

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.

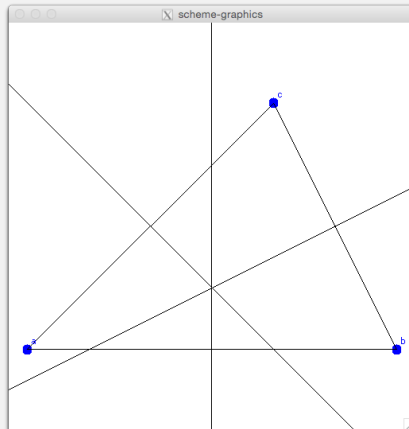
Code Example 3.1: Basic Figure Example

```
1 (define (triangle-with-perp-bisectors)
2   (let-geo* ((a (make-point 0 0))
3             (b (make-point 1.5 0))
4             (c (make-point 1 1))
5             (t (polygon-from-points a b c))
6             (pb1 (perpendicular-bisector (make-segment a b)))
7             (pb2 (perpendicular-bisector (make-segment b c)))
8             (pb3 (perpendicular-bisector (make-segment c a))))
9   (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))
```



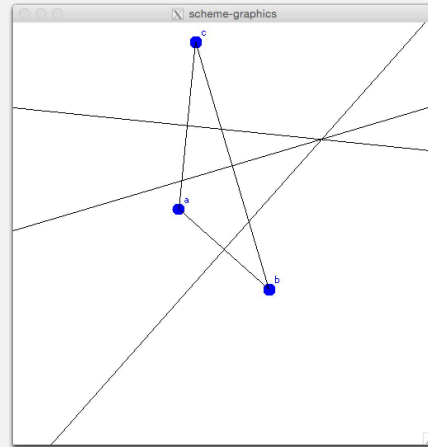
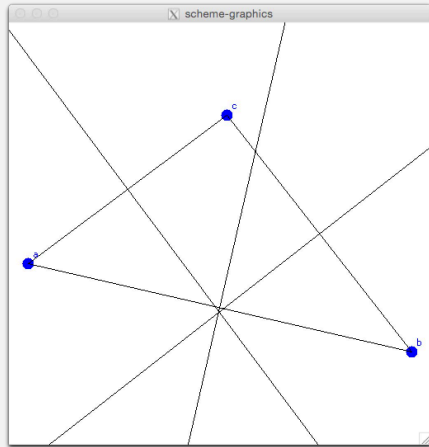
In the first figure, the coordinates of the point were explicitly specified yielding a deterministic instance of the figure. However, as geometry figures often involve arbitrary choices, the construction abstractions support random choices. Figure 3.3 demonstrates the creation of a figure involving an arbitrary triangle. The second formulation (`simple-random-triangle-with-perp-bisectors`) displays a syntax ex-

