**Instructions.** Your task is to write an SML program and a Ruby program, as well as test cases for those programs. There are no written problems on this homework. See the end of this document for instructions on how to submit your solution. In case you’re curious, our reference solution to the (non-karma) programming problems contains less than 250 lines of code, excluding comments, blank lines, and the provided code described next.

**Provided code.** Download `hw5template.sml` and `hw5template.rb` from the course website. As usual, there are parts of the provided code that you may not change. We also provide some public test cases on the course website; these are by no means exhaustive, but you might find them helpful as a basic sanity check on your implementation.

**Geopl**

Geopl (a geometric programming language, pronounced /ˈgee-pull/, rhymes with “people”) is our own domain-specific language for two-dimensional geometric objects. Geopl has five kinds of values and four other kinds of expressions:

- **Values and their representations:**
  - A **Point** value is a two-dimensional point with an x-coordinate and a y-coordinate, both floating-point numbers.
  - **NoPoints**, a value, is the empty set of points.
  - A **Line** value 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** value is an infinite vertical line in the plane, represented by its x-coordinate.
  - A **LineSegment** value is a (finite) line segment, represented by the x- and y-coordinates of its endpoints (hence a total of four floating-point numbers).

- **Expressions and their semantics (all values are also expressions):**
  - A **Let** expression has two subexpressions and a bound variable name. The first subexpression is evaluated and the result bound to the variable, which is added to the dynamic environment for evaluating the second subexpression.
  - A **Var** expression is for using variables in the environment: it evaluates to the geometric value that the variable is bound to in the dynamic environment.
  - An **Intersect** expression has two subexpressions. Both subexpressions are evaluated in the same dynamic environment. The result of the entire expression is a value that is the geometric intersection of the two subresults. For example, the intersection of two lines could be **NoPoints** (if the lines are parallel), a **Point** (if the lines intersect), a **Line** (if the two lines have the same slope and intercept), or a **VerticalLine**.
  - A **Shift** expression has a $\Delta X$ (a floating-point number), a $\Delta Y$ (a floating-point number), and a subexpression. The semantics is to evaluate the subexpression and then shift the result by $\Delta X$ (in the x-direction; positive is “to the right”) and $\Delta Y$ (in the y-direction; positive is “up”). More specifically, shifting for each form of value is as follows:
    - **NoPoints** remains **NoPoints**.
    - A **Point** representing $(x, y)$ becomes a **Point** representing $(x + \Delta X, y + \Delta Y)$.
    - A **Line** with slope $m$ and intercept $b$ becomes a **Line** with an unchanged slope and an intercept of $b + \Delta Y - m \cdot \Delta X$.
    - A **VerticalLine** becomes a **VerticalLine** shifted by $\Delta X$; the $\Delta Y$ is ignored.
    - A **LineSegment** has its endpoints shift by $\Delta X$ and $\Delta Y$.  

---

1
Your task. Your task is to implement two interpreters for Geopl, one in SML, the other in Ruby. We provide you with some of the code for these interpreters. Here’s some really important advice:

- Understand the high-level structure of the code, and how SML and Ruby use different high-level structures, before diving into the details.
- For most of the Ruby implementation, you can port the corresponding part of the SML solution. Doing so makes your task easier (e.g., you don’t have figure out geometric facts).

Porting existing code to a new language is a useful and realistic skill to develop. It also helps you learn the similarities and differences between languages.

SML interpreter for Geopl. We have already implemented most of an interpreter for Geopl in SML; that implementation is part of the provided code for the homework. Invoke the SML interpreter with `eval e`, where `e` is a Geopl expression.

The implementation is organized around a datatype-definition for expressions, functions for the different operations, and pattern-matching to identify different cases. Your task (further explained below) is to add preprocessing (problem 1) and Shift expressions (problem 2). Our reference solution adds about 25 lines of code.

Ruby interpreter for Geopl. We have provided the beginnings of a Ruby implementation for you, but most of the Ruby solution is not given to you. Invoke the Ruby interpreter with `e.eval`, where `e` is a Geopl expression.

