Automated Elementary Geometry Theorem Discovery via Inductive Diagram Manipulation

by

Lars Erik Johnson

S.B., Massachusetts Institute of Technology (2015)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
June 23, 2015

Certified by. . . . .
Gerald Jay Sussman
Panasonic Professor of Electrical Engineering, Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Automated Elementary Geometry Theorem Discovery via Inductive Diagram Manipulation

by

Lars Erik Johnson

Submitted to the
Department of Electrical Engineering and Computer Science
on June 23, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I created and analyzed an interactive computer system capable of exploring geometry concepts through inductive investigation. My system begins with a limited set of knowledge about basic geometry and enables a user interacting with the system to "teach" the system additional geometry concepts and theorems by suggesting investigations the system should explore to see if it "notices anything interesting." The system uses random sampling and physical simulations to emulate the more human-like processes of manipulating diagrams "in the mind's eye." It then uses symbolic pattern matching and a propagator-based truth maintenance system to appropriately generalize findings and propose newly discovered theorems. These theorems can be rigorously proved using external proof assistants, but also be used by the system to assist in its explorations of new, higher-level concepts. Through a series of simple investigations similar to an introductory course in geometry, the system has been able to propose and learn a few dozen standard geometry theorems. [and through more self-directed explorations, it has discovered several interesting properties and theorems not typically covered in standard mathematics courses.]

Thesis Supervisor: Gerald Jay Sussman
Title: Panasonic Professor of Electrical Engineering

Acknowledgments

I am thankful for my family, friends, advisors, and teachers for their continued support of my pursuits.

Contents

Chapter 1

Introduction

In this thesis, I develop and analyze an interactive computer system that emulates a student learning geometry concepts through inductive investigation. Although geometry knowledge can be conveyed via a series of factual definitions, theorems, and proofs, my system focuses on a more investigative approach in which an external teacher guides the student to "discover" new definitions and theorems via explorations and self-directed inquiry.

My system emulates such a student by beginning with a fairly limited knowledge set regarding basic definitions in geometry and providing a means by which a user interacting with the system can "teach" additional geometric concepts and theorems by suggesting investigations the system should explore to see if it "notices anything interesting."

To enable such learning, my project includes the combination of four intertwined modules: an imperative geometry construction interpreter to build constructions, a declarative geometry constraint solver to solve and test specifications, an observation-based perception module to notice interesting properties, and a learning module to analyze information from the other modules and integrate it into new definition and theorem discoveries.

To evaluate its recognition of such concepts, my system provides means for a user to extract the observations and apply its findings to new scenarios. Through a series of simple investigations similar to an introductory course in geometry, the system has been able to propose and learn a few dozen standard geometry theorems. [Furthermore, through more self-directed explorations, it has discovered several interesting properties and theorems not typically covered in standard mathematics courses.]

1.1 Document Structure

Chapter 2 further discusses motivation of the system and presents some examples of diagram manipulation, emphasizing the technique of visualizing diagrams "in the mind's eye."

Chapter 3 provides some sample interactions with the system and introduces the general system components.

Chapter 4 further introduces the system modules and discusses how they work together in the discovery of new definitions and theorems.

Chapters 5 - 8 describes the implementation and function of the four primary modules:

- **Chapter 5** describes the implementation and function of the imperative construction module that enables the system to carry out constructions.
- **Chapter 6** describes the implementation and function of the perception module focused on observing interesting properties in diagrams. A key question involves determining "what is interesting".
- **Chapter 7** describes the implementation and function of the propagator-based declarative geometry constraint solver that builds instances of diagrams satisfying declarative constraints.
- **Chapter 8** describes the analyzer module which integrates results from the other systems to create new discoveries. Main features include filtering out obvious or known results to focus on the most interesting discoveries, the persistence and storage of definitions and theorems, and an interface to apply these findings to new situations.

Chapter 9 discusses some related work to automated geometry theorem discovery and proof, as well as a comparison with existing dynamic geometry systems.

Chapter 10 evaluates the strengths and weaknesses of the system. Future work and possible extensions are discussed.

Finally, Appendix A includes the full listing for code used in the system.

Chapter 2

Motivation and Examples

Understanding elementary geometry is a fundamental reasoning skill, and encompasses a domain both constrained enough to model effectively, yet rich enough to allow for interesting insights. Although elementary geometry knowledge can be conveyed via series of factual definitions, theorems, and proofs, a particularly intriguing aspect of geometry is the ability for students to learn and develop an understanding of core concepts through visual investigation, exploration, and discovery.

These visual reasoning skills reflect many of the cognitive activities used as one interacts with his or her surroundings. Day-to-day decisions regularly rely on visual reasoning processes such as imagining what three dimensional objects look like from other angles, or mentally simulating the effects of one's actions on objects based on a learned understanding of physics and the object's properties. Such skills and inferred rules are developed through repeated observation, followed by the formation and evaluation of conjectures.

Similar to such day-to-day three-dimensional reasoning, visualizing and manipulating 2D geometric diagrams "in the mind's eye" allows one to explore questions such as "what happens if..." or "is it always true that..." to discover new conjectures. Further investigation of examples can increase one's belief in such a conjecture, and an accompanying system of deductive reasoning from basic axioms could prove that an observation is correct.

As an example, a curious student might notice that in a certain drawing of a triangle, the three perpendicular bisectors of the edges are concurrent, and that a circle constructed with center at the point of concurrence intersects all three vertices of the triangle. Given this “interesting observation”, the student might explore other triangles to see if this behavior is just coincidence, or conjecture about whether it applies to certain classes of triangles or all triangles in general. After investigating several other examples, the student might have sufficient belief in the conjecture to explore using previously-proven theorems (in this case, correspondences in congruent triangles) to prove the conjecture. My proposed project is a software system that simulates and automates this inductive thought process.

Automating geometric reasoning is not new, and has been an active field in computing and artificial intelligence. Dynamic geometry software, automated proof assistants, deductive databases, and several reformulations into abstract algebra models have been proposed in the last few decades. Although many of these projects have focused on the end goal of obtaining rigorous proofs of geometric theorems, I am particularly interested in exploring and modeling the more creative human-like thought processes of inductively exploring and manipulating diagrams to *discover* new insights about geometry.

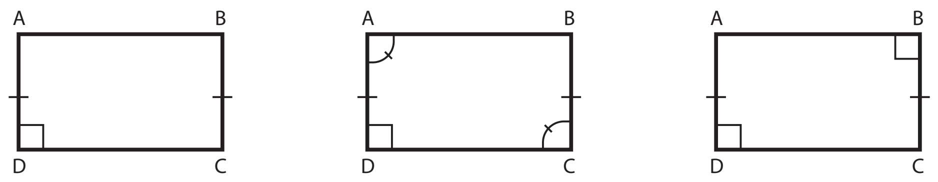
The interactive computer system presented in this thesis emulates the curious student described above, and is capable of exploring geometric concepts through inductive investigation. The system begins with a fairly limited set of factual knowledge regarding basic definitions in geometry and provides means by which a user interacting with the system can “teach” the system additional geometric concepts and theorems by suggesting investigations the system should explore to see if it “notices anything interesting.”

To evaluate its recognition of such concepts, the interactive system provide means for a user to extract the observations and apply such findings to new scenarios. In addition to the automated reasoning and symbolic artificial intelligence aspects of a system that can learn and reason inductively about geometry, the project also has some interesting opportunities to explore educational concepts related to experiential learning, and several extensions to integrate it with existing construction synthesis and proof systems.

2.1 Manipulating Diagrams "In the Mind's Eye"

Although the field of mathematics has developed a rigorous structure of deductive proofs explaining most findings in geometry, much of human intuition and initial reasoning about geometric ideas come not from applying formal rules, but rather from visually manipulating diagrams "in the mind's eye." Consider the following example:

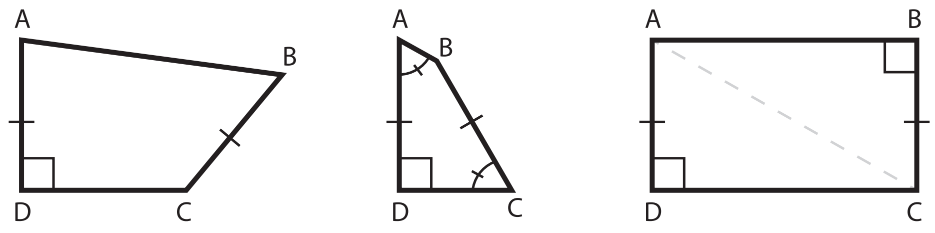
2.1.1 An Initial Example



Example 1: Of the three diagrams above, determine which have constraints sufficient to restrict the quadrilateral $ABCD$ to always be a rectangle.

An automated deductive solution to this question could attempt to use forward-chaining of known theorems to determine whether there was a logical path that led from the given constraints to the desired result that the quadrilateral shown is a rectangle. However, getting the correct results would require having a rich enough set of inference rules and a valid logic system for applying them.

A more intuitive visual-reasoning approach usually first explored by humans is to initially verify that the marked constraints hold for the instance of the diagram as drawn and then mentally manipulate or "wiggle" the diagram to see if one can find a nearby counter-example that still satisfies the given constraints, but is not a rectangle. If the viewer is unable to find a counter-example after several attempts, he or she may be sufficiently convinced the conclusion is true, and could commit to exploring a more rigorous deductive proof.



Solution to Example 1: As the reader likely discovered, the first two diagrams can be manipulated to yield instances that are not rectangles, while the third is sufficiently constrained to always represent a rectangle. (This can be proven by adding a diagonal and using the Pythagorean theorem.)

2.1.2 Diagrams, Figures, and Constraints

This example of manipulation using the "mind's eye" also introduces some terminology helpful in discussing the differences between images as drawn and the spaces of geometric objects they represent. For clarity, a *figure* will refer to an actual configuration of points, lines, and circles drawn on a page. Constraint annotations (congruence or measure) added to a figure create a *diagram*, which represents the entire space of figure *instances* that satisfy the constraints.

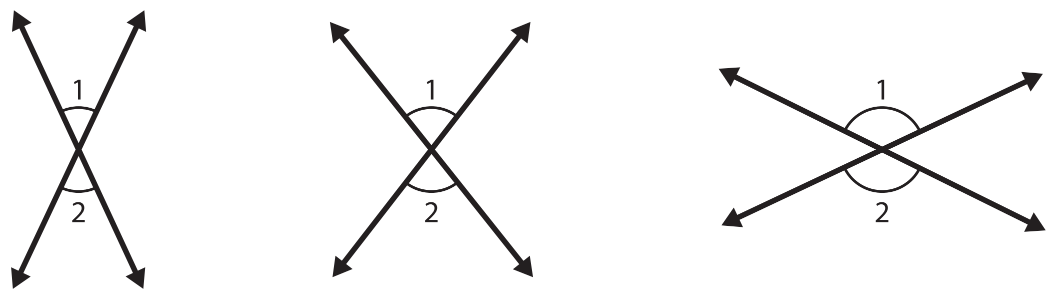
An annotated figure presented on a page is typically an instance of its corresponding diagram. However, it is certainly possible to add annotations to a figure that are not satisfied by that figure, yielding impossible diagrams. In such a case the diagram represents an empty set of satisfying figures.

In the initial example above, the three quadrilaterals figures are drawn as rectangles. It is true that all quadrilateral figures in the space represented by the third diagram are rectangles. However, the space of quadrilaterals represented by the first two diagrams include instances that are not rectangles, as shown above. At this time, the system only accepts diagrams whose constraints are satisfied in a given figure. However, detecting and explaining impossible diagrams, purely from their set of constraints could be an interesting extension.

2.2 Geometry Investigation

These same “mind’s eye” reasoning techniques can be used to discover and learn new geometric theorems. Given some “interesting properties” in a particular figure, one can construct other instances of the diagram to examine if the properties appear to hold uniformly, or if they were just coincidences in the initial drawing. Properties that are satisfied repeatedly can be further explored and proved using deductive reasoning. The examples below provide several demonstrations of such inductive investigations.

2.2.1 Vertical Angles

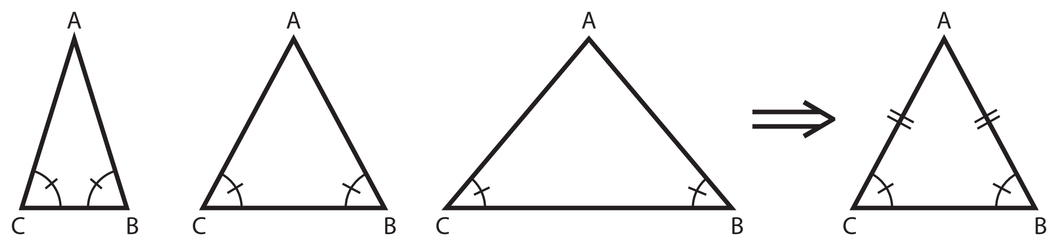


Investigation 1: Construct a pair of vertical angles. Notice anything interesting?

Often one of the first theorems in a geometry course, the fact that vertical angles are equal is one of the simplest examples of applying “mind’s eye” visual reasoning. Given the diagram on the left, one could “wiggle” the two lines in his or her mind and imagine how the angles respond. In doing so, one would notice that the lower angle’s measure increases and decreases proportionately with that of the top angle. This mental simulation, perhaps accompanied by a few drawn and measured figures, could sufficiently convince the viewer that vertical angles always have equal measure.

Of course, this fact can also be proved deductively by adding up pairs of angles that sum to 180 degrees, or by using a symmetry arguments. However, the inductive manipulations are more reflective of the initial, intuitive process one typically takes when first presented with understanding a problem.

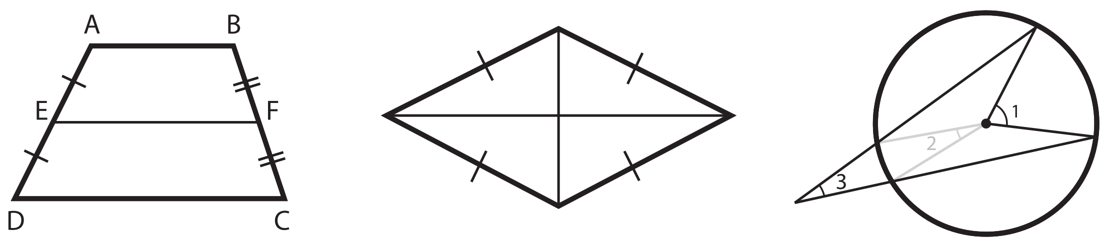
2.2.2 Elementary Results



Investigation 2: Construct a triangle ABC with $\angle B = \angle C$. Notice anything interesting?

A slightly more involved example includes discovering that if a triangle has two congruent angles, it is isoceles. As above, this fact has a more rigorous proof that involves dropping an altitude from point A and using corresponding parts of congruent triangles to demonstrate the equality of AB and AC. However, the inductive investigation of figures that satisfy the constraints can yield the same conjecture, give students better intuition for what is happening, and help guide the discovery and assembly of known rules to be applied in future situations.

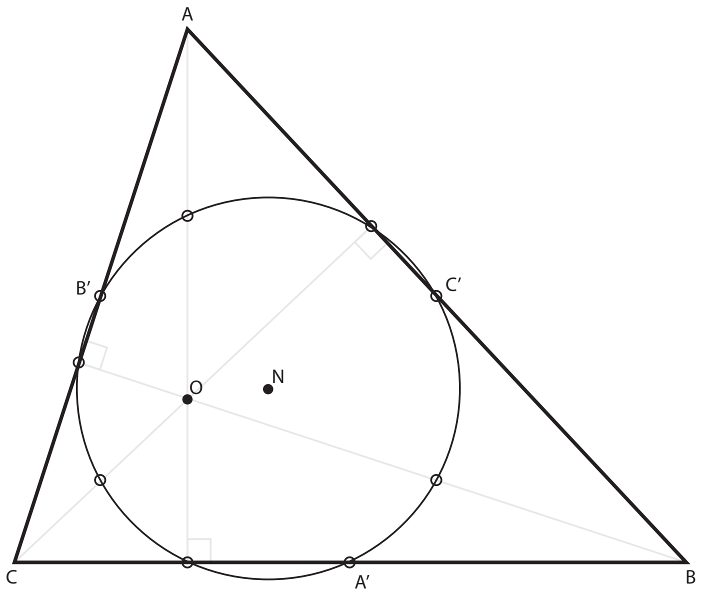
In this and further examples, an important question becomes what properties are considered “interesting” and worth investigating in further instances of the diagram, as discussed in section 4.3.3. As suggested by the examples in Investigation 3, this can include relations between segment and angle lengths, concurrent lines, collinear points, or parallel and perpendicular lines.



Investigation 3: What is interesting about the relationship between AB, CD, and EF in the trapezoid? What is interesting about the diagonals of a rhombus? What is interesting about $\angle 1$, $\angle 2$, and $\angle 3$?

2.2.3 Nine Point Circle and Euler Segment

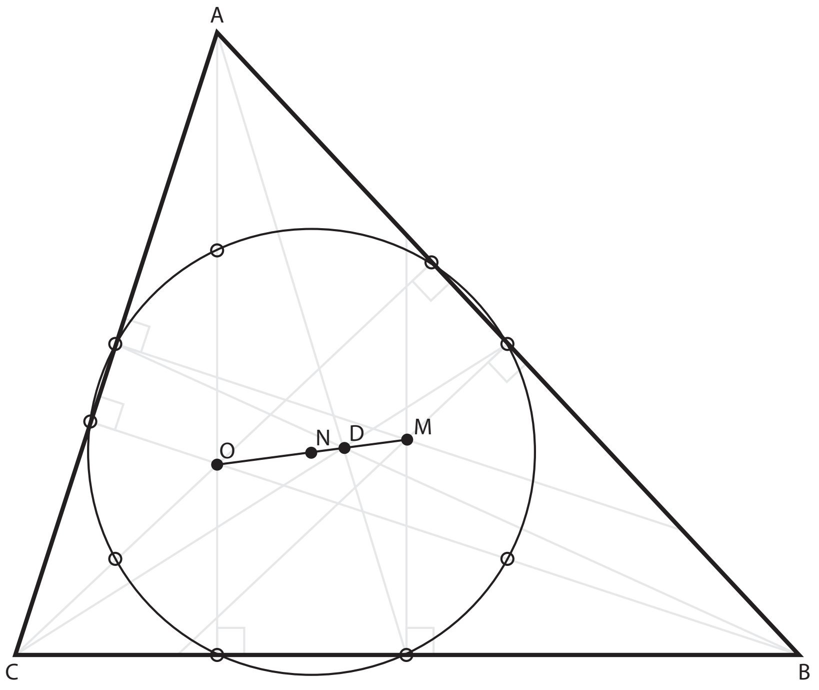
Finally, this technique can be used to explore and discover conjectures well beyond the scope of what one can visualize in his or her head:



Investigation 4a: In triangle ABC, construct the side midpoints A', B', C', and orthocenter O (from altitudes). Then, construct the midpoints of the segments connecting the orthocenter with each triangle vertex. Notice anything interesting?

As a more complicated example, consider the extended investigation of the Nine Point Circle and Euler Segment. As shown in Investigation 4a, the nine points created (feet of the altitudes, midpoints of sides, and midpoints of segments from orthocenter to vertices) are all concentric, lying on a circle with center labeled N.

Upon first constructing this figure, this fact seems almost beyond chance. However, as shown in Investigation 4b (below), further "interesting properties" continue to appear as one constructs the centroid and circumcenter: All four of these special points (O, N, D, and M) are collinear on what is called the *Euler Segment*, and the ratios $ON : ND : DM$ of 3 : 1 : 2 hold for any triangle.



Investigation 4b: Continue the investigation from 4a by also constructing the centroid D (from medians) and circumcenter M (from perpendicular bisectors). Notice anything interesting?

(Maybe I'll try to add in some more concluding remarks about this "mind's eye" concept.)

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.

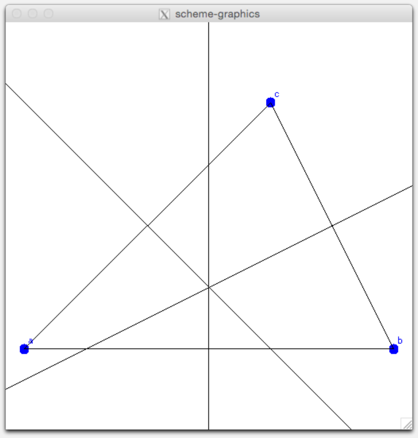
Code Example 3.1: Basic Figure Example

```
(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

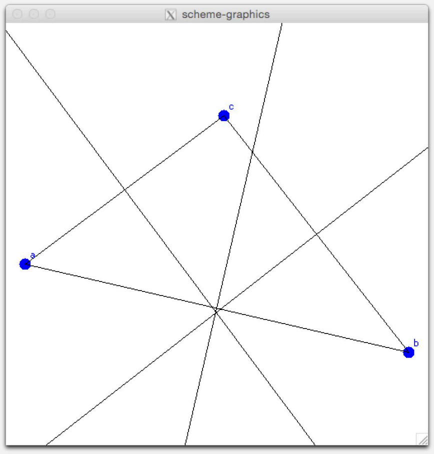
```
=> (show-figure (triangle-with-perp-bisectors))
```

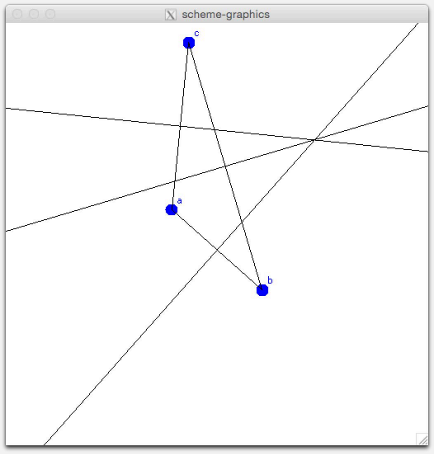


In the first figure, the coordinates of the point were explicitly specified yielding a deterministic instance of the figure. However, as geometry figures often involve arbitrary choices, the construction abstractions support random choices. Figure 3.3 demonstrates the creation of a figure involving an 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)))
```

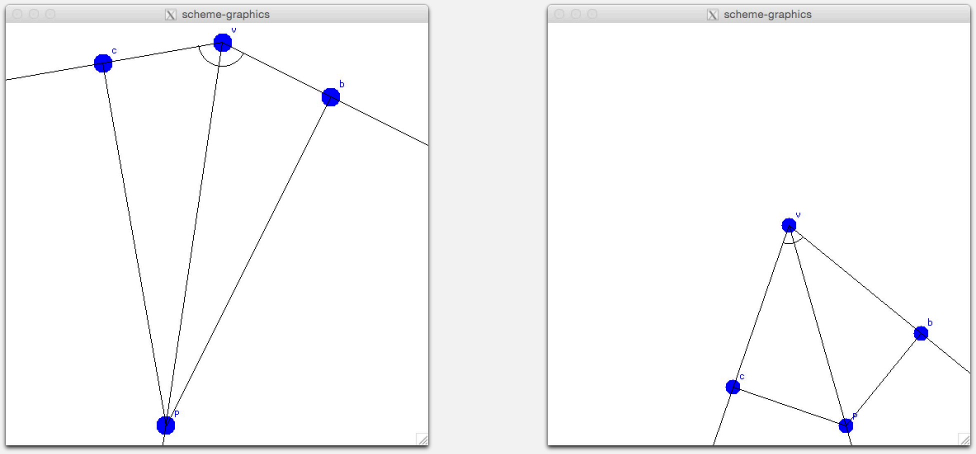




Finally, as examples of more involved constructions, Examples 3.4 and 3.5 demonstrate working with other objects (angles, rays, circles) and construction procedures. Notice that in the angle bisector example the pattern matching syntax extracts the components of an angle (ray, vertex, ray) and segment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

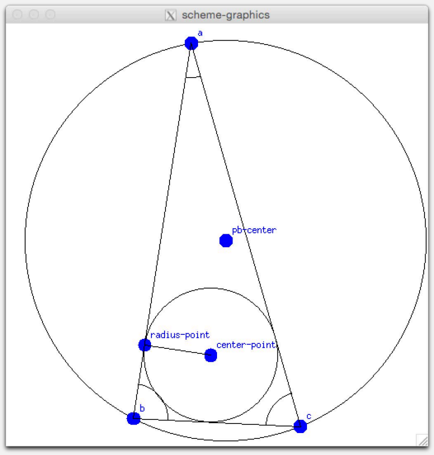
Interaction Example 3.4: Angle Bisector Distance

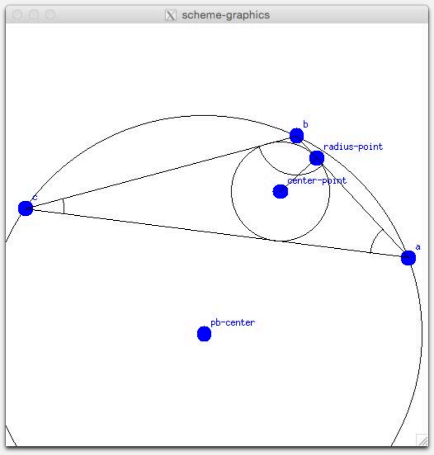
```
(define (angle-bisector-distance)
  (let-geo* (((a (r-1 v r-2)) (random-angle))
             (ab (angle-bisector a))
             (p (random-point-on-ray ab))
             ((s-1 (p b)) (perpendicular-to r-1 p))
             ((s-2 (p c)) (perpendicular-to r-2 p)))
     (figure a r-1 r-2 ab p s-1 s-2)))

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



Interaction Example 3.5: 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)))
             (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))
```





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 [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. 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

```
(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))))
```

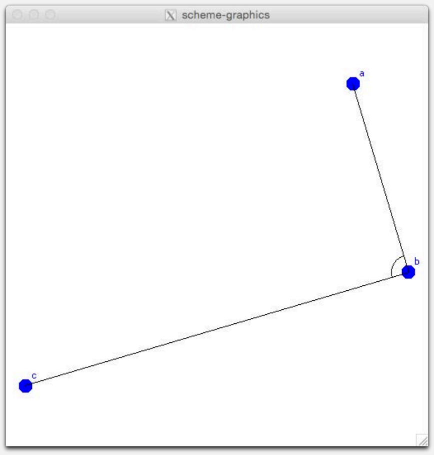
Building a mechanism involves first assembling the bars and joints together so that the named points are identified with one another. Initially, each bar has unknown length and direction, each joint has an unknown angle, and each endpoint has unknown position. Constraints for the bar and joint properties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

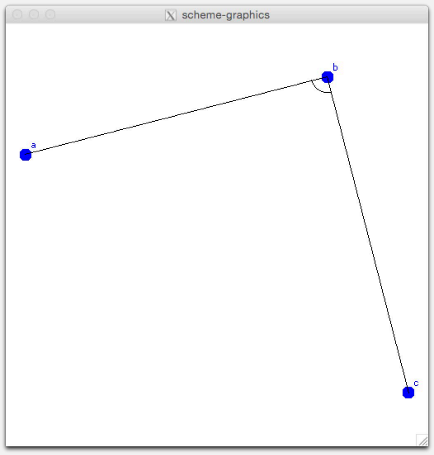
Solving the mechanism involves repeatedly selecting position, lengths, angles, and directions that are not fully specified and selecting values within the domain of that value's current partial information. As values are specified, the wiring of the propagator model propagates further partial information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

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 `m:establish-polygon-topology` procedure simplifies the specification of a closed polygon of joints.

Code Example 3.11: Describing an Arbitrary Triangle

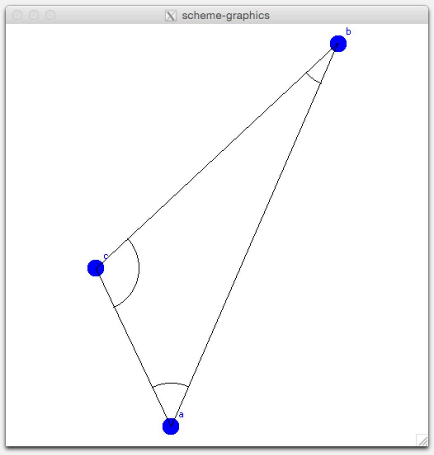
```
(define (arbitrary-triangle)
  (m:mechanism
   (m:make-named-bar 'a 'b)
   (m:make-named-bar 'b 'c)
   (m:make-named-bar 'c 'a)
   (m:make-named-joint 'a 'b 'c)
   (m:make-named-joint 'b 'c 'a)
   (m:make-named-joint 'c 'a 'b)))

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

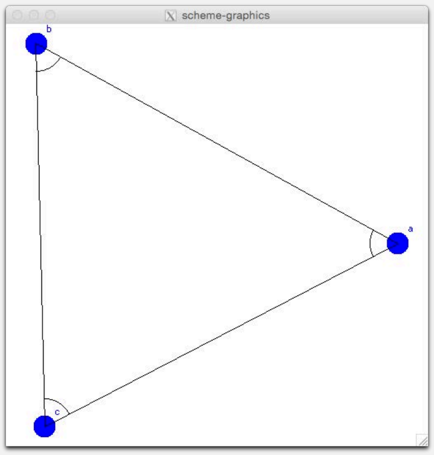
In example 3.12, Once joints b and c have had their angles specified, propagation sets the angle of joint a to a unique value. The only parameter to specify is the length of one of the bars. The two `initializing-` steps don't affect the resulting shape but determine its position and orientation on the 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. In some scenarios, specifying values in the wrong order can yield premature contradictions. A planned future extension will attempt to recover from such situations more gracefully by trying other orderings of 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)
```



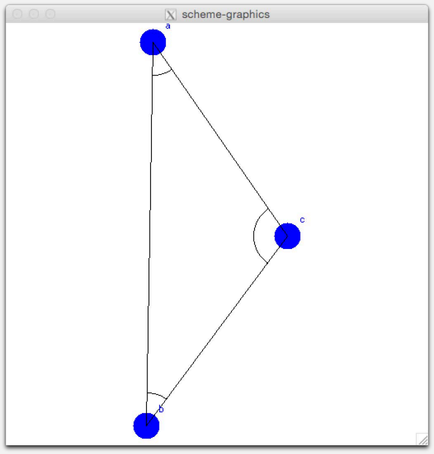


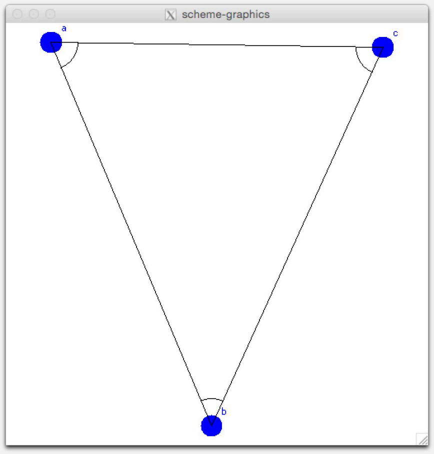
Example 3.13 shows the solving steps involved in solving an isoceles triangle from the fact that its base angles are congruent. Notice that the only two values that must be specified are one joint angle and one bar length. The rest is handled by propagation.

