# 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 implementation, I will present and discuss several sample interactions with the system. Further details details can be found in subsequent chapters.

The overall goal of the system is to emulate a student learning geometry via an investigative approach. To accomplish this, 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 will explore interactions with these modules in increasing complexity, building up to a full demonstration of the system achieving its learning goals in Section 3.4.

## 3.1 Imperative Figure Construction

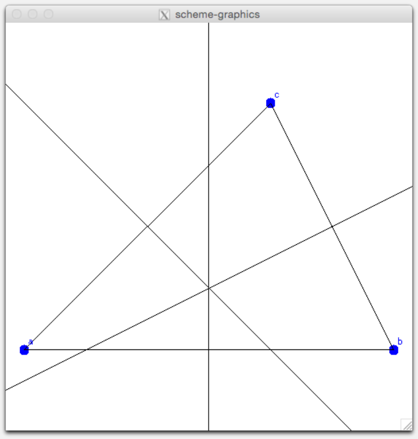
At its foundation, the system provides a language and engine for performing geometry constructions and building figures. Example 3.1 presents a simple specification of a figure. Primitives of points, lines, segments, rays, and circles can be combined into polygons and figures, and complicated constructions such as the perpendicular bisector of a segment can be abstracted into higher-level 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 the 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)))
```

## Interaction Example 3.2: Rendering the Basic Figure

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



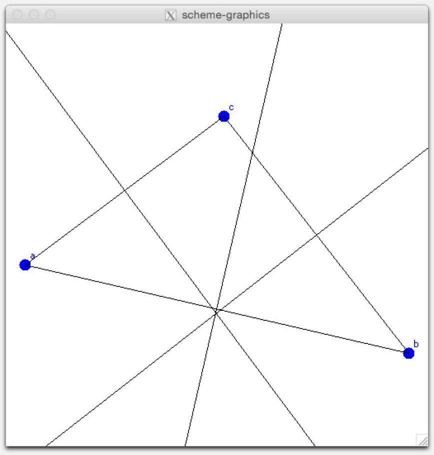
Given such an imperative description, the system can construct and display an instance of the figure as shown in Example 3.2. The graphics system uses the underlying X window system-based graphics interfaces in MIT Scheme, labels named points (`a`, `b`, `c`), and repositions the coordinate system to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

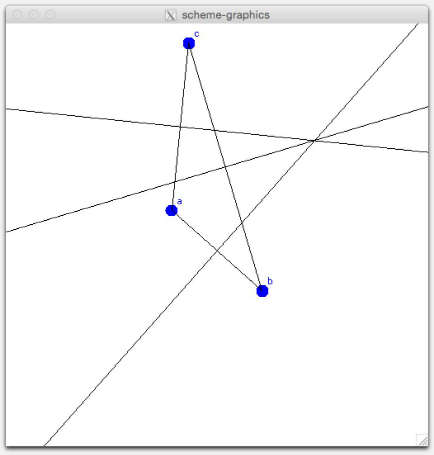
In the first figure, the coordinates of the points were explicitly specified yielding a deterministic instance of the figure. However, to represent entire spaces of diagram instances, the construction abstractions support random choices. Example 3.3 demonstrates the creation of a figure involving an arbitrary triangle.

The second formulation, `(simple-random-triangle-perp-bisectors)`, is equivalent to the first. It displays a syntax extension provided by `let-geo*` that shortens the common pattern of accessing and naming the components of an object. In this case, `((t (a b c)) (random-triangle))` will assign to the variable `t` the resulting random triangle, and to the variables `a`, `b`, and `c` the resulting triangle's vertices.

**Interaction Example 3.3: Introducing Randomness**

