Chapter 3

Demonstration

My system uses this idea of manipulating diagrams "in the mind's eye" to explore and discover geometry theorems. Before describing its internal representations and modules, I will present and discuss several sample interactions with the system. Further implementation details can be found in subsequent chapters.

The system is divided into four main modules: an imperative construction system, a perception-based analyzer, a declarative constraint solver, and a synthesizing learning module. The following examples explore interactions with these modules in increasing complexity.

3.1 Imperative Figure Construction

At its foundation, the system provides a language and engine for performing geometry constructions and building figures.

Example 3.1 presents a simple specification of a figure. Primitives of points, lines, segments, rays, and circles can be combined into polygons and figures and complicated constructions such as the perpendicular bisector of a segment can be abstracted into higher-level construction procedures. The custom special form `let-geo*` emulates the standard `let*` form in Scheme but also annotates the resulting objects with the names and dependencies as specified in this construction.

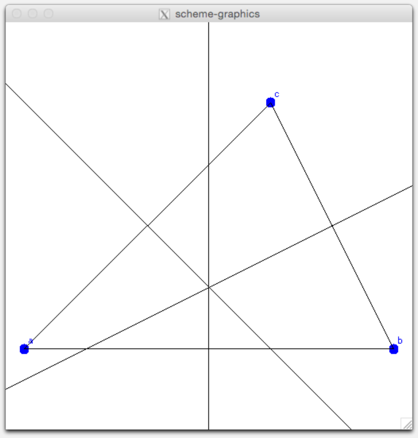
Code Example 3.1: Basic Figure Example

```
(define (triangle-with-perp-bisectors)
  (let-geo* ((a (make-point 0 0))
             (b (make-point 1.5 0))
             (c (make-point 1 1))
             (t (polygon-from-points a b c))
             (pb1 (perpendicular-bisector (make-segment a b)))
             (pb2 (perpendicular-bisector (make-segment b c)))
             (pb3 (perpendicular-bisector (make-segment c a))))
    (figure t pb1 pb2 pb3)))
```

Given such an imperative description of a figure, the system can construct and display an instance of the figure as shown in Example 3.2. The graphics system uses the underlying X window system-based graphics interfaces in MIT Scheme, labels named points (a, b, c), and repositions the coordinate system to display interesting features.

Interaction Example 3.2: Rendering the Basic Figure

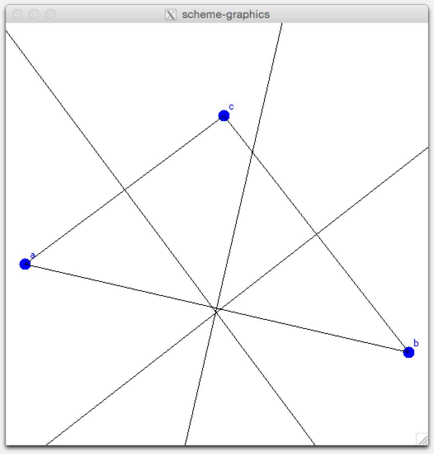
```
=> (show-figure (triangle-with-perp-bisectors))
```

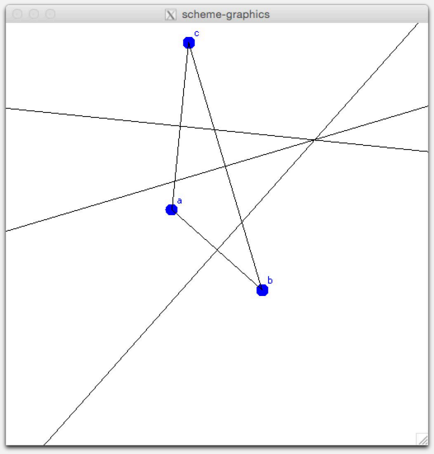


In the first figure, the coordinates of the point were explicitly specified yielding a deterministic instance of the figure. However, as geometry figures often involve arbitrary choices, the construction abstractions support random choices. Figure 3.3 demonstrates the creation of a figure involving an arbitrary triangle. The second formulation `(simple-random-triangle-with-perp-bisectors)` displays a syntax extension provided by `let-geo*` that shortens the common pattern of accessing and naming the components of a random object.

