Data Structures and Functional Programming	Problem Set 4
CS 3110, Fall 2013	Due at 11:59 PM, Thursday, October 17
Version: 7	Last Modified: October 27, 2013

Instructions

Compile Errors

All code you submit must compile. **Programs that do not compile will be heavily penalized.** If your submission does not compile, we will notify you immediately. You will have 48 hours after the submission 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 will be using an automatic grading script, 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 are **treated as compile errors** and you will have to submit a patch.

Code Style

Finally, please pay attention to style. Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct may still lose points. Take the extra time to think out the problems and find the most elegant solutions before coding them up. Good programming style is important for all assignments throughout the semester.

Late Assignments

Please carefully review the course website's policy on late assignments, as **all** assignments handed in after the deadline will be considered late. Verify on CMS that you have submitted the correct version, **before** the deadline. Submitting the incorrect version before the deadline and realizing that you have done so after the deadline will be counted as a late submission.

Getting started

This problem set is long; we recommend you start thinking about all of the parts early. Parts two and three depend on part one, but you only need to implement one of the modules (`Floats`) in part one to get started on them. The things you have to do are indicated by "Exercise" in the writeup, and are labeled "TODO" in the release code.

The GUI for this problem set requires some additional OCaml libraries; the file `README.txt` in the release contains instructions for compiling and running the project.

Introduction

In many applications, users want to rearrange a collection of geometric shapes while keeping them from overlapping with each other. For example, users like to rearrange windows on their desktops while keeping each window entirely visible. Without an automated way to keep them from overlapping, users must carefully adjust the boundaries of the windows if they want to avoid large gaps between the windows. The same user interface feature is useful in other applications, such as computer aided design, graphics applications, and games.

In this assignment, you will be building a library that allows users to drag and drop shapes while keeping them from intersecting each other. You will use the library to build a simple "tangrams" game.

The correctness of geometric algorithms depends on the properties of the numbers used to represent the coordinates of the shapes. You will implement a variety of number types to investigate the tradeoffs of various representations.

Finally, this assignment contains a small problem related to mutability.

Part One: Numbers

For this portion of the assignment, you will implement a variety of modules, each defining a representation of a set of numbers and the operations available for those numbers.

Your number modules will each implement one of the following interfaces (we have provided these signatures in `numbers.ml`):

- A `Quotient` represents a set of elements, which we will refer to as `numbers`. The only operation provided by `Quotient` is the `(===)` function, which defines the notion of equality for the set of `numbers`.
- Just as `Quotient` defines a notion of equality between numbers, `Group` defines a notion of addition. In addition to the `(+)` operation, `Group` requires a number called `zero`, and for each number x, an additive inverse `-`x. Note that OCaml writes the negation operation as `(~-)`. That is, `(~-) x` is the same as `-x`.
- `Ring` extends `Group` by adding a notion of multiplication. It also requires a multiplicative identity called `one`.
- A `Field` is a ring where every number x has a multiplicative inverse x^{-1}. This allows us to define division: x/y is just xy^{-1}.
- `OrderedRing` and `OrderedField` add a notion of ordering to rings and fields respectively. We only require implementors to provide a function that determines if a number is negative, but comparison operations like `(<)` and `(>)` can be defined in terms of `is_non_neg` (you will do this in exercise 2 below).
- `NiceRing` and `NiceField` require additional useful functions: the ability to convert a `number` to an OCaml `float`, and a function for printing a `number` onto the console. You can

register these formatting functions with the OCaml toplevel using the `#install_printer` command; this will cause the toplevel to use that function to print out values of your number type. See the toplevel documentation and the documentation for the OCaml `Format` module for more details.

These concepts (with the exception of `NiceRing` and `NiceField`) are borrowed from the field of abstract algebra[1]. In the same way that modular programming allows us to apply single functions to a large number of data types, these modular definitions have allowed algebraists to prove single theorems that apply to a large number of mathematical structures.