tension 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)))
```

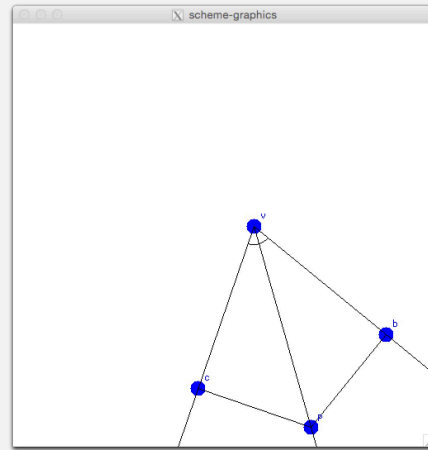
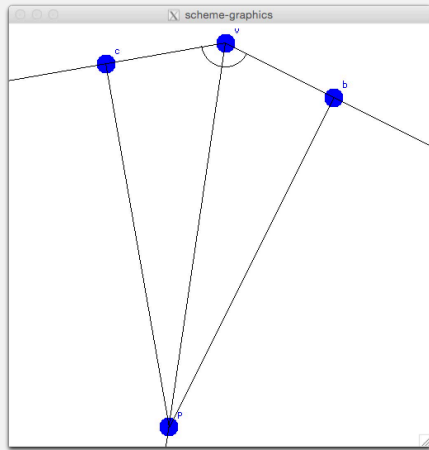


Finally, as examples of more involved constructions, Examples 3.4 and 3.5 demonstrate working with other objects (angles, rays, circles) and construction procedures. Notice that in the angle bisector example the pattern matching syntax extracts the components of an angle (ray, vertex, ray) and segment (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

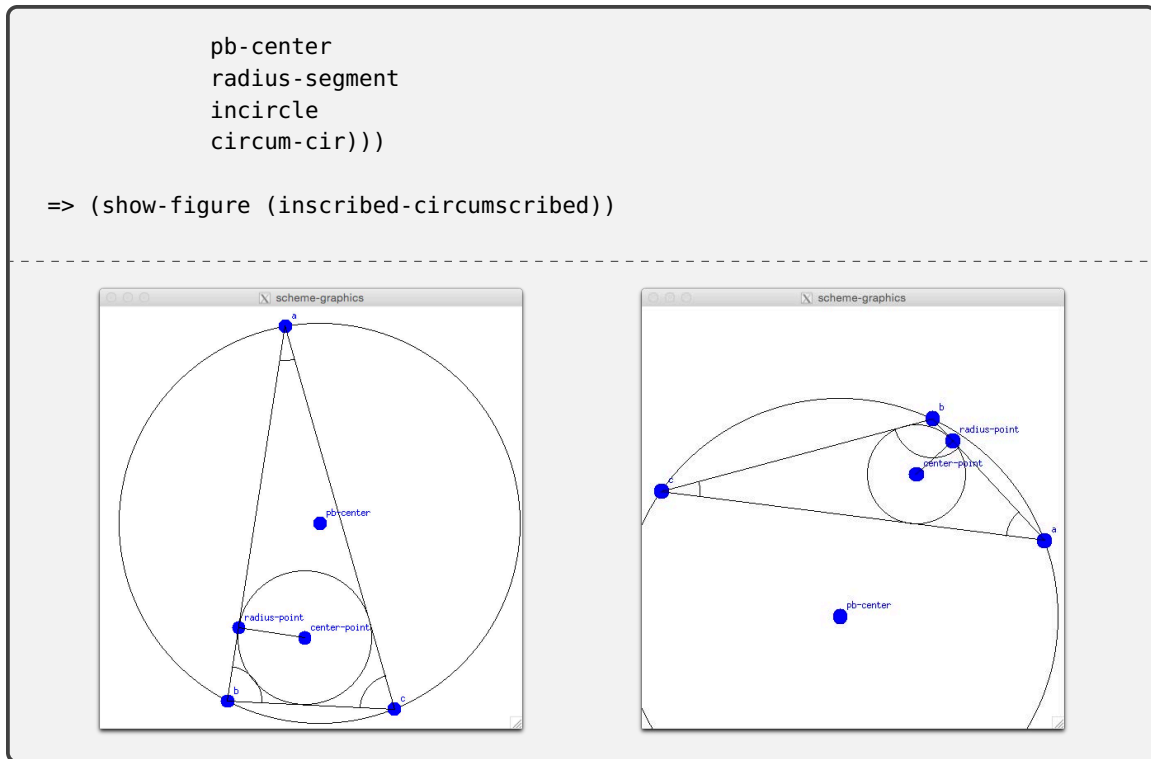
```
(define (angle-bisector-distance)
  (let-geo* (((a (r-1 v r-2)) (random-angle))
            (ab (angle-bisector a))
            (p (random-point-on-ray ab))
            ((s-1 (p b)) (perpendicular-to r-1 p))
            ((s-2 (p c)) (perpendicular-to r-2 p)))
    (figure a r-1 r-2 ab p s-1 s-2)))