```
(define (random-triangle-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-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)))
```

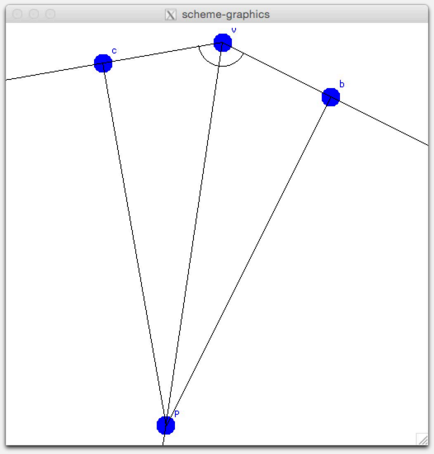


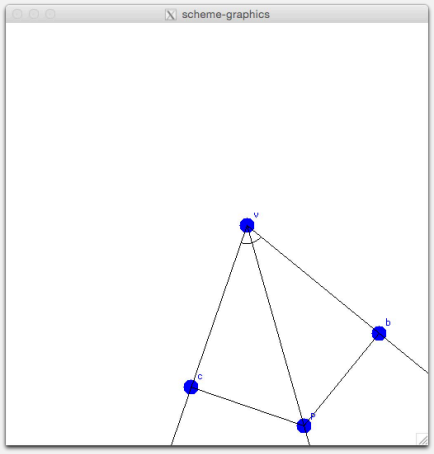


As examples of more involved constructions, Examples 3.4 and 3.5 demonstrate working with other objects (angles, rays, circles) and construction procedures. Notice that, in the angle bisector example, the pattern matching syntax extracts the components of an angle (ray, vertex, ray) and segment (endpoints),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

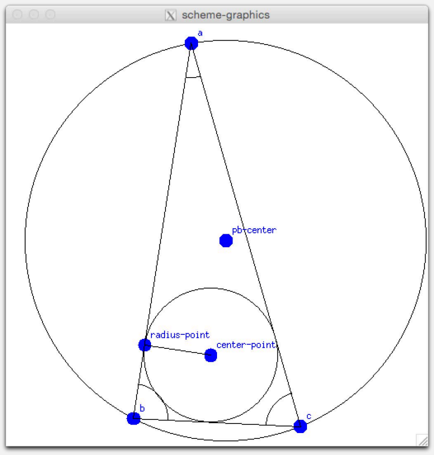


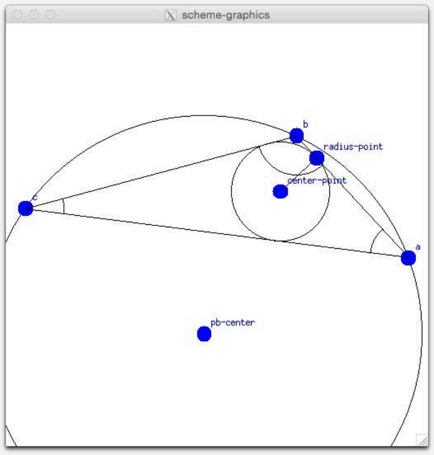


## Interaction Example 3.5: Inscribed and Circumscribed Circles

```
(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)))
             (ins-circle (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-circle (circle-from-points pb-center a)))
    (figure t a-1 a-2 a-3 pb-center radius-segment
            ins-circle circum-circle)))
```

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





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 can lead to undesirable 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 returned are compound structures that maintain properties of the underlying relationships and references to the original objects under consideration. Dependency information about how these original objects were construction will be later used to generalize these observations into conjectures. For now, my custom printer `print-observations` can use the name information of the objects to display the observations 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 perpendicular to that segment is not 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 construction operation such as perpendicular bisector is 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 known 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 I demonstrate the system learning definitions for polygons and simplifying such definitions to minimal sufficient constraint sets. 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)))
```

`all-observations` will report all reasonable observations found, but as will be shown in Section 3.4, as the system learns new terms and concepts, a request for `interesting-observations` will use such learn concepts to eliminate redundant observations and filter out previously-discovered facts. In this case, once a definition for parallelogram is learned, `interesting-observations` would simply report that `p` is a parallelogram and omit observations implied by that fact.

## 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 mind’s 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 [12] 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. Unlike the sequential, Scheme variable based `let-geo*` specification of constructions in the imperative construction system, to specify mechanisms, `m:mechanism` is applied to a list of linkage and constraint declarations containing symbolic identifiers. This example mechanism is composed of two bar linkages with one joint linkage between the bars, along with a constraint that the joint is a right angle.