In order for the operations defined by these concepts to make sense, they must obey certain properties (axioms). For reference, we have provided modules that encode these properties as functions. For example, the multiplication operation only makes sense if it is commutative, associative, distributes over addition, and if one is a multiplicative identity. These properties are tested (for individual numbers) by the `times_commutative`, `times_associative`, `times_distributive` and `times_identity` functions. A module `R` that implements `Ring` is only valid if for all inputs a, b, and c, each of the functions of the `RingProperties (R)` module return true. Similarly, the other `Properties` modules (`QuotientProperties`, `GroupProperties`, `FieldProperties`, and so on) encode the properties of the other module types.

Exercise 1: Simple number types

Implement the following modules and functors in `numbers.ml` (you may want to implement some of these in terms of others):

- `module Ints : NiceRing`, using `int` as the number type.
- `module Integers : NiceRing`, using large integers as the number type. You may either use your implementation of big integers from problem set 2, or OCaml's built-in `Big_int` module.
- `module Floats : NiceField`, using `float` as the number type. Because floating point arithmetic is inexact, you should consider two `float`s to be equal if they differ by less than 10^{-6}.

  > **Warning:** This should be your only number module that uses the `float` type (with the exception of the `float_of_number` functions).

- `module Root23 : NiceRing` that contains numbers of the form $a + b\sqrt{2} + c\sqrt{3} + d\sqrt{6}$ where a,b,c and d are integers. In addition to the `NiceRing` functions, this module should expose the `numbers` `sqrt2`, `sqrt3` and `sqrt6` representing $\sqrt{2}$, $\sqrt{3}$ and $\sqrt{6}$ respectively.

[1] note that our groups are technically Abelian groups, and our rings are technically commutative rings with identity

> **Hint 1:** $a_1 + a_2\sqrt{2} + a_3\sqrt{3} + a_6\sqrt{6}$ is zero if and only if a_1, a_2, a_3 and a_6 are zero. This fact may help you with `Root23.(===)`.

> **Hint 2:** For `Root23.is_non_neg`, try working out the solution for numbers of the form $a + b\sqrt{2}$ first, and then writing numbers in the form $(a + b\sqrt{2}) + (c + d\sqrt{2})\sqrt{3}$.

- `module FieldOfFractions (R : NiceRing) : NiceField` should represent numbers as fractions of `R.numbers`. You are not required to store the fractions in lowest terms.
- `module Rationals : NiceField` should implement the rational numbers. You are not required to store the fractions in lowest terms.
- `module Rot15 : NiceField` should contain the rationals as well as the numbers $\cos 45°$, $\sin 45°$, $\cos 30°$, and $\sin 30°$.

  > **Hint:** these can all be expressed in terms of $\sqrt{2}$ and $\sqrt{3}$.

You may find the `include` and `open` statements useful for this problem (and throughout the remainder of the problem set).

Exercise 2: Implement utility functors

In our definitions of `Quotient`, `Group`, `Ring` and so on, we provided a minimal set of functions; this makes it easier to implement these interfaces. For example, a module that implements `OrderedRing` only needs to provide the `is_non_neg` function; but when using numbers from ordered fields, it is useful to have ordering functions such as `(>)`, `(<)`, `(>=)`, and `(<=)`.

Implement the following utility functors in `numbers.ml` to provide these useful operations:

- `module QuotientUtils (Q : Quotient)` should define the not-equal function `(<>)` in terms of the `(===)` function on `Q`.
- `module GroupUtils (G : Group)` should define the binary `(-)` function in terms of the group operations on `G`.
- `module RingUtils (R : Ring)` should define the `number_of_int` function[2], which converts an integer into an `R.number`.
- `module FieldUtils (F : Field)` defines the division function for `F.numbers`.

[2] For those of you who like abstract algebra, the existence and properties of this function means that the integers are initial in the category of rings

- `module OrderedRingUtils (R : OrderedRing)` provides the comparison operations mentioned above (`(<)`, `(>)`, `(<=)` and `(>=)`, `min` and `max`). It should also adapt the ordered ring interface to the standard `Set.OrderedType` interface, so that elements of an ordered ring can be used in OCaml `Sets` and `Maps`.

- `module OrderedFieldUtils (F : OrderedField)` should simply combine the operations from `OrderedRingUtils` and `FieldUtils`.