=> (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)))
```



The sample images shown alongside these constructions represent images from separate executions of the figure. An additional method for viewing and displaying involves “running an animation” of these constructions in which several instances of the figure are created and displayed, incrementally wiggling each random choice. In generating and wiggling the random values, some effort is taken to avoid degenerate cases or instances where points are too close to one another, as such cases lead to floating-point errors in the numerical analysis.

3.2 Perception and Observation

Given the imperative construction module that enables the specification and construction of geometry figures, the second module focuses on perception and extracting interesting observations from these figures.

Example 3.6 demonstrates the interface for obtaining observations from a figure. An observation is a structure that associates a relationship (concurrent, equal length, parallel) with objects in the figure that satisfy the relationship. Relationships are

represented as predicates over typed n-tuples and are checked against all such n-tuples found in the figure under analysis. For example, the perpendicular relationship is checked against all pairs of linear elements in the figure.

The observation objects are complex structures that maintain properties of the underlying relationships and references to the original objects under consideration. However, my custom printer `print-observations` displays them in a more human-readable format.

Interaction Example 3.6: Simple Analysis

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

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

=> (print-observations (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)))
```

The fact that the perpendicular bisector of a segment is equal to that segment isn't very interesting. Thus, as shown in Example 3.7, the analysis module also provides an interface for reporting only the interesting observations. Currently, information about the interesting relationships formed by a perpendicular bisector are specified alongside instructions for how to perform the operation, but a further extension of the learning module could try to infer inductively which properties result from various construction operations.

Interaction Example 3.7: Interesting Analysis

```
=> (print-observations (interesting-observations
                       (triangle-with-perp-bisectors)))

((concurrent pb1 pb2 pb3))
```

For an example with more relationships, Example 3.8 demonstrates the observations and relationships found in a figure with a random parallelogram. These analysis

results will be used again later when we demonstrate the system learning definitions for polygons. Note that although the segments, angles, and points were not explicitly listed in the figure, they are extracted from the polygon that is listed. Extensions to the observation model can extract additional points and segments not explicitly listed in the original figure.

Interaction Example 3.8: Parallelogram Analysis

```
(define (parallelogram-figure)
  (let-geo* (((p (a b c d)) (random-parallelogram)))
    (figure p)))

=> (pprint (all-observations (parallelogram-figure)))

((equal-length (segment a b) (segment c d))
 (equal-length (segment b c) (segment d a))
 (equal-angle (angle a) (angle c))
 (equal-angle (angle b) (angle d))
 (supplementary (angle a) (angle b))
 (supplementary (angle a) (angle d))
 (supplementary (angle b) (angle c))
 (supplementary (angle c) (angle d))
 (parallel (segment a b) (segment c d))
 (parallel (segment b c) (segment d 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 3.9 demonstrates the specification of a very simple mechanism. Mechanisms are created by specifying the bars and joints involved as well as any additional constraints that must be satisfied. This example mechanism is composed of two bars with one joint between them that is constrained to be a right angle.

Code Example 3.9: Very Simple Mechanism