Interaction Example 3.3: Introducing Randomness

```
(define (random-triangle-with-perp-bisectors)
  (let-geo* ((t (random-triangle))
             (a (polygon-point-ref t 0))
             (b (polygon-point-ref t 1))
             (c (polygon-point-ref t 2))
             (pb1 (perpendicular-bisector (make-segment a b)))
             (pb2 (perpendicular-bisector (make-segment b c)))
             (pb3 (perpendicular-bisector (make-segment c a))))
    (figure t pb1 pb2 pb3)))

(define (simple-random-triangle-with-perp-bisectors)
  (let-geo* (((t (a b c)) (random-triangle))
             (pb1 (perpendicular-bisector (make-segment a b)))
             (pb2 (perpendicular-bisector (make-segment b c)))
             (pb3 (perpendicular-bisector (make-segment c a))))
    (figure t pb1 pb2 pb3)))
```

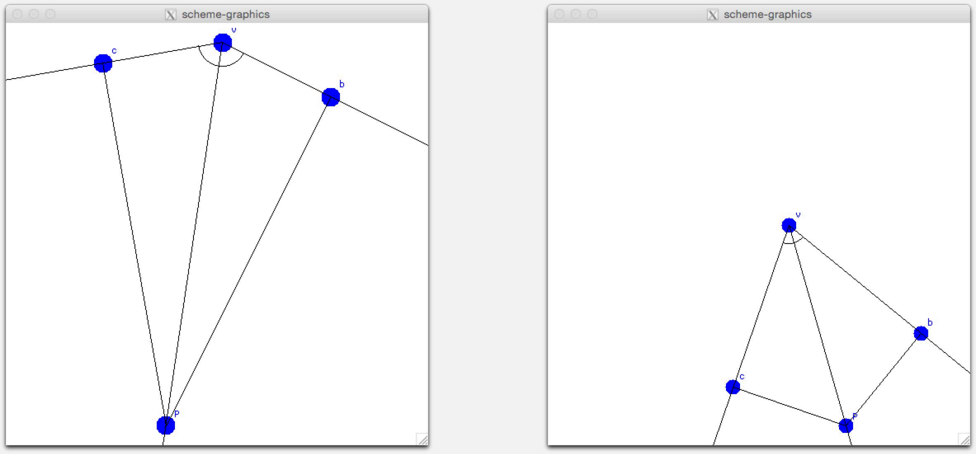




Finally, as examples of more involved constructions, Examples 3.4 and 3.5 demonstrate working with other objects (angles, rays, circles) and construction procedures. Notice that in the angle bisector example the pattern matching syntax extracts the components of an angle (ray, vertex, ray) and segment (endpoints), and that in the Inscribed/Circumscribed example, some intermediary elements are omitted from the final figure list and will not be displayed or analyzed.

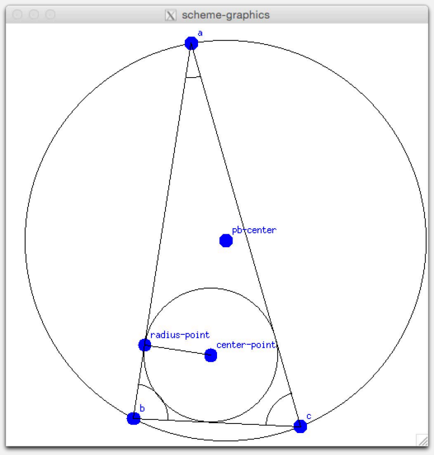
Interaction Example 3.4: Angle Bisector Distance

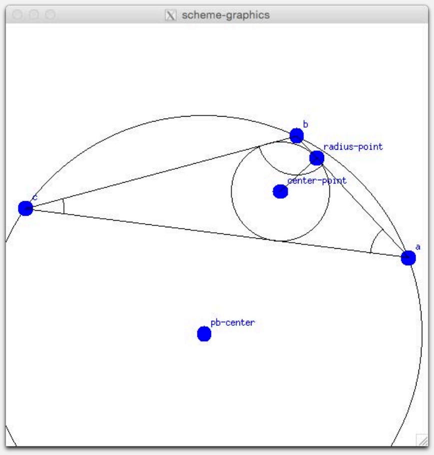
```
(define (angle-bisector-distance)
  (let-geo* (((a (r-1 v r-2)) (random-angle))
             (ab (angle-bisector a))
             (p (random-point-on-ray ab))
             ((s-1 (p b)) (perpendicular-to r-1 p))
             ((s-2 (p c)) (perpendicular-to r-2 p)))
     (figure a r-1 r-2 ab p s-1 s-2)))

=> (show-figure (angle-bisector-distance
```