The values used in the 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 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. 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)
```

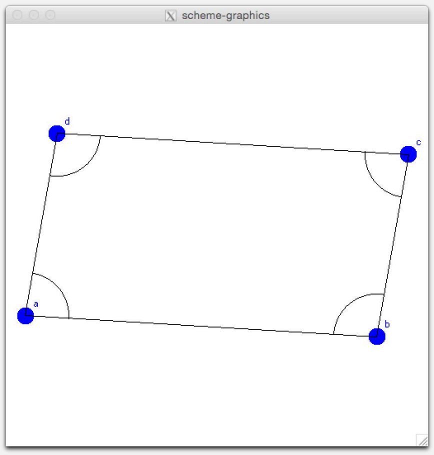


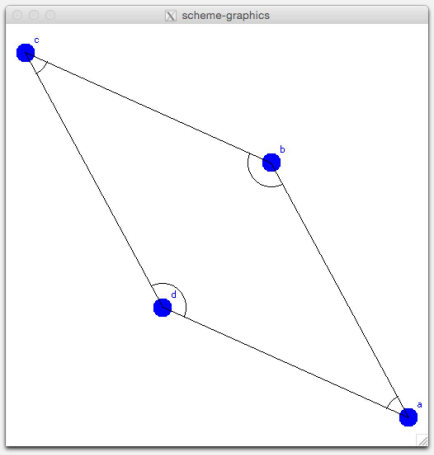


Example 3.14 continues our analysis of properties of the parallelogram. In this case, our constraint solver is able to build figures given the fact that its opposite angles are equal. The fact that these all happen to be parallelograms will be used by the learning module to produce a simpler definition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





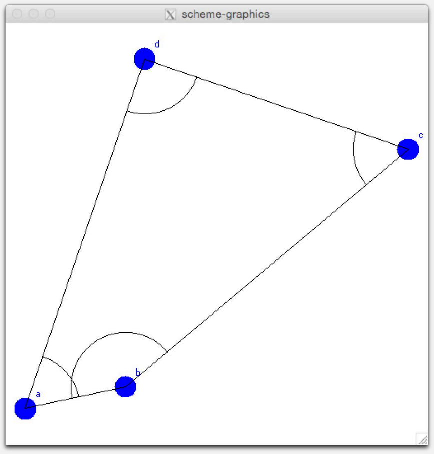
As a more complicated example, Example 3.15 demonstrates the constraint solving from the middle "Is this a rectangle?" question from Chapter 2. Try working this constraint problem by hand. As we see in 3.16, solutions are not all rectangles. Chapter 7 includes a more detailed walkthrough of how this example is solved.

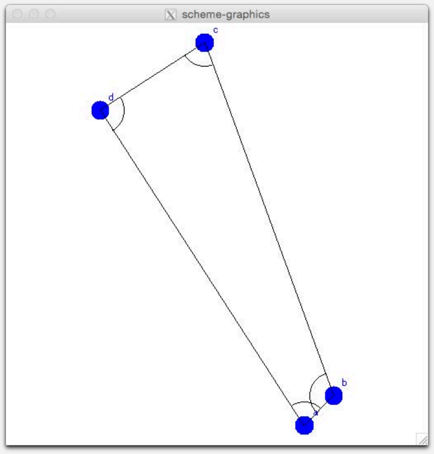
Code Example 3.15: Rectangle Constraints Example

```
(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))))
```

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)
```





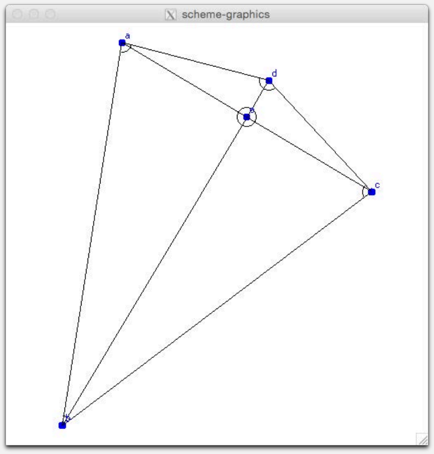
Finally, in addition to solving constraints of the angles and sides for a single polygon, the mechanism system allows for the creation of arbitrary topologies of bars and linkages. In the following examples, we build the topology of a quadrilaterals whose diagonals intersect at a point e and explore the effects of various constraints on these diagonal segments.

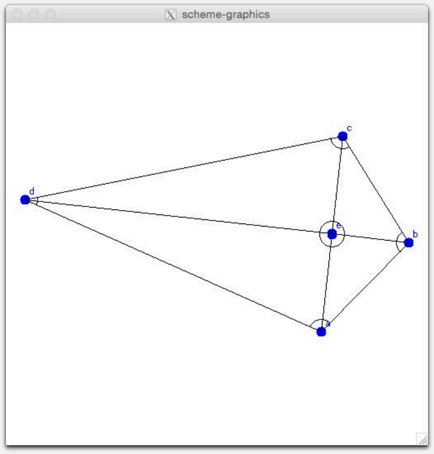
Code Example 3.17: More Involved Topologies for Constraint Solving

```scheme
(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

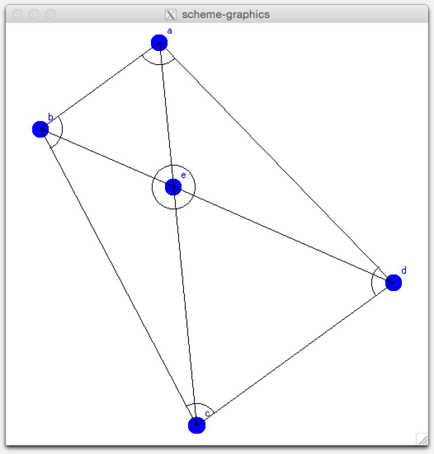
```scheme
(define (kite-from-diagonals)
  (m:mechanism
   (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
   (m:c-right-angle (m:joint 'b 'e 'c)) ;; Right Angle in Center
   (m:c-length-equal (m:bar 'c 'e) (m:bar 'a 'e))))

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

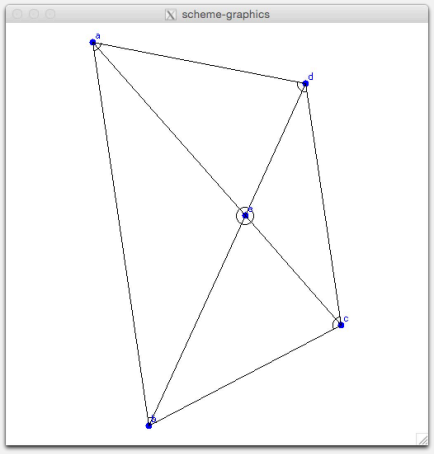




Interaction Example 3.19: Isoceles Trapezoids from Diagonals

```
(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)
```

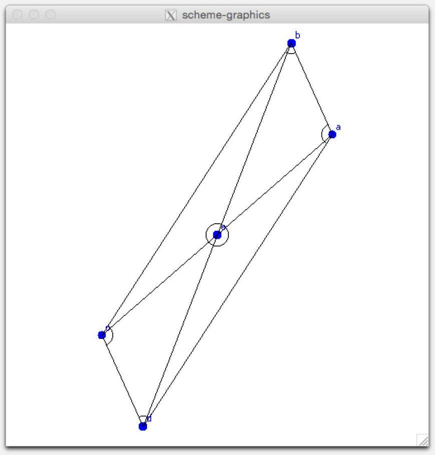


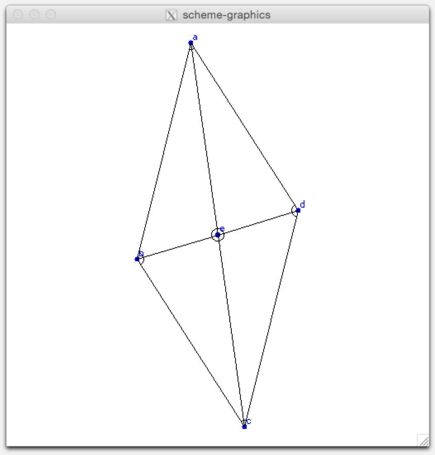


Interaction Example 3.20: Parallelograms from Diagonal Properties

```
(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))))
```





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.

Example 3.21 shows that the system begins with some knowledge of primitive objects (point, line, ray), and the most basic polygon terms (triangle, quadrilateral). However, upon startup, it knows nothing about higher-level terms such as trapezoids, parallelograms, or isoceles triangles.

Interaction Example 3.21: Querying Terms

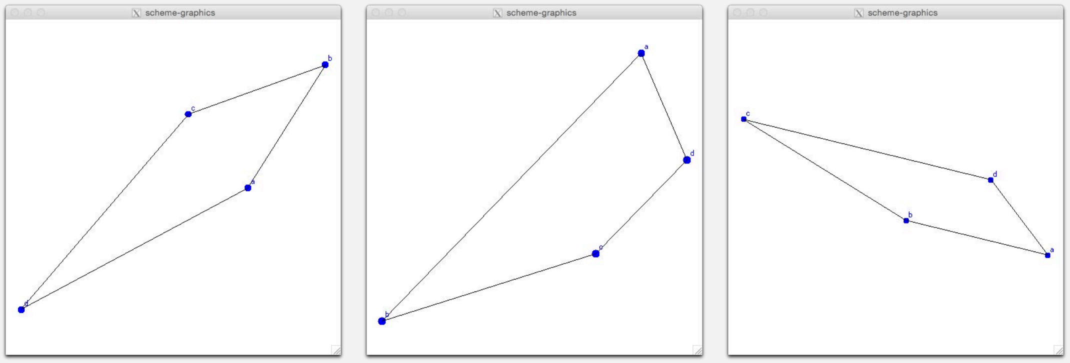
```
=> (what-is 'trapezoid)
unknown

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

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

```
(triangle (polygon)
          ((n-sides-3 identity)))
```

A user can create an investigation to help the system learn a new definition by creating a procedure that creates random elements satisfying that definition. 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 about the given element. In this case (3.23), we see the full dependencies of the elements under consideration instead of simply their names.

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 abilities, the system can be taught new definitions by providing a term ('pl for parallelogram) and a generator procedure that produces instances of that element. As shown in example 3.24, after being instructed to learn what a parallelogram is from the `random-parallelogram` procedure, when queried for a definition, we're given the term, then the base definition of this element, then all properties known to be true of such objects.

Interaction Example 3.24: Learning Parallelogram Definition

```
=> (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>))))
```

To use such learned knowledge, we can use `is-a?` to test whether other elements are satisfy the definition of the term. As shown in example 3.25, 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 3.25: Testing Definitions

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

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

=> (is-a? 'pl (polygon-from-points
                (make-point 0 0)
                (make-point 1 0)
                (make-point 2 1)
                (make-point 1 1)))
#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)
```

Learning individual definitions is nice, but cool properties arise when definitions build 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 lattice of terms. In example 3.26, we see that after learning definitions of kites and rhombuses, the resulting definition of a rhombus is that it a parallelogram and kite that satisfies two additional properties. Later, after learning a rectangle, amazingly, the system shows us that the definition of a square is just a rhombus and rectangle with no additional properties.

Interaction Example 3.26: 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) ())
```

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 3.27, 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 3.27: 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)

((invalid-definition ())
 (valid-definition
  ((equal-length (segment a b) (segment c a))))
 (valid-definition
  ((equal-angle (angle b) (angle c))))
 (valid-definition
  ((equal-length (segment a b) (segment c a))
   (equal-angle (angle b) (angle c)))))
```

In the parallelogram example 3.28, 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 3.28: Learning Simple Parallelogram Definitions

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

((invalid-definition ())
 (invalid-definition ((equal-length (segment a b) (segment c d))))
 (invalid-definition ((equal-angle (angle a) (angle c))))
 (invalid-definition ((equal-angle (angle b) (angle d))))
 (valid-definition
  ((equal-length (segment a b) (segment c d))
   (equal-length (segment b c) (segment d a))))
 (invalid-definition
  ((equal-length (segment b c) (segment d a))
   (equal-angle (angle b) (angle d))))
 (valid-definition
  ((equal-angle (angle a) (angle c))
   (equal-angle (angle b) (angle d))))
 (valid-definition
  ((equal-length (segment a b) (segment c d))
   (equal-length (segment b c) (segment d a))
   (equal-angle (angle a) (angle c))))
 (valid-definition
  ((equal-length (segment a b) (segment c d))
   (equal-length (segment b c) (segment d a))
   (equal-angle (angle b) (angle d))))
 (valid-definition
  ((equal-length (segment a b) (segment c d))
   (equal-angle (angle a) (angle c))
   (equal-angle (angle b) (angle d))))
 (valid-definition
  ((equal-length (segment b c) (segment d a))
   (equal-angle (angle a) (angle c))
   (equal-angle (angle b) (angle d))))
 (valid-definition
  ((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)))))
```

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)

Chapter 4

System Overview

My system uses this idea of manipulating diagrams "in the mind's eye" to explore and discover geometry theorems. Before discussing some of the internal representations and modules, I will briefly describe the goals of the system to provide direction and context to understand the components.

4.1 Goals

The end goal of the system is for it to be to notice and learn interesting concepts in Geometry from inductive explorations.

Because these ideas are derived from inductive observation, we will typically refer to them as conjectures. Once the conjectures are reported, they can easily be integrated into existing automated proof systems if a deductive proof is desired.

The conjectures explored in this system can be grouped into three areas: definitions, properties, and theorems.

Properties Properties include all the facts derived from a single premise. "Opposite angles in a rhombus are equal" or "The midpoint of a segment divides it into two equal-length segments".

Definitions Definitions classify and differentiate an object from other objects. For instance"What is a rhombus?" yields the definition that it is a quadrilateral (classification) with four equal sides (differentiation). For definitions, the system will attempt to simplify definition properties to more minimal sets, provide alternative formations, and use pre-existing definitions when possible: "A Square is a rhombus and a rectangle"

Theorems Theorems are very similar to properties but involve several premises. For instance, theorems about triangles may involve the construction of angle bisectors, incenters or circumcenters, or the interaction among several polygons in the same diagram.

Finally, given a repository of these conjectures about geometry, the system will be able to apply its findings in future investigations by examining elements to display its knowledge of definitions, and focusing future investigations by omitting results implied by prior theorems.

4.2 Diagram Representations

The system and modules are built around three core representations. As discussed in the motivation section, we use the term "diagram" to represent the abstract geometric object represented by these means:

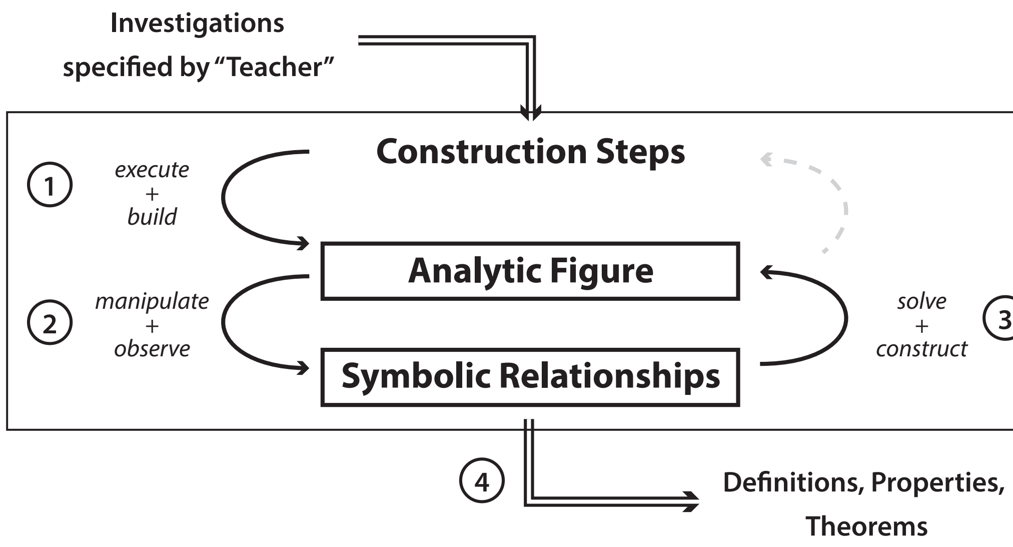
Construction Steps The main initial representation of most diagrams is a series of construction steps. These generally make up the input investigation from an external user trying to teach the system a concept. In some investigations, the actual construction steps are opaque to the system (as in a teacher that provides a process to "magically" produce rhombuses), but often, the construction steps use processes known by the system so that the resulting figures can include dependency information about how the figure was built.

Analytic Figure The second representation is an analytic figure for a particular instance of a diagram. This representation can be drawn and includes coordinates for all points in the diagram. This representation is used by the perception module to observe interesting relationships.

Symbolic Relationships Finally, the third representation is a collection of symbolic relationships or constraints on elements of the diagram. These are initially formed from the results of the perception module, but may also be introduced as known properties for certain premises and construction steps. These symbolic relationships can be further tested and simplified to discover which sets of constraints subsume one another.

While construction steps are primarily used as input and to generate examples, as the system investigates a figure, the analytic figure and symbolic relationship models get increasingly intertwined. The "mind's eye" perception aspects of observing relationships in the analytic figure lead to new symbolic relationships and a propagator-like approach of wigging solutions to the symbolic constraints yields new analytic figures.

As relationships are verified and simplified, results are output and stored in the student's repository of geometry knowledge. This process is depicted in the figure below and components are described in the following chapters.



System Overview: Given construction steps for an investigation an external teacher wishes the student perform, the system first (1) uses its imperative construction module to execute these construction steps and build an analytic instance of the diagram. Then, (2) it will manipulate the diagram by "wiggling" random choices and use the perception module to observe interesting relationships. Given these relationships, it will (3) use the declarative propagator-based constraint solver to reconstruct a diagram satisfying a subset of the constraints to determine which are essential in the original diagram. Finally (4), a learning module will monitor the overall process, omit already-known results, and assemble a repository of known definitions, properties, and theorems.

4.2.1 Modules

These four modules include an imperative geometry construction interpreter used to build diagrams, a declarative geometry constraint solver to solve and test specifications, an observation-based perception module to notice interesting properties, and a learning module to analyze information from the other modules and integrate it into new definition and theorem discoveries.

4.3 Steps in a Typical Interaction

This core system provides an interpreter to accept input of construction instructions, an analytic geometry system that can create instances of such constructions, a pattern-finding process to discover “interesting properties”, and an interface for reporting findings.

4.3.1 Interpreting Construction Instructions

The first step in such explorations is interpreting an input of the diagram to be explored. To avoid the problems involved with solving constraint systems and the possibility of impossible diagrams, the core system takes as input explicit construction steps that results in an instance of the desired diagram. These instructions can still include arbitrary selections (let P be some point on the line, or let A be some acute angle), but otherwise are restricted to basic construction operations using a compass and straight edge.

To simplify the input of more complicated diagrams, some of these steps can be abstracted into a library of known construction procedures. For example, although the underlying figures are be limited to very simple objects of points, lines, and angles, the steps of constructing a triangle (three points and three segments) or bisecting a line or angle can be encapsulated into single steps.

4.3.2 Creating Figures

Given a language for expressing the constructions, the second phase of the system is to perform such constructions to yield an instance of the diagram. This process mimics “imagining” manipulations and results in an analytic representation of the figure with coordinates for each point. Arbitrary choices in the construction (“Let Q be some point not on the line.”) are chosen via an random process, but with an attempt to keep the figures within a reasonable scale to ease human inspection.

4.3.3 Noticing Interesting Properties

Having constructed a particular figure, the system examines it to find interesting properties. These properties involve facts that appear to be “beyond coincidence”. This generally involves relationships between measured values, but can also include "unexpected" configurations of points, lines, and circles. As the system discovers interesting properties, it will reconstruct the diagram using different choices and observe if the observed properties hold true across many instances of a diagram.

4.3.4 Reporting Findings

Finally, once the system has discovered some interesting properties that appear repeatedly in instances of a given diagram, it reports its results to the user via the learning module. Although this includes a simple list of all simple relationships, effort is taken to avoid repeating observations that obvious in the construction. For example, if a perpendicular bisector of segment AB is requested, the fact that it bisects that segment in every instance is not informative. To do so, the construction process interacts with properties known in the learning module to maintain a list of facts that can be reasoned from construction assumptions so that these can be omitted in the final reporting.

Chapter 5

Imperative Construction System

5.1 Overview

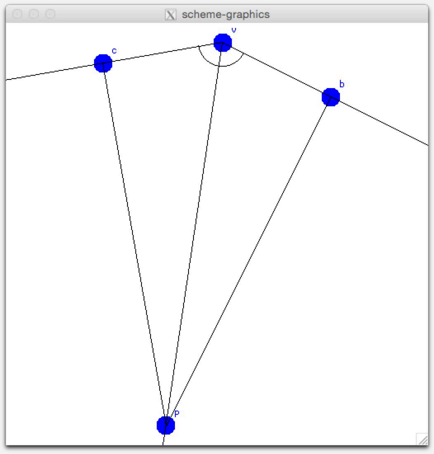
The first module is an imperative system for performing geometry constructions. This is the typical input method for generating coordinate-backed, analytic instances of figures.

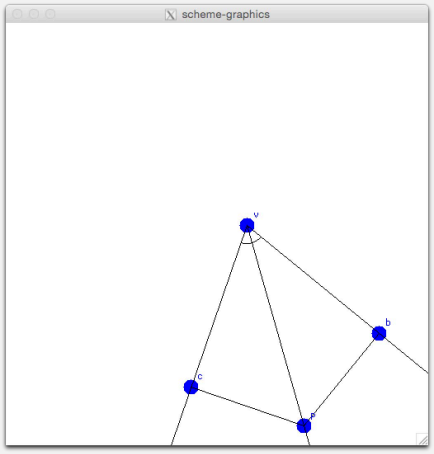
The construction system is comprised of a large, versatile library of useful utility and construction procedures for creating figures. To appropriately focus the discussion of this module, I will concentrate on the implementation of structures and procedures necessary for the sample construction seen in Example 5.1. Full code and more usage examples are provided in Appendix A.

In doing so, I will first describe the basic structures and essential utility procedures before presenting some higher-level construction procedures, polygons, and figures. Then, I will Then, I will explore the use of randomness in the system and examine how construction language macros handle names, dependencies, and multiple assignment of components. Finally, I will briefly discuss the interface and implementation for animating and displaying figures.

Interaction Example 5.1: Sample Construction

```
(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))
```





5.2 Basic Structures

The basic structures in the imperative construction system are points, segments, rays, lines, angles, and circles. These structures, as with all structures in the system are implemented using Scheme record structures as seen in Listings 5.2 and 5.3. In the internal representations, lines and segments are directioned. Predicates exist to allow other procedures to work with or ignore these directions.

Code Listing 5.2: Basic Structures

```
(define-record-type <point>
  (make-point x y)
  point?
  (x point-x)
  (y point-y))

```

```scheme
(define-record-type <segment>
  (%segment p1 p2)
  segment?
  (p1 segment-endpoint-1)
  (p2 segment-endpoint-2))

(define-record-type <line>
  (%make-line point dir)
  line?
  (point line-point) ;; Point on the line
  (dir line-direction))
```

Code Listing 5.3: Angle and Circle Structures

```scheme
(define-record-type <angle>
  (make-angle dir1 vertex dir2)
  angle?
  (dir1 angle-arm-1)
  (vertex angle-vertex)
  (dir2 angle-arm-2))

(define-record-type <circle>
  (make-circle center radius)
  circle?
  (center circle-center)
  (radius circle-radius))
```

5.2.1 Creating Elements

Elements can be created explicitly using the underlying `make-*` constructors defined with the record types. However, several higher-order constructors are provided to simplify construction as shown in Listings 5.4 and 5.5. In `angle-from-lines`, we make use of the fact that lines are directioned to uniquely specify an angle. As with the angle construction case, in several instances, we use generic operations to handle mixed types of geometry elements.

Code Listing 5.4: Higher-order Constructors

```scheme
(define (line-from-points p1 p2)
  (make-line p1 (direction-from-points p1 p2)))
```

Code Listing 5.5: Generic Constructors for Creating Angles

```
(define angle-from (make-generic-operation 2 'angle-from))

(define (angle-from-lines l1 l2)
  (let ((d1 (line->direction l1))
        (d2 (line->direction l2))
        (p (intersect-lines l1 l2)))
    (make-angle d1 p d2)))
(defhandler angle-from angle-from-lines line? line?)
```

5.2.2 Essential Math Utilities

Several math utility structures support these constructors and other geometry procedures. One particularly useful abstraction is a `direction` that fixes a direction in the interval $[0, 2\pi]$. Listing 5.6 provides a taste of some operations using such abstractions.

Code Listing 5.6: Directions

```
(define (subtract-directions d2 d1)
  (if (direction-equal? d1 d2)
      0
      (fix-angle-0-2pi (- (direction-theta d2)
                          (direction-theta d1)))))

(define (direction-perpendicular? d1 d2)
  (let ((difference (subtract-directions d1 d2)))
    (or (close-enuf? difference (/ pi 2))
        (close-enuf? difference (* 3 (/ pi 2))))))
```

5.3 Higher-order Procedures and Structures

Higher-order construction procedures and structures are built upon these basic elements and utilities. Listing 5.7 shows the implementation of the perpendicular constructions used in the sample figure.

Code Listing 5.7: Perpendicular Constructions

```
;; Constructs line through point perpendicular to linear-element
(define (perpendicular linear-element point)
  (let* ((direction (->direction linear-element))
         (rotated-direction (rotate-direction-90 direction)))
    (make-line point rotated-direction)))

;;; Constructs perpendicular segment from point to linear-element
(define (perpendicular-to linear-element point)
  (let ((pl (perpendicular linear-element point)))
    (let ((i (intersect-linear-elements pl (->line linear-element))))
      (make-segment point i))))
```

Although traditional constructions generally avoid using rulers and protractors, Listing 5.8 shows the implementation of the `angle-bisector` procedure from our sample figure that uses measurements to simplify construction.

Code Listing 5.8: Angle Bisector Construction

```
(define (angle-bisector a)
  (let* ((d2 (angle-arm-2 a))
         (vertex (angle-vertex a))
         (radians (angle-measure a))
         (half-angle (/ radians 2))
         (new-direction (add-to-direction d2 half-angle)))
    (make-ray vertex new-direction)))
```

5.3.1 Polygons and Figures

Polygons record structures contain an ordered list of points in counter-clockwise order, and provide procedures such as `polygon-point-ref` or `polygon-segment` to obtain particular points, segments, and angles specified by indices.

Figures are simple groupings of geometry elements and provide procedures for extracting all points, segments, angles, and lines contained in the figure, including ones extracted from within polygons or subfigures.

5.4 Random Choices

To allow figures to represent general spaces of diagrams, random choices are commonly used to instantiate diagrams. In our sample figure, we use `random-angle` and `random-point-on-ray`, implementations of which are shown in listing 5.9. Underlying these procedures are calls to Scheme's random function over a specified range ($[0, 2\pi]$ for `random-angle-measure`, for instance). Since infinite ranges are not well supported and to ensure the figures stay reasonable legible for a human viewer, `extend-ray-to-max-segment` clips a ray at the current working canvas so a point on the ray can be selected within the working canvas.

Code Listing 5.9: Random Constructors

```scheme
(define (random-angle)
  (let* ((v (random-point))
         (d1 (random-direction))
         (d2 (add-to-direction
              d1
              (rand-angle-measure))))
    (make-angle d1 v d2)))

(define (random-point-on-ray r)
  (random-point-on-segment
   (extend-ray-to-max-segment r)))

(define (random-point-on-segment seg)
  (let* ((p1 (segment-endpoint-1 seg))
         (p2 (segment-endpoint-2 seg))
         (t (safe-rand-range 0 1.0))
         (v (sub-points p2 p1)))
    (add-to-point p1 (scale-vec v t))))

(define (safe-rand-range min-v max-v)
  (let ((interval-size (max 0 (- max-v min-v))))
    (rand-range
     (+ min-v (* 0.1 interval-size))
     (+ min-v (* 0.9 interval-size)))))
```

Other random elements are created by combining these random choices, such as the random parallelogram in Listing 5.10.

Code Listing 5.10: Random Parallelogram

```
(define (random-parallelogram)
  (let* ((r1 (random-ray))
         (p1 (ray-endpoint r1))
         (r2 (rotate-about (ray-endpoint r1)
                           (rand-angle-measure)
                           r1))
         (p2 (random-point-on-ray r1))
         (p4 (random-point-on-ray r2))
         (p3 (add-to-point
              p2
              (sub-points p4 p1))))
    (polygon-from-points p1 p2 p3 p4)))
```

5.4.1 Backtracking

The module currently only provides limited support for avoiding degenerate cases, or cases where randomly selected points happen to be very nearly on top of existing points. Several random choices use `safe-rand-range` (seen in Listing 5.9) to avoid the edge cases of ranges, but further extensions could improve this system to periodically check for unintended relationships and backtrack to choose other values.

5.5 Construction Language Support

To simplify specification of figures, the module provides the `leg-geo*` macro which allows for a multiple-assignment-like extraction of components from elements and automatically tags resulting elements with their variable names for future reference. Listing 5.11 shows the expansion of a simple usage of `let-geo*` and listing 5.12 shows some of the macros' implementation.

Code Example 5.11: Expansion of let-geo* macro

```
(let-geo* (((a (r-1 v r-2)) (random-angle)))
  (figure a r-1 r-2 ...))

(let* ((a (random-angle))
       (r-1 (element-component a 0))
```

```
      (v   (element-component a 1))
      (r-2 (element-component a 2)))
 (set-element-name! a   'a)
 (set-element-name! r-1 'r-1)
 (set-element-name! v   'v)
 (set-element-name! r-2 'r-2)
 (figure a r-1 r-2 ...))
```

Code Listing 5.12: Multiple and Component Assignment

```
(define (expand-compound-assignment lhs rhs)
  (if (not (= 2 (length lhs)))
      (error "Malformed compound assignment LHS (needs 2 elements): " lhs))
  (let ((key-name (car lhs))
        (component-names (cadr lhs)))
    (if (not (list? component-names))
        (error "Component names must be a list:" component-names))
    (let ((main-assignment (list key-name rhs))
          (component-assignments (make-component-assignments
                                  key-name
                                  component-names)))
      (cons main-assignment
            component-assignments))))

(define (make-component-assignments key-name component-names)
  (map (lambda (name i)
         (list name `(element-component ,key-name ,i)))
       component-names
       (iota (length component-names))))
```

Once expanded, a generic `element-component` operator shown in Listing 5.13 defines what components are extracted from what elements (endpoints for segments, vertices for polygons, (ray, angle, ray) for angles).

Code Listing 5.13: Generic Element Component Handlers

```
(declare-element-component-handler polygon-point-ref polygon?)

(declare-element-component-handler
 (component-procedure-from-getters
  ray-from-arm-1
  angle-vertex
  ray-from-arm-2)
 angle?)
```

5.6 Graphics and Animation

The system integrates with Scheme's graphics system for the X Window System to display the figures for the users. The graphical viewer can include labels and highlight specific elements, as well as display animations representing the "wiggling" of the diagram. Implementations of core procedures of these components are shown in Listings 5.14 and 5.15.

Code Listing 5.14: Drawing Figures

```scheme
(define (draw-figure figure canvas)
  (set-coordinates-for-figure figure canvas)
  (clear-canvas canvas)
  (for-each
   (lambda (element)
     (canvas-set-color canvas (element-color element))
     ((draw-element element) canvas))
   (all-figure-elements figure))
  (for-each
   (lambda (element)
     (canvas-set-color canvas (element-color element))
     ((draw-label element) canvas))
   (all-figure-elements figure))
  (graphics-flush (canvas-g canvas)))
```

To animate a figure, constructions can call `animate` with a procedure f that takes an argument in $[0, 1]$. When the animation is run, the system will use fluid variables to iteratively wiggle each successive random choice through its range of $[0, 1]$. `animate-range` provides an example where a user can specify a range to wiggle over.

Code Listing 5.15: Animation

```scheme
(define (animate f)
  (let ((my-index *next-animation-index*))
    (set! *next-animation-index* (+ *next-animation-index* 1))
    (f (cond ((< *animating-index* my-index) 0)
             ((= *animating-index* my-index) *animation-value*)
             ((> *animating-index* my-index) 1)))))

(define (animate-range min max)
  (animate (lambda (v)
             (+ min
                (* v (- max min))))))
```

Chapter 6

Perception Module

6.1 Overview

The perception module focuses on "seeing" figures in our mind's eye. Given analytic figures represented using structures of the imperative construction module, the perception module is concerned with finding and reporting interesting relationships seen in the figure. In a generate-and-test-like fashion, it is rather liberal in the observations it returns. The module attempts to omit completely obvious properties, but leaves the filtering of new discoveries to the learning module (discussed further in Chapter 8).

To explain the module, I will first describe the implementation of underlying relationship and observation structures before examining the full analyzer routine. I will conclude with a discussion of extensions to the module, including some attempted techniques used to extract auxillary relationships from figures.

6.2 Relationships

Relationships are the primary structure defining what constitutes an interesting observation in a figure. Relationships are represented as predicates over typed n-tuples and are checked against all such n-tuples found in the figure under analysis.

Code Listing 6.1: Relationships

```scheme
(define-record-type <relationship>
  (%make-relationship type arity predicate)
  relationship?
  (type relationship-type)
  (arity relationship-arity)
  (predicate relationship-predicate))

(define equal-length-relationship
  (%make-relationship 'equal-length 2 segment-equal-length?))