```
1 (define (simple-mechanism)
2   (m:mechanism
3     (m:make-named-bar 'a 'b)
4     (m:make-named-bar 'b 'c)
5     (m:make-named-joint 'a 'b 'c)
6     (m:c-right-angle (m:joint 'b))))
```

Building a mechanism involves first assembling the bars and joints together so that the named points are identified with one another. Initially, each bar has unknown length and direction, each joint has an unknown angle, and each endpoint has unknown position. Constraints for the bar and joint properties are introduced alongside any explicitly specified constraints.

Solving the mechanism involves repeatedly selecting position, lengths, angles, and directions that are not fully specified and selecting values within the domain of that value's current partial information. As values are specified, the wiring of the propagator model propagates further partial information to other values.

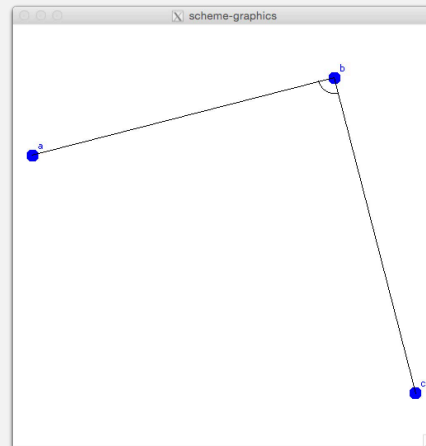
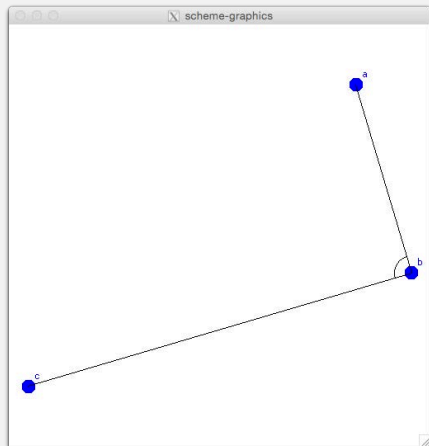
The printed statements in Example 3.10 demonstrate that solving the simple mechanism above involves specifying the location of point a, then specifying the length

of bar a-b and the direction from a that the bar extends. After those specifications, the joint angle is constrained to be a right angle and the location of point b is known by propagating information about point a and bar a-b's position and length. Thus, the only remaining property to fully specify the figure is the bar length of bar b-c. After building and solving the mechanism, run-mechanism converts it into a figure using the underlying primitives and displays it:

Interaction Example 3.10: Solving the Very Simple Mechanism

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

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



3.3.2 Geometry Examples

Code Example 3.11: Describing an Arbitrary Triangle

```
1 (define (arbitrary-triangle)
2   (m:mechanism
3     (m:make-named-bar 'a 'b)
4     (m:make-named-bar 'b 'c)
5     (m:make-named-bar 'c 'a)
6     (m:make-named-joint 'a 'b 'c)
7     (m:make-named-joint 'b 'c 'a))
```

```

8   (m:make-named-joint 'c 'a 'b)))
9
10  (define (simpler-arbitrary-triangle)
11    (m:mechanism
12      (m:establish-polygon-topology 'a 'b 'c)))

```

When solving the mechanism

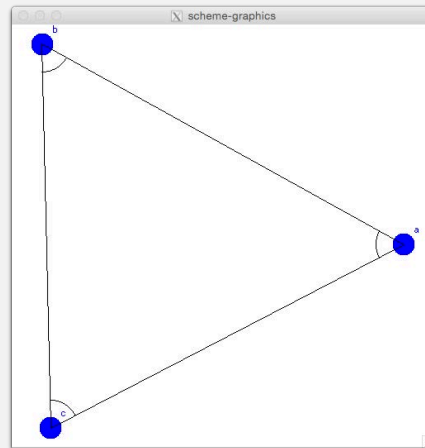
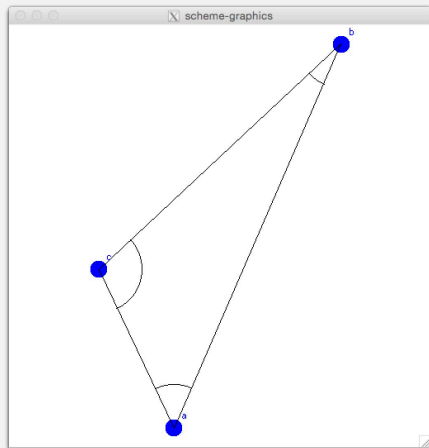
Interaction Example 3.12: Solving the Triangle

```
=> (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-length m:bar:c:a .4027149730292784)

```



Interaction Example 3.13: Constraint Solving for Isoceles Triangle