Once you have defined these modules, you can work with the numbers in a given group, ring or field very easily:

```
# module R  = Numbers.Root23;;
# module RU = Numbers.OrderedRingUtils (R);;
# open R;;
# open RU;;
# #install_printer format;;

# zero;;
- : R.number = 0

# one;;
- : R.number = 1

# one + sqrt2;;
- : R.number = 1+√2

# (one + sqrt2) * (one - sqrt3) + one + sqrt2 - sqrt6;;
- : R.number = 2+2√2-√3-2√6

# let x = one + sqrt2;;
val x : R.number = 1+√2

# let y = (number_of_int 7) - sqrt6;;
val y : R.number = 7-√6

# x * y;;
- : R.number = 7+7√2-2√3-√6
```

> **Note:** The way that the toplevel displays your output will depend on your implementation of `Root23.format`. This function is intended for your own benefit while debugging; you are free to format your numbers any way you choose.

Exercise 3: Implement real numbers

Implement arbitrary precision real numbers in `numbers.ml`. Your implementation should implement the `NiceField` interface.

You should provide a function `approximate` which given an integer k, and a real number x, produces a rational number that is within 10^{-k} of x. That is,

$$\left|\text{x} - (\text{approximate x k})\right| < 10^{-\text{k}}$$

In addition, you should provide a function `create` that accepts a function `f` such that the sequence `f 0`, `f 1`, `f 2`, ... converges to a real number x; `create f` should return x. The input to create is assumed to converge at a rate of 10^{-k}; the behavior of `create` is undefined for sequences that do not converge.

Your implementations of `(===)` and `is_non_neg` are not required to terminate, but they should terminate when possible.

> **Hint:** If you get stuck while trying to come up with a representation for `Real.number`, think carefully about the types and specifications of `create` and `approximate`. If you can implement these functions well, the rest of the `NiceField` operations will follow.

To give you some interesting real numbers to play with, you should provide an exact representation of the numbers π and e. There are many techniques for doing so; one possibility is to use the Taylor series expansions for the arctangent and exponential functions respectively.

Part Two: Geometry

In this portion of the assignment, you will build on the number types you defined in part one to construct a collision-avoiding drag-and-drop library for convex polygons in the plane.

> **Note:** This portion of the assignment depends on part 1, but you do not need to implement all of part 1 to get started on part 2. We recommend that you implement `Numbers.Floats` first, and use it to get started on this portion.

Suppose a user clicks on a shape S and drags it from a point x to a point y (see Figure 1. Suppose further that we wish to prevent S from overlapping with any of the obstacles O_1, O_2, ..., O_n. If placing S at y does not cause it to intersect the obstacles, then it would make sense to simply place S at y. However, if placing S at y would cause an intersection, what should we do?

One approach is to place S at the closest point to y that doesn't cause any overlaps. It turns out that this specification provides a very natural drag and drop experience, because it allows the user to easily adjust the position of adjacent objects. This scheme is illustrated in Figure 1.

We can compute these points as follows. When the user clicks on a shape, we compute a representation of the set of points y such that if S were placed at y, there would be an overlap. If $O = O_1 \cup O_2 \cup \cdots \cup O_n$, then this set consists of the points $\{p - q \mid p \in O, q \in S\}$. This is known as the Minkowski difference of O and S; it is often written $O \ominus S$. It is the shaded area in Figure 1.

Whenever the user moves the mouse to a new location y, we can use this representation to determine whether $y \in O \ominus S$, and if it is, we can find the closest point to y on the boundary of $O \ominus S$. We can then place S at this point.

For this project, we will assume that the shape to be dragged and the obstacles are all convex polygons, represented as lists of points. Points will be represented as pairs of `numbers` from a given `Numbers.OrderedField`. This means that your geometry code must be included in a functor parameterized on an `OrderedField`.

Exercise 4: Implement Minkowski difference for convex polygons

In the file `geometry.ml`, write a function `minkowski_difference_convex` that computes the Minkowski difference of two convex polygons represented as `point list`s.

Computing the Minkowski difference of convex polygons is a straightforward process, because it is simply the convex hull of the differences of all pairs of polygon corners from each of the two input polygons. See Figure 2 for an illustration.

We leave it up to you to find and implement an algorithm for computing the convex hull. We encourage you to use the internet to find an algorithm, and you may even look at open source implementations if you wish, but you must write your code yourself.