**Code Example 3.9: 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))))
```

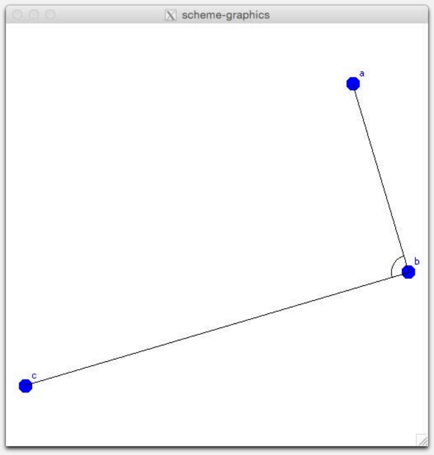
Assembling a mechanism involves first adjoining the bars and joints together so that the named points are identified with one another. Initially, each bar has unknown length and direction, each joint has an unknown angle, and each endpoint has unknown position. Constraints for the bar and joint properties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

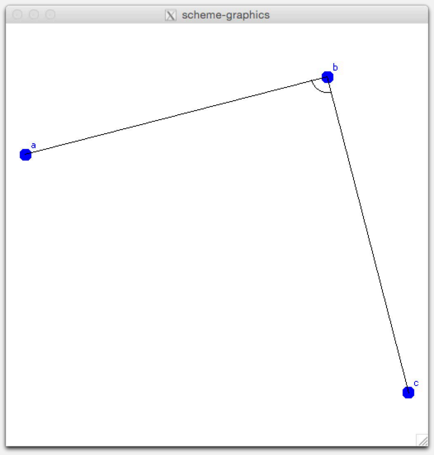
Solving a mechanism involves repeatedly selecting position, lengths, angles, and directions that are not fully specified and selecting values within the domain of that element's current partial information. As values are specified, the wiring of the propagator model propagates further partial information 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, point `c` is known to be on a ray extending outwards from `b` and the only remaining property needed to fully specify the figure is the length of bar `b-c`. The command `m:run-mechanism` builds and solves the mechanism, then converts the result into an analytic figure 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

These bar and linkage mechanisms can be used to represent the topologies of several geometry figures. Bars correspond to segments and joints correspond to angles. Example 3.11 demonstrates the set of linkages necessary to specify the topology of a triangle. The second formulation, `(simpler-arbitrary-triangle)` is equivalent to the first since the utility procedure `m:establish-polygon-topology` expands to create the set of $n$ bars and $n$ joint specifications needed to represent a closed polygon for the given $n$ vertex names.

**Code Example 3.11: Describing an Arbitrary Triangle**

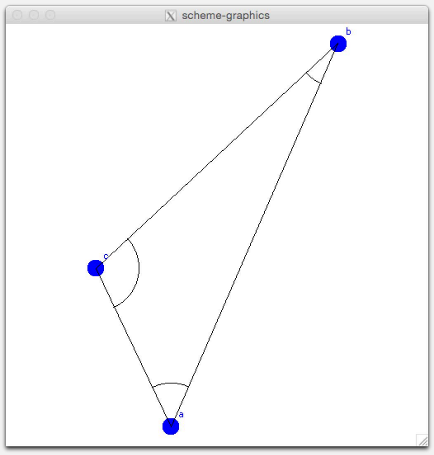
```scheme
(define (arbitrary-triangle)
  (m:mechanism
   (m:make-named-bar 'a 'b)
   (m:make-named-bar 'b 'c)
   (m:make-named-bar 'c 'a)
   (m:make-named-joint 'a 'b 'c)
   (m:make-named-joint 'b 'c 'a)
   (m:make-named-joint 'c 'a 'b)))

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

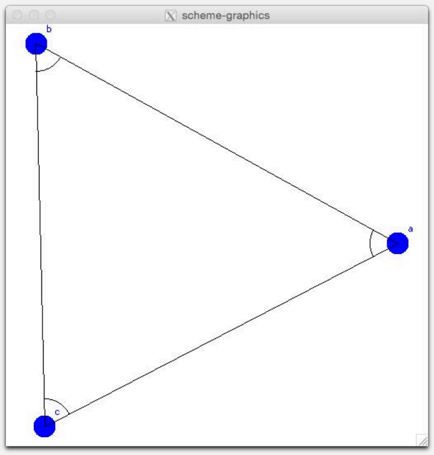
As seen in Example 3.12 (next page), once joints `b` and `c` have had their angles specified, propagation fully determines the angle of joint `a`. The only parameter remaining is the length of one of the bars. The two `initializing-` steps don't affect the resulting shape but determine its position and orientation on the canvas.

In this case, joint angles are specified first. The ordering of what is specified is guided by a heuristic that helps all of the examples shown in this chapter converge to solutions. The heuristic generally prefers specifying the most constrained values first. However, in some scenarios, specifying values in the wrong order can yield premature contradictions. A planned extension will attempt to recover from such situations more gracefully by trying other orderings for specifying components.