```

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

```

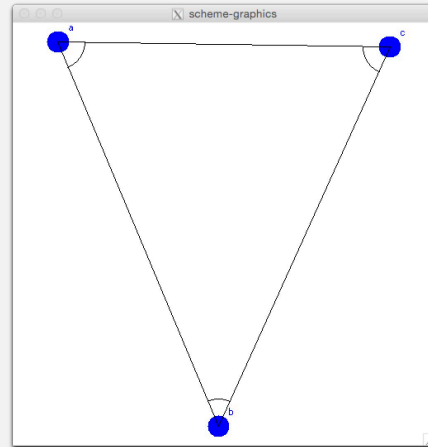
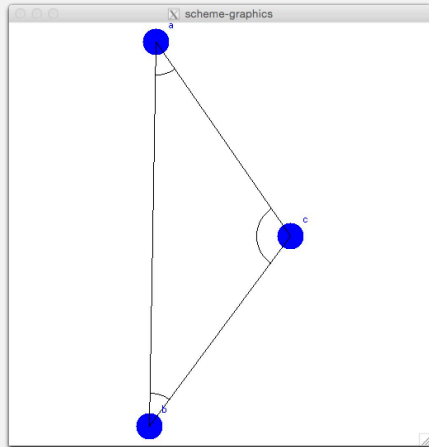
```
=> (m:run-mechanism isoeles-triangle-by-angles)
```

```

(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-length m:bar:b:c .3557699722973674)



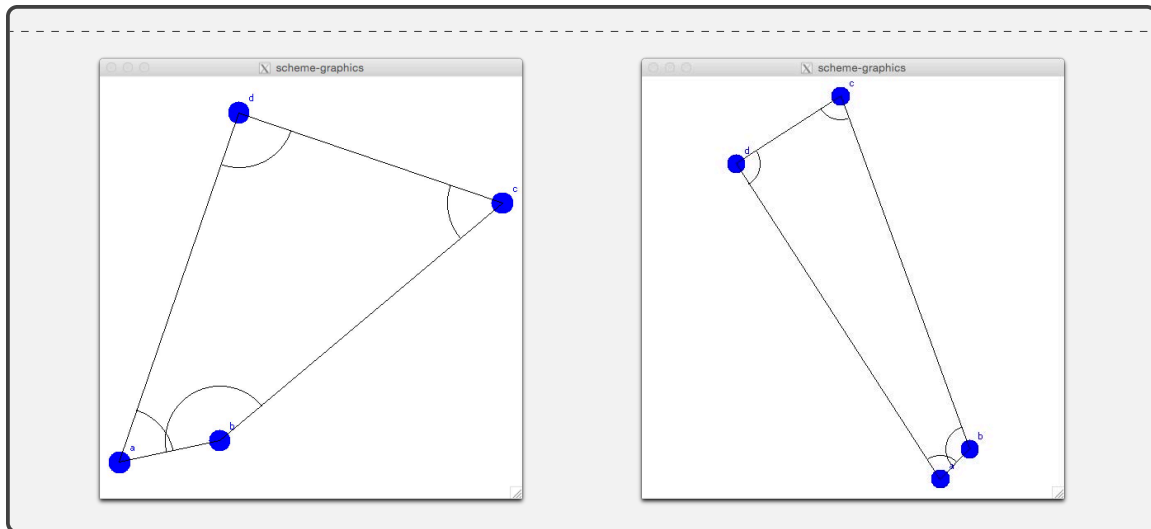
Code Example 3.14: Rectangle Constraints Example

```
1 (define (is-this-a-rectangle-2)
2   (m:mechanism
3     (m:establish-polygon-topology 'a 'b 'c 'd)
4     (m:c-length-equal (m:bar 'a 'd)
5                       (m:bar 'b 'c))
6     (m:c-right-angle (m:joint 'd))
7     (m:c-angle-equal (m:joint 'a)
8                      (m:joint 'c))))
```