Interaction Example 3.5: Incircle and Circumcircle

```
(define (inscribed-circumscribed)
  (let-geo* (((t (a b c)) (random-triangle))
             (((a-1 a-2 a-3)) (polygon-angles t))
             (ab1 (angle-bisector a-1))
             (ab2 (angle-bisector a-2))
             ((radius-segment (center-point radius-point))
              (perpendicular-to (make-segment a b)
                                (intersect-linear-elements ab1 ab2)))
             (incircle (circle-from-points
                        center-point
                        radius-point))
             (pb1 (perpendicular-bisector
                   (make-segment a b)))
             (pb2 (perpendicular-bisector
                   (make-segment b c)))
             (pb-center (intersect-lines pb1 pb2))
             (circum-cir (circle-from-points
                          pb-center
                          a)))
    (figure t a-1 a-2 a-3
            pb-center
            radius-segment
            incircle
            circum-cir)))

=> (show-figure (inscribed-circumscribed))
```





3.2 Perception and Observation

Interaction Example 3.6: Simple Analysis

```
=> (all-observations (triangle-with-perp-bisectors))

(#[observation 77] #[observation 78] #[observation 79] #[observation 80])

=> (pprint (all-observations (triangle-with-perp-bisectors)))

((concurrent pb1 pb2 pb3)
 (perpendicular pb1 (segment a b))
 (perpendicular pb2 (segment b c))
 (perpendicular pb3 (segment c a)))
```

3.3 Mechanism-based Declarative Constraint Solver

The first two modules focus on performing imperative constructions to build diagrams and analyze them to obtain interesting symbolic observations and relationships. Alone, these modules could assist a mathematician in building, analyzing, and exploring geometry concepts.

However, an important aspect of automating learning theorems and definitions involves reversing this process and obtaining instances of diagrams by solving provided symbolic constraints and relationships. When we are told to “Imagine a triangle ABC in which AB = BC”, we visualize in our minds eye an instance of such a triangle before continuing with the instructions.

Thus, the third module is a declarative constraint solver. To model the physical concept of building and wiggling components until constraints are satisfied, the system is formulated around solving mechanisms built from bars and joints that must satisfy certain constraints. Such constraint solving is implemented by extending the Propagator Model created by Alexey Radul and Gerald Jay Sussman [**??**] to handle partial information and constraints about geometry positions. Chapter 7 discusses further implementation details.

3.3.1 Bars and Joints

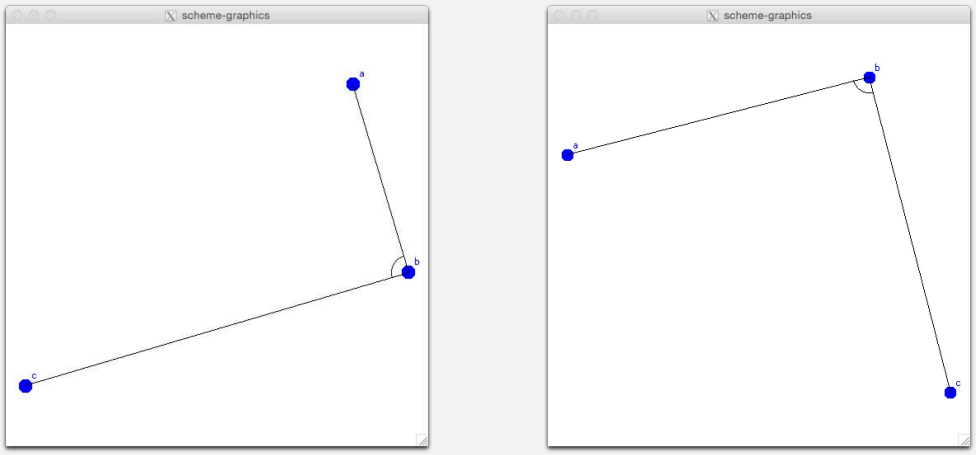
Example 3.7 demonstrates the specification of a very simple mechanism.