(define concurrent-relationship
  (%make-relationship 'concurrent 3 concurrent?))

(define concentric-relationship
  (%make-relationship 'concentric 4 concentric?))
...
```

Listing 6.1 displays some representative relationships. The relationship predicates can be arbitrary Scheme procedures and often use constructions and utilities from the underlying imperative system as seen with `concurrent?` (Listing 6.2) and `concurrent?` (Listing 6.3). Concurrent is checked over all 3-tuples of linear elements (lines, rays, segments) and Concentric is checked against all 4-tuples of points.

Code Listing 6.2: Concurrent Relationship

```scheme
(define (concurrent? l1 l2 l3)
  (let ((i-point (intersect-linear-elements-no-endpoints l1 l2)))
    (and i-point
         (on-element? i-point l3)
         (not (element-endpoint? i-point l3)))))
```

Code Listing 6.3: Concentric Relationship

```scheme
(define (concentric? p1 p2 p3 p4)
  (and (distinct? p1 p2 p3 p4)
       (let ((pb-1 (perpendicular-bisector
                    (make-segment p1 p2)))
             (pb-2 (perpendicular-bisector
                    (make-segment p2 p3)))
             (pb-3 (perpendicular-bisector
                    (make-segment p3 p4))))
         (concurrent? pb-1 pb-2 pb-3))))
```

6.2.1 What is Interesting?

The system currently checks for concurrent, parallel, and perpendicular linear elements, segments of equal length, supplementary and complementary angles, angles of equal measure, coincident and concentric points, and three concentric points with a fourth as its center. These relationships covered most of the basic observations used in our investigations, but further relationships can be easily added.

6.3 Observations

Observations (Listing 6.4) are structures used to report the analyzer's findings. They combine the relevant relationship structure with the actual element arguments from the figure that satisfy that relationship. Maintaining references to the actual figure elements allows helper procedures to print names or extract dependencies as needed.

Code Listing 6.4: Observations

```
(define-record-type <observation>
  (make-observation relationship args)
  observation?
  (relationship observation-relationship)
  (args observation-args))
```

An important question with observations is to determine when they are equivalent to one another, to avoid reporting redundant or uninteresting relationships. Listing 6.5

Code Listing 6.5: Equivalent Observations

```
(define (observations-eqivalent? obs1 obs2)
  (and (eq? (observation-relationship obs1)
            (observation-relationship obs2))
       (let ((rel-eqv-test
               (relationship-equivalence-predicate
                (observation-relationship obs1)))
              (args1 (observation-args obs1))
              (args2 (observation-args obs2)))
         (rel-eqv-test args1 args2))))
```

6.4 Analysis Procedure

Given these relationship and observation structures, Listing 6.6 presents the main analyzer routine in this module. After extracting various types of elements from the figure, it examines the relationships relevant for each set of elements and gathers all resulting observations.

Code Listing 6.6: Analyzer Routine

```
(define (analyze figure)
  (let* ((points (figure-points figure))
         (angles (figure-angles figure))
         (linear-elements (figure-linear-elements figure))
         (segments (figure-segments figure)))
    (append
     (extract-relationships points
                           (list concurrent-points-relationship
                                 concentric-relationship
                                 concentric-with-center-relationship))
     (extract-relationships segments
                            (list equal-length-relationship))
     (extract-relationships angles
                            (list equal-angle-relationship
                                  supplementary-angles-relationship
                                  complementary-angles-relationship))
     (extract-relationships linear-elements
                            (list parallel-relationship
                                  concurrent-relationship
                                  perpendicular-relationship)))))
```

The workhorses of `extract-relationships` and `report-n-wise` shown in Listing 6.7 generate the relevant n-tuples and report observations for those that satisfy the relationship under consideration. For these homogeneous cases, `all-n-tuples` returns all (unordered) subsets of size n as lists.

Code Listing 6.7: Extracting Relationships

```
(define (extract-relationship elements relationship)
  (map (lambda (tuple)
         (make-observation relationship tuple))
       (report-n-wise
        (relationship-arity relationship)
        (relationship-predicate relationship)
        elements)))
```

```scheme

(define (report-n-wise n predicate elements)
  (let ((tuples (all-n-tuples n elements)))
    (filter (nary-predicate n predicate) tuples)))
```

6.5 Extensions

The analysis routine was initially one large, arbitrarily complicated procedure in which individual checks were added. This reformulation to use relationships and observations has simplified the complexity.

In addition, further efforts described below explored extracting relationships for elements not explicitly specified in a figure, such as auxiliary segments between all pairs of points in the figure, treating all intersections as points, or extracting angles. These are areas for future work.

6.5.1 Auxiliary Segments

In some circumstances, the system can insert and consider segments between all pairs of points. Although this can sometimes produce interesting results, it can often lead to too many elements being considered. This option is off by default but could be extended and enabled in a self-exploration mode.

6.5.2 Extracting Angles

In addition, I briefly explored a system in which the construction module also maintains a graph-like representation of the connectedness and adjacencies in the figure. In addition to the complexity of determining which angles to keep, keeping track of "obvious" relationships between such extracted angles due to parallel lines, for instance, is quite a challenge.

Chapter 7

Declarative Geometry Constraint Solver

7.1 Overview

The third module is a declarative geometric constraint solver. Given a user-specified topology of a diagram and various constraints on segments and angles, this module solves the specification and if possible, instantiates a figure that satisfies the constraints.

The solver is implemented using propagators, uses new types of partial information about point regions and direction intervals, and focuses on emulating the mental process of wiggling constrained figures in the mind's eye. The physical nature of this process is captured by forming analogies between geometry diagrams and mechanical linkages of bars and joints.

After providing a brief overview of the mechanical analogies and quick background on the propagator system, I examine an example of the system solving a set of constraints for an under-constrained rectangle. Then, I describe the module implementation, starting with the new partial information representations and linkage constraints before explaining how mechanisms are assembled and solved. Finally, some limitations and extensions are discussed.

7.2 Mechanical Analogies

The geometry constraint solver: physical manipulation, simulation, and “wiggling”.

7.3 Propagator System

GJS / Radul Propagator System

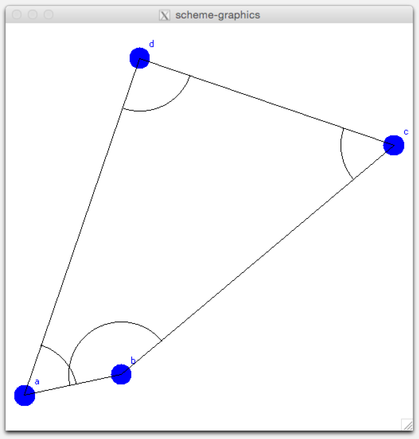
7.4 Example of Solving Geometric Constraints

Code Example 7.1: Rectangle Constraints Example

```
{images/rect-demo-2.png}
(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 7.2: Solved Constraints

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

(m:run-mechanism rect-demo-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)
```



7.5 Partial Information Structures

7.5.1 Regions

Propagating partial information across bars and joints yields a new region system: Regions include point sets of one or more possible points, an entire ray, or an entire arc. These rays and arcs are from an anchored bar with only one of direction or length specified, for instance.

7.5.2 Direction Intervals

Ranges of intervals. Full circle + invalid intervals. Adding and subtracting intervals of direction and thetas gets complicated at times.

Challenges with intersection, multiple segments. Eventually just return nothing is okay.

7.6 Bar and Joint Linkages

Bars have endpoints, directions and length. Joints have a vertex point and two directions. Currently, most joints are directioned and have max value of 180 degrees.

7.7 Propagator Constraints

System uses propagators to solve these mechanism constraints.

7.7.1 Basic Linkage Constraints

Direction, dx, dy, length, thetas. “Bars” + “Joints”

Code Listing 7.3: Basic Bar Constraints

```
(define (m:make-bar bar-id)
  (let ((bar-key (m:make-bar-name-key bar-id)))
    (let ((p1 (m:make-point (symbol bar-key '-p1)))
          (p2 (m:make-point (symbol bar-key '-p2))))
      (name! p1 (symbol bar-key '-p1))
      (name! p2 (symbol bar-key '-p2))
      (let ((v (m:make-vec bar-key)))
        (c:+ (m:point-x p1)
             (m:vec-dx v)
             (m:point-x p2))
        (c:+ (m:point-y p1)
             (m:vec-dy v)
             (m:point-y p2))
        (let ((bar (%m:make-bar p1 p2 v)))
          (m:p1->p2-bar-propagator p1 p2 bar)
          (m:p2->p1-bar-propagator p2 p1 bar)
          bar)))))
```

7.7.2 User-specified Constraints

Code Listing 7.4: User Constraints

```
(define-record-type <m:constraint>
  (m:make-constraint type args constraint-procedure)
  m:constraint?
  (type m:constraint-type)
  (args m:constraint-args)
  (constraint-procedure m:constraint-procedure))

(define (m:c-length-equal bar-id-1 bar-id-2)
  (m:make-constraint
   'm:c-length-equal
   (list bar-id-1 bar-id-2)
   (lambda (m)
     (let ((bar-1 (m:lookup m bar-id-1))
           (bar-2 (m:lookup m bar-id-2)))
       (c:id
        (m:bar-length bar-1)
        (m:bar-length bar-2))))))
```

Angle sum of polygon, or scan through polygon and ensure that the angles don't not match. Example is equilateral triangle, for instance... Could also observe always "60 degrees" as an interesting fact and put that in as a constraint. They're alebgraically quite similar, but my propagators currently don't perform symbolic algebra.

7.7.3 Building Mechanisms

The Mechanism in our declarative system is analogous to Figure, grouping elements. Also computes various caching and lookup tables to more easily access elements.

Code Listing 7.5: Establishing Topology

```
(define (m:establish-polygon-topology . point-names)
  (if (< (length point-names) 3)
      (error "Min polygon size: 3"))
  (let ((extended-point-names
         (append point-names
                 (list (car point-names) (cadr point-names)))))
    (let ((bars
           (map (lambda (p1-name p2-name)
                  (m:make-named-bar p1-name p2-name))
                point-names
                (cdr extended-point-names)))
          (joints
           (map (lambda (p1-name vertex-name p2-name)
                  (m:make-named-joint p1-name vertex-name p2-name))
                (cddr extended-point-names)
                (cdr extended-point-names)
                point-names)))
      (append bars joints
              (list (m:polygon-sum-slice
                     (map m:joint-name joints)))))))
```

Code Listing 7.6: Building Mechanisms

```scheme
(define (m:build-mechanism m)
  (m:identify-vertices m)
  (m:assemble-linkages (m:mechanism-bars m)
                       (m:mechanism-joints m))
  (m:apply-mechanism-constraints m)
  (m:apply-slices m))
```

7.8 Solving Mechanisms

Code Listing 7.7: Solving Mechanisms

```scheme
(define (m:solve-mechanism m)
  (m:initialize-solve)
  (let lp ()
    (run)
    (cond ((m:mechanism-contradictory? m)
           (m:draw-mechanism m c)
           #f)
          ((not (m:mechanism-fully-specified? m))
           (if (m:specify-something m)
               (lp)
               (error "Couldn't find anything to specify.")))
          (else 'mechanism-built))))
```

Given a wired diagram, process is repeatedly specifying values for elements

7.8.1 Backtracking

If it can't build a figure with a given set of specifications, it will first try some neighboring values, then backtrack and try a new value for the previous element. After a number of failed attempts, it will abort and claim that at this time, it is unable to build a diagram satisfying the constraints.

(This doesn't mean that it is impossible: Add analysis/info about what it can/-can't solve)

7.8.2 Interfacing with existing diagrams

Converts between figures and symbolic relationships.

7.9 Extensions

Future efforts involve an improved backtrack-search mechanism if constraints fail, and a system of initializing the diagram with content from an existing figure, kicking out and wiggling arbitrary premises, and seeing how the resulting diagram properties respond.

Chapter 8

Learning Module

8.1 Overview

As the final module, the learning module integrates information from the other modules and provides the primary, top-level interface for interacting with the system. It provides means for users to query its knowledge and provide investigations for the system to carry out. Through performing such investigations, the learning module formulates conjectures based on its observations and maintains a repository of information representing a student's understanding of geometry concepts.

I will first discuss the interface for interacting with the system. Then, after describing the structures for representing and storing definitions and conjectures, I demonstrate how the system module new terms and conjectures. Finally, I will explain the cyclic interaction between the imperative and declarative modules used to simplify definitions and discuss some limitations and future extensions.

8.2 Interface with Student

The learning module provides the primary interface by which users interact with the system. As such, it provides means by which users can both query the system to discover and use what it has known, as well as to teach the system information by suggesting investigations it should undertake.

Code Example 8.1: Using Definitions

```
(define (what-is term)
  (pprint (lookup term)))

(define (is-a? term obj)
  (let ((def (lookup term)))
    (if (unknown? def)
        `(,term unknown)
        ((definition-predicate def) obj))))

(define (show-me term)
  (let ((def (lookup term)))
    (if (unknown? def)
        `(,term unknown)
        (show-element ((definition-generator def))))))

(define (examine object)
  (show-element object)
  (let ((applicable-terms
         (filter (lambda (term)
                   (is-a? term object))
                 (all-known-terms))))
    applicable-terms))
```

8.2.1 Querying

A simple way of interacting with the learning module is to ask it for what it knows about various geometry concepts or terms. For definitions, the results provide the classification (that a rhombus is a parallelogram), and a set of minimal properties that differentiates that object from its classification. Further querying can present all known properties of the named object as well as theorems involving that term.

8.3 Representing Definitions and Conjectures

Discoveries are represented within a lattice of premises (discoveries about quadrilaterals $<$ discoveries about rhombuses $<$ discoveries about squares, but are separate from discoveries about circles or segments).

8.4 Learning new Terms and Conjectures

To learn a new definition, the system must be given the name of the term being learned as well as a procedure that will generate arbitrary instances of that definition. To converge to the correct definition, that random procedure should present a wide diversity of instances (i.e. the random-parallelogram procedure should produce all sorts of parallelograms, not just rectangles). However, reconciling mixed information about what constitutes a term could be an interesting extension.

Code Listing 8.2: Learning a new term

```scheme
(define (learn-term term object-generator)
  (let ((v (lookup term)))
    (if (not (eq? v 'unknown))
        (pprint `(already-known ,term))
        (let ((example (name-polygon (object-generator))))
          (let* ((base-terms (examine example))
                 (simple-base-terms (simplify-base-terms base-terms))
                 (base-definitions (map lookup base-terms))
                 (base-conjectures (flatten (map definition-conjectures
                                                 base-definitions)))
                 (fig (figure (with-dependency '<premise> example)))
                 (observations (analyze-figure fig))
                 (conjectures (map conjecture-from-observation observations))
                 (simplified-conjectures
                  (simplify-conjectures conjectures base-conjectures)))
            (pprint conjectures)
            (let ((new-def
                   (make-restrictions-definition
                    term
                    simple-base-terms
                    simplified-conjectures
                    object-generator)))
              (add-definition! *current-student* new-def)
              'done))))))
```

8.4.1 Predicates from Observations

Code Listing 8.3: Building Predicates for Definitions

```
(define (build-predicate-for-definition s def)
  (let ((classifications (definition-classifications def))
        (conjectures (definition-conjectures def)))
    (let ((classification-predicate
           (lambda (obj)
             (every
              (lambda (classification)
                (or ((definition-predicate (student-lookup s classification))
                     obj)
                    (begin (if *explain*
                               (pprint `(failed-classification
                                         ,classification)))
                           #f)))
              classifications))))
      (lambda args
        (and (apply classification-predicate args)
             (every (lambda (o) (satisfies-conjecture o args))
                    conjectures))))))
```

8.4.2 Performing Investigations

Investigations are similar to analyzing various figures above except that they have the intent of the analysis results being placed in the geometry knowledge repository. This separation also allows for dependence information about where properties were derived from.

Given the lattice structure of definitions, an interesting question when exploring new investigations is whether the given

8.5 Simplifying Definitions

To Simplify definitions, we interface with the constraint solver observations->figure to convert our observations back into a figure.

Code Listing 8.4: Simplifying Definitions

```scheme
(define (get-simple-definitions term)
  (let ((def (lookup term)))
    (if (unknown? def)
        (error "Unknown term" term))
    (let* ((object ((definition-generator def)))
           (observations
            (filter
             observation->constraint
             (all-observations
              (figure (name-polygon object))))))
      (map
       (lambda (obs-subset)
         (pprint obs-subset)
         (let* ((topology (topology-for-object object))
                (new-figure
                 (observations->figure topology obs-subset)))
           (if new-figure
               (let ((new-polygon
                      (polygon-from-figure new-figure)))
                 (pprint new-polygon)
                 (if (is-a? term new-polygon)
                     (list 'valid-definition
                           obs-subset)
                     (list 'invalid-definition
                           obs-subset)))
               (list 'unknown-definition
                     obs-subset))))
       (all-subsets observations)))))
```

Code Listing 8.5: Converting Observations to a Figure

```scheme
(define (observations->figure topology observations)
  (initialize-scheduler)
  (pprint (observations->constraints observations))
  (let ((m (apply
            m:mechanism
            (list
             topology
             (observations->constraints observations)))))
    (m:build-mechanism m)
    (if (not (m:solve-mechanism m))
        (begin
          (pp "Could not solve mechanism")
          #f)
        (let ((f (m:mechanism->figure m)))
          (pp "Solved!")
          (show-figure f)
          f))))
```

8.6 Discussion

Chapter 9

Related Work

The topics of automating geometric proofs and working with diagrams are areas of active research. Several examples of related work can be found in the proceedings of annual conferences such as *Automated Deduction in Geometry* [16] and *Diagrammatic Representation and Inference* [1]. In addition, two papers from the past year combine these concepts with a layer of computer vision interpretation of diagrams. Chen, Song, and Wang present a system that infers what theorems are being illustrated from images of diagrams [2], and a paper by Seo and Hajishirzi describes using textual descriptions of problems to improve recognition of their accompanying figures [13].

Further related work includes descriptions of the educational impacts of dynamic geometry approaches and some software to explore geometric diagrams and proofs. However, such software typically uses alternate approaches to automate such processes, and few focus on inductive reasoning.

9.1 Dynamic Geometry

From an education perspective, there are several texts that emphasize an investigative, conjecture-based approach to teaching such as *Discovering Geometry* by Michael Serra [14], the text I used to learn geometry and that served as an inspiration to this thesis project. Some researchers praise these investigative methods [10] while others question whether they appropriately encourages deductive reasoning skills [7].

9.2 Software

Some of these teaching methods include accompanying software such as Cabri Geometry [5] and the Geometer's Sketchpad [6] designed to enable students to explore constructions interactively. These programs occasionally provide scripting features, but have no proof-related automation.

A few more academic analogs of these programs introduce some proof features. For instance, GeoProof [9] integrates diagram construction with verified proofs using a number of symbolic methods carried out by the Coq Proof Assistant, and Geometry Explorer [15] uses a full-angle method of chasing angle relations to check assertions requested by the user. However, almost none of the software described simulates the exploratory, inductive investigation process used by students first discovering new conjectures.

The closest example is Geometer

9.3 Mechanical Analogies to Geometry Constraint Solving

Books about Mechanical Analogies

9.4 Automated Proof and Discovery

Although there are several papers that describe automated discovery or proof in geometry, most of these use alternate, more algebraic methods to prove theorems. These approaches include an area method [11], Wu's Method involving systems of polynomial equations [4], and a system based on Gröbner Bases [8]. Some papers discuss reasoning systems including the construction and application of a deductive database of geometric theorems [3]. However, all of these methods focused either on deductive reasoning or complex algebraic reformulations.

9.5 Other Resources

GJS / Radul Propagators, ghelper code, etc.

Chapter 10

Conclusion

10.1 Overview

The system presented in this thesis provides a good foundation for building, exploring, and analyzing geometry diagrams.

10.2 Limitations

Despite these successes, there are some limitations and

10.2.1 Numerical Accuracy

A big issue with any numerical analysis is dealing with numerical accuracy. Use of `close-enuf?`

10.2.2 Negative Relations and Definitions

Negative properties are hard to determine: “Scalene” means not isoceles or not equilateral, for instance.

10.2.3 Generality of Theorems

Similarly, some theorems involve more complicated statements “the shortest distance from a point to a line is along the perpendicular to the line”. The current system is primarily focused around theorems arising from simple premises.

10.3 Extensions

10.3.1 Deductive Proof Systems

Possible extensions include integrating with existing automated proof systems (Coq, etc.)

10.3.2 Learning Constructions

Also: learning construction procedures from the declarative constraint solver’s solution.

Appendix A

Code Listings

This chapter contains code listings for the thesis presented in this thesis. Code is implemented using MIT Scheme 9.2

A.1 Repository Structure

A.2 External Dependencies

GJS Propagator system, some scmutils, ghelper, eq-properties.

Listings

Listing A.1: load.scm

```scheme
;;; load.scm -- Load the system

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Utilities ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (reset)
  (ignore-errors (lambda () (close)))
  (ge (make-top-level-environment))
  (load "load"))

(define (load-module subdirectory)
  (let ((cur-pwd (pwd)))
    (cd subdirectory)
    (load "load")
    (cd cur-pwd)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Load Modules ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(for-each (lambda (m) (load-module m))
          '("lib"
            "core"
            "figure"
            "graphics"
            "manipulate"
            "perception"
            "learning"
            "content"))
(load "main")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Initialize ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(set! *random-state* (fasload "a-random-state"))
(initialize-scheduler)
(initialize-student)

'done-loading
```

Listing A.2: main.scm

```scheme
(define (i-t-figure)
  (let-geo* (((t (a b c)) (random-isoceles-triangle)))
    (figure t)))


(define (midpoint-figure)
  (let-geo* (((s (a b)) (random-segment))
             (m (segment-midpoint s)))
    (figure s m)))

(define (random-rhombus-figure)
  (let-geo* (((r (a b c d)) (random-rhombus)))
    (figure r)))

;;; Other Examples:

(define (debug-figure)
  (let-geo* (((r (a b c d)) (random-parallelogram))
             (m1 (midpoint a b))
             (m2 (midpoint c d)))
    (figure r m1 m2 (make-segment m1 m2))))

(define (demo-figure)
  (let-geo* (((t (a b c)) (random-isoceles-triangle))
             (d (midpoint a b))
             (e (midpoint a c))
             (f (midpoint b c))

             (l1 (perpendicular (line-from-points a b) d))
             (l2 (perpendicular (line-from-points a c) e))
             (l3 (perpendicular (line-from-points b c) f))

             (i1 (intersect-lines l1 l2))
             (i2 (intersect-lines l1 l3))

             (cir (circle-from-points i1 a)))

    (figure
     (make-segment a b)
     (make-segment b c)
     (make-segment a c)
     a b c l1 l2 l3 cir
     i1 i2)))

(define (circle-line-intersect-test)
  (let-geo* ((cir (random-circle))
             ((rad (a b)) (random-circle-radius cir))
             (p (random-point-on-segment rad))
             (l (random-line-through-point p))
             (cd (intersect-circle-line cir l))
             (c (car cd))
             (d (cadr cd)))
    (figure cir rad p l c d)))

(define (circle-test)
  (let-geo* ((a (random-point))
             (b (random-point))
             (d (distance a b))
             (r (rand-range
                 (* d 0.5)
                 (* d 1)))
             (c1 (make-circle a r))
             (c2 (make-circle b r))
             (cd (intersect-circles c1 c2))
             (c (car cd))
             (d (cadr cd)))
```

```
   (figure (polygon-from-points a c b d))))

(define (line-test)
  (let-geo* ((a (random-point))
             (b (random-point))
             (c (random-point))
             (d (random-point))
             (l1 (line-from-points a b))
             (l2 (line-from-points c d))
             (e (intersect-lines l1 l2))
             (f (random-point-on-line l1))
             (cir (circle-from-points e f)))
    (figure a b c d l1 l2 e f cir)))

(define (incircle-circumcircle)
  (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)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Run commands

(define current-figure demo-figure)

(define c
  (if (environment-bound? (the-environment) 'c)
      c
      (canvas)))

(define (close)
  (ignore-errors (lambda () (graphics-close (canvas-g c)))))

(define *num-inner-loop* 5)
(define *num-outer-loop* 5)


(define (run-figure current-figure-proc)
  (let ((analysis-data (make-analysis-collector)))
    (run-animation
     (lambda ()
       (let ((current-figure (current-figure-proc)))
         (draw-figure current-figure c)
         (let ((analysis-results (analyze-figure current-figure)))
           (save-results (print analysis-results) analysis-data))
         )))
    (display "--- Results ---\n")
    (print-analysis-results analysis-data)))

(define interesting-figures
  (list
   debug-figure
   parallel-lines-converse
   perpendicular-bisector-equidistant
   perpendicular-bisector-converse
   demo-figure
   linear-pair
   vertical-angles
   corresponding-angles
   cyclic-quadrilateral))

(define (r)
  (for-each (lambda (figure)
              (run-figure figure))
            interesting-figures)
  'done)

;(r)
```

Listing A.3: figure/load.scm

```
;;; load.scm -- Load figure
(for-each (lambda (f) (load f))
          '("core"
            "metadata"
            "line"
            "direction"
            "direction-interval"
            "vec"
            "measurements"
            "angle"
            "bounds"
            "circle"
            "point"
            "constructions"
            "intersections"
            "figure"
            "math-utils"
            "polygon"
            "dependencies"
            "randomness"
            "transforms"))
```

Listing A.4: figure/core.scm

```
;;; core.scm --- Core definitions used throughout the figure elements

;;; Commentary:

;; Ideas:
;; - Some gemeric handlers used in figure elements

;; Future:
;; - figure-element?, e.g.

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;; Element Component ;;;;;;;;;;;;;;;;;;;;;;;;;

(define element-component
  (make-generic-operation
   2 'element-component
   (lambda (el i)
     (error "No component procedure for element" el))))

(define (component-procedure-from-getters . getters)
  (let ((num-getters (length getters)))
    (lambda (el i)
      (if (not (<= 0 i (- num-getters 1)))
          (error "Index out of range for component procedure: " i))
      ((list-ref getters i)
       el))))

(define (declare-element-component-handler handler type)
  (defhandler element-component handler type number?))

(declare-element-component-handler list-ref list?)

#|
Example Usage:

(declare-element-component-handler
 (component-procedure-from-getters car cdr)
 pair?)

(declare-element-component-handler vector-ref vector?)

(element-component '(3 . 4 ) 1)
;Value: 4

(element-component #(1 2 3) 2)
;Value: 3
|#
```

Listing A.5: figure/line.scm

```
;;; line.scm --- Line

;;; Commentary:

;; Ideas:
;; - Linear Elements: Segments, Lines, Rays
;; - All have direction
;; - Conversions to directions, extending.
;; - Lines are point + direction, but hard to access point
;; - Means to override dependencies for random segments

;; Future:
;; - Simplify direction requirements
;; - Improve some predicates, more tests
;; - Fill out more dependency information

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Segments ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <segment>
  (%segment p1 p2)
  segment?
  (p1 segment-endpoint-1)
  (p2 segment-endpoint-2))

(define (set-segment-dependency! segment dependency)
  (set-dependency! segment dependency)
  (set-dependency!
   (segment-endpoint-1 segment)
   `(segment-endpoint-1 segment))
  (set-dependency!
   (segment-endpoint-2 segment)
   `(segment-endpoint-2 segment)))

(defhandler print
  element-name
  segment?)

;;; Alternate, helper constructors

(define (make-segment p1 p2)
  (let ((seg (%segment p1 p2)))
    (with-dependency
     `(segment ,p1 ,p2)
     seg)))

(define (make-auxiliary-segment p1 p2)
  (with-dependency
   `(aux-segment ,p1 ,p2)
   (make-segment p1 p2)))

```

```
(declare-element-component-handler
 (component-procedure-from-getters segment-endpoint-1
                                   segment-endpoint-2)
 segment?)

(defhandler generic-element-name
  (lambda (seg)
    `(segment ,(element-name (segment-endpoint-1 seg))
              ,(element-name (segment-endpoint-2 seg))))
  segment?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Lines ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <line>
  (%make-line point dir)
  line?
  (point line-point) ;; Point on the line
  (dir line-direction))

(defhandler print
  element-name
  line?)

(define make-line %make-line)

(define (line-from-points p1 p2)
  (make-line p1 (direction-from-points p1 p2)))

(define (line-from-point-direction p dir)
  (make-line p dir))

;;; TODO, use for equality tests?
(define (line-offset line)
  (let ((direction (direction-from-points p1 p2))
        (x1 (point-x p1))
        (y1 (point-y p1))
        (x2 (point-x p2))
        (y2 (point-y p2)))
    (let ((offset (/ (- (* x2 y1)
                        (* y2 x1))
                     (distance p1 p2))))
      (%make-line direction offset))))

;;; TODO: Figure out dependencies for these
(define (two-points-on-line line)
  (let ((point-1 (line-point line)))
   (let ((point-2 (add-to-point
                   point-1
                   (unit-vec-from-direction (line-direction line)))))
     (list point-1 point-2))))

(define (line-p1 line)
  (car (two-points-on-line line)))

(define (line-p2 line)
  (cadr (two-points-on-line line)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Rays ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <ray>
  (make-ray initial-point direction)
  ray?
  (initial-point ray-endpoint)
  (direction ray-direction))

(define (ray-from-point-direction p dir)
  (make-ray p dir))

(define (ray-from-points endpoint p1)
  (make-ray endpoint (direction-from-points endpoint p1)))

(define (shorten-ray-from-point r p)
  (if (not (on-ray? p r))
      (error "Can only shorten rays from points on the ray"))
  (ray-from-point-direction p (ray-direction r)))

;;;;;;;;;;;;;;;;;;;; Constructors from angles ;;;;;;;;;;;;;;;;;;;;

(define (ray-from-arm-1 a)
  (let ((v (angle-vertex a))
        (dir (angle-arm-1 a)))
    (make-ray v dir)))

(define (ray-from-arm-2 a)
  (ray-from-arm-1 (reverse-angle a)))

(define (line-from-arm-1 a)
  (ray->line (ray-from-arm-1 a)))

(define (line-from-arm-2 a)
  (ray->line (ray-from-arm-2 a)))

;;;;;;;;;;;;;;;;;;;;;;;;;;; Transforms ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define flip (make-generic-operation 1 'flip))

(define (flip-line line)
  (make-line
   (line-point line)
   (reverse-direction (line-direction line))))
(defhandler flip flip-line line?)

(define (flip-segment s)
  (make-segment (segment-endpoint-2 s) (segment-endpoint-1 s)))
(defhandler flip flip-segment segment?)

