CSE 341, Winter 2022, Assignment 7
Due: Friday March 11, 5:00PM

Set-up:

You will complete and extend two implementations of an interpreter for a small "language" for two-dimensional geometry objects. An OCaml implementation is mostly completed for you. A Racket implementation is mostly not completed. The OCaml implementation is structured with functions and pattern-matching. The Racket implementation is structured with subclasses and methods, including some mind-bending double dispatch and other dynamic dispatch to stick with an OOP style even where your instructor thinks the functional style is easier to understand. *Your Racket code should take a full OOP approach.*

Download and edit `hw7.ml` and `hw7.rkt` from the course website. Some example tests are also provided.

Language Semantics:

Our "language" has five kinds of values and four other kinds of expressions. The representation of expressions depends on the metalanguage (OCaml or Racket), with this same semantics:

- A `NoPoints` or `no-points%` represents the empty set of two-dimensional points.
- A `Point` or `point%` represents a two-dimensional point with an x-coordinate and a y-coordinate. Both coordinates are floating-point numbers.
- A `Line` or `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` or `vertical-line%` is an infinite vertical line in the plane, represented by its x-coordinate.
- A `LineSegment` or `line-segment%` is a (finite) line segment, represented by the x- and y-coordinates of its endpoints (so four total floating-point numbers).
- An `Intersect` or `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 geometric intersection of the two subresults. For example, the intersection of two lines could be one of:
  - `NoPoints` or `no-points%`, if the lines are parallel
  - a `Point` or `point%`, if the lines intersect
  - a `Line` or `line%`, if the lines have the same slope and intercept (see the note below about what "the same" means for floating-point numbers)
- A `Let` or `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` or `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` or `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:
  - `NoPoints` or `no-points%` remains the same.
  - A `Point` or `point%` representing (x, y) becomes a `Point` or `point%` representing $(x + deltaX, y + deltaY)$.

- A `Line` or `line%` with slope m and intercept b becomes a `Line` or `line%` with slope m and an intercept of $b + deltaY - m \cdot deltaX$.
- A `VerticalLine` or `vertical-line%` becomes a `VerticalLine` or `vertical-line%` shifted by $deltaX$; the $deltaY$ is irrelevant.
- A `LineSegment` or `line-segment%` 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 to decide if two floating-point numbers are "close enough" (for our purposes, within .00001) and all your code should follow this approach as needed. 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 `LineSegment` or `line-segment%` anywhere in the expression has endpoints that are the same as (i.e., real close to) each other. Such a line-segment should be replaced with the appropriate `Point` or `point%`. For example, in OCaml `LineSegment (3.2,4.1,3.2,4.1)` preprocesses to `Point (3.2,4.1)`.
- Every `LineSegment` (or `line-segment%`) anywhere in the expression has its first endpoint (the first two `float` values in OCaml) to the *right* (*higher* x-value) of the second endpoint. If the x-coordinates of the two endpoints are the same (real close), then the `LineSegment` (`line-segment%`) has its first endpoint *above* (*higher* y-value) the second endpoint. For any line-segment not meeting this requirement, replace it with a line-segment with the same endpoints reordered. (Admittedly this order can seem "backwards" especially when reading the OCaml code but sometimes reading code involves accepting a convention decided by someone else. Maintain this convention in both the OCaml and the Racket code.)

The OCaml Code:

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

Notice the OCaml code is organized around a variant type 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 (so with 5 kinds of values there are 25 cases though some are handled together via pattern-matching). The surprisingly complicated part is the algorithm for intersecting two line segments.

The Racket Code:

Much of the Racket solution is not given to you. To get you started in the desired way, we have defined classes for each kind of expression in our language, as well as appropriate superclasses. We have implemented parts of each class and left comments with what you need to do to complete the implementation as described in more detail in problems 3 and 4. The sample solution added about 120 lines of Racket code. As always, line counts are just a rough guide.

Notice the Racket code is organized around classes where each class has methods for various operations. All kinds of expressions need methods for preprocessing and evaluation. They are subclasses of `geometry-expression%` just like all OCaml constructors are part of the `geom_exp` type (though the `geometry-expression%` class turns out not to be so useful). The value subclasses also need methods for shifting and intersection and they subclass `geometry-value%` so that some shared methods can be inherited (in analogy with some uses of wildcard patterns and helper functions in OCaml).

Your Racket code should follow these general guidelines:

- All your geometry-expression objects should be *immutable*: assign to private state only when initializing an object. To "change a field," create a new object.
- The geometry-expression objects have public getter methods: like in the OCaml code, the entire program can assume the expressions have various coordinates, subexpressions, etc.
- Unlike in OCaml, you do not need to define exceptions since without a type-checker we can just "assume" the right objects are used in the right places. You can also use `error` as appropriate.
- Follow OOP-style. In particular, operations should be methods and you should **not use methods like `is-a?`, `implementation?`, etc. This makes problem 4 much more difficult, which is the purpose of the problem.**

Advice for Approaching the Assignment:

- Understand the high-level structure of the code and how the OCaml and Racket files 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 Racket problems before the OCaml problems).
- Because almost all the OCaml code is given to you, for much of the Racket implementation, you can port the corresponding part of the OCaml solution. Doing so makes your job **much** 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.
- Be sure to test each line of your Racket code. Dynamically typed languages require testing things that other languages catch for you statically.