The implementation is organized around subclasses for expressions, where each subclass has methods for the different operations. Your task (further explained below) is to finish that implementation. To get you started on the right path, we have defined classes for each kind of expression, as well as appropriate superclasses. We have implemented parts of each class, leaving comments with what you need to do to complete the implementation (problems 3 and 4). Our reference solution adds about 200 lines of Ruby code, many of which are simply `end`.

Your Ruby code should adhere to these guidelines:

- All your geometry-expression objects should be immutable: The only method that assigns to instance variables should be `initialize`. No other method should ever mutate an instance variable. To “change” an instance variable, instead create a new object with a new value for that variable.
- Geometry-expression objects can and should have public getter methods: like in the SML code, the entire program can assume the expressions have various coordinates, subexpressions, etc.
- Raise an `UnboundVariableError` when necessary. Unlike in SML, you don’t need to raise other kinds of exceptions, because with dynamic typing the programmer just assumes that the right objects are used in the right places.
- Follow “pure” OOP style. In particular, operations should be instance methods. And your solution must implement intersection with double dispatch. So you may not use methods like `is_a?`, `instance_of?`, `class`, etc. This restriction does make problem 4 more difficult, but its purpose is to expose you to double dispatch.

It is critical to test your Ruby code thoroughly. Dynamically-typed languages require testing to expose bugs that statically-typed languages automatically find for you. For example, it’s easy to to write `[1.0, 2.0]` where you mean really mean `[[1.0, 2.0]]`—and there’s no type system to catch that error for you. As another example, it’s easy to misspell method names, instance variables, etc. Unless you write enough test cases to exercise every single line of code, you can’t be sure you spelled all those names correctly. On this assignment, more so than any of the previous ones, it is easy to believe your code is correct even when it doesn’t work correctly!
Expression preprocessing. To simplify both interpreters, we first preprocess expressions. Preprocessing takes an expression and produces a new, equivalent expression with the following invariants:

- No LineSegment anywhere in the expression has endpoints that are the same as each other. Such a line-segment should be replaced with the appropriate Point. For example (in SML syntax), LineSegment(3.2, 4.1, 3.2, 4.1) should be replaced with Point(3.2, 4.1).

- 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.

What it means to be the “same.” Because arithmetic with floating-point numbers can introduce small rounding errors, it is inappropriate to use equality to decide if two floating-point numbers are the same. Instead, the provided code uses a helper function (SML) or method (Ruby) to decide whether two floating-point numbers are “really close.” All your enhancements must use that same function or method.

Problems. We strongly recommend doing these problems in order.

1. Finish the implementation of SML function preprocess_prog to implement expression preprocessing as defined above. (Note how this adds a new operation to the program.)

2. Add shift expressions as defined above to the SML implementation by adding the constructor Shift of real * real * geopl_exp to the definition of geopl_exp. (The first real is ΔX and the second is ΔY.) Add appropriate branches to eval_prog and preprocess_prog. Do not change any other functions. In particular, there is no need to change intersect. (Note how this adds a new variant to the program. The compiler tells you where to add new branches by non-exhaustive pattern match errors.)

3. Complete the Ruby implementation except for intersection. (So don’t add anything to the Intersect class, and don’t modify methods related to intersection in other classes.) Don’t modify the provided code in any way that would cause our test cases to break. For example, don’t change class names, number or order of arguments to methods, etc. Follow this approach:

- Every subclass of GeoplExp should have a preprocess_prog method that takes no arguments and returns the object that is the result of preprocessing self. To avoid mutation, return a new instance of the same class—unless it is trivial to determine that self is already an appropriate result.

- Every subclass of GeoplExp should also have an eval_prog method that takes one argument, the environment. Represent an environment as an array whose elements are themselves two-element arrays: the variable name, as a Ruby string in index 0; and the geometric value, as an object in index 1. Pass the appropriate environment when evaluating subexpressions. The result of eval_prog is the result of evaluating the expression represented by self.