(define (reverse-ray r)
  (make-ray (ray-endpoint r)
            (reverse-direction (ray-direction r))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Operations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (segment-length seg)
  (distance (segment-endpoint-1 seg)
            (segment-endpoint-2 seg)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (linear-element? x)
  (or (line? x)
      (segment? x)
      (ray? x)))

(define (parallel? a b)
  (direction-parallel? (->direction a)
                       (->direction b)))

(define (perpendicular? a b)
  (direction-perpendicular? (->direction a)
                            (->direction b)))

(define (segment-equal? s1 s2)
  (and
   (point-equal? (segment-endpoint-1 s1)
                 (segment-endpoint-1 s2))
   (point-equal? (segment-endpoint-2 s1)
                 (segment-endpoint-2 s2))))

(define (segment-equal-ignore-direction? s1 s2)
  (or (segment-equal? s1 s2)
      (segment-equal? s1 (flip-segment s2))))

(define (segment-equal-length? seg-1 seg-2)
  (close-enuf? (segment-length seg-1)
               (segment-length seg-2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Conversions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Ray shares point p1
(define (segment->ray segment)
  (make-ray (segment-endpoint-1 segment)
            (direction-from-points
             (segment-endpoint-1 segment)
             (segment-endpoint-2 segment))))

(define (ray->line ray)
  (make-line (ray-endpoint ray)
             (ray-direction ray)))

(define (segment->line segment)
  (ray->line (segment->ray segment)))

(define (line->direction l)
  (line-direction l))

(define (ray->direction r)
  (ray-direction r))

(define (segment->direction s)
  (direction-from-points
   (segment-endpoint-1 s)
   (segment-endpoint-2 s)))

(define (segment->vec s)
  (sub-points
   (segment-endpoint-2 s)
   (segment-endpoint-1 s)))

(define ->direction (make-generic-operation 1 '->direction))
(defhandler ->direction line->direction line?)
(defhandler ->direction ray->direction ray?)
(defhandler ->direction segment->direction segment?)

(define ->line (make-generic-operation 1 '->line))
(defhandler ->line identity line?)
(defhandler ->line segment->line segment?)
(defhandler ->line ray->line ray?)
```

Listing A.6: figure/direction.scm

```
;;; direction.scm --- Low-level direction structure

;;; Commentary:

;; A Direction is equivalent to a unit vector pointing in some direction.

;; Ideas:
;; - Ensures range [0, 2pi]

;; Future:
;; - Could generalize to dx, dy or theta

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;; Direction Structure ;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <direction>
  (%direction theta)
  direction?
  (theta direction-theta))

(define (make-direction theta)
  (%direction (fix-angle-0-2pi theta)))

(define (print-direction dir)
  `(direction ,(direction-theta dir)))
(defhandler print print-direction direction?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Arithemtic ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (add-to-direction dir radians)
  (make-direction (+ (direction-theta dir)
                     radians)))
;;; D2 - D1
(define (subtract-directions d2 d1)
  (if (direction-equal? d1 d2)
      0
      (fix-angle-0-2pi (- (direction-theta d2)
                          (direction-theta d1)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Operations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; CCW
(define (rotate-direction-90 dir)
  (add-to-direction dir (/ pi 2)))

(define (reverse-direction dir)
  (add-to-direction dir pi))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (direction-equal? d1 d2)
  (or (close-enuf? (direction-theta d1)
                   (direction-theta d2))
      (close-enuf? (direction-theta (reverse-direction d1))
                   (direction-theta (reverse-direction d2)))))

(define (direction-opposite? d1 d2)
  (close-enuf? (direction-theta d1)
               (direction-theta (reverse-direction d2))))

(define (direction-perpendicular? d1 d2)
  (let ((difference (subtract-directions d1 d2)))
    (or (close-enuf? difference (/ pi 2))
        (close-enuf? difference (* 3 (/ pi 2))))))

(define (direction-parallel? d1 d2)
  (or (direction-equal? d1 d2)
      (direction-opposite? d1 d2)))
```

Listing A.7: figure/vec.scm

```
;;; vec.scm --- Low-level vector structures

;;; Commentary:

;; Ideas:
;; - Simplifies lots of computation, cartesian coordiates
;; - Currently 2D, could extend

;; Future:
;; - Could generalize to allow for polar vs. cartesian vectors

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;; Vector Structure ;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <vec>
  (make-vec dx dy)
  vec?
  (dx vec-x)
  (dy vec-y))

;;; Transformations of Vectors
(define (vec-magnitude v)
  (let ((dx (vec-x v))
        (dy (vec-y v)))
    (sqrt (+ (square dx) (square dy)))))

;;;;;;;;;;;;;;;;;;;;; Alternate Constructors ;;;;;;;;;;;;;;;;;;;;;

(define (unit-vec-from-direction direction)
  (let ((theta (direction-theta direction)))
   (make-vec (cos theta) (sin theta))))

(define (vec-from-direction-distance direction distance)
  (scale-vec (unit-vec-from-direction direction) distance))

;;;;;;;;;;;;;;;;;;;;;;;;;;; Conversions ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (vec->direction v)
  (let ((dx (vec-x v))
        (dy (vec-y v)))
    (make-direction (atan dy dx))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Operations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Returns new vecs

(define (rotate-vec v radians)
  (let ((dx (vec-x v))
        (dy (vec-y v))
        (c (cos radians))
        (s (sin radians)))
    (make-vec (+ (* c dx) (- (* s dy)))
              (+ (* s dx) (* c dy)))))

(define (scale-vec v c)
  (let ((dx (vec-x v))
        (dy (vec-y v)))
    (make-vec (* c dx) (* c dy))))

(define (scale-vec-to-dist v dist)
```

```scheme
  (scale-vec (unit-vec v) dist))

(define (reverse-vec v)
  (make-vec (- (vec-x v))
            (- (vec-y v))))

(define (rotate-vec-90 v)
  (let ((dx (vec-x v))
        (dy (vec-y v)))
    (make-vec (- dy) dx)))

(define (unit-vec v)
  (scale-vec v (/ (vec-magnitude v))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (vec-equal? v1 v2)
  (and (close-enuf? (vec-x v1)  (vec-x v2))
       (close-enuf? (vec-y v1)  (vec-y v2))))

(define (vec-direction-equal? v1 v2)
  (direction-equal?
   (vec->direction v1)
   (vec->direction v2)))

(define (vec-perpendicular? v1 v2)
  (close-enuf?
   (* (vec-x v1) (vec-x v2))
   (* (vec-y v1) (vec-y (reverse-vec v2)))))
```

Listing A.8: figure/measurements.scm

```scheme
;;; measurements.scm

;;; Commentary:

;; Ideas:
;; - Measurements primarily for analysis
;; - Occasionally used for easily duplicating angles or segments

;; Future:
;; - Arc Measure

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Distance ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (distance p1 p2)
  (sqrt (+ (square (- (point-x p1)
                      (point-x p2)))
           (square (- (point-y p1)
                      (point-y p2))))))

;;; Sign of distance is positive if the point is to the left of
;;; the line direction and negative if to the right.
(define (signed-distance-to-line point line)
  (let ((p1 (line-p1 line))
        (p2 (line-p2 line)))
    (let ((x0 (point-x point))
          (y0 (point-y point))
          (x1 (point-x p1))
          (y1 (point-y p1))
          (x2 (point-x p2))
          (y2 (point-y p2)))
      (/ (+ (- (* x0 (- y2 y1)))
            (* y0 (- x2 x1))
            (- (* x2 y1))
            (* y2 x1))
         (* 1.0
            (sqrt (+ (square (- y2 y1))
                     (square (- x2 x1)))))))))

(define (distance-to-line point line)
  (abs (signed-distance-to-line point line)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Angles ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (angle-measure a)
  (let* ((d1 (angle-arm-1 a))
         (d2 (angle-arm-2 a)))
    (subtract-directions d1 d2)))

;;;;;;;;;;;;;;;;;;;;;;;; Measured Elements ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (measured-point-on-ray r dist)
  (let* ((p1 (ray-p1 r))
         (p2 (ray-p2 r))
         (v (sub-points p1 p2))
         (scaled-v (scale-vec-to-dist v dist)))
    (add-to-point p1 scaled-v)))

(define (measured-angle-ccw p1 vertex radians)
  (let* ((v1 (sub-points p1 vertex))
         (v-rotated (rotate-vec v (- radians))))
    (angle v1 vertex v-rotated)))

(define (measured-angle-cw p1 vertex radians)
  (reverse-angle (measured-angle-ccw p1 vertex (- radians))))
```

Listing A.9: figure/angle.scm

```scheme
;;; angle.scm --- Angles

;;; Commentary:

;; Ideas:
;; - Initially three points, now vertex + two directions
;; - Counter-clockwise orientation
;; - Uniquely determining from elements forces directions
;; - naming of "arms" vs. "directions"

;; Future Ideas:
;; - Automatically discover angles from diagrams (e.g. from a pile of
;;     points and segments)
;; - Angle intersections

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Angles ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; dir1 and dir2 are directions of the angle arms
;;; The angle sweeps from dir2 *counter clockwise* to dir1
(define-record-type <angle>
  (make-angle dir1 vertex dir2)
  angle?
  (dir1 angle-arm-1)
  (vertex angle-vertex)
  (dir2 angle-arm-2))

(declare-element-component-handler
 (component-procedure-from-getters
  ray-from-arm-1
  angle-vertex
  ray-from-arm-2)
 angle?)

(defhandler generic-element-name
  (lambda (angle)
    `(angle ,(element-name (angle-vertex angle))))
  angle?)

(defhandler print
  element-name
  angle?)

;;;;;;;;;;;;;;;;;;;; Transformations on Angles ;;;;;;;;;;;;;;;;;;;;;

(define (reverse-angle a)
  (let ((d1 (angle-arm-1 a))
        (v (angle-vertex a))
        (d2 (angle-arm-2 a)))
    (make-angle d2 v d1)))

(define (smallest-angle a)
  (if (> (angle-measure a) pi)
      (reverse-angle a)
      a))

;;;;;;;;;;;;;;;;;;;;;; Alternate Constructors ;;;;;;;;;;;;;;;;;;;;;;

(define (angle-from-points p1 vertex p2)
  (let ((arm1 (direction-from-points vertex p1))
        (arm2 (direction-from-points vertex p2)))
    (make-angle arm1 vertex arm2)))

(define (smallest-angle-from-points p1 vertex p2)
  (smallest-angle (angle-from-points p1 vertex p2)))

;;;;;;;;;;;;;;;;;; Angle from pairs of elements ;;;;;;;;;;;;;;;;;;

(define angle-from (make-generic-operation 2 'angle-from))

(define (angle-from-lines l1 l2)
  (let ((d1 (line->direction l1))
        (d2 (line->direction l2))
        (p (intersect-lines l1 l2)))
    (make-angle d1 p d2)))
(defhandler angle-from angle-from-lines line? line?)

(define (angle-from-line-ray l r)
  (let ((vertex (ray-endpoint r)))
    (assert (on-line? vertex l)
            "Angle-from-line-ray: Vertex of ray not on line")
    (let ((d1 (line->direction l))
          (d2 (ray->direction r)))
      (make-angle d1 vertex d2))))
(defhandler angle-from angle-from-line-ray line? ray?)

(define (angle-from-ray-line r l)
  (reverse-angle (angle-from-line-ray l r)))
(defhandler angle-from angle-from-ray-line ray? line?)

(define (angle-from-segment-segment s1 s2)
  (define (angle-from-segment-internal s1 s2)
    (let ((vertex (segment-endpoint-1 s1)))
      (let ((d1 (segment->direction s1))
            (d2 (segment->direction s2)))
        (make-angle d1 vertex d2))))
  (cond ((point-equal? (segment-endpoint-1 s1)
                       (segment-endpoint-1 s2))
         (angle-from-segment-internal s1 s2))
        ((point-equal? (segment-endpoint-2 s1)
                       (segment-endpoint-1 s2))
         (angle-from-segment-internal (flip s1) s2))
        ((point-equal? (segment-endpoint-1 s1)
                       (segment-endpoint-2 s2))
         (angle-from-segment-internal s1 (flip s2)))
        ((point-equal? (segment-endpoint-2 s1)
                       (segment-endpoint-2 s2))
         (angle-from-segment-internal  (flip s1) (flip s2)))
        (else (error "Angle-from-segment-segment must share vertex"))))
(defhandler angle-from angle-from-segment-segment segment? segment?)

(define (smallest-angle-from a b)
  (smallest-angle (angle-from a b)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; Predicates on Angles ;;;;;;;;;;;;;;;;;;;;;;;;

(define (angle-measure-equal? a1 a2)
  (close-enuf? (angle-measure a1)
               (angle-measure a2)))

(define (supplementary-angles? a1 a2)
  (close-enuf? (+ (angle-measure a1)
                  (angle-measure a2))
               pi))

(define (complementary-angles? a1 a2)
  (close-enuf? (+ (angle-measure a1)
                  (angle-measure a2))
               (/ pi 2.0)))

;;;;;;;;;;;;;;;;;;;;;;;;;;; Definitions ;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; TODO? Consider learning or putiting elsewhere
(define (linear-pair? a1 a2)
  (define (linear-pair-internal? a1 a2)
    (and (point-equal? (angle-vertex a1)
                       (angle-vertex a2))
         (direction-equal? (angle-arm-2 a1)
                           (angle-arm-1 a2))
         (direction-opposite? (angle-arm-1 a1)
                              (angle-arm-2 a2))))
  (or (linear-pair-internal? a1 a2)
      (linear-pair-internal? a2 a1)))

(define (vertical-angles? a1 a2)
  (and (point-equal? (angle-vertex a1)
                     (angle-vertex a2))
       (direction-opposite? (angle-arm-1 a1)
                            (angle-arm-1 a2))
       (direction-opposite? (angle-arm-2 a1)
                            (angle-arm-2 a2))))
```

Listing A.10: figure/bounds.scm

```
;;; bounds.scm --- Graphics Bounds

;;; Commentary:

;; Ideas:
;; - Logic to extend segments to graphics bounds so they can be drawn.

;; Future:
;; - Separate logical bounds of figures from graphics bounds
;; - Combine logic for line and ray (one vs. two directions)
;; - Should these be a part of "figure" vs. "graphics"
;; - Remapping of entire figures to different canvas dimensions

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Bounds ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <bounds>
  (make-bounds x-interval y-interval)
  bounds?
  (x-interval bounds-x-interval)
  (y-interval bounds-y-interval))

(define (bounds-xmin b) (interval-low (bounds-x-interval b)))
(define (bounds-xmax b) (interval-high (bounds-x-interval b)))
(define (bounds-ymin b) (interval-low (bounds-y-interval b)))
(define (bounds-ymax b) (interval-high (bounds-y-interval b)))

(define (print-bounds b)
  `(bounds ,(bounds-xmin b)
           ,(bounds-xmax b)
           ,(bounds-ymin b)
           ,(bounds-ymax b)))
(defhandler print print-bounds bounds?)

;;;;;;;;;;;;;;;;;;;;;;;; Bounds Constants ;;;;;;;;;;;;;;;;;;;;;;;;

;;; Max bounds of the graphics window

(define *g-min-x* -2)
(define *g-max-x*  2)
(define *g-min-y* -2)
(define *g-max-y*  2)

;;;;;;;;;;;;;;; Conversion to segments for Graphics ;;;;;;;;;;;;;;;

(define (extend-to-max-segment p1 p2)
  (let ((x1 (point-x p1))
        (y1 (point-y p1))
        (x2 (point-x p2))
        (y2 (point-y p2)))
    (let ((dx (- x2 x1))
          (dy (- y2 y1)))
      (cond
       ((= 0 dx) (make-segment
                  (make-point x1 *g-min-y*)
                  (make-point x1 *g-max-y*)))
       ((= 0 dy) (make-segment
                  (make-point *g-min-x* y1)
                  (make-point *g-min-y* y1)))
       (else
        (let ((t-xmin (/ (- *g-min-x* x1) dx))
              (t-xmax (/ (- *g-max-x* x1) dx))
              (t-ymin (/ (- *g-min-y* y1) dy))
              (t-ymax (/ (- *g-max-y* y1) dy)))
          (let* ((sorted (sort (list t-xmin t-xmax t-ymin t-ymax) <))
                 (min-t (cadr sorted))
                 (max-t (caddr sorted))
```

```
               (min-x (+ x1 (* min-t dx)))
               (min-y (+ y1 (* min-t dy)))
               (max-x (+ x1 (* max-t dx)))
               (max-y (+ y1 (* max-t dy))))
          (make-segment (make-point min-x min-y)
                        (make-point max-x max-y)))))))))

(define (ray-extend-to-max-segment p1 p2)
  (let ((x1 (point-x p1))
        (y1 (point-y p1))
        (x2 (point-x p2))
        (y2 (point-y p2)))
    (let ((dx (- x2 x1))
          (dy (- y2 y1)))
      (cond
       ((= 0 dx) (make-segment
                  (make-point x1 *g-min-y*)
                  (make-point x1 *g-max-y*)))
       ((= 0 dy) (make-segment
                  (make-point *g-min-x* y1)
                  (make-point *g-min-y* y1)))
       (else
        (let ((t-xmin (/ (- *g-min-x* x1) dx))
              (t-xmax (/ (- *g-max-x* x1) dx))
              (t-ymin (/ (- *g-min-y* y1) dy))
              (t-ymax (/ (- *g-max-y* y1) dy)))
          (let* ((sorted (sort (list t-xmin t-xmax t-ymin t-ymax) <))
               (min-t (cadr sorted))
               (max-t (caddr sorted))
               (min-x (+ x1 (* min-t dx)))
               (min-y (+ y1 (* min-t dy)))
               (max-x (+ x1 (* max-t dx)))
               (max-y (+ y1 (* max-t dy))))
            (make-segment p1
                        (make-point max-x max-y)))))))))

;;;;;;;;;;;;;;;;;;;;;;;;;; Rescale Figure ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define empty-bounds (make-bounds (make-interval 0 0)
                                  (make-interval 0 0)))

(define (extend-interval i new-value)
  (let ((low (interval-low i))
        (high (interval-high i)))
    (make-interval (min low new-value)
                   (max high new-value))))

(define (interval-length i)
  (- (interval-high i)
     (interval-low i)))

(define (extend-bounds bounds point)
  (let ((px (point-x point))
        (py (point-y point)))
    (make-bounds
     (extend-interval (bounds-x-interval bounds)
                      px)
     (extend-interval (bounds-y-interval bounds)
                      py))))

(define (bounds-width bounds)
  (interval-length (bounds-x-interval bounds)))

(define (bounds-height bounds)
  (interval-length (bounds-y-interval bounds)))

(define (bounds->square bounds)
  (let ((new-side-length
         (max (bounds-width bounds)
              (bounds-height bounds))))
    (recenter-bounds bounds
                     new-side-length
                     new-side-length)))

(define (recenter-interval i new-length)
  (let* ((min (interval-low i))
         (max (interval-high i))
         (old-half-length (/ (- max min) 2))
         (new-half-length (/ new-length 2)))
    (make-interval (- (+ min old-half-length) new-half-length)
                   (+ (- max old-half-length) new-half-length))))

(define (recenter-bounds bounds new-width new-height)
  (make-bounds
   (recenter-interval (bounds-x-interval bounds) new-width)
   (recenter-interval (bounds-y-interval bounds) new-height)))

(define (scale-bounds bounds scale-factor)
  (recenter-bounds
   bounds
   (* (bounds-width bounds) scale-factor)
   (* (bounds-height bounds) scale-factor)))

(define (extract-bounds figure)
  (let ((all-points (figure-points figure)))
    (let lp ((bounds empty-bounds)
             (points all-points))
      (if (null? points)
          bounds
          (extend-bounds (lp bounds (cdr points))
                         (car points))))))
```

Listing A.11: figure/circle.scm

```
;;; circle.scm --- Circles

;;; Commentary:

```

```
;; Ideas:
;; - Currently rather limited support for circles

;; Future:
;; - Arcs, tangents, etc.

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;; Circle structure ;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <circle>
  (make-circle center radius)
  circle?
  (center circle-center)
  (radius circle-radius))

;;;;;;;;;;;;;;;;;;;;; Alternate Constructions ;;;;;;;;;;;;;;;;;;;;;;

(define (circle-from-points center radius-point)
  (make-circle center
          (distance center radius-point)))

;;;;;;;;;;;;;;;;;;;;;;;;; Points on circle ;;;;;;;;;;;;;;;;;;;;;;;;

(define (point-on-circle-in-direction cir dir)
  (let ((center (circle-center cir))
        (radius (circle-radius cir)))
    (add-to-point
     center
     (vec-from-direction-distance
      dir radius))))
```

Listing A.12: figure/point.scm

```
;;; point.scm --- Point

;;; Commentary:

;; Ideas:
;; - Points are the basis for most elements

;; Future:
;; - Transform to different canvases
;; - Have points know what elements they are on.

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;; Point Structure ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <point>
  (make-point x y)
  point?
  (x point-x)
  (y point-y))

(define (print-point p)
  `(point ,(point-x p) ,(point-y p)))

(defhandler print
  print-point point?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (point-equal? p1 p2)
  (and (close-enuf? (point-x p1)
                    (point-x p2))
       (close-enuf? (point-y p1)
                    (point-y p2))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Operations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; P2 - P1
(define (sub-points p2 p1)
  (let ((x1 (point-x p1))
        (x2 (point-x p2))
        (y2 (point-y p2))
        (y1 (point-y p1)))
    (make-vec (- x2 x1)
              (- y2 y1))))

;;; Direction from p1 to p2
(define (direction-from-points p1 p2)
  (vec->direction (sub-points p2 p1)))

(define (add-to-point p vec)
  (let ((x (point-x p))
        (y (point-y p))
        (dx (vec-x vec))
        (dy (vec-y vec)))
    (make-point (+ x dx)
                (+ y dy))))
```

Listing A.13: figure/constructions.scm

```
;;; constructions.scm --- Constructions

;;; Commentary:

;; Ideas:
;; - Various logical constructions that can be peformed on elements
;; - Some higher-level constructions...

;; Future:
;; - More constructions?
;; - Separation between compass/straightedge and compound?
;; - Experiment with higher-level vs. learned constructions

;;; Code:
```

```

;;;;;;;;;;;;;;;;;;;;;; Segment Constructions ;;;;;;;;;;;;;;;;;;;;;;;

(define (midpoint p1 p2)
  (let ((newpoint
         (make-point (avg (point-x p1)
                          (point-x p2))
                     (avg (point-y p1)
                          (point-y p2)))))
    (with-dependency
     `(midpoint ,(element-dependency p1) ,(element-dependency p2))
     (with-source (lambda (premise)
                    (midpoint
                     ((element-source p1) premise)
                     ((element-source p1) premise)))
                  newpoint))))

(define (segment-midpoint s)
  (let ((p1 (segment-endpoint-1 s))
        (p2 (segment-endpoint-2 s)))
    (with-dependency
     `(segment-midpoint ,s)
     (with-source (lambda (premise)
                    (segment-midpoint
                     ((element-source s) premise)))
                  (midpoint p1 p2)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; TODO: Where to put these?
(define (on-segment? p seg)
  (let ((seg-start (segment-endpoint-1 seg))
        (seg-end (segment-endpoint-2 seg)))
    (or (point-equal? seg-start p)
        (point-equal? seg-end p)
        (let ((seg-length (distance seg-start seg-end))
              (p-length (distance seg-start p))
              (dir-1 (direction-from-points seg-start p))
              (dir-2 (direction-from-points seg-start seg-end)))
          (and (direction-equal? dir-1 dir-2)
               (< p-length seg-length))))))

(define (on-line? p l)
  (let ((line-pt (line-point l))
        (line-dir (line-direction l)))
    (or (point-equal? p line-pt)
        (let ((dir-to-p (direction-from-points p line-pt)))
          (or (direction-equal? line-dir dir-to-p)
              (direction-equal? line-dir (reverse-direction
                   dir-to-p)))))))

(define (on-ray? p r)
  (let ((ray-endpt (ray-endpoint r))
        (ray-dir (ray-direction r)))
    (or (point-equal? ray-endpt p)
        (let ((dir-to-p (direction-from-points ray-endpt p)))
          (direction-equal? dir-to-p ray-dir)))))

;;;;;;;;;;;;;;;;;;;;;; Construction of lines ;;;;;;;;;;;;;;;;;;;;;;;

(define (perpendicular linear-element point)
  (let* ((direction (->direction linear-element))
         (rotated-direction (rotate-direction-90 direction)))
    (make-line point rotated-direction)))

;;; endpoint-1 is point, endpoint-2 is on linear-element
(define (perpendicular-to linear-element point)
  (let ((pl (perpendicular linear-element point)))
    (let ((i (intersect-linear-elements pl (->line linear-element))))
      (make-segment point i))))

(define (perpendicular-line-to linear-element point)
  (let ((pl (perpendicular linear-element point)))
    pl))

(define (perpendicular-bisector segment)
  (let ((midpt (segment-midpoint segment)))
    (let ((pb (perpendicular (segment->line segment)
                             midpt)))
      (save-obvious-observation!
       (make-observation perpendicular-relationship
                         (list pb segment)))
      pb)))

(define (angle-bisector a)
  (let* ((d1 (angle-arm-1 a))
         (d2 (angle-arm-2 a))
         (vertex (angle-vertex a))
         (radians (angle-measure a))
         (half-angle (/ radians 2))
         (new-direction (add-to-direction d2 half-angle)))
    (make-ray vertex new-direction)))

(define (polygon-angle-bisector polygon vertex-angle)
  (angle-bisector (polygon-angle polygon vertex-angle)))

;;;;;;;;;;;;;;;;;;;; Higher-order constructions ;;;;;;;;;;;;;;;;;;;;

(define (circumcenter t)
  (let ((p1 (polygon-point-ref t 0))
        (p2 (polygon-point-ref t 1))
        (p3 (polygon-point-ref t 2)))
    (let ((l1 (perpendicular-bisector (make-segment p1 p2)))
          (l2 (perpendicular-bisector (make-segment p1 p3))))
      (intersect-linear-elements l1 l2))))

;;;;;;;;;;;;;;;;;;;; Concurrent Linear Elements ;;;;;;;;;;;;;;;;;;;;

```

```
(define (concurrent? l1 l2 l3)
  (let ((i-point (intersect-linear-elements-no-endpoints l1 l2)))
    (and i-point
         (on-element? i-point l3)
         (not (element-endpoint? i-point l3)))))

(define (concentric? p1 p2 p3 p4)
  (and (not (point-equal? p1 p2))
       (not (point-equal? p1 p3))
       (not (point-equal? p1 p4))
       (not (point-equal? p2 p3))
       (not (point-equal? p2 p4))
       (not (point-equal? p3 p4))
       (let ((pb-1 (perpendicular-bisector
                    (make-segment p1 p2)))
             (pb-2 (perpendicular-bisector
                    (make-segment p2 p3)))
             (pb-3 (perpendicular-bisector
                    (make-segment p3 p4))))
         (concurrent? pb-1 pb-2 pb-3))))

(define (concentric-with-center? center p1 p2 p3)
  (let ((d1 (distance center p1))
        (d2 (distance center p2))
        (d3 (distance center p3)))
    (and (close-enuf? d1 d2)
         (close-enuf? d1 d3))))
```

Listing A.14: figure/intersections.scm

```
;;; intersections.scm --- Intersections

;;; Commentary:

;; Ideas:
;; - Unified intersections
;; - Separation of core computations

;; Future:
;; - Amb-like selection of multiple intersections, or list?
;; - Deal with elements that are exactly the same

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Computations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; http://en.wikipedia.org/wiki/Line%E2%80%93line_intersection
;;; line 1 through p1, p2 with line 2 through p3, p4
(define (intersect-lines-by-points p1 p2 p3 p4)
  (let ((x1 (point-x p1))
        (y1 (point-y p1))
        (x2 (point-x p2))
        (y2 (point-y p2))
        (x3 (point-x p3))
        (y3 (point-y p3))
        (x4 (point-x p4))
        (y4 (point-y p4)))
    (let* ((denom
            (det (det x1 1 x2 1)
                 (det y1 1 y2 1)
                 (det x3 1 x4 1)
                 (det y3 1 y4 1)))
           (num-x
            (det (det x1 y1 x2 y2)
                 (det x1  1 x2  1)
                 (det x3 y3 x4 y4)
                 (det x3  1 x4  1)))
           (num-y
            (det (det x1 y1 x2 y2)
                 (det y1  1 y2  1)
                 (det x3 y3 x4 y4)
                 (det y3  1 y4  1))))
      (if (= denom 0)
          '()
          (let
              ((px (/ num-x denom))
               (py (/ num-y denom)))
            (list (make-point px py)))))))

;;; http://mathforum.org/library/drmath/view/51836.html
(define (intersect-circles-by-centers-radii c1 r1 c2 r2)
  (let* ((a (point-x c1))
         (b (point-y c1))
         (c (point-x c2))
         (d (point-y c2))
         (e (- c a))
         (f (- d b))
         (p (sqrt (+ (square e)
                     (square f))))
         (k (/ (- (+ (square p) (square r1))
                  (square r2))
               (* 2 p))))
    (if (> k r1)
        (error "Circle's don't intersect")
        (let* ((t (sqrt (- (square r1)
                           (square k))))
               (x1 (+ a (/ (* e k) p)))
               (y1 (+ b (/ (* f k) p)))
               (dx (/ (* f t) p))
               (dy (- (/ (* e t) p))))
          (list (make-point (+ x1 dx)
                            (+ y1 dy))
                (make-point (- x1 dx)
                            (- y1 dy)))))))

;;; Intersect circle centered at c with radius r and line through
;;; points p1, p2
;;; http://mathworld.wolfram.com/Circle-LineIntersection.html
```

```
(define (intersect-circle-line-by-points c r p1 p2)
  (let ((offset (sub-points (make-point 0 0) c)))
    (let ((p1-shifted (add-to-point p1 offset))
          (p2-shifted (add-to-point p2 offset)))
      (let ((x1 (point-x p1-shifted))
            (y1 (point-y p1-shifted))
            (x2 (point-x p2-shifted))
            (y2 (point-y p2-shifted)))
        (let* ((dx (- x2 x1))
               (dy (- y2 y1))
               (dr (sqrt (+ (square dx) (square dy))))
               (d (det x1 x2 y1 y2))
               (disc (- (* (square r) (square dr)) (square d))))
          (if (< disc 0)
              (list)
              (let ((x-a (* d dy))
                    (x-b (* (sgn dy) dx (sqrt disc)))
                    (y-a (- (* d dx)))
                    (y-b (* (abs dy) (sqrt disc))))
                (let ((ip1 (make-point
                            (/ (+ x-a x-b) (square dr))
                            (/ (+ y-a y-b) (square dr))))
                      (ip2 (make-point
                            (/ (- x-a x-b) (square dr))
                            (/ (- y-a y-b) (square dr)))))
                  (if (close-enuf? 0 disc) ;; Tangent
                      (list (add-to-point ip1 (reverse-vec offset)))
                      (list (add-to-point ip1 (reverse-vec offset))
                            (add-to-point ip2 (reverse-vec
                                                offset)))))))))))

;;;;;;;;;;;;;;;;;;;;;; Basic Intersections ;;;;;;;;;;;;;;;;;;;;;;;

(define (intersect-lines-to-list line1 line2)
  (let ((p1 (line-p1 line1))
        (p2 (line-p2 line1))
        (p3 (line-p1 line2))
        (p4 (line-p2 line2)))
    (intersect-lines-by-points p1 p2 p3 p4)))

(define (intersect-lines line1 line2)
  (let ((i-list (intersect-lines-to-list line1 line2)))
    (if (null? i-list)
        (error "Lines don't intersect")
        (car i-list))))

(define (intersect-circles cir1 cir2)
  (let ((c1 (circle-center cir1))
        (c2 (circle-center cir2))
        (r1 (circle-radius cir1))
        (r2 (circle-radius cir2)))
    (intersect-circles-by-centers-radii c1 r1 c2 r2)))

(define (intersect-circle-line cir line)
  (let ((center (circle-center cir))
        (radius (circle-radius cir))
        (p1 (line-p1 line))
        (p2 (line-p2 line)))
    (intersect-circle-line-by-points center radius p1 p2)))

(define standard-intersect
  (make-generic-operation 2 'standard-intersect))

(defhandler standard-intersect
  intersect-lines-to-list line? line?)

(defhandler standard-intersect
  intersect-circles circle? circle?)

(defhandler standard-intersect
  intersect-circle-line circle? line?)

(defhandler standard-intersect
  (flip-args intersect-circle-line) line? circle?)