The Problems (Finally):

1. Implement an OCaml function `preprocess_prog` of type `geom_exp -> geom_exp` 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 `[]` is the empty list for the empty environment.
2. Add shift expressions as defined above to the OCaml implementation by adding the constructor `Shift of float * float * geom_exp` to the definition of `geom_exp` and adding appropriate branches to `eval_prog` and `preprocess_prog`. (The first `float` is *deltaX* and the second is *deltaY*.) Do *not* change other functions. In particular, there is no need to change `intersect` because this function is used only for values in our geometry language and shift expressions are not geometry values.
3. Complete the Racket implementation *except for intersection*, which means skip for now additions to the `intersect%` class and, more importantly, methods related to intersection in other classes. Do not modify the code given to you. Follow this approach:
   - Every subclass of `geometry-expression%` should have a `preprocess-prog` method that takes no arguments and returns the geometry object that is the result of preprocessing `this`. To avoid mutation, return a new instance of the same class unless it is trivial to determine that `this` is already an appropriate result.
   - Every subclass of `geometry-expression%` should have an `eval-prog` method that takes one argument, the environment, which you should represent as a list of pairs: a Racket string (the variable name) in the car and an object that is a value in our language in the cdr. As in any interpreter, pass the appropriate environment when evaluating subexpressions. (This is fairly easy since we do not have closures.) To make sure you handle both scope and shadowing correctly:

  - Do not ever mutate an environment; create a new environment as needed instead.
  - The `eval-prog` method in `var%` is given to you. Make sure the environments you create work correctly with this definition.

  The result of `eval-prog` is the result of "evaluating the expression represented by `this`," so, as we expect with OOP style, the cases of OCaml's `eval_prog` are spread among our classes, just like with `preprocess_prog`.

- Every subclass of `geometry-value%` should have a `shift` method that takes two arguments `dx` and `dy` and returns the result of shifting `this` by `dx` and `dy`. In other words, all values in the language "know how to shift themselves to create new objects." Hence the `eval-prog` method in the `shift%` class should be very short.
- Remember you should not use Racket features like `is-a?` or `implementation?`.
- Analogous to OCaml, an overall program `e` would be evaluated via `(send (send e preprocess-prog) eval-prog null)`.

4. Implement intersection in your Racket solution following the directions here, in which we require both double dispatch and a separate use of dynamic dispatch for the line-segment case. Remember all the different cases in OCaml will appear in the Racket OOP solution, just arranged very differently.

   - Implement `preprocess-prog` and `eval-prog` in the `intersect%` class. This is not difficult, much like your prior work in the `shift%` class is not difficult. This is because every subclass of `geometry-value%` will have an `intersect` method that "knows how to intersect itself" with another geometry-value passed as an argument.
   - Every subclass of `geometry-value%` needs an `intersect` method, but these will be short. The argument is another geometry-value, but we do not know what kind. So we use double dispatch and call the appropriate method on the argument passing `this` to the method. For example, the `point%` class has an `intersect` method that calls `intersect-point` with `this`.
   - So methods `intersect-no-points`, `intersect-point`, `intersect-line`, `intersect-vertical-line`, and `intersect-line-segment` defined in each of our 5 subclasses of `geometry-value%` handle the 25 possible intersection combinations:
     - The 9 cases involving `no-points%` are done for you. See the `geometry-value%` class — there is nothing more you need to do.
     - Next do the 9 remaining cases involving combinations that do not involve line segments. You will need to understand double-dispatch to avoid `is-a?`. As in the OCaml code, 3 of these 9 cases can just use one of the other cases because intersection is commutative.
     - What remains are the 7 cases where one value is a `line-segment%` and the other is not a `no-points%`. These cases are all "done" for you because all subclasses of `geometry-value%` inherit an `intersect-line-segment` method that will be correct for all of them. But it calls `intersect-with-segment-as-line-result`, which you need to implement for each subclass of `geometry-value%`. Here is how this method should work:
       * It takes one argument, which is a line segment. (In OCaml the corresponding variable was a `float*float*float*float`, but here it will actually be an instance of `line-segment%` and you can use the getter methods `get-x1`, `get-y1`, `get-x2`, and `get-y2` as needed.)
       * It *assumes* that `this` is the intersection of (1) some not-provided geometry-value and (2) the line (vertical or not) containing the segment given as an argument.
       * It *returns* the intersection of the not-provided geometry-value and the segment given as an argument.

     Together the 5 `intersect-with-segment-as-line-result` methods you write will implement the same algorithm as on lines 79–130 of the OCaml code.

5. **Challenge Problem:** Make a third version of your solution in **Java**. Follow the structure of your Racket solution, with no use of Java's `instanceof` or type casts. Use abstract methods as necessary for type-checking.

Turn-in Instructions: Upload **four** files to Gradescope:

- The OCaml and Racket files with your solutions
- OCaml and Racket files used for testing
- If you do the challenge problem, turn in your Java files as well.