- Every subclass of GeoplVal should have a shift method that takes two arguments dx and dy and returns the result of shifting self by dx and dy. In other words, all values “know how to shift themselves.” So the eval_prog method in Shift should be quite short.

- Remember that you may not use methods like is_a?, instance_of?, class, etc.

(Not how you are adding new operations and new variants to the program as you port the SML code.)

4. Complete your Ruby solution by implementing intersection, according to the following instructions. (Note how you are adding a new operation to the program, and how you are using double dispatch to do so.) Remember that all the different cases in your SML solution will appear somewhere in your Ruby solution, just arranged differently.
• Implement `preprocess_prog` and `eval_prog` in the `Intersect` class. For `eval_prog`, assume that every subclass of `GeoplVal` has an `intersect` method (which you’ll implement, next) that takes another geometric value and returns the result of intersecting `self` and that value. In other words, all values “know how to intersect themselves.” This is about as easy as implementing `eval_prog` in `Shift` was, for similar reasons.

• Implement an `intersect` method in every subclass of `GeoplVal`. These will all be short. The argument is another geometric value, but the method doesn’t know what kind of value. So use `double dispatch` to call the appropriate method on the argument, passing `self` to the method. For example, the `Point` class has an `intersect` method that calls `intersect_Point` with `self`.

• Implement all the remaining methods needed for double dispatch: `intersect_NoPoints`, `intersect_Point`, `intersect_Line`, `intersect_VerticalLine`, and `intersect_LineSegment`. There are 25 possible intersection combinations:
  – The nine combinations involving `NoPoints` are already done for you in the `GeoplVal` class. There’s nothing left for you to do here.
  – Implement the nine remaining combinations that do not involve a `LineSegment`. Remember that you may not use methods like `is_a?`, `instance_of?`, `class`, etc. You will need to understand double dispatch to avoid such methods. As in the SML code, three of these nine combinations can delegate to another combination because of symmetry of intersection.
  – Now there are seven combinations remaining. In each, one value is a `LineSegment` and the other is not `NoPoints`. Class `GeoplVal` already implements `intersect_LineSegment` correctly for all these seven combinations. But that implementation calls `intersect_LineSegment_as_Line`, which you will need to implement for each subclass of `GeoplVal`. Here is how that method should work:
    * It takes one argument `seg`, which is a `LineSegment`.
    * It assumes that `self` is already the intersection of some unknown geometric value with some line \( \ell \), and that \( \ell \) contains `seg` as a segment.
    * It returns the intersection of `self` with `seg`.

5. **Karma problem:** *This is an extremely educational karma problem. We recommend that you do it, even if you don’t normally do karma problems, and even though your work won’t be graded.* Implement a third interpreter for Geopl by porting your Ruby implementation to Java. Follow the structure of your Ruby code. Don’t use type casts or `instanceof`. You will need to use abstract classes and methods.

**Further Instructions**

**Testing.** You are required to test your functions. Put your testing code in separate files. We will not directly grade it, but you must turn it in. Good test cases might help you get some partial credit if your solution is erroneous.

**Submission Instructions**

Submissions that do not adhere to these criteria will lose points:

• **Do not archive** (zip, rar, etc.) your files.

• Put all your SML solution code in one file, `netid_hw5.sml`, your Ruby solution code in another file, `netid_hw5.rb`, and your Java solution code (if you do the karma problem) in a third file, `netid_hw5.java`.

• Put all the tests you wrote in two or three other files, `netid_hw5_test.sml`, `netid_hw5_test.rb`, and `netid_hw5_test.java`. 
• The first line of every file should be a comment with your name, GW NetId, and the phrase Homework 5.

• Upload all the files to the Homework 5 assignment on BlackBoard.

Evaluation Criteria

Solutions will be evaluated on correctness with respect to the specifications in this assignment; style, including indentation and line breaks, with respect to the SML style guide on the course website (we don’t have a Ruby style guide); elegance, which is an ineffable quality that includes beauty, effectiveness, and simplicity; and adherence to using only those language features permitted in this homework.