**Interaction Example 3.12: Solving the Triangle**

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

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



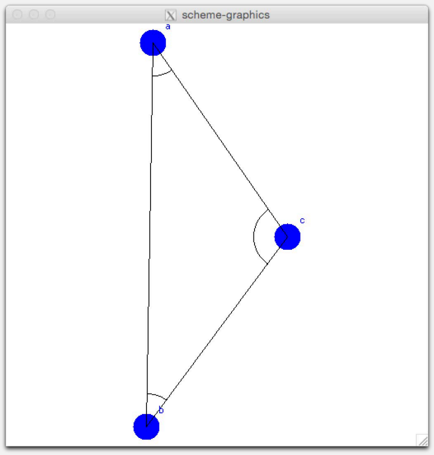


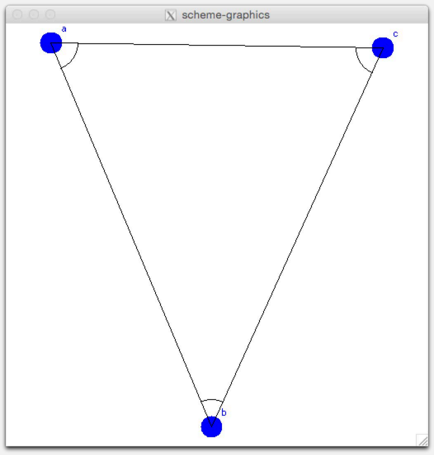
To include some user-specified constraints, Example 3.13 shows the steps involved in solving an isoceles triangle from the fact that its base angles are congruent. Notice that the only two values that must be specified are one joint angle and one bar length. The rest is handled by propagation.

Propagation involves representing the partial information of where points and angles can be. A specified angle constrains a point to a ray and a specified length constrains a point to be on an arc of a circle. As information about a point is merged from several sources, intersecting these rays and circles yields unique solutions for where the points must exist. Then, as the location of points are determined, the bidirectional propagation continues to update the corresponding bar lengths and joint angles Although not as dynamic, these representations correspond to physically wiggling and extending the bars until they reach one another.

**Interaction Example 3.13: 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-angle 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)
```



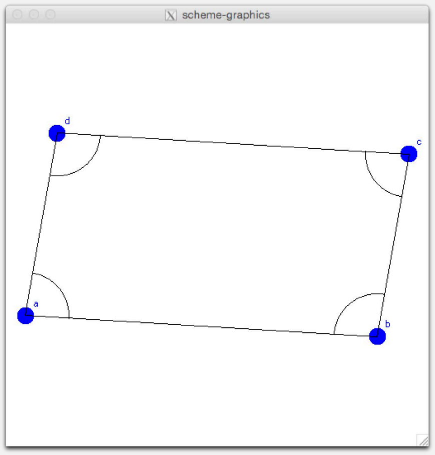


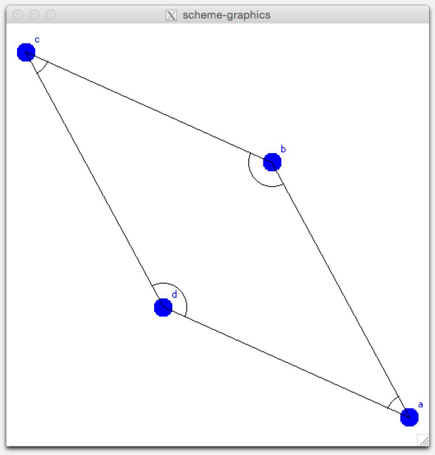
Example 3.14 continues the analysis of properties of the parallelogram. In this case, the constraint solver is able to build figures given the fact that its opposite angles are equal. The fact that these all happen to be parallelograms will be used by the learning module to produce a simpler definition for a parallelogram.

**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-angle 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-length m:bar:a:b .8152792207652096)
(specifying-bar-length m:bar:b:c .42887899934327023)
```





To demonstrate the constraint solving working on a more complicated example, Example 3.15 represents the constraints from the middle “Is this a rectangle?” question from Chapter 2. This question asks whether a quadrilateral in which a pair of opposite sides are congruent, a pair of opposite angles are congruent, and one of the other angles is a right angle is always a rectangle. Try working this constraint problem by hand or in your mind’s eye.

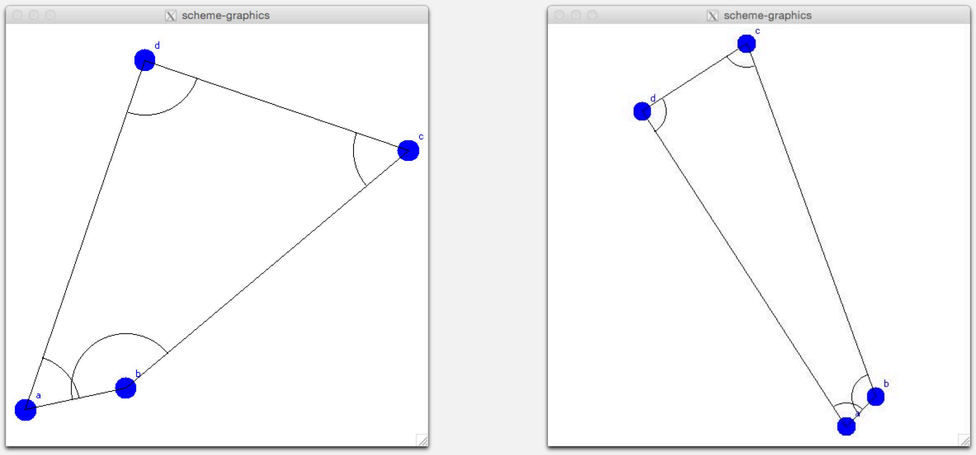
**Code Example 3.15: Rectangle Constraints Example**

```
(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))))
```

As seen in Example 3.16, solutions are not all rectangles. Chapter 7 includes a more detailed walkthrough of how this example is solved. Interestingly, once the initial scale is determined by the first bar length, the remaining shape only has one degree of freedom.