;;;;;;;;;;;;;;;;;;;;;; Generic intersection ;;;;;;;;;;;;;;;;;;;;;;

(define (intersect-linear-elements el-1 el-2)
  (let ((i-list (standard-intersect (->line el-1)
                                    (->line el-2))))
    (if (null? i-list)
        #f
        (let ((i (car i-list)))
          (if (or (not (on-element? i el-1))
                  (not (on-element? i el-2)))
              #f
              i)))))

(define (intersect-linear-elements-no-endpoints el-1 el-2)
  (let ((i (intersect-linear-elements el-1 el-2)))
    (and (or i
             (element-endpoint? i el-1)
             (element-endpoint? i el-2))
         i)))

;;;;;;;;;;;;;;;;;;;;;;;;;;; On Elements ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define on-element? (make-generic-operation 2 'on-element?))

(defhandler on-element? on-segment? point? segment?)
(defhandler on-element? on-line? point? line?)
(defhandler on-element? on-ray? point? ray?)

;;;;;;;;;;;;;;;;;;;;; Element Endpoint Test ;;;;;;;;;;;;;;;;;;;;;;

(define element-endpoint? (make-generic-operation 2 'on-endpoint?
                                                  (lambda (p el) #f)))
```

```
(define (segment-endpoint? p seg)
  (or (point-equal? p (segment-endpoint-1 seg))
      (point-equal? p (segment-endpoint-2 seg))))
(defhandler element-endpoint? segment-endpoint? point? segment?)

(define (ray-endpoint? p ray)
  (point-equal? p (ray-endpoint seg)))
(defhandler element-endpoint? ray-endpoint? point? ray?)
```

Listing A.15: figure/figure.scm

```
;;; figure.scm --- Figure

;;; Commentary:

;; Ideas:
;; - Gathers elements that are part of a figure
;; - Helpers to extract relevant elements

;; Future:
;; - Convert to record type like other structures
;; - Extract points automatically?

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;; Figure Structure ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (figure . elements)
  (cons 'figure elements))
(define (figure-elements figure)
  (cdr figure))

(define (all-figure-elements figure)
  (append (figure-elements figure)
          (figure-points figure)
          (figure-linear-elements figure)))

(define (figure? x)
  (and (pair? x)
       (eq? (car x 'figure))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Getters ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (figure-filter predicate figure)
  (filter predicate (figure-elements figure)))

(define (figure-points figure)
  (dedupe-by point-equal?
             (append (figure-filter point? figure)
                     (append-map (lambda (polygon) (polygon-points
                          polygon))
                                 (figure-filter polygon? figure))
                     (append-map (lambda (s)
                                   (list (segment-endpoint-1 s)
                                         (segment-endpoint-2 s)))
                                 (figure-filter segment? figure))
                     (map (lambda (a)
                            (angle-vertex a))
                          (figure-filter angle? figure)))))

(define (figure-angles figure)
  (append (figure-filter angle? figure)
          (append-map (lambda (polygon) (polygon-angles polygon))
                      (figure-filter polygon? figure))))

(define (figure-segments figure)
  (append (figure-filter segment? figure)
          (append-map (lambda (polygon) (polygon-segments polygon))
                      (figure-filter polygon? figure))))

(define (figure-linear-elements figure)
  (append (figure-filter linear-element? figure)
          (append-map (lambda (polygon) (polygon-segments polygon))
                      (figure-filter polygon? figure))))
```

Listing A.16: figure/math-utils.scm

```
;;; math-utils.scm --- Math Helpers

;;; Commentary:

;; Ideas:
;; - All angles are [0, 2pi]
;; - Other helpers

;; Future:
;; - Add more as needed, integrate with scmutils-basic

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Angles ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define pi (* 4 (atan 1)))

(define (fix-angle-0-2pi a)
  (float-mod a (* 2 pi)))

(define (rad->deg rad)
  (* (/ rad (* 2 pi)) 360))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Modular ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (float-mod num mod)
  (- num
     (* (floor (/ num mod))
        mod)))

;;;;;;;;;;;;;;;;;;;;;;;;; Basic Operators ;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```scheme

(define (avg a b)
  (/ (+ a b) 2))

(define (sgn x)
  (if (< x 0) -1 1))

;;;;;;;;;;;;;;;;;;;;;;;;;; Linear Alegbra ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (det a11 a12 a21 a22)
  (- (* a11 a22) (* a12 a21)))

;;;;;;;;;;;;;;;;;;;;;; Extensions of Max/Min ;;;;;;;;;;;;;;;;;;;;;;;

(define (min-positive . args)
  (min (filter (lambda (x) (>= x 0)) args)))

(define (max-negative . args)
  (min (filter (lambda (x) (<= x 0)) args)))
```

Listing A.17: figure/polygon.scm

```scheme
;;; polygon.scm --- Polygons

;;; Commentary:

;; Ideas:
;; - Points and (derived) segments define polygon

;; Future
;; - Figure out dependencies better
;; - Other operations, angles? diagonals? etc.

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;; Polygon Structure ;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Data structure for a polygon, implemented as a list of
;;; points in counter-clockwise order.
;;; Drawing a polygon will draw all of its points and segments.
(define-record-type <polygon>
  (%polygon n-points points)
  polygon?
  (n-points polygon-n-points)
  (points %polygon-points))

(define (polygon-from-points . points)
  (let ((n-points (length points)))
    (%polygon n-points points)))

(define ((ngon-predicate n) obj)
  (and (polygon? obj)
       (= n (polygon-n-points obj))))

;;;;;;;;;;;;;;;;;;;;;;;;;; Polygon Points ;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Internal reference for polygon points
(define (polygon-point-ref polygon i)
  (if (not (<= 0 i (- (polygon-n-points polygon) 1)))
      (error "polygon point index not in range"))
  (list-ref (%polygon-points polygon) i))

(define (polygon-points polygon)
  (map (lambda (i) (polygon-point polygon i))
       (iota (polygon-n-points polygon))))

;;; External polygon points including dependencies
(define (polygon-point polygon i)
  ;;: TODO: Handle situations where polygon isn't terminal dependency
  (with-dependency ;;-if-unknown
   `(polygon-point ,i ,(element-dependency polygon))
   (with-source
    (lambda (p) (polygon-point (car p) i))
    (polygon-point-ref polygon i))))

(declare-element-component-handler
 polygon-point
 polygon?)

(define (polygon-index-from-point polygon point)
  (index-of
   point
   (%polygon-points polygon)
   point-equal?))

(define (name-polygon polygon)
  (for-each (lambda (i)
              (set-element-name! (polygon-point-ref polygon i)
                                 (nth-letter-symbol (+ i 1))))
            (iota (polygon-n-points polygon)))
  polygon)

;;;;;;;;;;;;;;;;;;;;;;;; Polygon Segments ;;;;;;;;;;;;;;;;;;;;;;;;;

;;; i and j are indices of adjacent points
(define (polygon-segment polygon i j)
  (let ((n-points (polygon-n-points polygon)))
   (cond
    ((not (or (= i (modulo (+ j 1) n-points))
              (= j (modulo (+ i 1) n-points))))
     (error "polygon-segment must be called with adjacent indices"))
    ((or (>= i n-points)
         (>= j n-points))
     (error "polygon-segment point index out of range"))
    (else
     (let* ((p1 (polygon-point-ref polygon i))
            (p2 (polygon-point-ref polygon j))
            (segment (make-segment p1 p2)))
      ;;: TODO: Handle situations where polygon isn't terminal
           dependency
      (with-dependency
       `(polygon-segment ,i ,j ,polygon)
       (with-source
        (lambda (p) (polygon-segment (car p) i j))
        segment)))))))

(define (polygon-segments polygon)
  (let ((n-points (polygon-n-points polygon)))
    (map (lambda (i)
           (polygon-segment polygon i (modulo (+ i 1) n-points)))
         (iota n-points))))

;;;;;;;;;;;;;;;;;;;;;;;;;; Polygon Angles ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define polygon-angle
  (make-generic-operation 2 'polygon-angle))

(define (polygon-angle-by-index polygon i)
  (let ((n-points (polygon-n-points polygon)))
    (cond
     ((not (<= 0 i (- n-points 1)))
      (error "polygon-angle point index out of range"))
     (else
      (let* ((v (polygon-point-ref polygon i))
             (a1p (polygon-point-ref polygon
                                     (modulo (- i 1)
                                             n-points)))
             (a2p (polygon-point-ref polygon
                                     (modulo (+ i 1)
                                             n-points)))
             (angle (angle-from-points a1p v a2p)))
        (with-dependency
         `(polygon-angle ,i ,polygon)
         (with-source
          (lambda (p) (polygon-angle-by-index (car p) i))
          angle)))))))

(defhandler polygon-angle
  polygon-angle-by-index
  polygon? number?)

(define (polygon-angle-by-point polygon p)
  (let ((i (polygon-index-from-point polygon p)))
    (if (not i)
        (error "Point not in polygon" (list p polygon)))
    (polygon-angle-by-index polygon i)))

(defhandler polygon-angle
  polygon-angle-by-point
  polygon? point?)

(define (polygon-angles polygon)
  (map (lambda (i) (polygon-angle-by-index polygon i))
       (iota (polygon-n-points polygon))))
```

Listing A.18: figure/metadata.scm

```
;;; metadata.scm - Element metadata

;;; Commentary:

;; Ideas:
;; - Currently, names
;; - Dependencies grew here, but are now separate

;; Future:
;; - Point/Linear/Circle adjacency - walk like graph

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Names ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (set-element-name! element name)
  (eq-put! element 'name name)
  element)

(define (element-name element)
  (or (eq-get element 'name)
      (generic-element-name element)))

(define *unnamed* (list 'unnamed))
(define (is-unnamed? x) (eq? *unnamed* x))

(define generic-element-name
  (make-generic-operation 1 'generic-element-name
                          (lambda (el) *unnamed*)))

(define (named? element)
  (not (is-unnamed? (element-name element-name))))
```

Listing A.19: figure/dependencies.scm

```
;;; dependencies.scm --- Dependencies of figure elements

;;; Commentary:

;; Ideas:
;; - Use eq-properties to set dependencies of elements
;; - Some random elements are gien external/random dependencies
;; - For some figures, override dependencies of intermediate elements

;; Future:
;; - Expand to full dependencies
;; - Start "learning" and generalizing

```

```
;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Sources ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (set-source! element source)
  (eq-put! element 'source source))

(define (with-source source element)
  (set-source! element source)
  element)

(define (element-source element)
  (or (eq-get element 'source)
      '*unknown-source*))

;;;;;;;;;;;;;;;;;;;;;;; Setitng Dependencies ;;;;;;;;;;;;;;;;;;;;;;;

(define (set-dependency! element dependency)
  (eq-put! element 'dependency dependency))

(define (with-dependency dependency element)
  (set-dependency! element dependency)
  element)


(define (with-dependency-if-unknown dependency element)
  (if (dependency-unknown? element)
      (with-dependency dependency element)
      element))
;;;;;;;;;;;;;;;;;;;;;;; Unknown Dependencies ;;;;;;;;;;;;;;;;;;;;;;;

(define *unknown-dependency* (list '*unknown-dependency*))
(define (unknown-dependency? x)
  (eq? x *unknown-dependency*))

(define (dependency-unknown? element)
  (unknown-dependency? (element-dependency element)))

(define dependency-known? (notp dependency-unknown?))
;;;;;;;;;;;;;;;;;;;;;; Accessing Dependencies ;;;;;;;;;;;;;;;;;;;;;;

(define (element-dependency element)
  (or (eq-get element 'dependency)
      *unknown-dependency*))

;;;;;;;;;;;;;;;;;;;;;;; Random Dependencies ;;;;;;;;;;;;;;;;;;;;;;;;
(define (make-random-dependency tag)
  (%make-random-dependency tag 0))

(define-record-type <random-dependency>
  (%make-random-dependency tag num)
  random-dependency?
  (tag random-dependency-tag)
  (num %random-dependency-num set-random-dependency-num!))

(define (random-dependency-num rd)
  (let ((v (%random-dependency-num rd)))
    (if (= v 0)
        0
        v)))

(define (print-random-dependency rd)
  (list (random-dependency-tag rd)
        (random-dependency-num rd)))
(defhandler print print-random-dependency random-dependency?)

(define (number-figure-random-dependencies! figure)
  (define *random-dependency-num* 1)
  (map (lambda (el)
         (let ((dep (element-dependency el)))
           (cond ((random-dependency? dep)
                  (set-random-dependency-num!
                   dep
                   *random-dependency-num*)
                  (set! *random-dependency-num*
                        (+ *random-dependency-num* 1))))))
       (figure-elements figure))
  'done)

(define element-dependencies->list
  (make-generic-operation
   1 'element-dependencies->list
   (lambda (x) x)))

(define (element-dependency->list el)
  (element-dependencies->list
   (element-dependency el)))

(defhandler element-dependencies->list
  element-dependency->list
  dependency-known?)

(defhandler element-dependencies->list
  print-random-dependency
  random-dependency?)

(defhandler element-dependencies->list
  (lambda (l)
    (map element-dependencies->list l))
  list?)




;;;;;;;;;;;;;;;;;;;;; Formatting Dependencies ;;;;;;;;;;;;;;;;;;;;;

(define (format-dependencies object)
  (element-dependencies->list object))
```

Listing A.20: figure/randomness.scm

```
;;; randomness.scm --- Random creation of elements

;;; Commentary:

;; Ideas:
;; - Random points, segments, etc. essential to system
;; - Separated out animation / persistence across frames

;; Future:
;; - Better random support
;; - Maybe separating out "definitions" (random square, etc.)

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;; Base: Random Scalars ;;;;;;;;;;;;;;;;;;;;;;;

(define (internal-rand-range min-v max-v)
  (if (close-enuf? min-v max-v)
      (error "range is too close for rand-range"
             (list min-v max-v))
      (let ((interval-size (max *machine-epsilon* (- max-v min-v))))
        (persist-value (+ min-v (random (* 1.0 interval-size)))))))

(define (safe-internal-rand-range min-v max-v)
  (let ((interval-size (max 0 (- max-v min-v))))
    (internal-rand-range
     (+ min-v (* 0.1 interval-size))
     (+ min-v (* 0.9 interval-size)))))

;;;;;;;;;;;;;;;;;;;;;;;;; Animated Ranges ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *wiggle-ratio* 0.15)

;;; Will return floats even if passed integers
;;; TODO: Rename to animated?
(define (rand-range min max)
  (let* ((range-size (- max min))
         (wiggle-amount (* range-size *wiggle-ratio*))
         (v (internal-rand-range min (- max wiggle-amount))))
    (animate-range v (+ v wiggle-amount))))

(define (safe-rand-range min-v max-v)
  (let ((interval-size (max 0 (- max-v min-v))))
    (rand-range
     (+ min-v (* 0.1 interval-size))
     (+ min-v (* 0.9 interval-size)))))

;;; Random Values - distances, angles

(define (rand-theta)
  (rand-range 0 (* 2 pi)))

(define (rand-angle-measure)
  (rand-range (* pi 0.05) (* .95 pi)))

(define (rand-obtuse-angle-measure)
  (rand-range (* pi 0.55) (* .95 pi)))

(define (random-direction)
  (let ((theta (rand-theta)))
    (make-direction theta)))

;;;;;;;;;;;;;;;;;;;;;;;;;; Random Points ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *point-wiggle-radius* 0.05)
(define (random-point)
  (let ((x (internal-rand-range -0.8 0.8))
        (y (internal-rand-range -0.8 0.8)))
    (random-point-around (make-point x y))))

(define (random-point-around p)
  (let ((x (point-x p))
        (y (point-y p)))
    (let ((theta (internal-rand-range 0 (* 2 pi)))
          (d-theta (animate-range 0 (* 2 pi))))
      (let ((dir (make-direction (+ theta d-theta))))
        (with-dependency
         (make-random-dependency 'random-point)
         (add-to-point
          (make-point x y)
          (vec-from-direction-distance dir *point-wiggle-radius*)))))))

;;; TODO: Maybe separate out reflection about line?
(define (random-point-left-of-line line)
  (let* ((p (random-point))
         (d (signed-distance-to-line p line))
         (v (rotate-vec-90
             (unit-vec-from-direction
              (line-direction line)))))
    (if (> d 0)
        p
        (add-to-point p (scale-vec v (* 2 (- d)))))))

(define (random-point-between-rays r1 r2)
  (let ((offset-vec (sub-points (ray-endpoint r2)
                                (ray-endpoint r1))))
    (let ((d1 (ray-direction r1))
          (d2 (ray-direction r2)))
      (let ((dir-difference (subtract-directions d2 d1)))
        (let ((new-dir (add-to-direction
                        d1
                        (internal-rand-range 0.05 dir-difference))))
          (random-point-around
           (add-to-point
            (add-to-point (ray-endpoint r1)
                          (vec-from-direction-distance
                           new-dir
                           (internal-rand-range 0.05 0.9)))
            (scale-vec offset-vec
                       (internal-rand-range 0.05 0.9))))))))

(define (random-point-on-segment seg)
  (let* ((p1 (segment-endpoint-1 seg))
         (p2 (segment-endpoint-2 seg))
         (t (rand-range 0.05 1.0))
         (v (sub-points p2 p1)))
    (add-to-point p1 (scale-vec v t))))

;;; TODO: Fix this for new construction
(define (random-point-on-line l)
  (let* ((p1 (line-p1 l))
         (p2 (line-p2 l))
         (seg (extend-to-max-segment p1 p2))
         (sp1 (segment-endpoint-1 seg))
         (sp2 (segment-endpoint-2 seg))
         (t (rand-range 0.0 1.0))
         (v (sub-points sp2 sp1)))
    (add-to-point sp1 (scale-vec v t))))

(define (random-point-on-ray r)
  (let* ((p1 (ray-endpoint r))
         (dir (ray-direction r))
         (p2 (add-to-point p1 (unit-vec-from-direction dir)))
         (seg (ray-extend-to-max-segment p1 p2))
         (sp1 (segment-endpoint-1 seg))
         (sp2 (segment-endpoint-2 seg))
         (t (rand-range 0.05 1.0))
         (v (sub-points sp2 sp1)))
    (add-to-point sp1 (scale-vec v t))))


#|
(define (random-point-on-ray r)
  (random-point-on-segment
   (ray-extend-to-max-segment r)))
|#

(define (random-point-on-circle c)
  (let ((dir (random-direction)))
    (point-on-circle-in-direction c dir)))

(define (n-random-points-on-circle-ccw c n)
  (let* ((thetas
          (sort
           (make-initialized-list n (lambda (i) (rand-theta)))
           <)))
    (map (lambda (theta)
           (point-on-circle-in-direction c
                                         (make-direction theta)))
         thetas)))

;;;;;;;;;;;;;;;;;;;;;; Random Linear Elements ;;;;;;;;;;;;;;;;;;;;;;

(define (random-line)
  (let ((p (random-point)))
    (with-dependency
     (make-random-dependency 'random-line)
     (random-line-through-point p))))

(define (random-segment)
  (let ((p1 (random-point))
        (p2 (random-point)))
    (let ((seg (make-segment p1 p2)))
      (set-segment-dependency!
       seg
       (make-random-dependency 'random-segment))
      seg)))

(define (random-ray)
  (let ((p (random-point)))
    (random-ray-from-point p)))

(define (random-line-through-point p)
  (let ((v (random-direction)))
    (line-from-point-direction p v)))

(define (random-ray-from-point p)
  (let ((v (random-direction)))
    (ray-from-point-direction p v)))

(define (random-horizontal-line)
  (let ((p (random-point))
        (v (make-vec 1 0)))
    (line-from-point-vec p v)))

(define (random-vertical-line)
  (let ((p (random-point))
        (v (make-vec 0 1)))
    (line-from-point-vec p v)))

;;;;;;;;;;;;;;;;;;;;;; Random Circle Elements ;;;;;;;;;;;;;;;;;;;;;;

(define (random-circle-radius circle)
  (let ((center (circle-center circle))
        (radius (circle-radius circle))
        (angle (random-direction)))
    (let ((radius-vec
           (scale-vec (unit-vec-from-direction
                       (random-direction))
                      radius)))
      (let ((radius-point (add-to-point center radius-vec)))
        (make-segment center radius-point)))))

(define (random-circle)
  (let ((pr1 (random-point))
        (pr2 (random-point)))
    (circle-from-points (midpoint pr1 pr2) pr1)))

(define (random-angle)
  (let* ((v (random-point))
         (d1 (random-direction))
         (d2 (add-to-direction
              d1
              (rand-angle-measure))))
    (make-angle d1 v d2)))

;;;;;;;;;;;;;;;;;;;;;;;;; Random Polygons ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (random-n-gon n)
  (if (< n 3)
      (error "n must be > 3"))
  (let* ((p1 (random-point))
         (p2 (random-point)))
    (let ((ray2 (reverse-ray (ray-from-points p1 p2))))
      (let lp ((n-remaining (- n 2))
               (points (list p2 p1)))
        (if (= n-remaining 0)
            (apply polygon-from-points (reverse points))
            (lp (- n-remaining 1)
                (cons (random-point-between-rays
                       (reverse-ray (ray-from-points (car points)
                                                     (cadr points)))
                       ray2)
                      points)))))))

(define (random-polygon)
  (random-n-gon (+ 3 (random 5))))

;;;;;;;;;;;;;;;;;;;;;;;;; Random Triangles ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (random-triangle)
  (let* ((p1 (random-point))
         (p2 (random-point))
         (p3 (random-point-left-of-line (line-from-points p1 p2))))
    (with-dependency
     (make-random-dependency 'random-triangle)
     (polygon-from-points p1 p2 p3))))

(define (random-equilateral-triangle)
  (let* ((s1 (random-segment))
         (s2 (rotate-about (segment-endpoint-1 s1)
                           (/ pi 3)
                           s1)))
    (with-dependency
     (make-random-dependency 'random-equilateral-triangle)
     (polygon-from-points
      (segment-endpoint-1 s1)
      (segment-endpoint-2 s1)
      (segment-endpoint-2 s2)))))

(define (random-isoceles-triangle)
  (let* ((s1 (random-segment))
         (base-angle (rand-angle-measure))
         (s2 (rotate-about (segment-endpoint-1 s1)
                           base-angle
                           s1)))
    (with-dependency
     (make-random-dependency 'random-isoceles-triangle)
     (polygon-from-points
      (segment-endpoint-1 s1)
      (segment-endpoint-2 s1)
      (segment-endpoint-2 s2)))))

;;;;;;;;;;;;;;;;;;;;;; Random Quadrilaterals ;;;;;;;;;;;;;;;;;;;;;;

(define (random-quadrilateral)
  (with-dependency
   (make-random-dependency 'random-quadrilateral)
   (random-n-gon 4)))

(define (random-square)
  (let* ((s1 (random-segment))
         (p1 (segment-endpoint-1 s1))
         (p2 (segment-endpoint-2 s1))
         (p3 (rotate-about p2
                           (- (/ pi 2))
                           p1))
         (p4 (rotate-about p1
                           (/ pi 2)
                           p2)))
    (with-dependency
     (make-random-dependency 'random-square)
     (polygon-from-points p1 p2 p3 p4))))

(define (random-rectangle)
  (let* ((r1 (random-ray))
         (p1 (ray-endpoint r1))
         (r2 (rotate-about (ray-endpoint r1)
                           (/ pi 2)
                           r1))
         (p2 (random-point-on-ray r1))
         (p4 (random-point-on-ray r2))
         (p3 (add-to-point
              p2
              (sub-points p4 p1))))
    (with-dependency
     (make-random-dependency 'random-rectangle)
     (polygon-from-points
      p1 p2 p3 p4))))

(define (random-parallelogram)
  (let* ((r1 (random-ray))
        (p1 (ray-endpoint r1))
        (r2 (rotate-about (ray-endpoint r1)
                          (rand-angle-measure)
                          r1))
        (p2 (random-point-on-ray r1))
        (p4 (random-point-on-ray r2))
        (p3 (add-to-point
             p2
             (sub-points p4 p1))))
    (with-dependency
     (make-random-dependency 'random-parallelogram)
     (polygon-from-points p1 p2 p3 p4))))

(define (random-kite)
  (let* ((r1 (random-ray))
        (p1 (ray-endpoint r1))
        (r2 (rotate-about (ray-endpoint r1)
                          (rand-obtuse-angle-measure)
                          r1))
        (p2 (random-point-on-ray r1))
        (p4 (random-point-on-ray r2))
        (p3 (reflect-about-line
             (line-from-points p2 p4)
             p1)))
    (with-dependency
     (make-random-dependency 'random-parallelogram)
     (polygon-from-points p1 p2 p3 p4))))

(define (random-rhombus)
  (let* ((s1 (random-segment))
        (p1 (segment-endpoint-1 s1))
        (p2 (segment-endpoint-2 s1))
        (p4 (rotate-about p1
                          (rand-angle-measure)
                          p2))
        (p3 (add-to-point
             p2
             (sub-points p4 p1))))
    (with-dependency
     (make-random-dependency 'random-rhombus)
     (polygon-from-points p1 p2 p3 p4))))

(define (random-trapezoid)
  (let* ((r1 (random-ray))
        (r2 (translate-randomly r1))
        (p1 (ray-endpoint r1))
        (p2 (random-point-on-ray r1))
        (p3 (random-point-on-ray r2))
        (p4 (ray-endpoint r2)))
    (with-dependency
     (make-random-dependency 'random-trapezoid)
     (polygon-from-points p1 p2 p3 p4))))
```

Listing A.21: figure/transforms.scm

```
;;; transforms.scm --- Transforms on Elements

;;; Commentary:

;; Ideas:
;; - Generic transforms - rotation and translation
;; - None mutate points, just return new copies.

;; Future:
;; - Translation or rotation to match something
;; - Consider mutations?
;; - Reflections?

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Rotations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (rotate-point-about rot-origin radians point)
  (let ((v (sub-points point rot-origin)))
    (let ((rotated-v (rotate-vec v radians)))
      (add-to-point rot-origin rotated-v))))

(define (rotate-segment-about rot-origin radians seg)
  (define (rotate-point p) (rotate-point-about rot-origin radians p))
  (make-segment (rotate-point (segment-endpoint-1 seg))
                (rotate-point (segment-endpoint-2 seg))))

(define (rotate-ray-about rot-origin radians r)
  (define (rotate-point p) (rotate-point-about rot-origin radians p))
  (make-ray (rotate-point-about rot-origin radians (ray-endpoint r))
            (add-to-direction (ray-direction r) radians)))

(define (rotate-line-about rot-origin radians l)
  (make-line (rotate-point-about rot-origin radians (line-point l))
             (add-to-direction (line-direction l) radians)))

(define rotate-about (make-generic-operation 3 'rotate-about))
(defhandler rotate-about rotate-point-about point? number? point?)
(defhandler rotate-about rotate-ray-about point? number? ray?)
(defhandler rotate-about rotate-segment-about point? number? segment?)
(defhandler rotate-about rotate-line-about point? number? line?)

(define (rotate-randomly-about p elt)
  (let ((radians (rand-angle-measure)))
    (rotate-about p radians elt)))

;;;;;;;;;;;;;;;;;;;;;;;;;;; Translations ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (translate-point-by vec point)
  (add-to-point point vec))

(define (translate-segment-by vec segment)
```

```scheme
  (define (translate-point p) (translate-point-by vec p))
  (make-segment (translate-point (segment-endpoint-1 seg))
                (translate-point (segment-endpoint-2 seg))))

(define (translate-ray-by vec r)
  (make-ray (translate-point-by vec (ray-endpoint r))
            (ray-direction r)))

(define (translate-line-by vec l)
  (make-line (translate-point-by vec (line-point l))
             (line-direction l)))

(define (translate-angle-by vec a)
  (define (translate-point p) (translate-point-by vec p))
  (make-angle (angle-arm-1 a)
              (translate-point (angle-vertex a))
              (angle-arm-2 a)))

(define translate-by (make-generic-operation 2 'rotate-about))
(defhandler translate-by translate-point-by vec? point?)
(defhandler translate-by translate-ray-by vec? ray?)
(defhandler translate-by translate-segment-by vec? segment?)
(defhandler translate-by translate-line-by vec? line?)
(defhandler translate-by translate-angle-by vec? angle?)

;;; Reflections

(define (reflect-about-line line p)
  (if (on-line? p line)
      p
      (let ((s (perpendicular-to line p)))
        (let ((v (segment->vec s)))
          (add-to-point
           p
           (scale-vec v 2))))))

;;;;;;;;;;;;;;;;;;;;;;;;; Random Translation ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (translate-randomly-along-line l elt)
  (let* ((vec (unit-vec-from-direction (line->direction l)))
         (scaled-vec (scale-vec vec (rand-range 0.5 1.5))))
    (translate-by vec elt)))

(define (translate-randomly elt)
  (let ((vec (rand-translation-vec-for elt)))
    (translate-by vec elt)))

(define (rand-translation-vec-for-point p1)
  (let ((p2 (random-point)))
    (sub-points p2 p1)))

(define (rand-translation-vec-for-segment seg)
  (rand-translation-vec-for-point (segment-endpoint-1 seg)))

(define (rand-translation-vec-for-ray r )
  (rand-translation-vec-for-point (ray-endpoint r)))

(define (rand-translation-vec-for-line l)
  (rand-translation-vec-for-point (line-point l)))

(define rand-translation-vec-for
  (make-generic-operation 1 'rand-translation-vec-for))
(defhandler rand-translation-vec-for
  rand-translation-vec-for-point point?)
(defhandler rand-translation-vec-for
  rand-translation-vec-for-segment segment?)
(defhandler rand-translation-vec-for
  rand-translation-vec-for-ray ray?)
(defhandler rand-translation-vec-for
  rand-translation-vec-for-line line?)
```

Listing A.22: perception/load.scm

```scheme
;;; load.scm -- Load perception
(for-each (lambda (f) (load f))
          '("relationship"
            "observation"
            "analyzer"))
```

Listing A.23: perception/observation.scm

```scheme
;;; observation.scm -- observed relationships

;;; Commentary:

;; Future:
;; - Observation equality is more complicated!

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Observation ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <observation>
  (make-observation relationship args)
  observation?
  (relationship observation-relationship)
  (args observation-args))

(define (observation-equal? obs1 obs2)
  (equal? (print-observation obs1)
          (print-observation obs2)))

(define (print-observation obs)
  (cons
   (print (observation-relationship obs))
   (map print (observation-args obs))))

```

```
(defhandler print print-observation observation?)

(define (print-observations obs-list)
  (map print-observation obs-list))

(define (observation-with-premises obs)
  (cons (observation-relationship obs)
        (map element-dependencies->list (observation-args obs))))

(define (observations-eqivalent? obs1 obs2)
  (and (eq? (observation-relationship obs1)
            (observation-relationship obs2))
       (let ((rel-eqv-test
              (relationship-equivalence-predicate
               (observation-relationship obs1)))
             (args1 (observation-args obs1))
             (args2 (observation-args obs2)))
         (rel-eqv-test args1 args2))))
```

Listing A.24: perception/analyzer.scm

```
;;; analyzer.scm --- Tools for analyzing Diagram

;;; Commentary

;; Ideas:
;; - Analyze figrue to dermine properties "beyond coincidence"
;; - Use dependency structure to eliminate some obvious examples.

;; Future:
;; - Add More "interesting properties"
;; - Create storage for learned properties.
;; - Output format, add names
;; - Separate "discovered" from old properties.