Exercise 5: Implement Minkowski difference

The next step is to compute the Minkowski difference of the dragged shape with the entire set of obstacles. This can be accomplished by computing the Minkowski difference of the dragged shape with each obstacle, and the taking the union of the resulting differences.

Computing the union of many polygons is a difficult problem, because the union of two polygons may contain holes, and even degenerate zero or one dimensional holes. These holes are quite important for drag and drop, because they represent spaces between the obstacles where one may want to place the dragged shape.

In the `Region` module we have provided you with an implementation of polygon union. Use this module to write a function `minkowski_difference` that computes the Minkowski difference of a list of obstacles (represented as `point list`s) with the dragged shape. This function should return a `Region.region`.

Part Three: Tangrams

In the last portion of the assignment, you will integrate your geometry code into the tangrams application.

The logic of the game is very simple; users get a handful of simple shapes which they are supposed to arrange into a single large shape. The shapes are shown in Figure 3; in fact this is a screenshot of what you will see as soon as you implement a `NiceField` and run the user interface.

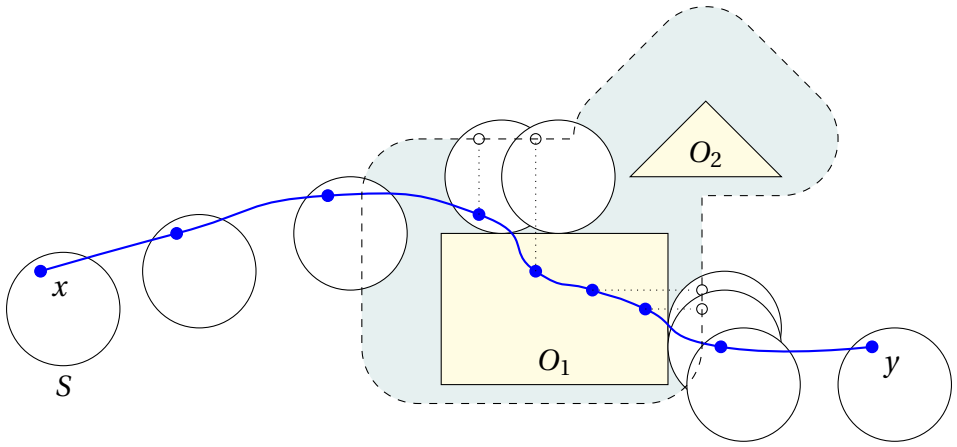

Figure 1: The positions that the circle S takes as it is dragged along the blue path. The yellow shapes O_1 and O_2 are the obstacles. The shaded area contains all points y such that placing S at y would cause S to intersect one of the obstacles. At each point, the circle is placed at the closest point to the mouse location that is not in this set. The circle is only for illustration; you will only be handling polygons.

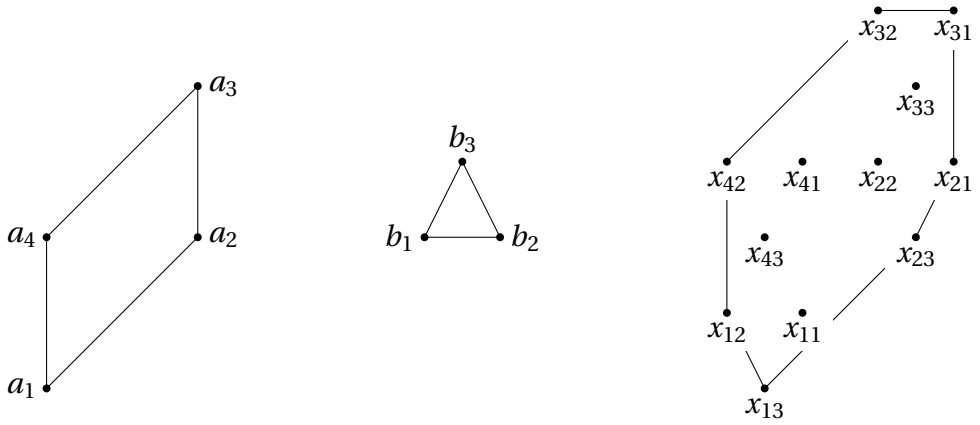

Figure 2: The Minkowski difference of convex polygons can be computed by finding the convex hull of the differences of all pairs of points. In the diagram, $x_{ij} = a_i - b_j$.