## Interaction Example 3.16: 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-angle m:joint:c:b:a 2.65583669872538)
```



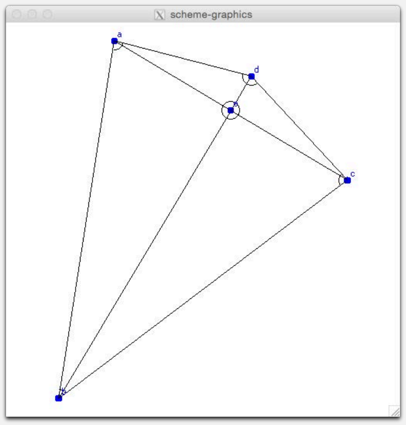
As a final mechanism example, in addition to solving constraints of the angles and sides for a *single* polygon, the mechanism system can support the creation of arbitrary topologies of bars and joints. In the following examples, by using several calls to the `establish-polygon-topology utility`, I build the topology of a quadrilaterals whose diagonals intersect at a point `e` and explore the effects of various constraints on these diagonal segments. `m:quadrilateral-with-intersecting-diagonals` will simplify specification of this topology in the following examples.

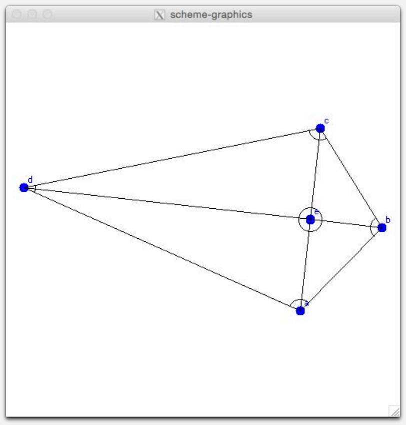
## Code Example 3.17: More Involved Topologies for Constraint Solving

```
(define (m:quadrilateral-with-intersecting-diagonals a b c d e)
  (list (m:establish-polygon-topology a b e)
        (m:establish-polygon-topology b c e)
        (m:establish-polygon-topology c d e)
        (m:establish-polygon-topology d a e)
        (m:c-line-order c e a)
        (m:c-line-order b e d)))
```

## Interaction Example 3.18: Kites from Diagonal Properties

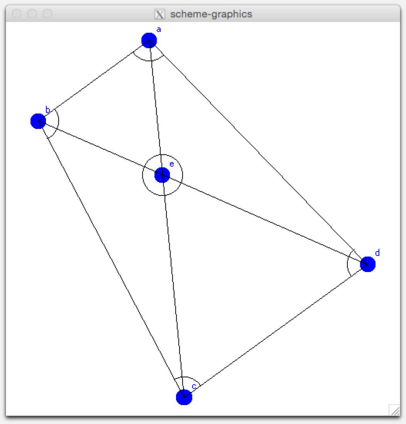
```
(define (kite-from-diagonals)
  (m:mechanism
   (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
   (m:c-right-angle (m:joint 'b 'e 'c)) ;; Right Angle in Center
   (m:c-length-equal (m:bar 'c 'e) (m:bar 'a 'e))))

=> (m:run-mechanism kite-from-diagonals)
```

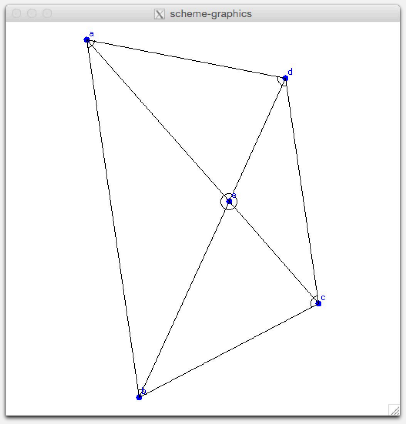




## Interaction Example 3.19: Isoceles Trapezoids from Diagonals

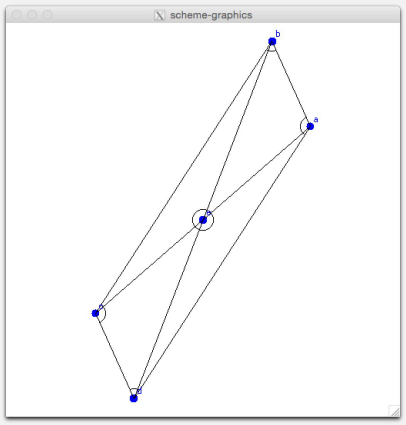
```
(define (isoceles-trapezoid-from-diagonals)
  (m:mechanism
   (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
   (m:c-length-equal (m:bar 'a 'e) (m:bar 'b 'e))
   (m:c-length-equal (m:bar 'c 'e) (m:bar 'd 'e))))