Interaction Example 3.15: Solved Constraints

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

(specifying-bar-length 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)
```

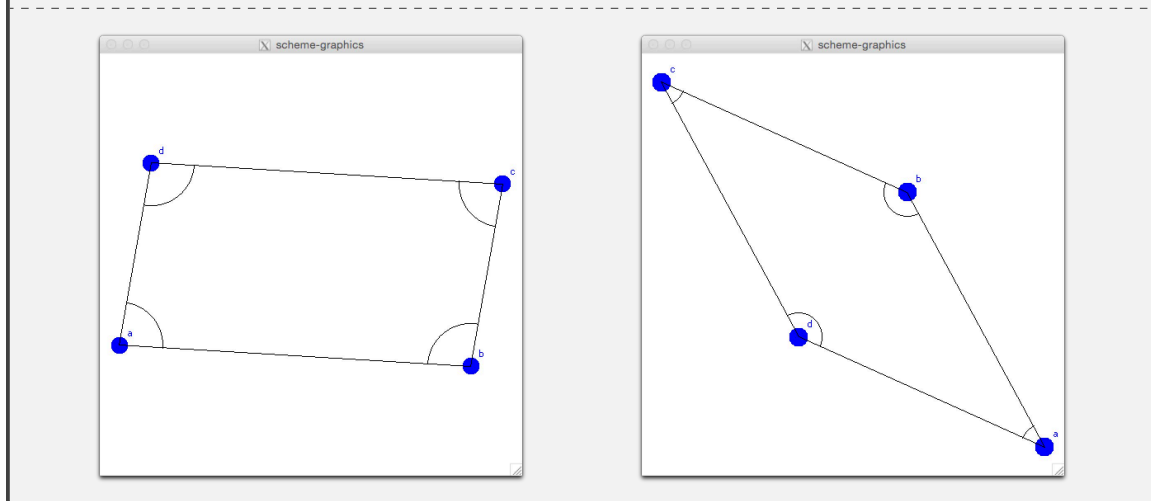


Interaction Example 3.16: Constraint Solving for Parallelogram

```
(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-pl (0 0))
(specifying-bar-length m:bar:a:b .8152792207652096)
(specifying-bar-length m:bar:b:c .42887899934327023)
```



3.4 Learning Module

Finally, given these modules for performing constructions, observing interesting symbolic relationships, and rebuilding figures that satisfy such relationship, a learning module interfaces with these properties to emulate a student that is actively learning geometry.

A user representing the teacher can interact with the system by querying what it knows, teaching it new terms, and asking it to apply its knowledge to new situations.

Interaction Example 3.17: Querying Terms

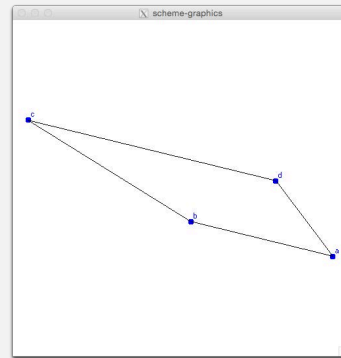
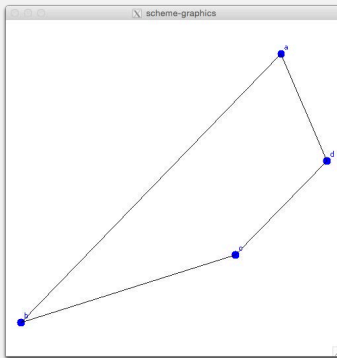
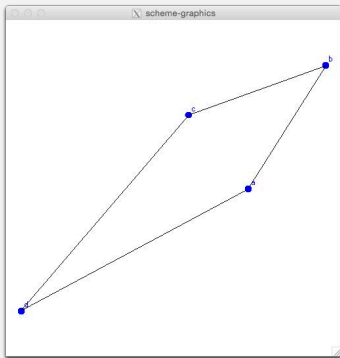
```
=> (what-is 'trapezoid)
unknown

=> (what-is 'line)
primitive-definition

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

Interaction Example 3.18: Random Figures

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



Interaction Example 3.19:

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

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

Interaction Example 3.20:

```
> (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 3.21: 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 3.22: 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) ())

```