Code Example 3.7: Very Simple Mechanism

```
(define (simple-mechanism)
  (m:mechanism
   (m:make-named-bar 'a 'b)
   (m:make-named-bar 'b 'c)
   (m:make-named-joint 'a 'b 'c)
   (m:c-right-angle (m:joint 'b))))
```

Interaction Example 3.8: Solving the Simple Mechanism

```
=> (m:run-mechanism simple-mechanism)

(initializing-point m:bar:a:b-p1 (0 0))
(specifying-bar m:bar:a:b .5644024854677596)
(initializing-direction m:bar:a:b-dir (direction 4.999857164003272))
(specifying-bar m:bar:b:c 1.1507815910257295)
```



3.3.2 Geometry Examples

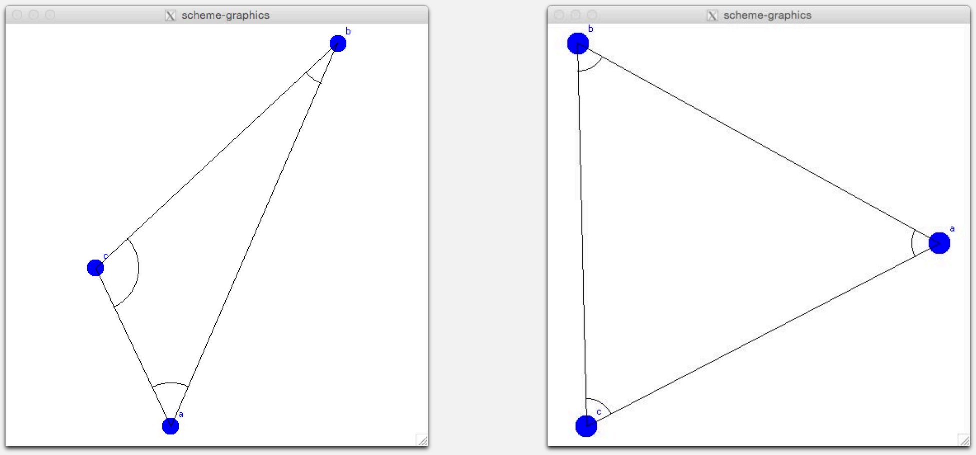
Code Example 3.9: Describing an Arbitrary Triangle

```
(define (arbitrary-triangle)
  (m:mechanism
   (m:make-named-bar 'a 'b)
   (m:make-named-bar 'b 'c)
   (m:make-named-bar 'c 'a)
   (m:make-named-joint 'a 'b 'c)
   (m:make-named-joint 'b 'c 'a)
   (m:make-named-joint 'c 'a 'b)))

(define (simpler-arbitrary-triangle)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)))
```

When solving the mechanism

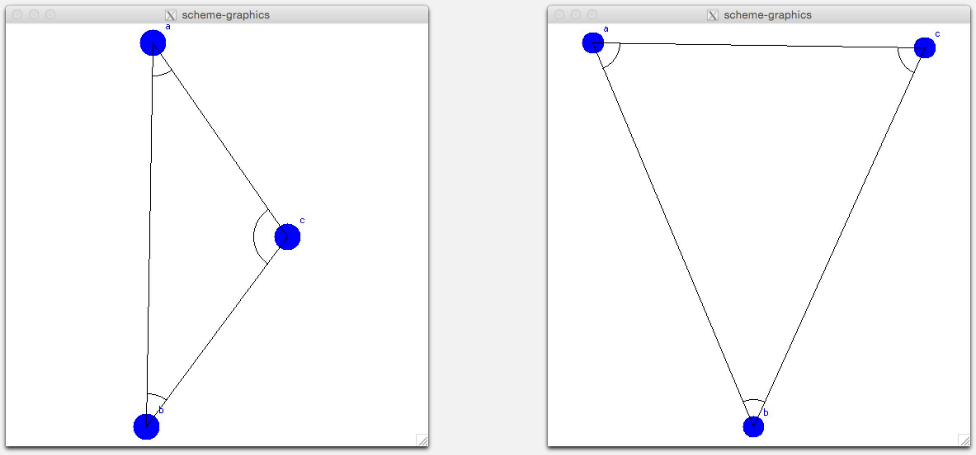
Interaction Example 3.10: Solving the Triangle

```
=> (m:run-mechanism (arbitrary-triangle))

(specifying-joint m:joint:c:b:a .41203408293499)
(initializing-direction m:joint:c:b:a-dir-1 (direction 3.888926311421853))
(specifying-joint m:joint:a:c:b 1.8745808264593105)
(initializing-point m:bar:c:a-p1 (0 0))
(specifying-bar m:bar:c:a .4027149730292784)
```