=> (m:run-mechanism isoceles-trapezoid-from-diagonals)
```

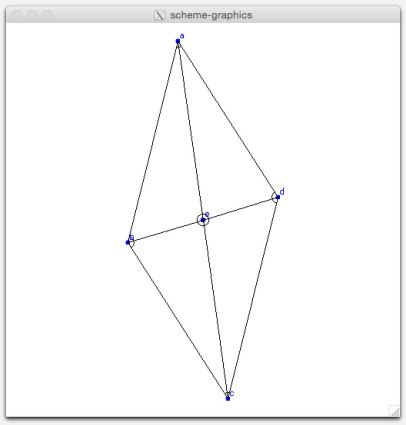




**Interaction Example 3.20: Parallelograms from Diagonal Properties**

```
(define (parallelogram-from-diagonals)
  (m:mechanism
   (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
   (m:c-length-equal (m:bar 'a 'e) (m:bar 'c 'e))
   (m:c-length-equal (m:bar 'b 'e) (m:bar 'd 'e))))
```





As seen in Examples 3.18 through 3.20, simple specifications on the diagonals of a quadrilateral can fully constrain such quadrilaterals to particular classes. Such results are interesting to be able to explore via this module alone, but also becomes a powerful tool as the learning module combines imperative and declarative information.

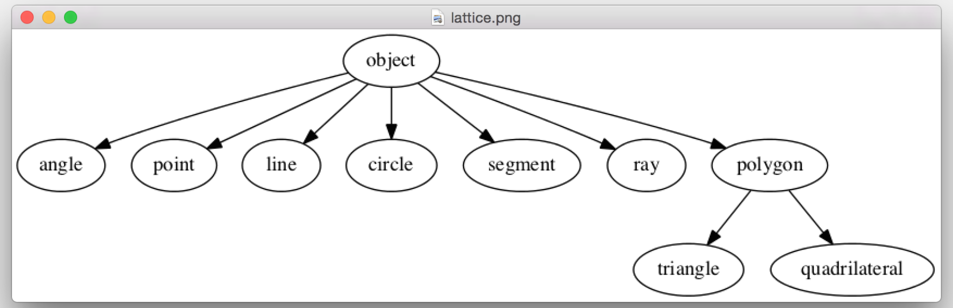
## 3.4 Learning Module

The previous sections described modules for performing constructions, observing interesting symbolic relationships, and rebuilding figures that satisfy such relationships. As the final module, the learning module interfaces with these modules to achieve the end goal of emulating a student learning geometry via an investigative approach.

Although we have seen examples of various higher-level terms and objects, the learning module begins with very limited knowledge about geometry. The lattice in Example 3.21 represents the built-in objects the system understands. Although it has some knowledge of points, segments, lines, rays, angles, circles and polygons, upon startup, it knows nothing about higher-level terms such as trapezoids, parallelograms, or isoceles triangles.

**Interaction Example 3.21: Initial Lattice**

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

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

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

=> (show-definition-lattice)
```

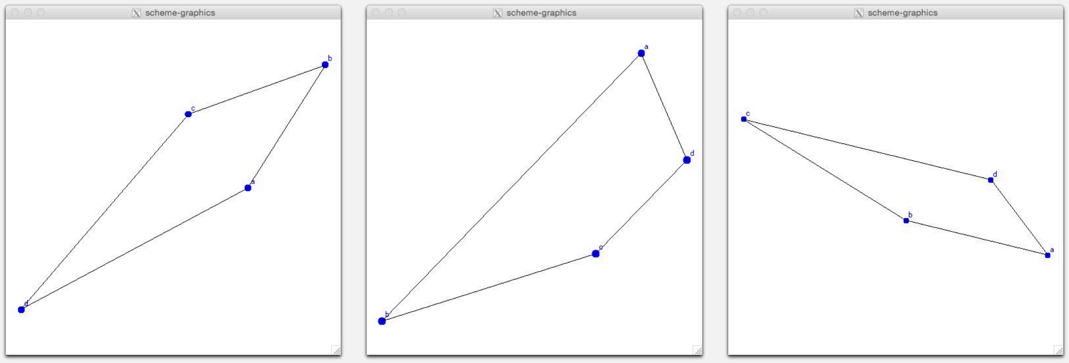


A user representing a "teacher" can interact with the system by creating investigations using these primitives. These investigations are typically steps to construct a diagram instance, but can include other specifications. The system will construct and examine the figure resulting from such investigations, and sometimes perform related investigations of its own. Interesting relationships invariant across the instances are generalized into new concepts and theorems. To evaluate the system's learning, the system provides means for a user to query its knowledge or apply it to new situations.

