CSE 341, Fall 2011, Assignment 7
Due: Friday December 9, 11:00PM

Set-up: For this assignment, you will complete and extend two implementations of an interpreter for a small "language" for two-dimensional geometry objects. One implementation is in SML and is mostly completed for you. The other implementation is in Ruby and you will do most of it. The SML solution is structured with functions and pattern-matching. The Ruby solution is structured with subclasses and methods, including a use of double dispatch.

Download and edit `hw7provided.sml` and `hw7provided.rb` from the course website.

Language Semantics:

Our "language" has four kinds of values and three other kinds of expressions. The representation of expressions depends on the metalanguage (SML or Ruby), but we can define the language in general:

- A `PointSet` is a set of two-dimensional points. Each point has an x-coordinate and a y-coordinate, both floating-point numbers. See the important note below on comparing floating-point numbers.
- A `Line` is a non-vertical infinite line in the plane, represented by a *slope* and an *intercept* (as in $y = mx + b$ where m is the slope and b is the intercept), both floating-point numbers.
- A `VerticalLine` is an infinite vertical line in the plane, represented by its x-coordinate.
- A `LineSegment` is a (finite) line segment, represented by the x- and y-coordinates of its endpoints (so four total floating-point numbers).
- An `Intersect` expression is not a value. It has two subexpressions. The semantics is to evaluate the subexpressions (in the same environment) and then return the value that is the intersection (in the geometric sense) of the two subresults. For example, the intersection of two lines could be the `PointSet` with zero points (if the lines are parallel), a `PointSet` with one point (if the lines intersect), or a line (if the two lines have the same slope and intercept).
- A `Let` expression is not a value. It is like let-expressions in other languages we have studied: The first subexpression is evaluated and the result bound to a variable that is added to the environment for evaluating the second subexpression.
- A `Var` expression is not a value. It is for using variables in the environment: We look up a string in the environment to get a geometric value.
- A `Shift` expression is not a value. It has a *deltaX* (a floating-point number), a *deltaY* (a floating-point number), and a subexpression. The semantics is to evaluate the subexpression and then *shift* the result by `deltaX` (in the x-direction; positive is "to the right") and `deltaY` (in the y-direction; positive is "up"). More specifically, shifting for each form of value is as follows:
  - A `PointSet` becomes a `PointSet` of the same size where the point (x, y) becomes the point $(x + deltaX, y + deltaY)$.
  - A `Line` becomes a `Line` with an unchanged slope and an intercept of $b + deltaY - m \cdot deltaX$ where m is the slope and b is the old intercept.
  - A `VerticalLine` becomes a `VerticalLine` shifted by *deltaX*; the *deltaY* is irrelevant.
  - A `LineSegment` has its endpoints shift by *deltaX* and *deltaY*.

Note on Floating-Point Numbers:

Because arithmetic with floating-point numbers can introduce small rounding errors, it is rarely appropriate to use equality to decide if two floating-point numbers are "the same." Instead, the provided code uses a helper function/method to decide if two floating-point numbers are "real close" (for our purposes, within .00001) and all your enhancements should do the same. For example, two points are the same if their x-coordinates are within .00001 and their y-coordinates are within .00001.

Expression Preprocessing:

To simplify the interpreter, we first preprocess expressions. Preprocessing takes an expression and produces a new, equivalent expression with the following invariants:

- No `PointSet` anywhere in the expression has any repeated points (see above about what it means for two points to be the same).
- No `LineSegment` anywhere in the expression has endpoints that are the same as each other. Such a line-segment should be replaced with a one-element `PointSet` containing the repeated end-point.
- Every `LineSegment` has its first endpoint (the first two `real` values in SML) to the left (lower x-value) of the second endpoint. If the x-coordinates of the two endpoints are the same, then the first endpoint has its first endpoint below (lower y-value) the second endpoint. For any `LineSegment` not meeting this requirement, replace it with a `LineSegment` with the same endpoints reordered.

The SML Code:

Most of the SML solution is given to you. All you have to do is add preprocessing (problem 1) and `Shift` expressions (problem 2). The sample solution added less than 50 lines of code. As always, line counts are just a rough guide.

Notice the SML code is organized around a datatype-definition for expressions, functions for the different operations, and pattern-matching to identify different cases. The interpreter `eval_prog` uses a helper function `intersect` with cases for every combination of geometric value.

The Ruby Code:

Much of the Ruby solution is not given to you. To get you started in the desired way, we defined `GeometryExpression` with useful helper methods and two of the subclasses, `PointSet` and `Line`. These subclasses are complete except for the methods needed to implement intersection (problem 4). You need to implement the subclasses for other kinds of expressions (`VerticalLine`, `LineSegment`, `Let`, `Var`, `Shift`, and `Intersect`), mostly in problem 3. Some more specific information is given in problems 3 and 4. The sample solution added 200–250 lines of Ruby code, many of which were `end`. As always, line counts are just a rough guide.