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;; Main Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (all-observations figure)
  (analyze figure))

(define (analyze-figure figure)
  (all-observations figure))

;;; Given a figure, report what's interesting
(define (all-observations figure)
  (number-figure-random-dependencies! figure)
  (let* ((points (figure-points figure))
         (angles (figure-angles figure))
         (implied-segments '() ; (point-pairs->segments (all-pairs
              points))
                           )
         (linear-elements (append
                           (figure-linear-elements figure)
                           implied-segments))
         (segments (append (figure-segments figure)
                           implied-segments)))
    (append
     (extract-relationships points
                            (list concurrent-points-relationship
                                  concentric-relationship
                                  concentric-with-center-relationship))
     (extract-relationships segments
                             (list equal-length-relationship))
     (extract-relationships angles
                             (list equal-angle-relationship
                                   supplementary-angles-relationship
                                   complementary-angles-relationship))
     (extract-relationships linear-elements
                             (list parallel-relationship
                                   concurrent-relationship
                                   perpendicular-relationship
                                   )))))

(define (extract-relationships elements relationships)
  (append-map (lambda (r)
                (extract-relationship elements r))
              relationships))

(define (extract-relationship elements relationship)
  (map (lambda (tuple)
         (make-observation relationship tuple))
       (report-n-wise
        (relationship-arity relationship)
        (relationship-predicate relationship)
        elements)))

;;;;;;;;;;;;;;;;;;;;;; Interesting Observations ;;;;;;;;;;;;;;;;;;;;;

(define (interesting-observations figure-proc)
  (set! *obvious-observations* '())
  (let ((all-obs (all-observations (figure-proc))))
    (pprint *obvious-observations*)
    (pprint all-obs)
    (set-difference all-obs *obvious-observations*
                    observation-equal?)))

(define *obvious-observations* #f)

(define (save-obvious-observation! obs)
  (if *obvious-observations*
      (begin
        (pprint obs)
        (set! *obvious-observations*
              (cons obs
                    *obvious-observations*)))))

```

```scheme
;;;;;;;;;;;;;;;;;;;;;;; Cross products, pairs ;;;;;;;;;;;;;;;;;;;;;;;;

;;; General proceudres for generating pairs
(define (all-pairs elements)
  (all-n-tuples 2 elements))

(define (all-n-tuples n elements)
  (cond ((zero? n) '(()))
        ((< (length elements) n) '())
        (else
         (let lp ((elements-1 elements))
           (if (null? elements-1)
               '()
               (let ((element-1 (car elements-1))
                     (n-minus-1-tuples
                      (all-n-tuples (- n 1) (cdr elements-1))))
                 (append
                  (map
                   (lambda (rest-tuple)
                     (cons element-1 rest-tuple))
                   n-minus-1-tuples)
                  (lp (cdr elements-1)))))))))

;;;;;;;;;;;;;;;;;;;;;;;; Obvious Segments ;;;;;;;;;;;;;;;;;;;;;;;;

(define (segment-for-endpoint p1)
  (let ((dep (element-dependency p1)))
    (and dep
         (or (and (eq? (car dep) 'segment-endpoint-1)
                  (cadr dep))
             (and (eq? (car dep) 'segment-endpoint-2)
                  (cadr dep))))))

(define (derived-from-same-segment? p1 p2)
  (and
   (segment-for-endpoint p1)
   (segment-for-endpoint p2)
   (eq? (segment-for-endpoint p1)
        (segment-for-endpoint p2))))

(define (polygon-for-point p1)
  (let ((dep (element-dependency p1)))
    (and dep
         (and (eq? (car dep) 'polygon-point)
              (cons (caddr dep)
                    (cadr dep))))))

(define (adjacent-in-same-polygon? p1 p2)
  (let ((poly1 (polygon-for-point p1))
        (poly2 (polygon-for-point p2)))
    (and poly1 poly2
         (eq? (car poly1) (car poly2))
         (or (= (abs (- (cdr poly1)
                        (cdr poly2)))
                1)
             (and (= (cdr poly1) 0)
                  (= (cdr poly2) 3))
             (and (= (cdr poly1) 3)
                  (= (cdr poly2) 0))))))

(define (point-pairs->segments ppairs)
  (filter (lambda (segment) segment)
          (map (lambda (point-pair)
                 (let ((p1 (car point-pair))
                       (p2 (cadr point-pair)))
                   (and (not (point-equal? p1 p2))
                        (not (derived-from-same-segment? p1 p2))
                        (not (adjacent-in-same-polygon? p1 p2))
                        (make-auxiliary-segment
                         (car point-pair)
                         (cadr point-pair))))) ; TODO: Name segment
               ppairs)))

;;;;;;;;;;;;;;;;;;;;;;;; Dealing with pairs ;;;;;;;;;;;;;;;;;;;;;;;;

;;; Check for pairwise equality
(define ((nary-predicate n predicate) tuple)
  (apply predicate tuple))

;;; Merges "connected-components" of pairs
(define (merge-pair-groups elements pairs)
  (let ((i 0)
        (group-ids (make-key-weak-eq-hash-table))
        (group-elements (make-key-weak-eq-hash-table))) ; Map from pair
    (for-each (lambda (pair)
                (let ((first (car pair))
                      (second (cadr pair)))
                  (let ((group-id-1 (hash-table/get group-ids first i))
                        (group-id-2 (hash-table/get group-ids second i)))
                    (cond ((and (= group-id-1 i)
                                (= group-id-2 i))
                           ;; Both new, new groups:
                           (hash-table/put! group-ids first group-id-1)
                           (hash-table/put! group-ids second group-id-1))
                          ((= group-id-1 i)
                           (hash-table/put! group-ids first group-id-2))
                          ((= group-id-2 i)
                           (hash-table/put! group-ids second
                                group-id-1)))
                    (set! i (+ i 1)))))
              pairs)
    (for-each (lambda (elt)
                (hash-table/append group-elements
                                   (hash-table/get group-ids elt
                                        'invalid)
                                   elt))
              elements)
    (hash-table/remove! group-elements 'invalid)
    (hash-table/datum-list group-elements)))

(define (report-n-wise n predicate elements)
  (let ((tuples (all-n-tuples n elements)))
    (filter (nary-predicate n predicate) tuples)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Results: ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-analysis-collector)
  (make-equal-hash-table))

(define (save-results results data-table)
  (hash-table/put! data-table results
                   (+ 1 (hash-table/get data-table results 0))))

(define (print-analysis-results data-table)
  (hash-table/for-each
   data-table
   (lambda (k v)
     (pprint (list v (cons 'discovered k))))))
```

Listing A.25: graphics/load.scm

```
;;; load.scm -- Load graphics
(for-each (lambda (f) (load f))
          '("appearance"
            "graphics"))
```

Listing A.26: graphics/appearance.scm

```
(define (with-color color element)
  (eq-put! element 'color color)
  element)

(define default-element-color
  (make-generic-operation 1
                          'default-element-color
                          (lambda (e) "black")))

(defhandler default-element-color (lambda (e) "blue") point?)
(defhandler default-element-color (lambda (e) "black") segment?)

(define (element-color element)
  (or (eq-get element 'color)
      (default-element-color element)))
```

Listing A.27: graphics/graphics.scm

```
;;; graphics.scm -- Graphics Commands

;;;;;;;;;;;;;;;;;;;;;;;;;;; Main Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (draw-figure figure canvas)
  (set-coordinates-for-figure figure canvas)
  (clear-canvas canvas)
  (for-each
   (lambda (element)
     (canvas-set-color canvas (element-color element))
     ((draw-element element) canvas))
   (all-figure-elements figure))
  (for-each
   (lambda (element)
     (canvas-set-color canvas (element-color element))
     ((draw-label element) canvas))
   (all-figure-elements figure))
  (graphics-flush (canvas-g canvas)))

(define (set-coordinates-for-figure figure canvas)
  (let* ((bounds (scale-bounds (bounds->square (extract-bounds figure))
                               1.1)))
    (graphics-set-coordinate-limits
     (canvas-g canvas)
     (bounds-xmin bounds)
     (bounds-ymin bounds)
     (bounds-xmax bounds)
     (bounds-ymax bounds))))

(define draw-element
  (make-generic-operation 1 'draw-element
                          (lambda (e) (lambda (c) 'done))))

(define draw-label
  (make-generic-operation 1 'draw-label (lambda (e) (lambda (c)'done))))

(define (add-to-draw-element! predicate handler)
  (defhandler draw-element
    (lambda (element)
      (lambda (canvas)
        (handler canvas element)))
    predicate))

(define (add-to-draw-label! predicate handler)
  (defhandler draw-label
    (lambda (element)
      (lambda (canvas)
        (handler canvas element)))
    predicate))


(define *point-radius* 0.02)
(define (draw-point canvas point)
  (canvas-fill-circle canvas
                      (point-x point)
                      (point-y point)
                      *point-radius*))
(define (draw-point-label canvas point)
  (canvas-draw-text canvas
                    (+ (point-x point) *point-radius*)
                    (+ (point-y point) *point-radius*)
                    (symbol->string (element-name point))))

(define (draw-segment canvas segment)
  (let ((p1 (segment-endpoint-1 segment))
        (p2 (segment-endpoint-2 segment)))
    (canvas-draw-line canvas
                      (point-x p1)
                      (point-y p1)
                      (point-x p2)
                      (point-y p2))))
(define (draw-segment-label canvas segment)
  (let ((v (vec-from-direction-distance (rotate-direction-90
                                         (segment->direction segment))
                                        (* 2 *point-radius*)))
        (m (segment-midpoint segment)))
    (let ((label-point (add-to-point m v)))
      (canvas-draw-text canvas
                        (point-x label-point)
                        (point-y label-point)
                        (symbol->string (element-name segment))))))

(define (draw-line canvas line)
  (let ((p1 (line-p1 line)))
   (let ((p2 (add-to-point
              p1
              (unit-vec-from-direction (line-direction line)))))
     (draw-segment canvas (extend-to-max-segment p1 p2)))))

(define (draw-ray canvas ray)
  (let ((p1 (ray-endpoint ray)))
    (let ((p2 (add-to-point
               p1
               (unit-vec-from-direction (ray-direction ray)))))
      (draw-segment canvas (ray-extend-to-max-segment p1 p2)))))

(define (draw-circle canvas c)
  (let ((center (circle-center c))
        (radius (circle-radius c)))
    (canvas-draw-circle canvas
                        (point-x center)
                        (point-y center)
                        radius)))

(define *angle-mark-radius* 0.05)
(define (draw-angle canvas a)
  (let* ((vertex (angle-vertex a))
         (d1 (angle-arm-1 a))
         (d2 (angle-arm-2 a))
         (angle-start (direction-theta d2))
         (angle-end (direction-theta d1)))
    (canvas-draw-arc canvas
                     (point-x vertex)
                     (point-y vertex)
                     *angle-mark-radius*
                     angle-start
                     angle-end)))

;;; Add to generic operations

(add-to-draw-element! point? draw-point)
(add-to-draw-element! segment? draw-segment)
(add-to-draw-element! circle? draw-circle)
(add-to-draw-element! angle? draw-angle)
(add-to-draw-element! line? draw-line)
(add-to-draw-element! ray? draw-ray)

(add-to-draw-label! point? draw-point-label)

;;; Canvas for x-graphics

(define (x-graphics) (make-graphics-device 'x))

(define (canvas)
  (let ((g (x-graphics)))
    (graphics-enable-buffering g)
    (list 'canvas g)))

(define (canvas-g canvas)
  (cadr canvas))

(define (canvas? x)
  (and (pair? x)
       (eq? (car x 'canvas))))

(define (clear-canvas canvas)
  (graphics-clear (canvas-g canvas)))

(define (canvas-draw-circle canvas x y radius)
  (graphics-operation (canvas-g canvas)
                      'draw-circle
                      x y radius))

(define (canvas-draw-text canvas x y text)
  (graphics-draw-text (canvas-g canvas) x y text))

(define (canvas-draw-arc canvas x y radius
                         angle-start angle-end)
  (let ((angle-sweep
         (fix-angle-0-2pi (- angle-end
                             angle-start))))
    (graphics-operation (canvas-g canvas)
                        'draw-arc
                        x y radius radius
                        (rad->deg angle-start)
                        (rad->deg angle-sweep)
                        #f)))

(define (canvas-fill-circle canvas x y radius)
  (graphics-operation (canvas-g canvas)
                      'fill-circle
                      x y radius))

(define (canvas-draw-line canvas x1 y1 x2 y2)
  (graphics-draw-line (canvas-g canvas)
                      x1 y1
                      x2 y2))

(define (canvas-set-color canvas color)
  (graphics-operation (canvas-g canvas) 'set-foreground-color color)
  )
```

Listing A.28: manipulate/load.scm

```
;;; load.scm -- Load manipulate
(for-each (lambda (f) (load f))
          '("linkages"
            "region"
            "constraints"
            "topology"
            "mechanism"
            "main"))
```

Listing A.29: manipulate/linkages.scm

```
;;; linkages.scm ---  Bar/Joint propagators between directions and coordinates

;;; Commentary:

;; Ideas:
;; - Join "Identify" bars and joints to build mechanism
;;   versions of diagrams
;; - Use propagator system to deal with partial information
;; - Used Regions for partial info about points,
;; - Direction Intervals for partial info about joint directions.

;; Future:
;; - Other Linkages?
;; - Draw partially assembled linkages

;;; Example:

#|
 (let* ((s1 (m:make-bar))
        (s2 (m:make-bar))
        (j (m:make-joint)))
   (m:instantiate (m:joint-theta j) (/ pi 2) 'theta)
   (c:id (m:bar-length s1)
         (m:bar-length s2))
   (m:instantiate-point (m:bar-p2 s1) 4 0 'bar-2-endpoint)
   (m:instantiate-point (m:bar-p1 s1) 2 -2 'bar-2-endpoint)
   (m:identify-out-of-arm-1 j s1)
   (m:identify-out-of-arm-2 j s2)
   (run)
   (m:examine-point (m:bar-p2 s2)))
|#

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;; TMS Interfaces ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:instantiate cell value premise)
  (add-content cell
               (make-tms (contingent value (list premise)))))

(define (m:examine-cell cell)
  (let ((v (content cell)))
    (cond ((nothing? v) v)
          ((tms? v)
           (contingent-info (tms-query v)))
          (else v))))

(defhandler print
  (lambda (cell) (print (m:examine-cell cell)))
  cell?)

(define (m:contradictory? cell)
  (contradictory? (m:examine-cell cell)))

;;;;;;;;;;;;;;;;;;;;;;; Reversing directions ;;;;;;;;;;;;;;;;;;;;;;;

(define m:reverse-direction
  (make-generic-operation 1 'm:reverse-direction))
(defhandler m:reverse-direction
  reverse-direction direction?)
(defhandler m:reverse-direction
  reverse-direction-interval direction-interval?)

(propagatify m:reverse-direction)

(define (ce:reverse-direction input-cell)
  (let-cells (output-cell)
    (name! output-cell (symbol 'reverse- (name input-cell)))
    (p:m:reverse-direction input-cell output-cell)
    (p:m:reverse-direction output-cell input-cell)
    output-cell))

;;;;;;;;;;;;;;;;;;;;;;; Adding to directions ;;;;;;;;;;;;;;;;;;;;;;;

(define (m:add-interval-to-direction d i)
  (if (empty-interval? i)
      (error "Cannot add empty interval to direction"))
  (make-direction-interval-from-start-dir-and-size
   (add-to-direction d (interval-low i))
   (- (interval-high i)
      (interval-low i))))

(define (m:add-interval-to-standard-direction-interval di i)
  (if (empty-interval? i)
      (error "Cannot add empty interval to direction"))
  (let ((di-size (direction-interval-size di))
        (i-size (- (interval-high i)
                   (interval-low i)))
        (di-start (direction-interval-start di)))
    (make-direction-interval-from-start-dir-and-size
     (add-to-direction di-start (interval-low i))
     (+ di-size i-size))))

(define (m:add-interval-to-full-circle-direction-interval fcdi i)
  (if (empty-interval? i)
      (error "Cannot add empty interval to direction"))
  fcdi)

(define (m:add-interval-to-invalid-direction-interval fcdi i)
  (if (empty-interval? i)
      (error "Cannot add empty interval to direction"))
  (error "Cannot add to invalid direction in"))

(define m:add-to-direction
  (make-generic-operation 2 'm:add-to-direction))

(defhandler m:add-to-direction
  m:add-interval-to-direction direction? interval?)

(defhandler m:add-to-direction
  add-to-direction direction? number?)

(defhandler m:add-to-direction
  m:add-interval-to-standard-direction-interval
  standard-direction-interval? interval?)

(defhandler m:add-to-direction
  m:add-interval-to-full-circle-direction-interval
  full-circle-direction-interval? interval?)

(defhandler m:add-to-direction
  m:add-interval-to-invalid-direction-interval
  invalid-direction-interval? interval?)

(defhandler m:add-to-direction
  shift-direction-interval direction-interval? number?)

(propagatify m:add-to-direction)

;;;;;;;;;;;;;;;;;;;;;; Subtracting directions ;;;;;;;;;;;;;;;;;;;;;;

(defhandler generic-negate
  (lambda (i) (mul-interval i -1))  % interval?)

(define (m:standard-direction-interval-minus-direction di d)
  (if (within-direction-interval? d di)
      (make-interval
       0
       (subtract-directions (direction-interval-end di) d))
      (make-interval
       (subtract-directions (direction-interval-start di) d)
       (subtract-directions (direction-interval-end di) d))))

(define (m:full-circle-direction-interval-minus-direction di d)
  (make-interval
   0 (* 2 pi)))

(define (m:direction-minus-standard-direction-interval d di)
  (if (within-direction-interval? d di)
      (make-interval
       0
       (subtract-directions d (direction-interval-start di)))
      (make-interval
       (subtract-directions d (direction-interval-end di))
       (subtract-directions d (direction-interval-start di)))))

(define (m:direction-minus-full-circle-direction-interval d di)
  (make-interval
   0 (* 2 pi)))

(define m:subtract-directions
  (make-generic-operation 2 'm:subtract-directions))

(defhandler m:subtract-directions
  subtract-directions direction? direction?)

;;; TODO: Support Intervals for thetas?
(defhandler m:subtract-directions
  (lambda (di1 di2)
    nothing)
  direction-interval? direction-interval?)

(defhandler m:subtract-directions
  m:standard-direction-interval-minus-direction
  standard-direction-interval? direction?)

(defhandler m:subtract-directions
  m:full-circle-direction-interval-minus-direction
  full-circle-direction-interval? direction?)

(defhandler m:subtract-directions
  m:direction-minus-standard-direction-interval
  direction? standard-direction-interval?)

(defhandler m:subtract-directions
  m:direction-minus-full-circle-direction-interval
  direction? full-circle-direction-interval?)

(propagatify m:subtract-directions)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Vec ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-record-type <m:vec>
  (%m:make-vec dx dy length direction)
  m:vec?
  (dx m:vec-dx)
  (dy m:vec-dy)
  (length m:vec-length)
  (direction m:vec-direction))


;;; Allocate and wire up the cells in a vec
(define (m:make-vec vec-id)
  (let-cells (dx dy length direction)
    (name! dx (symbol vec-id '-dx))
    (name! dy (symbol vec-id '-dy))
    (name! length (symbol vec-id '-len))
    (name! direction (symbol vec-id '-dir))

    (p:make-direction
     (e:atan2 dy dx) direction)
    (p:sqrt (e:+ (e:square dx)
                 (e:square dy))
            length)
    (p:* length (e:direction-cos direction) dx)
    (p:* length (e:direction-sin direction) dy)
    (%m:make-vec dx dy length direction)))

(define (m:print-vec v)
  `(m:vec (,(print (m:vec-dx v))
           ,(print (m:vec-dy v)))
          ,(print (m:vec-length v))
          ,(print (m:vec-direction v))))

(defhandler print m:print-vec m:vec?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Point ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-record-type <m:point>
  (%m:make-point x y region)
  m:point?
  (x m:point-x)
  (y m:point-y)
  (region m:point-region))

;;; Allocate cells for a point
(define (m:make-point id)
  (let-cells (x y region)
    (name! x (symbol id '-x))
    (name! y (symbol id '-y))
    (name! region (symbol id '-region))
    (p:m:x-y->region x y region)
    (p:m:region->x region x)
    (p:m:region->y region y)
    (%m:make-point x y region)))

(define (m:x-y->region x y)
  (m:make-singular-point-set (make-point x y)))

(propagatify m:x-y->region)

(define (m:region->x region)
  (if (m:singular-point-set? region)
      (point-x (m:singular-point-set-point region))
      nothing))

(define (m:region->y region)
  (if (m:singular-point-set? region)
      (point-y (m:singular-point-set-point region))
      nothing))

(propagatify m:region->x)
(propagatify m:region->y)

(define (m:instantiate-point p x y premise)
  (m:instantiate (m:point-x p)
                 x premise)
  (m:instantiate (m:point-y p)
                 y premise)
  (m:instantiate (m:point-region p)
                 (m:make-singular-point-set (make-point x y))
                 premise))

(define (m:examine-point p)
  (list 'm:point
        (m:examine-cell (m:point-x p))
        (m:examine-cell (m:point-y p))))

(define (m:print-point p)
  `(m:point ,(print (m:point-x p))
            ,(print (m:point-y p))
            ,(print (m:point-region p))))

(defhandler print m:print-point m:point?)

;;; Set p1 and p2 to be equal
(define (m:identify-points p1 p2)
  (for-each (lambda (getter)
              (c:id (getter p1)
                    (getter p2)))
            (list m:point-x m:point-y m:point-region)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Bar ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <m:bar>
```

```scheme
  (%m:make-bar p1 p2 vec)
  m:bar?
  (p1 m:bar-p1)
  (p2 m:bar-p2)
  (vec m:bar-vec))

(define (m:bar-direction bar)
  (m:vec-direction (m:bar-vec bar)))

(define (m:bar-length bar)
  (m:vec-length (m:bar-vec bar)))

(define (m:print-bar b)
  `(m:bar
    ,(print (m:bar-name b))
    ,(print (m:bar-p1 b))
    ,(print (m:bar-p2 b))
    ,(print (m:bar-vec b))))

(defhandler print m:print-bar m:bar?)

;;; Allocate cells and wire up a bar
(define (m:make-bar bar-id)
  (let ((bar-key (m:make-bar-name-key bar-id)))
    (let ((p1 (m:make-point (symbol bar-key '-p1)))
          (p2 (m:make-point (symbol bar-key '-p2))))
      (name! p1 (symbol bar-key '-p1))
      (name! p2 (symbol bar-key '-p2))
      (let ((v (m:make-vec bar-key)))
        (c:+ (m:point-x p1)
             (m:vec-dx v)
             (m:point-x p2))
        (c:+ (m:point-y p1)
             (m:vec-dy v)
             (m:point-y p2))
        (let ((bar (%m:make-bar p1 p2 v)))
          (m:p1->p2-bar-propagator p1 p2 bar)
          (m:p2->p1-bar-propagator p2 p1 bar)
          bar)))))

;;; TODO: Combine p1->p2 / p2->p1
(define (m:x-y-direction->region px py direction)
  (if (direction? direction)
      (let ((vertex (make-point px py)))
        (m:make-ray vertex direction))
      nothing))

(propagatify m:x-y-direction->region)

(define (m:x-y-length-di->region px py length dir-interval)
  (if (direction-interval? dir-interval)
      (let ((vertex (make-point px py)))
        (m:make-arc vertex length dir-interval))
      nothing))
(propagatify m:x-y-length-di->region)

(define (m:region-length-direction->region pr length dir)
  (if (direction-interval? dir)
      nothing
      (m:translate-region
       pr
       (vec-from-direction-distance dir length))))
(propagatify m:region-length-direction->region)


(define (m:p1->p2-bar-propagator p1 p2 bar)
  (let ((p1x (m:point-x p1))
        (p1y (m:point-y p1))
        (p1r (m:point-region p1))
        (p2r (m:point-region p2))
        (length (m:bar-length bar))
        (dir (m:bar-direction bar)))
    (p:m:x-y-direction->region p1x p1y dir p2r)
    (p:m:x-y-length-di->region p1x p1y length dir p2r)
    (p:m:region-length-direction->region p1r length dir p2r)))

(define (m:p2->p1-bar-propagator p2 p1 bar)
  (let ((p2x (m:point-x p2))
        (p2y (m:point-y p2))
        (p1r (m:point-region p1))
        (p2r (m:point-region p2))
        (length (m:bar-length bar))
        (dir (m:bar-direction bar)))
    (p:m:x-y-direction->region p2x p2y (ce:reverse-direction dir) p1r)
    (p:m:x-y-length-di->region p2x p2y length (ce:reverse-direction dir)
         p1r)
    (p:m:region-length-direction->region
     p2r length (ce:reverse-direction dir) p1r)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Joint  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Direction-2 is counter-clockwise from direction-1 by theta
(define-record-type <m:joint>
  (%m:make-joint vertex dir-1 dir-2 theta)
  m:joint?
  (vertex m:joint-vertex)
  (dir-1 m:joint-dir-1)
  (dir-2 m:joint-dir-2)
  (theta m:joint-theta))

(define *max-joint-swing* pi)

(define (m:make-joint joint-id)
  (let ((joint-key (m:make-joint-name-key joint-id)))
   (let ((vertex (m:make-point (symbol joint-key '-vertex))))
     (let-cells (dir-1 dir-2 theta)
       (name! dir-1 (symbol joint-key '-dir-1))
       (name! dir-2 (symbol joint-key '-dir-2))
       (name! theta (symbol joint-key '-theta))
      (name! vertex (symbol joint-key '-vertex))
      (p:m:add-to-direction
       dir-1 theta dir-2)
      (p:m:add-to-direction
       dir-2 (e:negate theta) dir-1)
      (p:m:subtract-directions
       dir-2 dir-1
       theta)
      (m:instantiate theta (make-interval 0 *max-joint-swing*) 'theta)
      (% m:make-joint vertex dir-1 dir-2 theta)))))

(define (m:print-joint j)
  `(m:joint
    ,(print (m:joint-name j))
    ,(print (m:joint-dir-1 j))
    ,(print (m:joint-vertex j))
    ,(print (m:joint-dir-2 j))
    ,(print (m:joint-theta j))))

(defhandler print m:print-joint m:joint?)

;;; TOOD: Abstract?
(define (m:identify-out-of-arm-1 joint bar)
  (m:set-endpoint-1 bar joint)
  (m:set-joint-arm-1 joint bar)
  (m:identify-points (m:joint-vertex joint)
                     (m:bar-p1 bar))
  (c:id (m:joint-dir-1 joint)
        (m:bar-direction bar)))

(define (m:identify-out-of-arm-2 joint bar)
  (m:set-endpoint-1 bar joint)
  (m:set-joint-arm-2 joint bar)
  (m:identify-points (m:joint-vertex joint)
                     (m:bar-p1 bar))
  (c:id (m:joint-dir-2 joint)
        (m:bar-direction bar)))

(define (m:identify-into-arm-1 joint bar)
  (m:set-endpoint-2 bar joint)
  (m:set-joint-arm-1 joint bar)
  (m:identify-points (m:joint-vertex joint)
                     (m:bar-p2 bar))
  (c:id (ce:reverse-direction (m:joint-dir-1 joint))
        (m:bar-direction bar)))

(define (m:identify-into-arm-2 joint bar)
  (m:set-endpoint-2 bar joint)
  (m:set-joint-arm-2 joint bar)
  (m:identify-points (m:joint-vertex joint)
                     (m:bar-p2 bar))
  (c:id (ce:reverse-direction (m:joint-dir-2 joint))
        (m:bar-direction bar)))

;;;;;;;;;;;;;;;;;;;;;;; Storing Adjacencies ;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:set-endpoint-1 bar joint)
  (eq-append! bar 'm:bar-endpoints-1 joint))

(define (m:bar-endpoints-1 bar)
  (or (eq-get bar 'm:bar-endpoints-1)
      '()))

(define (m:set-endpoint-2 bar joint)
  (eq-append! bar 'm:bar-endpoints-2 joint))

(define (m:bar-endpoints-2 bar)
  (or (eq-get bar 'm:bar-endpoints-2)
      '()))

(define (m:set-joint-arm-1 joint bar)
  (eq-put! joint 'm:joint-arm-1 bar))

(define (m:joint-arm-1 joint)
  (eq-get joint 'm:joint-arm-1))

(define (m:set-joint-arm-2 joint bar)
  (eq-put! joint 'm:joint-arm-2 bar))

(define (m:joint-arm-2 joint)
  (eq-get joint 'm:joint-arm-2))

;;;;;;;;;;;;;;;;;;;;;;;;;; Named Linkages  ;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:make-bar-name-key bar-id)
  (symbol 'm:bar:
          (m:bar-id-p1-name bar-id) ':
          (m:bar-id-p2-name bar-id)))

(define (m:make-joint-name-key joint-id)
  (symbol 'm:joint:
          (m:joint-id-dir-1-name joint-id) ':
          (m:joint-id-vertex-name joint-id) ':
          (m:joint-id-dir-2-name joint-id)))

(define (m:name-element! element name)
  (eq-put! element 'm:name name))

(define (m:element-name element)
  (or (eq-get element 'm:name)
      '*unnamed*))

(define (m:make-named-bar p1-name p2-name)
  (let ((bar (m:make-bar (m:bar p1-name p2-name))))
    (m:name-element! (m:bar-p1 bar) p1-name)
    (m:name-element! (m:bar-p2 bar) p2-name)
    bar))
```

```
(define (m:bar-name bar)
  (m:bar
   (m:element-name (m:bar-p1 bar))
   (m:element-name (m:bar-p2 bar))))

(define (m:bars-name-equivalent? bar-1 bar-2)
  (or (m:bar-id-equal?
       (m:bar-name bar-1)
       (m:bar-name bar-2))
      (m:bar-id-equal?
       (m:bar-name bar-1)
       (m:reverse-bar-id (m:bar-name bar-2)))))

(define (m:bar-p1-name bar)
  (m:element-name (m:bar-p1 bar)))

(define (m:bar-p2-name bar)
  (m:element-name (m:bar-p2 bar)))

(define (m:make-named-joint arm-1-name vertex-name arm-2-name)
  (let ((joint-id (m:joint arm-1-name
                           vertex-name
                           arm-2-name)))
   (let ((joint (m:make-joint joint-id)))
     (m:name-element! (m:joint-dir-1 joint) arm-1-name)
     (m:name-element! (m:joint-vertex joint) vertex-name)
     (m:name-element! (m:joint-dir-2 joint) arm-2-name)
     joint)))

(define (m:joint-name joint)
  (m:joint
   (m:joint-dir-1-name joint)
   (m:joint-vertex-name joint)
   (m:joint-dir-2-name joint)))

(define (m:joint-vertex-name joint)
  (m:element-name (m:joint-vertex joint)))

(define (m:joint-dir-1-name joint)
  (m:element-name (m:joint-dir-1 joint)))

(define (m:joint-dir-2-name joint)
  (m:element-name (m:joint-dir-2 joint)))

;;;;;;;;;;;;;;;;; Symbolic Bar / Joint Identifiers ;;;;;;;;;;;;;;;;;

;;; Maybe Move?

(define-record-type <m:bar-id>
  (%m:make-bar-id p1-name p2-name)
  m:bar-id?
  (p1-name m:bar-id-p1-name)
  (p2-name m:bar-id-p2-name))

(define (m:bar-id-equal? bar-id-1 bar-id-2)
  (and (eq? (m:bar-id-p1-name bar-id-1)
            (m:bar-id-p1-name bar-id-2))
       (eq? (m:bar-id-p2-name bar-id-1)
            (m:bar-id-p2-name bar-id-2))))

(define (m:bar p1-name p2-name)
  (%m:make-bar-id p1-name p2-name))

(defhandler print m:make-bar-name-key m:bar-id?)

(define (m:reverse-bar-id bar-id)
  (%m:make-bar-id (m:bar-id-p2-name bar-id)
                  (m:bar-id-p1-name bar-id)))

;;; Joints:

(define-record-type <m:joint-vertex-id>
  (%m:make-joint-verex-id vertex-name)
  m:joint-vertex-id?
  (vertex-name m:joint-vertex-id-name))

(define-record-type <m:joint-id>
  (%m:make-joint-id dir-1-name vertex-name dir-2-name)
  m:joint-id?
  (dir-1-name m:joint-id-dir-1-name)
  (vertex-name m:joint-id-vertex-name)
  (dir-2-name m:joint-id-dir-2-name))

(defhandler print m:make-joint-name-key m:joint-id?)