One example of this process involves the "teacher" user crafting an investigation procedure that creates instances of a new class of object. For instance, a user could define `random-trapezoid` to be a procedure that, each time it is called, returns randomly constructed trapezoid. Example 3.22 shows the full range of trapezoids created via the `random-trapezoid` procedure.

**Interaction Example 3.22: Random Figures**

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



The learning module can interface with the perception module to obtain observations about given element. In Example 3.23, the results show the full dependencies of elements under consideration instead of their names. These dependency structures are later used to convert the observations about this specific trapezoid into general conjectures that can be tested against other polygons.

**Interaction Example 3.23: Analyzing an Element**

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

((supplementary (polygon-angle 0 <premise>) (polygon-angle 3 <premise>))
 (supplementary (polygon-angle 1 <premise>) (polygon-angle 2 <premise>))
 (parallel (polygon-segment 0 1 <premise>) (polygon-segment 2 3 <premise>)))
```

With these analysis abilities, a user can teach the system new object classes by providing a term (`'trapezoid`) and a generator procedure that produces instances of that element as seen in Example 3.24.

**Interaction Example 3.24: Learning New Terms**

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

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

Although the internal implementations of user-provided generator procedures are opaque to the learning module, it is able to examine interesting relationships invariant across instances of such objects and discover properties for the new definition.

As shown in example 3.25, after being instructed to learn what a parallelogram is from the `random-parallelogram` procedure, when queried for a definition, we're given the term, the base classifications of the element, and all properties known to be true of such objects.

**Interaction Example 3.25: Asking about Terms**

```
=> (what-is 'parallelogram)
(parallelogram
 (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>))))
```

To use such learned knowledge, we can use `is-a?` to test whether other elements are satisfy the known definition of a term. As shown in example 3.26, results are correctly returned for any polygon that satisfies the observed properties. In cases where the properties are not satisfied, the system reports the failed conjectures or classifications (e.g. an equaliteral triangle is not a parallelogram: It failed the necessary classification that it must be a quadrilateral because it didn't have 4 sides).

**Interaction Example 3.26: Testing Definitions**

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

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

=> (is-a? 'parallelogram (polygon-from-points
                (make-point 0 0)
                (make-point 1 0)
                (make-point 2 1)
                (make-point 1 1)))
#t

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

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

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

Learning individual definitions is nice, but cool properties arise when definitions build upon one another. When a new term is learned, the system checks other related terms for overlapping properties to determine where the new definition fits in the current lattice of terms. In Example 3.27, we see that, after learning definitions of kites and rhombuses, the reported definition of a rhombus is that it a parallelogram and kite that satisfies two additional rhombus-specific properties about equal length sides. Later, after learning about rectangles, amazingly, the system shows us that the definition of a square has no additional properties beyond that of being both a rhombus and a rectangle. The system is able to make these same deductions and update definitions irrespective of the order in which it is taught the terms.

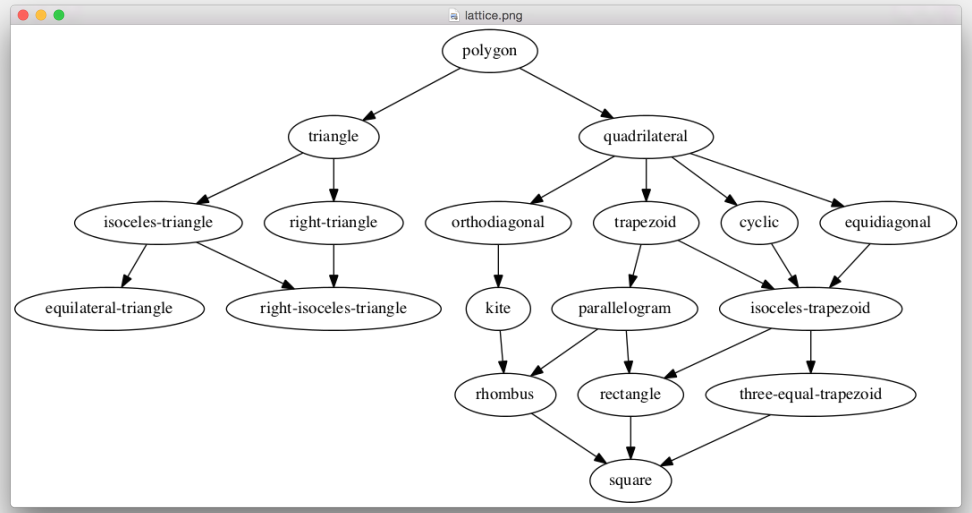
**Interaction Example 3.27: Building on Definitions**