Notice the Ruby code is organized around subclasses where each class has a method for various operations. All kinds of expressions need methods for preprocessing and evaluation. The value subclasses also need methods for shifting and intersection.

Your Ruby code should follow these general guidelines:

- All your geometry-expression objects should be *immutable*: assign to instance variables only when constructing an object. To "change" a field, create a new object.
- Your geometry-expression objects can/should have public getter methods: like in the SML code, the entire program can assume the expressions have various coordinates, subexpressions, etc.
- Unlike in SML, you do not need to define exceptions since `raise` can just be called with a string.
- Use arrays as appropriate, such as for points (2-element arrays), lists of points (in point-sets), and environments (holding 2-element arrays to implement association lists).
- Follow OOP-style. In particular, operations should be instance methods and you should not use methods like `is_a?` or `instance_of?`. See problem 4 for more discussion of the methods `isLine` and `isVerticalLine` in class `GeometryExpression`.

Advice for Approaching the Assignment:

- Understand the high-level structure of the code and how the SML and Ruby are structured in different ways before diving into the details.
- Approach the questions in order even though there is some flexibility (e.g., it is possible to do the Ruby problems before the SML problems).
- Because almost all the SML code is given to you, for much of the Ruby implementation, you can port the corresponding part of the SML solution. Doing so makes your job easier (e.g., you need not re-figure out facts about geometry). Porting existing code to a new language is a useful and realistic skill to develop. It also helps teach the similarities and differences between languages.

The Problems (Finally):

1. Implement an SML function `preprocess_prog` of type `geom_exp -> geom_exp` (and any helper functions you need) to implement expression preprocessing as defined above. The idea is that evaluating program `e` would be done with `eval_prog (preprocess_prog e, [])` where the second argument is the empty list for the empty environment.
2. Add shift expressions as defined above to the SML implementation by adding the constructor `Shift of real * real * geom_exp` and changing functions as necessary.
3. Complete the Ruby implementation *except for intersection*. Define classes `VerticalLine`, `LineSegment`, `Let`, `Var`, and `Shift`, including methods `initialize`, `eval_prog`, and `preprocess_prog`.
   - To make it easier for the course staff to test your code, have your `initialize` methods take the same arguments as the SML constructors in the same order.
   - Given immutability, `preprocess_prog` methods should, like in SML, return a new object unless `self` is an appropriate result.
   - Analogous to SML, an overall program `e` would be evaluated via `e.preprocess_prog.eval_prog []` (notice we use an array for the environment).
4. Complete your Ruby solution by implementing the `Intersect` class. To do so, the geometric-value classes should have `intersect` methods that are implemented via double-dispatch to cover all 16 cases (6 of each can be handled by calling 6 other cases since intersection is always commutative).

   In porting some of the SML cases involving line segments, the `isLine` and `isVerticalLine` methods are helpful, and you should override these methods appropriately. (We could also have an `isPointSet` method, but instead just assume that if certain results are not lines, then they must be point-sets.) *Do not overuse the methods* `isLine` *and* `isVerticalLine`. They are only useful where, like in the ML code, we are computing the intersection of a line segment with something and we need to know the "shape" of an intersection with the line containing the line segment.
5. **Challenge Problem:** Make a third version of your solution in **Java**. Follow the structure of your Ruby solution. Use abstract methods as necessary for type-checking.
6. **Challenge Problem:** Add **circles** to the language by extending your SML solution and your Ruby solution appropriately. Represent a circle via three floating-point numbers: the x-coordinate of the center, the y-coordinate of the center, and the radius. Have preprocessing ensure no radius is negative (nor zero: such a circle is actually a single point). Feel free to consult textbooks or the web for the geometry of how to determine the intersection of two circles or the intersection of a circle and a line — it is more difficult than you might think.

Turn-in Instructions

- Put your solutions in two files, `lastname_hw7.sml` and `lastname_hw7.rb`, which should include the provided code, where `lastname` is your last name. Put tests you wrote in `lastname_hw7_test.sml` and `lastname_hw7_test.rb`.
- Turn in your **four** files using the Catalyst dropbox link on the course website.
- If you do problem 5, put your solution in `lastname_hw7.java`, with tests in `lastname_hw7_test.java`. For this to work, you *cannot* declare your classes public, since a public class `Foo` must be in file `foo.java`. So give classes package-level visibility by writing `class Foo` instead of `public class Foo`.