Interaction Example 3.11: Constraint Solving for Isoceles Triangle

```
(define (isoceles-triangle-by-angles)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)
   (m:c-angle-equal (m:joint 'a)
                    (m:joint 'b))))

=> (m:run-mechanism  isoceles-triangle-by-angles)

(specifying-joint m:joint:c:b:a .6219719886662947)
(initializing-direction m:joint:c:b:a-dir-1 (direction .9330664240883363))
(initializing-point m:bar:b:c-p1 (0 0))
(specifying-bar m:bar:b:c .3557699722973674)
```

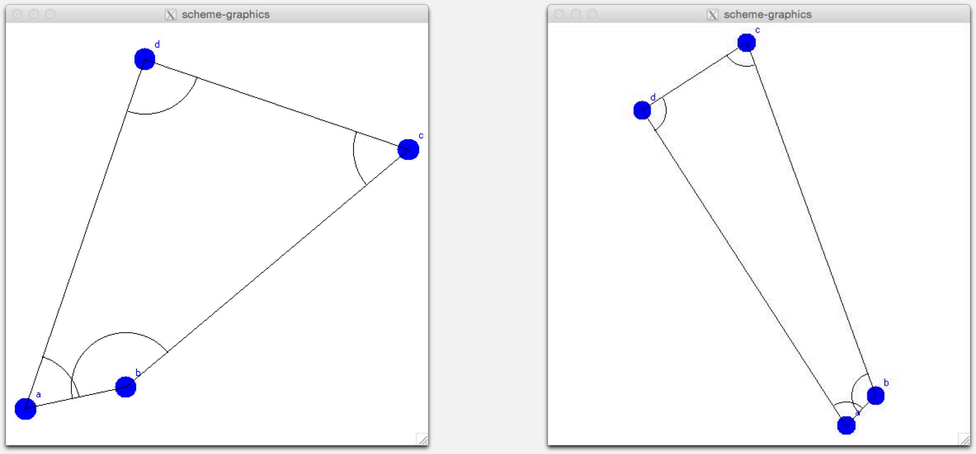


Code Example 3.12: Rectangle Constraints Example

```
(define (is-this-a-rectangle-2)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'd)
                     (m:bar 'b 'c))
   (m:c-right-angle (m:joint 'd))
   (m:c-angle-equal (m:joint 'a)
                    (m:joint 'c))))
```

Interaction Example 3.13: Solved Constraints

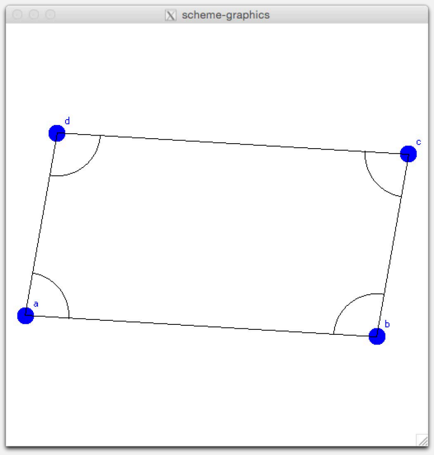
```
=> (m:run-mechanism (is-this-a-rectangle-2))

(specifying-bar m:bar:d:a .6742252545577186)
(initializing-direction m:bar:d:a-dir (direction 4.382829365403101))
(initializing-point m:bar:d:a-p1 (0 0))
(specifying-joint m:joint:c:b:a 2.65583669872538)
```

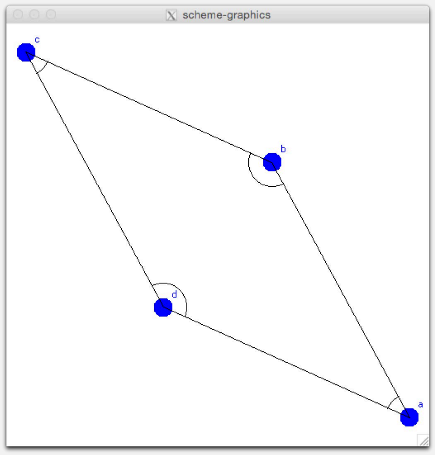


Interaction Example 3.14: Constraint Solving for Parallelogram

```
(define (parallelogram-by-angles)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-angle-equal (m:joint 'a)
                    (m:joint 'c))
   (m:c-angle-equal (m:joint 'b)
                    (m:joint 'd))))

=> (m:run-mechanism parallelogram-by-angles)
```

```
(specifying-joint m:joint:c:b:a 1.6835699856637936)
(initializing-angle m:joint:c:b:a-dir-1 (direction 1.3978162819212452))
(initializing-point m:bar:a:b-p1 (0 0))
(specifying-bar m:bar:a:b .8152792207652096)
(specifying-bar m:bar:b:c .42887899934327023)
```