(define (m:joint arg1 . rest)
  (cond ((null? rest)
         (%m:make-joint-verex-id arg1))
        ((= 2 (length rest))
         (%m:make-joint-id arg1 (car rest) (cadr rest)))
        (else
         (error "m:joint was called with the wrong number of
              arguments."))))

;;;;;;;;;;;;; Tables and Accessors for named linkages ;;;;;;;;;;;;;;
(define (m:make-bars-by-name-table bars)
  (let ((table (make-key-weak-eqv-hash-table)))
    (for-each (lambda (bar)
                (let ((key (m:make-bar-name-key (m:bar-name bar))))
                  (if (hash-table/get table key #f)
                      (error "Bar key already in bar name table" key))
                  (hash-table/put! table key bar)))
              bars)
    table))

;;; Unordered
(define (m:find-bar-by-id table bar-id)
  (or (hash-table/get table
                       (m:make-bar-name-key bar-id)
                       #f)
       (hash-table/get table
                       (m:make-bar-name-key (m:reverse-bar-id bar-id))
                       #f)))

;;; Joints:

(define (m:make-joints-by-vertex-name-table joints)
  (let ((table (make-key-weak-eq-hash-table)))
    (for-each
     (lambda (joint)
       (let ((key (m:joint-vertex-name joint)))
         (hash-table/put!
          table key
          (cons
           joint (hash-table/get table
                                  key
                                  '())))))
     joints)
    table))

(define (m:find-joint-by-vertex-name table vertex-name)
  (let ((joints (hash-table/get table
                                 vertex-name
                                 #f)))
    (cond ((null? joints) #f)
          ((= (length joints) 1)
           (car joints))
          (else (error "Vertex name not unique among joints"
                       (map m:joint-name joints))))))

(define (m:make-joints-by-name-table joints)
  (let ((table (make-key-weak-eq-hash-table)))
    (for-each (lambda (joint)
                (hash-table/put! table
                                  (m:make-joint-name-key (m:joint-name
                                       joint))
                                  joint))
              joints)
    table))

;;; dir-2 is CCW from dir-1
(define (m:find-joint-by-id table joint-id)
  (hash-table/get
   table
   (m:make-joint-name-key joint-id)
   #f))

;;;;;;;;;;;;;;;;;;;;;; Operations using Names ;;;;;;;;;;;;;;;;;;;;;;

(define (m:identify-joint-bar-by-name joint bar)
  (let ((vertex-name (m:joint-vertex-name joint))
        (dir-1-name (m:joint-dir-1-name joint))
        (dir-2-name (m:joint-dir-2-name joint))
        (bar-p1-name (m:bar-p1-name bar))
        (bar-p2-name (m:bar-p2-name bar)))
    (cond ((eq? vertex-name bar-p1-name)
           (cond ((eq? dir-1-name bar-p2-name)
                  (m:identify-out-of-arm-1 joint bar))
                 ((eq? dir-2-name bar-p2-name)
                  (m:identify-out-of-arm-2 joint bar))
                 (else (error "Bar can't be identified with joint - no
                      arm"
                              bar-p2-name))))
          ((eq? vertex-name bar-p2-name)
           (cond ((eq? dir-1-name bar-p1-name)
                  (m:identify-into-arm-1 joint bar))
                 ((eq? dir-2-name bar-p1-name)
                  (m:identify-into-arm-2 joint bar))
                 (else (error "Bar can't be identified with joint - no
                      arm"
                              bar-p1-name))))
          (else (error "Bar can't be identified with joint - no vertex"
                       vertex-name)))))

;;;;;;;;;;;;;;;;;;;;;;; Degrees of Freedom  ;;;;;;;;;;;;;;;;;;;;;;;

(define (m:specified? cell #!optional predicate)
  (let ((v (m:examine-cell cell)))
    (and
     (not (nothing? v))
     (or (default-object? predicate)
         (predicate v)))))

(define (m:bar-length-specified? bar)
  (m:specified? (m:bar-length bar)) number?)

(define (m:bar-direction-specified? bar)
  (m:specified? (m:bar-direction bar)) direction?)

(define (m:joint-theta-specified? joint)
  (m:specified? (m:joint-theta joint)) number?)

;;;;;;;;;;;;;;;;;;;;;;;; Point Predicates ;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:point-specified? p)
  (and (m:specified? (m:point-x p) number?)
       (m:specified? (m:point-y p) number?)))

(define (m:point-contradictory? p)
  (or (m:contradictory? (m:point-x p))
      (m:contradictory? (m:point-y p))
      (m:contradictory? (m:point-region p))))

;;;;;;;;;;;;;;;;;;;;;;;;; Bar Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:bar-p1-specified? bar)
  (m:point-specified? (m:bar-p1 bar)))

(define (m:bar-p2-specified? bar)
  (m:point-specified? (m:bar-p2 bar)))

(define (m:bar-p1-contradictory? bar)
  (m:point-contradictory? (m:bar-p1 bar)))

(define (m:bar-p2-contradictory? bar)
  (m:point-contradictory? (m:bar-p2 bar)))

(define (m:bar-anchored? bar)
  (or (m:bar-p1-specified? bar)
      (m:bar-p2-specified? bar)))

(define (m:bar-directioned? bar)
  (and (m:bar-anchored? bar)
       (m:specified? (m:bar-direction bar) direction?)))

(define (m:bar-direction-contradictory? bar)
  (or (m:contradictory? (m:bar-direction bar))
      (m:contradictory? (m:vec-dx (m:bar-vec bar)))
      (m:contradictory? (m:vec-dy (m:bar-vec bar)))))

(define (m:bar-length-specified? bar)
  (and (m:specified? (m:bar-length bar) number?)))

(define (m:bar-direction-specified? bar)
  (and (m:specified? (m:bar-direction bar) number?)))

(define (m:bar-length-contradictory? bar)
  (m:contradictory? (m:bar-length bar)))

(define (m:bar-length-dir-specified? bar)
  (and (m:bar-length-specified? bar)
       (m:bar-direction-specified? bar)))

(define (m:bar-fully-specified? bar)
  (and (m:bar-p1-specified? bar)
       (m:bar-p2-specified? bar)))

(define (m:bar-contradictory? bar)
  (or (m:bar-p1-contradictory? bar)
      (m:bar-p2-contradictory? bar)
      (m:bar-direction-contradictory? bar)
      (m:bar-length-contradictory? bar)))

;;;;;;;;;;;;;;;;;;;;;;;;; Joint Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:joint-dir-1-specified? joint)
  (m:specified? (m:joint-dir-1 joint) direction?))

(define (m:joint-dir-1-contradictory? joint)
  (m:contradictory? (m:joint-dir-1 joint)))


(define (m:joint-dir-2-specified? joint)
  (m:specified? (m:joint-dir-2 joint) direction?))

(define (m:joint-dir-2-contradictory? joint)
  (m:contradictory? (m:joint-dir-2 joint)))

(define (m:joint-theta-contradictory? joint)
  (m:contradictory? (m:joint-theta joint)))

(define (m:joint-anchored? joint)
  (or (m:joint-dir-1-specified? joint)
      (m:joint-dir-2-specified? joint)))

(define (m:joint-anchored-and-arm-lengths-specified? joint)
  (and (m:joint-anchored? joint)
       (m:bar-length-specified? (m:joint-arm-1 joint))
       (m:bar-length-specified? (m:joint-arm-2 joint))))

(define (m:joint-specified? joint)
  (m:specified? (m:joint-theta joint) number?))

(define (m:joint-dirs-specified? joint)
  (and
   (m:joint-dir-1-specified? joint)
   (m:joint-dir-2-specified? joint)))

(define (m:joint-fully-specified? joint)
  (and
   (m:point-specified? (m:joint-vertex joint))
   (m:joint-dir-1-specified? joint)
   (m:joint-dir-2-specified? joint)))

(define (m:joint-contradictory? joint)
  (or
   (m:point-contradictory? (m:joint-vertex joint))
   (m:joint-dir-1-contradictory? joint)
   (m:joint-dir-2-contradictory? joint)
   (m:joint-theta-contradictory? joint)))

;;;;;;;;;;;;;;;;;;;;;;;; Specifying Values ;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:joint-theta-if-specified joint)
  (let ((theta-v (m:examine-cell
                  (m:joint-theta joint))))
    (if (number? theta-v) theta-v
        0)))

(define (m:bar-max-inner-angle-sum bar)
  (let ((e1 (m:bar-endpoints-1 bar))
        (e2 (m:bar-endpoints-2 bar)))
    (if (or (null? e1)
            (null? e2))
        0
        (+ (apply max (map m:joint-theta-if-specified e1))
           (apply max (map m:joint-theta-if-specified e2))))))

(define (m:joint-bar-sums joint)
  (let ((b1 (m:joint-arm-1 joint))
        (b2 (m:joint-arm-2 joint)))
    (and (m:bar-length-specified? b1)
         (m:bar-length-specified? b2)
         (+ (m:examine-cell (m:bar-length b1))
            (m:examine-cell (m:bar-length b2))))))

(define (m:random-theta-for-joint joint)
  (let ((theta-range (m:examine-cell (m:joint-theta joint))))
    (if (interval? theta-range)
        (if (close-enuf? (interval-low theta-range)
                         (interval-high theta-range))
            (interval-low theta-range)
            (begin
              (safe-internal-rand-range
               (interval-low theta-range)
               (interval-high theta-range))))
        (error "Attempting to specify theta for joint"))))

(define (m:random-bar-length)
  (internal-rand-range 0.2 1.5))

(define (m:initialize-bar bar)
  (if (not (m:bar-anchored? bar))
      (m:instantiate-point (m:bar-p1 bar) 0 0 'initialize))
  (let ((random-dir (random-direction)))
    (m:instantiate (m:bar-direction bar)
                   random-dir 'initialize)
    (pp `(initializing-bar ,(print (m:bar-name bar))
                           ,(print random-dir)))))

(define (m:initialize-joint joint)
  (m:instantiate-point (m:joint-vertex joint) 0 0 'initialize)
  (pp `(initializing-joint ,(print (m:joint-name joint)))))

;;;;;;;;; Assembling named joints into diagrams ;;;;;;

(define (m:assemble-linkages bars joints)
  (let ((bar-table (m:make-bars-by-name-table bars)))
    (for-each
     (lambda (joint)
       (let ((vertex-name (m:joint-vertex-name joint))
             (dir-1-name (m:joint-dir-1-name joint))
             (dir-2-name (m:joint-dir-2-name joint)))
         (for-each
          (lambda (dir-name)
            (let ((bar (m:find-bar-by-id
                        bar-table
                        (m:bar vertex-name
                               dir-name))))
              (if (eq? bar #f)
                  (error "Could not find bar for" vertex-name dir-name))
              (m:identify-joint-bar-by-name joint bar)))
          (list dir-1-name dir-2-name))))
     joints)))

#|
 ;; Simple example of "solving for the third point"
 (begin
   (initialize-scheduler)
   (let ((b1 (m:make-named-bar 'a 'c))
         (b2 (m:make-named-bar 'b 'c))
         (b3 (m:make-named-bar 'a 'b))
         (j1 (m:make-named-joint 'b 'a 'c))
         (j2 (m:make-named-joint 'c 'b 'a))
         (j3 (m:make-named-joint 'a 'c 'b)))

     (m:assemble-linkages
      (list b1 b2 b3)
      (list j2 j3 j1))

     (m:initialize-joint j1)
     (c:id (m:bar-length b1) (m:bar-length b2))

     (m:instantiate (m:bar-length b3) 6 'b3-len)
     (m:instantiate (m:bar-length b1) 5 'b1-len)
     (run)
     (m:examine-point (m:bar-p2 b1))))
 ;Value: (m:point 3 4)

|#

;;;;;;;;;;;;;;;;;; Converstion to Figure Elements ;;;;;;;;;;;;;;;;;

;;; TODO: Extract dependencies from TMS? or set names

(define (m:point->figure-point m-point)
  (if (not (m:point-specified? m-point))
      (let ((r (m:examine-cell (m:point-region m-point))))
        (m:region->figure-elements r))
      (let ((p (make-point (m:examine-cell (m:point-x m-point))
                           (m:examine-cell (m:point-y m-point)))))
        (set-element-name! p (m:element-name m-point))
        p)))

(define (m:bar->figure-segment m-bar)
  (if (not (m:bar-fully-specified? m-bar))
      #f
      (let ((p1 (m:point->figure-point (m:bar-p1 m-bar)))
            (p2 (m:point->figure-point (m:bar-p2 m-bar))))
        (and (point? p1)
             (point? p2)
             (make-segment p1 p2)))))

```

```
(define (m:joint->figure-angle m-joint)
  (if (not (m:joint-fully-specified? m-joint))
      #f
      (make-angle (m:examine-cell (m:joint-dir-2 m-joint))
                  (m:point->figure-point (m:joint-vertex m-joint))
                  (m:examine-cell (m:joint-dir-1 m-joint)))))
```

Listing A.30: manipulate/region.scm

```
;;; regions.scm --- Region Information

;;; Commentary:

;; Ideas:
;; - Points, Lines, Circles, Intersections
;; - For now, semicircle (joints only go to 180deg to avoid
;;     multiple solns.)

;; Future:
;; - Differentiate regions with 2 deg. of freedom
;; - Improve contradiction objects

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Point Sets ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <m:point-set>
  (%m:make-point-set points)
  m:point-set?
  (points m:point-set-points))

(define (m:make-point-set points)
  (%m:make-point-set points))

(define (m:make-singular-point-set point)
  (m:make-point-set (list point)))

(define (m:in-point-set? p point-set)
  (pair? ((member-procedure point-equal?) p (m:point-set-points
      point-set))))

(define (m:singular-point-set? x)
  (and (m:point-set? x)
       (= 1 (length (m:point-set-points x)))))

(define (m:singular-point-set-point ps)
  (if (not (m:singular-point-set? ps))
      (error "Not a singular point set"))
  (car (m:point-set-points ps)))

(define (m:point-sets-equivalent? ps1 ps2)
  (define delp (delete-member-procedure list-deletor point-equal?))
  (define memp (member-procedure point-equal?))
  (let lp ((points-1 (m:point-set-points ps1))
           (points-2 (m:point-set-points ps2)))
    (if (null? points-1)
        (null? points-2)
        (let ((p1 (car points-1)))
          (if (memp p1 points-2)
              (lp (cdr points-1)
                  (delp p1 points-2))
              #f)))))

(define (m:print-point-set ps)
  (cons 'm:point-set
        (map (lambda (p) (list 'point (point-x p) (point-y p)))
             (m:point-set-points ps))))

(defhandler print
  m:print-point-set m:point-set?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Rays ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <m:ray>
  (%m:make-ray endpoint direction)
  m:ray?
  (endpoint m:ray-endpoint)
  (direction m:ray-direction))

(define m:make-ray %m:make-ray)

(define (m:ray->figure-ray m-ray)
  (with-color "red"
              (make-ray (m:ray-endpoint m-ray)
                        (m:ray-direction m-ray))))

(define (m:on-ray? p ray)
  (let ((endpoint (m:ray-endpoint ray)))
    (or (point-equal? p endpoint)
        (let ((dir (direction-from-points endpoint p)))
          (direction-equal? dir (m:ray-direction ray))))))

(define (m:p2-on-ray ray)
  (add-to-point (m:ray-endpoint ray)
                (unit-vec-from-direction (m:ray-direction ray))))

(define (m:rays-equivalent? ray1 ray2)
  (and (point-equal? (m:ray-endpoint ray1)
                     (m:ray-endpoint ray2))
       (direction-equal? (m:ray-direction ray1)
                         (m:ray-direction ray2))))

(define (m:print-ray ray)
  (let ((endpoint (m:ray-endpoint ray)))
    `(m:ray (,(point-x endpoint)
             ,(point-y endpoint))
            ,(direction-theta (m:ray-direction ray)))))

```

```
(defhandler print
  m:print-ray m:ray?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Arcs ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <m:arc>
  (m:make-arc center-point radius dir-interval)
  m:arc?
  (center-point m:arc-center)
  (radius m:arc-radius)
  (dir-interval m:arc-dir-interval))

;;; Start direction + ccw pi radian
(define (m:make-semi-circle center radius start-direction)
  (m:make-arc center radius
              (make-direction-interval start-direction
                                       (reverse-direction
                                            start-direction))))

(define (m:on-arc? p arc)
  (let ((center-point (m:arc-center arc))
        (radius (m:arc-radius arc)))
    (let ((distance (distance p center-point))
          (dir (direction-from-points center-point p)))
      (and (close-enuf? distance radius)
           (within-direction-interval?
            dir
            (m:arc-dir-interval arc))))))

(define (m:arcs-equivalent? arc1 arc2)
  (and (point-equal? (m:arc-center arc1)
                     (m:arc-center arc2))
       (close-enuf? (m:arc-radius arc1)
                    (m:arc-radius arc2))
       (direction-interval-equal?
        (m:arc-dir-interval arc1)
        (m:arc-dir-interval arc2))))

(define (m:print-arc arc)
  (let ((center-point (m:arc-center arc))
        (dir-interval (m:arc-dir-interval arc)))
    `(m:arc (,(point-x center-point)
            ,(point-y center-point))
           ,(m:arc-radius arc)
           (,(direction-theta (direction-interval-start dir-interval))
            ,(direction-theta (direction-interval-end dir-interval))))))

(defhandler print
  m:print-arc
  m:arc?)

;;;;;;;;;;;;;;;;;;;;;; Contradiction Objects ;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <m:region-contradiction>
  (m:make-region-contradiction error-regions)
  m:region-contradiction?
  (error-regions m:contradiction-error-regions))

;;; TODO: Maybe differeniate by error values
(define (m:region-contradictions-equivalent? rc1 rc2) #t)

(define (m:region-contradiction->figure-elements rc)
  (map m:region->figure-elements (m:contradiction-error-regions rc)))

;;;;;;;;;;;;;;;;;;;;; Specific Intersections ;;;;;;;;;;;;;;;;;;;;;

(define (m:intersect-rays ray1 ray2)
  (let ((endpoint-1 (m:ray-endpoint ray1))
        (endpoint-2 (m:ray-endpoint ray2))
        (dir-1 (m:ray-direction ray1))
        (dir-2 (m:ray-direction ray2)))
    (if (direction-equal? dir-1 dir-2)
        (cond ((m:on-ray? endpoint-1 ray2) ray1)
              ((m:on-ray? endpoint-2 ray1) ray2)
              ;; TODO: Determine error value
              (else (m:make-region-contradiction (list ray1 ray2))))
        (let ((ray1-p2 (m:p2-on-ray ray1))
              (ray2-p2 (m:p2-on-ray ray2)))
          (let ((intersections
                 (intersect-lines-by-points endpoint-1 ray1-p2
                                            endpoint-2 ray2-p2)))
            (if (not (= 1 (length intersections)))
                (m:make-region-contradiction (list ray1 ray2))
                (let ((intersection (car intersections)))
                 (if (and (m:on-ray? intersection ray1)
                          (m:on-ray? intersection ray2))
                     (m:make-point-set (list intersection))
                     ;; TODO: Determine error value
                     (m:make-region-contradiction (list ray1
                          ray2))))))))))

(define (m:intersect-arcs arc1 arc2)
  (let ((c1 (m:arc-center arc1))
        (c2 (m:arc-center arc2))
        (r1 (m:arc-radius arc1))
        (r2 (m:arc-radius arc2)))
    (if (point-equal? c1 c2)
        (if (close-enuf? r1 r2)
            (m:make-arc c1 r1
                        (intersect-direction-intervals
                         (m:arc-dir-interval arc1)
                         (m:arc-dir-interval arc2)))
            (m:make-region-contradiction (list arc1 arc2)))
        (let ((intersections
               (intersect-circles-by-centers-radii
                c1 r1 c2 r2)))
          (let ((points
                 (filter (lambda (p)
```

```scheme
                           (and (m:on-arc? p arc1)
                                (m:on-arc? p arc2)))
                         intersections)))
            (if (> (length points) 0)
                (m:make-point-set points)
                ;; TODO: Determine error value
                (m:make-region-contradiction (list arc1 arc2))))))))

(define (m:intersect-ray-arc ray arc)
  (let ((center (m:arc-center arc))
        (radius (m:arc-radius arc))
        (endpoint (m:ray-endpoint ray))
        (ray-p2 (m:p2-on-ray ray)))
    (let ((intersections
           (intersect-circle-line-by-points
            center radius endpoint ray-p2)))
      (let ((points
             (filter (lambda (p)
                       (and (m:on-ray? p ray)
                            (m:on-arc? p arc)))
                     intersections)))
        (if (> (length points) 0)
            (m:make-point-set points)
            ;; TODO: Determine error value
            (m:make-region-contradiction (list ray arc)))))))

(define (m:intersect-arc-ray arc ray)
  (m:intersect-ray-arc ray arc))

;;;;;;;;;;;;;;;;;;; Intersecting with Point Sets ;;;;;;;;;;;;;;;;;;;

(define m:in-region? (make-generic-operation 2 'm:in-region?))

(defhandler m:in-region? m:in-point-set? point? m:point-set?)
(defhandler m:in-region? m:on-ray? point? m:ray?)
(defhandler m:in-region? m:on-arc? point? m:arc?)
(defhandler m:in-region? (lambda (p r) #f) point?
     m:region-contradiction?)

(define (m:intersect-point-set-with-region ps1 region)
  (let ((results
         (let lp ((points-1 (m:point-set-points ps1))
                  (point-intersections '()))
           (if (null? points-1)
               point-intersections
               (let ((p1 (car points-1)))
                 (if (m:in-region? p1 region)
                     (lp (cdr points-1)
                         (cons p1 point-intersections))
                     (lp (cdr points-1)
                         point-intersections)))))))
    (if (> (length results) 0)
        (m:make-point-set results)
        ;;; TODO: Determine error value
        (m:make-region-contradiction (list ps1 region)))))

(define (m:intersect-region-with-point-set region ps)
  (m:intersect-point-set-with-region ps region))

;;;;;;;;;;;;;;;;;;;; Translating regions by Vec ;;;;;;;;;;;;;;;;;;;;

(define m:translate-region (make-generic-operation 2
     'm:translate-region))

(define (m:translate-point-set ps vec)
  (m:make-point-set
   (map (lambda (p) (add-to-point p vec))
        (m:point-set-points ps))))
(defhandler m:translate-region m:translate-point-set m:point-set? vec?)

(define (m:translate-ray ray vec)
  (m:make-ray
   (add-to-point (m:ray-endpoint ray) vec)
   (m:ray-direction ray)))
(defhandler m:translate-region m:translate-ray m:ray? vec?)

(define (m:translate-arc arc vec)
  (m:make-arc
   (add-to-point (m:arc-center arc) vec)
   (m:arc-radius arc)
   (m:arc-dir-interval arc)))
(defhandler m:translate-region m:translate-arc m:arc? vec?)

;;;;;;;;;;;;;;;; Generic Intersect Regions "Merge" ;;;;;;;;;;;;;;;;

(define m:intersect-regions (make-generic-operation 2
     'm:intersect-regions))

;;; Same Type
(defhandler m:intersect-regions
  m:intersect-rays m:ray? m:ray?)
(defhandler m:intersect-regions
  m:intersect-arcs m:arc? m:arc?)

;;; Arc + Ray
(defhandler m:intersect-regions
  m:intersect-ray-arc m:ray? m:arc?)
(defhandler m:intersect-regions
  m:intersect-arc-ray m:arc? m:ray?)

;;; Point Sets
(defhandler m:intersect-regions
  m:intersect-region-with-point-set any? m:point-set?)
(defhandler m:intersect-regions
  m:intersect-point-set-with-region m:point-set? any?)

;;; Contradictions
```

```
(defhandler m:intersect-regions (lambda (a b) a) m:region-contradiction?
     any?)
(defhandler m:intersect-regions (lambda (a b) b) any?
     m:region-contradiction?)

;;;;;;;;;;;;;;;;;;;;;;;;; Generic Equivalency ;;;;;;;;;;;;;;;;;;;;;;;;;

(define m:region-equivalent?
  (make-generic-operation 2 'm:region-equivalent? (lambda (a b) #f)))

(defhandler m:region-equivalent?
  m:point-sets-equivalent? m:point-set? m:point-set?)

(defhandler m:region-equivalent?
  m:rays-equivalent? m:ray? m:ray?)

(defhandler m:region-equivalent?
  m:arcs-equivalent? m:arc? m:arc?)

(defhandler m:region-equivalent?
  m:region-contradictions-equivalent?
  m:region-contradiction?
  m:region-contradiction?)

;;;;;;;;;;;;;;;;;;; Interface to Propagator System ;;;;;;;;;;;;;;;;;;;

(define (m:region? x)
  (or (m:point-set? x)
      (m:ray? x)
      (m:arc? x)
      (m:region-contradiction? x)))


(defhandler equivalent? m:region-equivalent? m:region? m:region?)

(defhandler merge m:intersect-regions m:region? m:region?)

(defhandler contradictory? m:region-contradiction? m:region?)

#|
 Simple Examples
 (pp (let-cells (c)
    (add-content c (m:make-arc (make-point 1 0) (sqrt 2)
                               (make-direction-interval
                                (make-direction (/ pi 8))
                                (make-direction (* 7 (/ pi 8))))))

    (add-content c (m:make-ray (make-point -3 1) (make-direction 0)))
    (add-content c (m:make-ray (make-point 1 2)
                 (make-direction (* 7 (/ pi 4)))))
    (content c)))

 (let ((a (make-point 0 0))
       (b (make-point 1 0))
       (c (make-point 0 1))
       (d (make-point 1 1)))
     (let-cells (cell)
       (add-content cell
                    (make-tms
                     (contingent (m:make-point-set (list a b c))
                                 '(a))))
       (add-content cell
                    (make-tms
                     (contingent (m:make-point-set (list a d))
                                 '(a))))
       (pp (tms-query (content cell)))))
|#
;;;;;;;;;;;;;;;;;;;;;;;;; To Figure elements ;;;;;;;;;;;;;;;;;;;;;;;;

(define m:region->figure-elements
  (make-generic-operation 1 'm:region->figure-elements (lambda (r) #f )))

(defhandler m:region->figure-elements
  m:ray->figure-ray
  m:ray?)

(defhandler m:region->figure-elements
  m:region-contradiction->figure-elements
  m:region-contradiction?)
```

Listing A.31: manipulate/constraints.scm

```
;;; constraints.scm --- Constraints for mechanisms

;;; Commentary:

;; Ideas:
;; - Abstraction for specifying constraints
;; - Length, angle equality
;; - Perpendicular / Parellel

;; Future:
;; - Constraints for other linkages?

;;; Code:

;;;;;;;;;;;;;;;;;;;;;; Constraint Structure ;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <m:constraint>
  (m:make-constraint type args constraint-procedure)
  m:constraint?
  (type m:constraint-type)
  (args m:constraint-args)
  (constraint-procedure m:constraint-procedure))

;;;;;;;;;;;;;;;;;;;;;;;;; Constraint Types ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:c-length-equal bar-id-1 bar-id-2)
  (m:make-constraint
   'm:c-length-equal
   (list bar-id-1 bar-id-2)
   (lambda (m)
     (let ((bar-1 (m:lookup m bar-id-1))
           (bar-2 (m:lookup m bar-id-2)))
       (c:id
        (m:bar-length bar-1)
        (m:bar-length bar-2))))))

(define (m:c-angle-equal joint-id-1 joint-id-2)
  (m:make-constraint
   'm:c-angle-equal
   (list joint-id-1 joint-id-2)
   (lambda (m)
     (let ((joint-1 (m:lookup m joint-id-1))
           (joint-2 (m:lookup m joint-id-2)))
       (c:id (m:joint-theta joint-1)
             (m:joint-theta joint-2))))))

(define (m:c-right-angle joint-id)
  (m:make-constraint
   'm:right-angle
   (list joint-id)
   (lambda (m)
     (let ((joint (m:lookup m joint-id)))
       (c:id
        (m:joint-theta joint)
        (/ pi 2))))))

;;; p2 between p1 p3 in a line
(define (m:c-line-order p1-id p2-id p3-id)
  (list
   (m:make-named-bar p1-id p2-id)
   (m:make-named-bar p2-id p3-id)
   (m:make-named-joint p1-id p2-id p3-id)
   (m:c-full-angle (m:joint p1-id p2-id p3-id))))

(define (m:c-full-angle joint-id)
  (m:make-constraint
   'm:full-angle
   (list joint-id)
   (lambda (m)
     (let ((joint (m:lookup m joint-id)))
       (c:id
        (m:joint-theta joint)
        pi)))))

(define (m:equal-joints-in-sum equal-joint-ids
                               all-joint-ids
                               total-sum)
  (m:make-constraint
   'm:equal-joints-in-sum
   all-joint-ids
   (lambda (m)
     (let ((all-joints (m:multi-lookup m all-joint-ids))
           (equal-joints (m:multi-lookup m equal-joint-ids)))
       (let ((other-joints
              (set-difference all-joints equal-joints eq?)))
         (c:id (m:joint-theta (car equal-joints))
               (ce:/
                (ce:- total-sum
                      (ce:multi+ (map m:joint-theta other-joints)))
                (length equal-joints))))))))

(define (n-gon-angle-sum n)
  (* n (- pi (/ (* 2 pi) n))))

(define (m:polygon-sum-slice all-joint-ids)
  (m:make-slice
   (m:make-constraint
    'm:joint-sum
    all-joint-ids
    (lambda (m)
      (let ((all-joints (m:multi-lookup m all-joint-ids))
            (total-sum (n-gon-angle-sum (length all-joint-ids))))
        (m:joints-constrained-in-sum all-joints total-sum))))))

;;;;;;;;;;;;; Applying and Marking Constrained Elements ;;;;;;;;;;;;;

(define (m:constrained? element)
  (not (null? (m:element-constraints element))))

(define (m:element-constraints element)
  (or (eq-get element 'm:constraints)
      '()))

(define (m:set-element-constraints! element constraints)
  (eq-put! element 'm:constraints constraints))

(define (m:mark-constraint element constraint)
  (m:set-element-constraints!
   element
   (cons constraint
         (m:element-constraints element))))

(define (m:apply-constraint m constraint)
  (for-each (lambda (element-id)
              (m:mark-constraint
               (m:lookup m element-id)
               constraint))
            (m:constraint-args constraint))
  ((m:constraint-procedure constraint) m))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Slices ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Slices are constraints that are processed once the normal
;;; constraints have been aplied.
```

```

(define-record-type <m:slice>
  (m:make-slice constraint)
  m:slice?
  (constraint m:slice-constraint))

(define (m:apply-slice m slice)
  (m:apply-constraint m (m:slice-constraint slice)))

;;;;;;;;;;;;;;;;;;;;;;;; Propagator Utils ;;;;;;;;;;;;;;;;;;;;;;;;

(define (ce:multi+ cells)
  (cond ((null? cells) 0)
        ((null? (cdr cells)) (car cells))
        (else
         (ce:+ (car cells)
               (ce:multi+ (cdr cells))))))

;;;;;;;;;;;;;;;;;;;;;;; Slices (for sums) ;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:equal-values-in-sum equal-cells all-cells total-sum)
  (let ((other-values (set-difference all-cells equal-cells eq?)))
    (c:id (car equal-cells)
          (ce:/
           (ce:- total-sum
                 (ce:multi+ other-values))
           (length equal-cells)))))

(define (m:sum-slice elements cell-transformer equality-predicate
     total-sum)
  (let ((equivalence-classes
         (partition-into-equivalence-classes elements
              equality-predicate))
        (all-cells (map cell-transformer elements)))
    (cons
     (c:id total-sum
           (ce:multi+ all-cells))
     (filter identity
             (map (lambda (equiv-class)
                    (and (> (length equiv-class) 1)
                         (begin
                           (m:equal-values-in-sum
                            (map cell-transformer equiv-class)
                            all-cells
                            total-sum))))
                  equivalence-classes)))))

(define (angle-equal-constraint? c)
  (eq? (m:constraint-type c) 'm:c-angle-equal))

(define (m:joints-constrained-equal-to-one-another? joint-1 joint-2)
  (let ((joint-1-constraints
         (filter angle-equal-constraint?
                 (m:element-constraints joint-1)))
        (joint-2-constraints
         (filter angle-equal-constraint?
                 (m:element-constraints joint-2))))
    (not (null? (set-intersection joint-1-constraints
                                  joint-2-constraints
                                  (member-procedure eq?))))))