```
=> (learn-term 'rhombus random-rhombus)
=> (learn-term 'kite random-kite)
=> (what-is 'rhombus)
(rhombus
 (parallelogram 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 'rectangle random-rectangle)
=> (learn-term 'square random-square)
=> (what-is 'square)
(square (rhombus rectangle) ())
```

As it learns definitions, the system constructs and maintains a lattice of known concepts in which child nodes are more specific classes of their parents. An example of the definition lattice the system generated after learning several more terms is shown in Example 3.28. We see that the accurate relations are expressed:

**Interaction Example 3.28: Expanded Definition Lattice**

```
=> (show-definition-sublattice 'polygon)
```



Although most terms can be distinguished from one another using the basic angle and side properties, in some cases the initial analysis of the polygon is insufficient. As seen in Example 3.29, when initially learning the orthodiagonal term, the system was not able to observe any differentiating properties between arbitrary quadrilaterals and orthodiagonal quadrilaterals. Orthodiagonal quadrilaterals are quadrilaterals with the property that their diagonals are perpendicular to one another.

**Interaction Example 3.29: Learning Orthodiagonal Quadrilaterals**

```
=> (learn-term 'orthodiagonal random-orthodiagonal-quadrilateral)
"Warning: No new known properties for term: orthodiagonal. Appears same as
    quadrilateral."
done
```

To handle such situations, and to allow the learning module to capture more general theorems about its objects,

— — — — Fun Example — — — —

Finally, the fun example that integrates all of these systems is learning simpler definitions for these terms. In these examples, `get-simple-definitions` takes a known term, looks up the known observations and properties for that term, and tests using all reasonable subsets of those properties as constraints via the constraint solver. For each subset of properties, if the constraint solver was able to create a diagram satisfying exactly those properties, the resulting diagram is examined as with “is-a” above to see if all the known properties of the original term hold.

If so, the subset of properties is reported as a valid definition of the term, and if the resulting diagram fails some properties, the subset is reported as an invalid (insufficient) set of constraints.

In the example 3.30, we see a trace of finding simple definitions for isoceles triangles and parallelograms. In the first example, the observed properties of an isoceles triangle are that its segments and angles are equal. Via the definitions simplification via constraint solving, we actually discover that the constraints of base angles equal or sides equal are sufficient.

**Interaction Example 3.30: Learning Simple Definitions**

```
=> (what-is 'isoceles-triangle)

(i-t
 (triangle)
 ((equal-length (polygon-segment 0 1 <premise>)
                (polygon-segment 2 0 <premise>))
  (equal-angle (polygon-angle 1 <premise>) (polygon-angle 2 <premise>))))

=> (get-simple-definitions 'isoceles-triangle)

((sufficient
  (((equal-angle (*angle* b) (*angle* c)))
   ((equal-length (*segment* a b) (*segment* c a)))))
 (insufficient (()))
 (unknown ()))
```

In the parallelogram example 3.31, some subsets are omitted because the constraint solver wasn't able to solve a diagram given those constraints (to be improved / retried more gracefully in the future). However, the results still show some interesting valid definitions such as the pair of equal opposite angles as explored in Example 3.14 or equal length opposite sides and correctly mark several sets of invalid definitions as not being specific enough.

**Interaction Example 3.31: Learning Simple Parallelogram Definitions**

```
=> (get-simple-definitions 'parallelogram)

((sufficient
  (((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)))))
 (insufficient
  (((equal-length (*segment* a b) (*segment* c d))
    (equal-angle (*angle* b) (*angle* d)))))
 (unknown
  (((equal-angle (*angle* a) (*angle* c)))
   ((equal-length (*segment* b c) (*segment* d a))))))
```

This simple definitions implementation is still a work in progress and has room for improvement. For instance, checking all possible subsets is wasteful as any superset of a valid definition is known to be valid and any subset of an invalid definition is known to be invalid. In addition to such checks, in the future I plan to use the knowledge about what properties the insufficient diagram is violating to use as a possible addition to the constraint set. Further extensions could involve generalizing this get-simple-definitions to support other topologies for the initial properties (such as the quadrilaterals being fully specified by their diagonal properties as in Example 3.17)