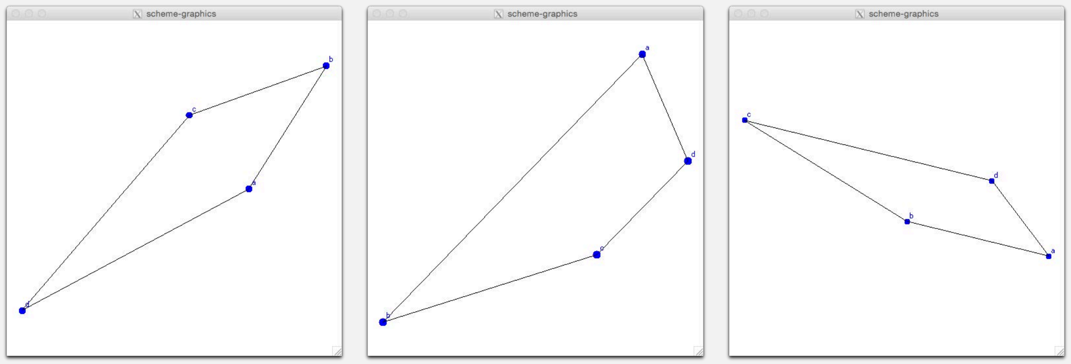
3.4 Learning Module

Interaction Example 3.15:

```
> (what-is 'square)
unknown

=> (what-is line)
primitive-definition

=> (what-is 'triangle)
(triangle (polygon)
          ((n-sides-3 identity)))
```

Interaction Example 3.16: Random Figures

```
=> (show-element (random-trapezoid))
```



Interaction Example 3.17:

```
> (pprint (analyze-element (random-trapezoid)))

((supplementary (angle a) (angle d))
 (supplementary (angle b) (angle c))
 (parallel (segment a b) (segment c d)))
```

Interaction Example 3.18:

```
> (learn-term 'pl random-parallelogram)
done

=> (what-is 'pl)
(pl
 (quadrilateral)
 ((equal-length (polygon-segment 0 1 <premise>)
                (polygon-segment 2 3 <premise>))
  (equal-length (polygon-segment 1 2 <premise>)
                (polygon-segment 3 0 <premise>))
  (equal-angle (polygon-angle 0 <premise>)
               (polygon-angle 2 <premise>))
  (equal-angle (polygon-angle 1 <premise>)
               (polygon-angle 3 <premise>))
  (supplementary (polygon-angle 0 <premise>)
                 (polygon-angle 1 <premise>))
  (supplementary (polygon-angle 0 <premise>)
                 (polygon-angle 3 <premise>))
  (supplementary (polygon-angle 1 <premise>)
                 (polygon-angle 2 <premise>))
  (supplementary (polygon-angle 2 <premise>)
                 (polygon-angle 3 <premise>))
  (parallel (polygon-segment 0 1 <premise>)
            (polygon-segment 2 3 <premise>))
  (parallel (polygon-segment 1 2 <premise>)
              (polygon-segment 3 0 <premise>))))
```

Interaction Example 3.19: Testing Definitions

```
=> (is-a? 'pl (random-parallelogram))
#t

=> (is-a? 'pl (random-rectangle))
#t

=> (is-a? 'pl (random-trapezoid))
(failed-conjecture
 (equal-length (polygon-segment 0 1 <premise>)
               (polygon-segment 2 3 <premise>)))

=> (is-a? 'pl (random-equilateral-triangle))
(failed-conjecture (n-sides-4 <premise>))
(failed-classification quadrilateral)

=> (is-a? 'pl (random-segment))
(failed-classification polygon)
(failed-classification quadrilateral)
```

Interaction Example 3.20: Building on Definitions

```
=> (learn-term 'kite random-kite)
done

=> (learn-term 'rh random-rhombus)
done

=> (what-is 'rh)
(rh
 (pl kite)
 ((equal-length (polygon-segment 0 1 <premise>)
                (polygon-segment 3 0 <premise>))
  (equal-length (polygon-segment 1 2 <premise>)
                (polygon-segment 2 3 <premise>))))

=> (learn-term 'rect random-rectangle)
done

=> (learn-term 'square random-square)
done

=> (what-is 'sq)
(sq (rh rectangle) ())
```