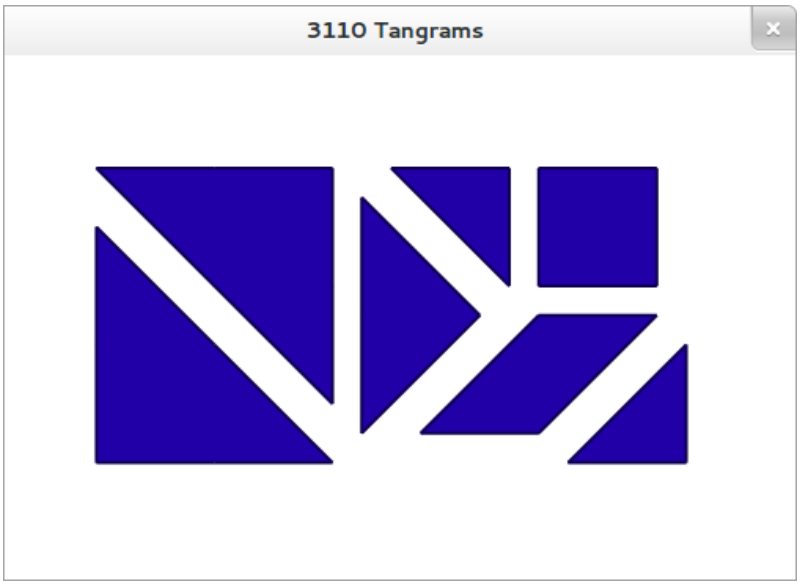

Figure 3: The tangrams user interface

Exercise 6: Implement drag and drop

Your goal is to implement the drag and drop behavior for the application in `game.ml`. The UI code will call the `click`, `move_to` and `unclick` functions whenever the user performs the corresponding actions. It will then call the `obstacles` and `selection` functions to determine what to draw on the screen.

The `obstacles` function should return the list of undragged polygons, while `selection` should return the polygon that is currently being dragged (or `None` if the user is not currently dragging). You are also free to return extra points or lines to draw using the `extra_points` or `extra_lines` functions; these will be drawn by the UI and may aid you in debugging.

As described above, when the user clicks, you should compute the Minkowski difference of the polygon they clicked on with all of the other polygons. When they drag the mouse to a point p, you should find the closest point to p that is not in that difference (you should use `Region.find_closest` for this).

The code that actually constructs and runs the UI is contained in `main.ml`. If you wish to test your code with a different number type you should modify that file to load the appropriate module. You can then build and run the application as described in `README.txt` in the release file.

Part Four: Written questions

Exercise 7: Written questions

Answer these questions in the file `written.txt` or `written.pdf`. Don't forget that most of the `Properties` modules include other `Properties` modules. For example, `FieldProperties` contains `plus_commutative` as well as `times_commutative`, because the `FieldProperties` extend the `GroupProperties`.

(a) We asked you to implement the `Ring` interface for the `Ints` and `Integers`, but integers also have a built-in division operator `(/)`. If we used integer division to implement the `Field` interface, which of the field properties would be violated? Give a set of inputs to the corresponding `FieldProperties` function that demonstrate that the property is violated.

(b) If you run your tangrams game using `FieldOfFractions(Ints).numbers`, you will see strange behavior. This is because some of the properties of `NiceRing` are not satisfied by the `Ints`. Give a set of inputs and a function from `OrderedRingProperties(Ints)` that demonstrate a property that is violated.

(c) Your tangrams game will also not work correctly using the `Floats`, although you may have to play with it a bit more to see visible errors. Again, these failures can be ascribed to a failure to implement the `FieldProperties` perfectly. Give a set of inputs and the name of a `FieldProperties(Float)` function that demonstrate the failure.

(d) Your arbitrary precision real numbers are able to implement every real number exactly. Explain in one or two sentences why they are not the ideal number implementation for your tangrams game.