solutions, please include any comments you have about the problem set or about your solutions. This would be a good place to list any known problems with your submission that you weren't able to fix, or to give us general feedback about how to improve the problem set. Also include any information about the karma problem, if you choose to solve it.

Include a statement of what work in this problem set was done by which partner. The ideal case is that each of you contributed to every problem. But (especially since this exercise is ungraded) please be honest about how you divided the work.