(define (m:joints-constrained-in-sum all-joints total-sum)
  (m:sum-slice
   all-joints
   m:joint-theta
   m:joints-constrained-equal-to-one-another?
   total-sum))
```

Listing A.32: manipulate/topology.scm

```
;;; topology.scm --- Helpers for establishing topology for mechanism

;;; Commentary:

;; Ideas:
;; - Simplify listing out all bar and joint orderings
;; - Start with basic polygons, etc.

;; Future:
;; - Figure out making multi-in/out joints: (all pairs?)

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;; Establish-topology ;;;;;;;;;;;;;;;;;;;;;;;

;;; CCW point names
(define (m:establish-polygon-topology . point-names)
  (if (< (length point-names) 3)
      (error "Min polygon size: 3"))
  (let ((extended-point-names
         (append point-names
                 (list (car point-names) (cadr point-names)))))
    (let ((bars
           (map (lambda (p1-name p2-name)
                  (m:make-named-bar p1-name p2-name))
                point-names
                (cdr extended-point-names)))
          (joints
           (map (lambda (p1-name vertex-name p2-name)
                  (m:make-named-joint p1-name vertex-name p2-name))
                (cddr extended-point-names)
                (cdr extended-point-names)
                point-names)))
      (append bars joints
              (list (m:polygon-sum-slice
                     (map m:joint-name joints)))))))
```

Listing A.33: manipulate/mechanism.scm

```scheme
;;; mechanism.scm --- Group of Bars / Joints

;;; Commentary:

;; Ideas:
;; - Grouping of bars and joints
;; - Integrate with establishing toplogy

;; Future:
;; - Also specify constraints with it
;; - Convert to Diagram

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;; Mechanism Structure ;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <m:mechanism>
    (%m:make-mechanism bars joints constraints slices
                       bar-table joint-table joint-by-vertex-table)
    m:mechanism?
    (bars m:mechanism-bars)
    (joints m:mechanism-joints)
    (constraints m:mechanism-constraints)
    (slices m:mechanism-slices)
    (bar-table m:mechanism-bar-table)
    (joint-table m:mechanism-joint-table)
    (joint-by-vertex-table m:mechanism-joint-by-vertex-table))

(define (m:make-mechanism bars joints constraints slices)
  (let ((bar-table (m:make-bars-by-name-table bars))
        (joint-table (m:make-joints-by-name-table joints))
        (joint-by-vertex-table (m:make-joints-by-vertex-name-table
             joints)))
    (%m:make-mechanism bars joints constraints slices
                       bar-table joint-table joint-by-vertex-table)))

(define (m:mechanism . args)
  (let ((elements (flatten args)))
    (let ((bars (m:dedupe-bars (filter m:bar? elements)))
          (joints (filter m:joint? elements))
          (constraints (filter m:constraint? elements))
          (slices (filter m:slice? elements)))
      (m:make-mechanism bars joints constraints slices))))

(define (m:print-mechanism m)
  `((bars ,(map print (m:mechanism-bars m)))
    (joints ,(map print (m:mechanism-joints m)))
    (constraints ,(map print (m:mechanism-constraints m)))))

(defhandler print m:print-mechanism m:mechanism?)

;;;;;;;;;;;;;;;;;;;;;;;;;; Deduplication ;;;;;;;;;;;;;;;;;;;;;;;;;;;


(define (m:dedupe-bars bars)
  (dedupe (member-procedure m:bars-name-equivalent?) bars))


;;;;;;;;;;;;;;;;;;;;;;;;;;;; Accessors ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:mechanism-joint-by-vertex-name m vertex-name)
  (m:find-joint-by-vertex-name
   (m:mechanism-joint-by-vertex-table m)
   vertex-name))

(define (m:mechanism-joint-by-names m dir-1-name vertex-name dir-2-name)
  (m:find-joint-by-names
   (m:mechanism-joint-table m)
   dir-1-name vertex-name dir-2-name))

(define (m:multi-lookup m ids)
  (map (lambda (id) (m:lookup m id)) ids))

(define (m:lookup m id)
  (cond ((m:bar-id? id) (m:find-bar-by-id
                         (m:mechanism-bar-table m)
                         id))
        ((m:joint-id? id) (m:find-joint-by-id
                           (m:mechanism-joint-table m)
                           id))
        ((m:joint-vertex-id? id) (m:find-joint-by-vertex-name
                                  (m:mechanism-joint-by-vertex-table m)
                                  (m:joint-vertex-id-name id)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Specified ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:mechanism-fully-specified? mechanism)
  (and (every m:bar-fully-specified? (m:mechanism-bars mechanism))
       (every m:joint-fully-specified? (m:mechanism-joints mechanism))))

(define (m:mechanism-contradictory? mechanism)
  (or (any m:bar-contradictory? (m:mechanism-bars mechanism))
      (any m:joint-contradictory? (m:mechanism-joints mechanism))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Specify ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Should these be in Linkages?

(define *any-dir-specified* #f)
(define *any-point-specified* #f)

(define (any-one l)
  (let ((i (random (length l))))
    (list-ref l i)))

(define (m:pick-bar bars)
  (car (sort-by-key bars (negatep m:bar-max-inner-angle-sum))))
```

```scheme

(define m:pick-joint-1 any-one)

(define (m:pick-joint joints)
  (car
   (append
    (sort-by-key
     (filter m:joint-bar-sums joints)
     m:joint-bar-sums)
    (filter (notp m:joint-bar-sums) joints))))

(define (m:specify-angle-if-first-time cell)
  (if (not *any-dir-specified*)
      (let ((dir (random-direction)))
        (set! *any-dir-specified* #t)
        (pp `(initializing-direction ,(name cell) ,(print dir)))
        (m:instantiate cell dir 'first-time-angle))))

(define (m:specify-point-if-first-time point)
  (if (not *any-point-specified*)
      (begin
        (set! *any-point-specified* #t)
        (pp `(initializing-point ,(name point) (0 0)))
        (m:instantiate-point point 0 0 'first-time-point))))

(define (m:specify-bar bar)
  (let ((v (m:random-bar-length)))
    (pp `(specifying-bar-length ,(print (m:bar-name bar)) ,v))
    (m:instantiate (m:bar-length bar) v 'specify-bar)
    (m:specify-angle-if-first-time (m:bar-direction bar))
    (m:specify-point-if-first-time (m:bar-p1 bar))))

(define (m:specify-joint joint)
  (let ((v (m:random-theta-for-joint joint)))
    (pp `(specifying-joint-angle ,(print (m:joint-name joint)) ,v))
    (m:instantiate (m:joint-theta joint) v 'specify-joint)
    (m:specify-angle-if-first-time (m:joint-dir-1 joint))))

(define (m:initialize-joint-vertex joint)
  (m:specify-point-if-first-time (m:joint-vertex joint)))

(define (m:initialize-joint-direction joint)
  (m:specify-angle-if-first-time (m:joint-dir-1 joint)))

(define (m:initialize-bar-p1 bar)
  (m:specify-point-if-first-time (m:bar-p1 bar)))

(define (m:specify-joint-if m predicate)
  (let ((joints (filter (andp predicate (notp m:joint-specified?))
                        (m:mechanism-joints m))))
    (and (not (null? joints))
         (m:specify-joint (m:pick-joint joints)))))

(define (m:initialize-joint-if m predicate)
  (let ((joints (filter (andp predicate (notp m:joint-specified?))
                        (m:mechanism-joints m))))
    (and (not (null? joints))
         (let ((j (m:pick-joint joints)))
           (m:initialize-joint-direction j)))))

(define (m:specify-bar-if m predicate)
  (let ((bars (filter (andp predicate (notp m:bar-length-specified?))
                      (m:mechanism-bars m))))
    (and (not (null? bars))
         (m:specify-bar (m:pick-bar bars)))))

(define (m:initialize-bar-if m predicate)
  (let ((bars (filter (andp predicate (notp m:bar-length-specified?))
                      (m:mechanism-bars m))))
    (and (not (null? bars))
         (m:initialize-bar-p1 (m:pick-bar bars)))))

(define (m:specify-something m)
  (or
   (m:specify-bar-if m m:constrained?)
   (m:specify-joint-if m m:constrained?)
   (m:specify-joint-if m m:joint-anchored-and-arm-lengths-specified?)
   (m:specify-joint-if m m:joint-anchored?)
   (m:specify-bar-if m m:bar-directioned?)
   (m:specify-bar-if m m:bar-anchored?)
   (m:initialize-joint-if m m:joint-dirs-specified?)
   (m:initialize-bar-if m m:bar-length-dir-specified?)
   (m:initialize-bar-if m m:bar-direction-specified?)
   (m:initialize-bar-if m m:bar-length-specified?)
   (m:initialize-joint-if m m:joint-anchored?)
   (m:initialize-joint-if m true-proc)
   (m:initialize-bar-if m true-proc)))

;;;;;;;;;;;;;;;;;;;;;; Applying constraints ;;;;;;;;;;;;;;;;;;;;;;

(define (m:apply-mechanism-constraints m)
  (for-each (lambda (c)
              (m:apply-constraint m c))
            (m:mechanism-constraints m)))

(define (m:apply-slices m)
  (for-each (lambda (s)
              (m:apply-slice m s))
            (m:mechanism-slices m)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Build ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:identify-vertices m)
  (for-each (lambda (joints)
              (let ((first-vertex (m:joint-vertex (car joints))))
                (for-each (lambda (joint)
                            (m:identify-points first-vertex
                                               (m:joint-vertex joint)))
```

```
                          (cdr joints))))
            (hash-table/datum-list (m:mechanism-joint-by-vertex-table
                 m))))

(define (m:build-mechanism m)
  (m:identify-vertices m)
  (m:assemble-linkages (m:mechanism-bars m)
                       (m:mechanism-joints m))
  (m:apply-mechanism-constraints m)
  (m:apply-slices m))

(define (m:initialize-solve)
  (set! *any-dir-specified* #f)
  (set! *any-point-specified* #f))

(define *m* #f)
(define (m:solve-mechanism m)
  (set! *m* m)
  (m:initialize-solve)
  (let lp ()
    (run)
    (cond ((m:mechanism-contradictory? m)
           (m:draw-mechanism m c)
           #f)
          ((not (m:mechanism-fully-specified? m))
           (if (m:specify-something m)
               (lp)
               (error "Couldn't find anything to specify.")))
          (else 'mechanism-built))))

(define (m:solve-mechanism-new m)
  (set! *m* m)
  (m:initialize-solve))

(define (m:specify-something-new m fail)
  (let ((linkages (append (m:mechanism-bars m)
                          (m:mechanism-joints m))))
    (let lp ((linkages (sort-linknages linkages)))
      (if (null? linkages)
          (fail)
          (let ((first-linkage (car linkages))
                (other-linkages (cdr linkages)))
            (m:specify-linkage m first-linkage
                               (lambda ()
                                 (lp (cdr linkages))))))))))

#|
 (begin
   (initialize-scheduler)
   (m:build-mechanism
    (m:mechanism
     (m:establish-polygon-topology 'a 'b 'c))))
|#

;;;;;;;;;;;;;;;;;;;;;;;; Conversion to Figure ;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:mechanism->figure m)
  (let ((points
         (map (lambda (joint)
                (m:point->figure-point (m:joint-vertex joint)))
              (m:mechanism-joints m)))
        (segments (map m:bar->figure-segment (m:mechanism-bars m)))
        (angles (map m:joint->figure-angle (m:mechanism-joints m))))
    (apply figure (flatten (filter (lambda (x) (or x))
                           (append points segments angles))))))

(define (m:draw-mechanism m c)
  (draw-figure (m:mechanism->figure m) c))

#|
(let lp ()
  (initialize-scheduler)
  (let ((m (m:mechanism
            (m:establish-polygon-topology 'a 'b 'c 'd))))
    (pp (m:joint-anchored? (car (m:mechanism-joints m))))
    (m:build-mechanism m)
    (m:solve-mechanism m)
    (let ((f (m:mechanism->figure m)))
      (draw-figure f c)
      (pp (analyze-figure f)))))
|#
```

Listing A.34: manipulate/main.scm

```
;;; main.scm --- Main definitions and code for running the
;;; manipulation / mechanism-based code

;;; Examples

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

(define (arbitrary-right-triangle)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)
   (m:c-right-angle (m:joint 'a))))

(define (arbitrary-right-triangle-2)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)
   (m:c-right-angle (m:joint 'c))))

(define (quadrilateral-with-diagonals a b c d)
  (list
   (m:establish-polygon-topology a b c d)
   (m:establish-polygon-topology a b c)
   (m:establish-polygon-topology b c d)
   (m:establish-polygon-topology c d a)
   (m:establish-polygon-topology d a c)))

(define (quadrilateral-with-diagonals-intersection a b c d e)
  (list
   (quadrilateral-with-diagonals a b c d)
   (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)))

(define (quad-diagonals)
  (m:mechanism
   ;; Setup abcd with e in the middle:
   ;(quadrilateral-with-diagonals-intersection 'a 'b 'c 'd 'e)

   (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)

   ;; Right Angle in Center:
   (m:c-right-angle (m:joint 'b 'e 'c))

   ;; Diagonals Equal
   ;;(m:c-length-equal (m:bar 'c 'a) (m:bar 'b 'd))
   (m:c-length-equal (m:bar 'c 'e) (m:bar 'a 'e))
   ;;(m:c-length-equal (m:bar 'b 'e) (m:bar 'd 'e))

   ;; Make it a square:
   ;;(m:c-length-equal (m:bar 'c 'e) (m:bar 'b 'e))
   ))

;;; Works:
(define (isoceles-triangle)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'b 'c))))

(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:equal-joints-in-sum
    (list (m:joint 'a) (m:joint 'b))
    (list (m:joint 'a) (m:joint 'b) (m:joint 'c))
    pi)))

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

;;; Often works:
(define (arbitrary-quadrilateral)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)))

;;; Always works:
(define (parallelogram-by-sides)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'c 'd))
   (m:c-length-equal (m:bar 'b 'c)
                     (m:bar 'd 'a))))

(define (kite-by-sides)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'b 'c))
   (m:c-length-equal (m:bar 'c 'd)
                     (m:bar 'd 'a))))

(define (kite-by-angles-sides)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'a 'd))
   (m:c-angle-equal (m:joint 'b)
                    (m:joint 'd))))

(define (rhombus-by-sides)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'b 'c))
   (m:c-length-equal (m:bar 'b 'c)
                     (m:bar 'c 'd))
   (m:c-length-equal (m:bar 'c 'd)
                     (m:bar 'a 'd))))

(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))))

```

```
(define *m*)
(define (m:run-mechanism mechanism-proc)
  (initialize-scheduler)
  (let ((m (mechanism-proc)))
    (set! *m* m)
    (m:build-mechanism m)
    (if (not (m:solve-mechanism m))
        (pp "Unsolvable!")
        (let ((f (m:mechanism->figure m)))
          (draw-figure f c)
          ;;(pp (analyze-figure f))
          ))))

#|
 (let lp ()
   (initialize-scheduler)
   (pp 'start)
   (m:run-mechanism
    (lambda ()
      (m:mechanism
       ;;(m:establish-polygon-topology 'a 'b 'c)
       (m:make-named-bar 'a 'b)
       (m:make-named-bar 'b 'c)
       (m:make-named-bar 'c 'a)
       (m:make-named-joint 'c 'b 'a)
       (m:make-named-joint 'a 'c 'b)
       (m:make-named-joint 'b 'a 'c)

       (m:make-named-bar 'a 'd)
       (m:make-named-bar 'b 'd)
       (m:make-named-joint 'd 'a 'b)
       (m:make-named-joint 'a 'b 'd)
       (m:make-named-joint 'b 'd 'a)

       (m:make-named-bar 'c 'd)
       (m:make-named-joint 'a 'd 'c)
       (m:make-named-joint 'c 'a 'd)
       (m:make-named-joint 'd 'c 'a))))
   (lp))

 (let lp ()
   (initialize-scheduler)
   (let ((m (m:mechanism
             (m:establish-polygon-topology 'a 'b 'c 'd))))
     (m:build-mechanism m)
     (m:solve-mechanism m)
     (let ((f (m:mechanism->figure m)))
       (draw-figure f c)
       (pp (analyze-figure f)))))
|#

(define (rect-demo-1)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'b 'c))
   (m:c-right-angle (m:joint 'd))))

(define (rect-demo-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))))

(define (rect-demo-3)
  (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-right-angle (m:joint 'b))))
```

Listing A.35: learning/load.scm

```
;;; load.scm -- Load learning module
(for-each (lambda (f) (load f))
          '("lattice"
            "definitions"
            "student"
            "conjecture"
            "simplifier"
            "core-knowledge"))
```

Listing A.36: learning/core-knowledge.scm

```
;;; core-knowledge.scm -- Core knowledge of a student

;;; Commentary:

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;; Adding to student ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (provide-core-knowledge student)
  (for-each (lambda (def)
              (add-definition! student def))
            primitive-definitions)
  (for-each (lambda (def)
              (add-definition! student def))
            built-in-definitions))

;;;;;;;;;;;;;;;;;;;;;;;;; Primitive definitions ;;;;;;;;;;;;;;;;;;;;;;;;

(define primitive-definitions
  (list
   (make-primitive-definition 'point point? random-point)
   (make-primitive-definition 'line line? random-line)
   (make-primitive-definition 'segment segment? random-segment)
   (make-primitive-definition 'polygon polygon? random-polygon)
   (make-primitive-definition 'circle circle? random-circle)
   (make-primitive-definition 'angle angle? random-angle)))

;;;;;;;;;;;;;;;;;;;;;;; Built-in Definitions ;;;;;;;;;;;;;;;;;;;;;;;

(define (polygon-n-sides-conjecture n)
  (make-conjecture
   '(polygon)
   '(<premise>)
   (list car)
   (make-polygon-n-sides-relationship n)))

(define built-in-definitions
  (list
   ;; Triangle
   (make-restrictions-definition
    'triangle '(polygon)
    (list (polygon-n-sides-conjecture 3))
    random-triangle)
   ;; Quadrilateral
   (make-restrictions-definition
    'quadrilateral '(polygon)
    (list (polygon-n-sides-conjecture 4))
    random-quadrilateral)

   ;; Isoceles Triangle!
   #|
   (make-restrictions-definition
    'isoceles-triangle 'triangle
    (list (lambda (t)
            (let* ((a (polygon-point-ref t 0))
                   (b (polygon-point-ref t 1))
                   (c (polygon-point-ref t 2)))
              (segment-equal-length? (make-segment a b)
                                     (make-segment a c)))))

   random-isoceles-triangle))
|#
   ))
```

Listing A.37: learning/lattice.scm

```
;;; lattice.scm -- code for general lattice

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Nodes ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <lattice-node>
  (%make-lattice-node key content parents children)
  lattice-node?
  (key lattice-node-key)
  (content lattice-node-content)
  (parents lattice-node-parents set-lattice-node-parents!)
  (children lattice-node-children set-lattice-node-children!))

(define (make-lattice-node key content)
  (%make-lattice-node key content '() '()))

(define (add-lattice-node-parent! node parent-node)
  (set-lattice-node-parents!
   node
   (cons parent-node (lattice-node-parents node))))

(define (add-lattice-node-child! node child-node)
  (set-lattice-node-children!
   node
   (cons child-node (lattice-node-children node))))

(define (add-lattice-node-children! node children-nodes)
  (for-each
   (lamdba (child)
           (add-lattice-node-child! node child))))

(define (print-lattice-node node)
  (list (lattice-node-key node)
        (lattice-node-content node)
        (map lattice-node-key (lattice-node-parents node))
        (map lattice-node-key (lattice-node-children node))))

(defhandler print print-lattice-node lattice-node?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Lattice ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Partial-order-proc is a procedure on keys that returns true if the
;;; first argument is a parent of "above" the second in the lattice

(define-record-type <lattice>
  (%make-lattice partial-order-proc root)
  lattice?
  (partial-order-proc lattice-partial-order-proc)
  (root lattice-root))

(define (make-lattice partial-order-proc root)
  (define (node-partial-order-proc parent-node child-node)
    (partial-order-proc
     (lattice-node-content parent-node)
     (lattice-node-content child-node)))
  (%make-lattice node-partial-order-proc root))

(define (add-to-lattice lattice new-node)
  (let ((visited '()))
    (define (visited? node)
      (memq (lattice-node-key node) visited))
    (define (mark-visited node)
      (set! visited (cons node visited)))
    (define (ancestor-of-new-node? node)
      ((lattice-partial-order-proc lattice)
       node new-node))
    (define (descendent-of-new-node? node)
      ((lattice-partial-order-proc lattice)
       new-node node))
    (define (get-unvisited-children node)
      (let ((unvisited-children
             (filter (notp visited?)
                     (lattice-node-children node))))
        (for-each mark-visited unvisited-children)
        unvisited-children))
    (define (save-as-parent parent-node)
      (add-lattice-node-parent! new-node parent-node)
      (let lp ((agenda (list parent-node)))
        (if (null? agenda) 'done
            (let ((node (car agenda)))
              (let ((unvisited-children
                     (get-unvisited-children node)))
                (let ((descendent-children
                       (filter descendent-of-new-node?
                               unvisited-children))
                      (nondescendent-children
                       (filter (notp descendent-of-new-node?)
                               unvisited-children)))
                  (add-lattice-node-children!
                   new-node descendent-children)
                  (lp (append (cdr agenda)
                              nondescendent-children))))))))
    (define (clean-children node)
      (let ((children (dedupe-by eq? (lattice-node-children node))))
        (set-lattice-node-children!
         node
         (remove-supplants
          (lattice-partial-order-proc lattice)
          children))))
    (define (clean-parents node)
      (let ((parents (dedupe-by eq? (lattice-node-parents node))))
        (set-lattice-node-parents!
         node
         (remove-supplants
          (flip-args (lattice-partial-order-proc lattice))
          parents))))
    (define (update-parent-child-pointers)
      (let ((parents-of-new-node (lattice-node-parents new-node))
            (children-of-new-node (lattice-node-children new-node)))
        (for-each (lambda (parent-node)
                    (set-lattice-node-children!
                     parent-node
                     (set-difference
                      (cons new-node (lattice-node-children parent-node))
                      children-of-new-node
                      eq?))
                    (clean-children parent-node))
                  parents-of-new-node)
        (for-each (lambda (child-node)
                    (set-lattice-node-parents!
                     child-node
                     (set-difference
                      (cons new-node (lattice-node-parents child-node))
                      parents-of-new-node
                      eq?))
                    (clean-parents child-node))
                  children-of-new-node)
        (clean-children new-node)
        (clean-parents new-node)
        ))
    (let lp ((agenda (list (lattice-root lattice))))
      (if (null? agenda)
          (update-parent-child-pointers)
          (let ((node (car agenda)))
            (let ((unvisited-children
                   (get-unvisited-children
                    node)))
              (let ((ancestor-children
                     (filter ancestor-of-new-node?
                             unvisited-children)))
                (if (null? ancestor-children)
                    (begin (save-as-parent node)
                           (lp (cdr agenda)))
                    (lp (append (cdr agenda)
                                ancestor-children)))))))))

;;; Example:

#|
(let* ((root (make-lattice-node 'root '()))
       (lattice (make-lattice eq-subset? root))
       (a (make-lattice-node 'a '(1)))
       (b (make-lattice-node 'b '(2)))
       (c (make-lattice-node 'c '(3)))
       (d (make-lattice-node 'd '(1 2)))
       (e (make-lattice-node 'e '(1 3)))
       (f (make-lattice-node 'f '(2 3 4)))
       (g (make-lattice-node 'g '(1 2 3)))
       (h (make-lattice-node 'h '(1 2 3 4))))
  (add-to-lattice lattice c)
  (add-to-lattice lattice h)
  (add-to-lattice lattice f)
  (add-to-lattice lattice e)
  (add-to-lattice lattice g)
  (add-to-lattice lattice a)
  (add-to-lattice lattice d)
  (add-to-lattice lattice b)
  (pprint root)
  (pprint a)
  (pprint b)
  (pprint c)
  (pprint d)
  (pprint e)
  (pprint f)
  (pprint g)
  (pprint h))

; ->
(root () () (a c b))
(a (1) (root) (e d))
(b (2) (root) (d f))
(c (3) (root) (f e))
(d (1 2) (a b) (g))
(e (1 3) (c a) (g))
(f (2 3 4) (c b) (h))
(g (1 2 3) (d e) (h))
(h (1 2 3 4) (g f) ())
|#
```

Listing A.38: learning/definitions.scm

```
;;; definitions.scm --- representation and interaction with definitions

;;; Commentary:

;; Ideas:
;; - primitive definitions

;; Future:
;; - relationship-based definitions

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;; Basic Structure ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <definition>
  (%make-definition name classifications conjectures predicate
       generator)
  definition?
  (name definition-name)
  (classifications definition-classifications)
  (conjectures definition-conjectures)
  (predicate definition-predicate set-definition-predicate!)
  (generator definition-generator))

(define (make-primitive-definition name predicate generator)
  (%make-definition name '() '()  predicate generator))

(define (primitive-definition? def)
  (and (definition? def)
       (null? (definition-classifications def))))

;;;;;;;;;;;;;;;;;;;;;; Higher-order Definitions ;;;;;;;;;;;;;;;;;;;;;;

(define (make-restrictions-definition
         name classifications conjectures generator)
  (%make-definition name classifications conjectures #f generator))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Formatting ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (print-definition def)
  (list (definition-name def)
        (definition-classifications def)
        (map print (definition-conjectures def))))

(defhandler print print-definition
  definition?)

(define (print-primitive-definition def)
  'primitive-definition)

(defhandler print print-primitive-definition
  primitive-definition?)
```

Listing A.39: learning/conjecture.scm

```
;; conjecture -- a proposed conjecture based on an observed relationship

;;; Commentary

;; Ideas:
;; - Higher-level than raw observations reported by perception/analyzer

;; Future:
;; - More complicated premises
;; - "Pattern-matching"

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Conjecture ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <conjecture>
  (make-conjecture premises constructions construction-procedures
                   relationship)
  conjecture?
  (premises conjecture-premises)
  (constructions conjecture-constructions)
  (construction-procedures conjecture-construction-procedures)
  (relationship conjecture-relationship))

(define (print-conjecture conj)
  (cons
   (print (conjecture-relationship conj))
   (conjecture-constructions conj)))

(defhandler print print-conjecture conjecture?)
```

```scheme

(define (conjecture-equal? conj1 conj2)
  (equal? (print conj1)
          (print conj2)))

;;; Whether
(define (satisfies-conjecture conj premise-instances)
  (let ((new-args
         (map
          (lambda (construction-proc)
            (construction-proc premise-instances))
          (conjecture-construction-procedures conj)))
        (rel (conjecture-relationship conj)))
    (or (relationship-holds rel new-args)
        #f
        (begin (if *explain*
                   (pprint `(failed-conjecture ,conj)))
               #f))))


(define (conjecture-from-observation obs)
  (make-conjecture
   '()
   (map element-dependencies->list (observation-args obs))
   (map element-source (observation-args obs))
   (observation-relationship obs)))


;;; Removing redundant conjectures

(define (simplify-conjectures conjectures base-conjectures)
  (define memp (member-procedure conjecture-equal?))
  (filter
   (lambda (o) (not (memp o base-conjectures)))
   conjectures))
```

Listing A.40: learning/simplifier.scm

```scheme
;;; simplifier.scm --- simplifies definitions

;;; Commentary:

;; Ideas:
;; - interfaces to manipulator

;; Future:
;; - Support more complex topologies.

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;; Main Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (observations->constraints observations)
  (filter identity (map observation->constraint observations)))

(define (observation->constraint obs)
  (let ((rel (observation-relationship obs))
        (args (observation-args obs)))
    (let ((constraint-proc (relationship->constraint rel))
          (linkage-ids (args->linkage-ids args)))
      (and constraint-proc
           (every identity linkage-ids)
           (apply constraint-proc
                  (args->linkage-ids args))))))

(define (relationship->constraint rel)
  (case (relationship-type rel)
    ((equal-length) m:c-length-equal)
    ((equal-angle) m:c-angle-equal)
    (else #f)))

(define (args->linkage-ids args)
  (map arg->linkage-id args))

(define arg->linkage-id (make-generic-operation 1 'arg->linkage-id
                                               false-proc))

(define (segment->bar-id segment)
  (m:bar (element-name (segment-endpoint-1 segment))
         (element-name (segment-endpoint-2 segment))))
(defhandler arg->linkage-id segment->bar-id segment?)

(define (angle->joint-id angle)
  (m:joint (element-name (angle-vertex angle))))
(defhandler arg->linkage-id angle->joint-id angle?)

(define (establish-polygon-topology-for-n-gon n-sides)
  (cond ((= n-sides 3)
         (m:establish-polygon-topology 'a 'b 'c))
        ((= n-sides 4)
         (m:establish-polygon-topology 'a 'b 'c 'd))))

(define (observations->figure topology observations)
  (initialize-scheduler)
  (pprint (observations->constraints observations))
  (let ((m (apply
            m:mechanism
            (list
             topology
             (observations->constraints observations)))))
    (m:build-mechanism m)
    (if (not (m:solve-mechanism m))
        (begin
          (pp "Could not solve mechanism")
          #f)
        (let ((f (m:mechanism->figure m)))
          (pp "Solved!")
          (show-figure f)
          f))))

(define (topology-for-object obj)
  (if (polygon? obj)
      (establish-polygon-topology-for-n-gon
       (polygon-n-points obj))
      (error "Object isn't a polygon")))

;;; TODO: Make more general
(define (polygon-from-figure figure)
  (let ((all-points (figure-points figure)))
    (let lp ((i 1)
             (pts '()))
      (let ((p (find-point all-points
                           (nth-letter-symbol i))))
        (if p
            (lp (+ i 1)
                (append pts (list p)))
            (apply polygon-from-points pts))))))

(define (find-point points name)
  (let ((pts (filter
              (lambda (p)
                (eq? (element-name p) name))
              points)))
    (and (not (null? pts))
         (car pts))))
```

Listing A.41: learning/student.scm

```scheme
;;; student.scm -- base model of a student's knowlege

;;; Commentary:

;; Ideas:
;; - Definitions, constructions, theorems
;; - "What is"

;; Future:
;; - Simplifiers of redudant / uninsteresting info
;; - Propose own investigations?

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;; Student Structure ;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <student>
  (%make-student definitions)
  student?
  (definitions student-definitions))

(define (make-student)
  (%make-student (make-key-weak-eq-hash-table)))


;;;;;;;;;;;;;;;;;;;;;;;;; Building Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (build-predicate-for-definition s def)
  (let ((classifications (definition-classifications def))
        (conjectures (definition-conjectures def)))
    (let ((classification-predicate
           (lambda (obj)
             (every
              (lambda (classification)
                (or ((definition-predicate (student-lookup s
                     classification))
                     obj)
                    (begin (if *explain*
                               (pprint `(failed-classification
                                         ,classification)))
                           #f)))
              classifications))))
      (lambda args
        (and (apply classification-predicate args)
             (every (lambda (o) (satisfies-conjecture o args))
                    conjectures))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;; Definitions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (add-definition! s def)
  (if (not (definition-predicate def))
      (set-definition-predicate!
       def
       (build-predicate-for-definition s def)))
  (hash-table/put! (student-definitions s)
                   (definition-name def)
                   def))

(define (lookup-definition s name)
  (hash-table/get (student-definitions s)
                  name
                  #f))

;;;;;;;;;;;;;;;;;;;;;;;;;; Current Student ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *current-student* #f)

(define (student-lookup s term)
  (or (lookup-definition s term)
      *unknown*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Query ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (lookup term)
  (student-lookup *current-student* term))

(define *unknown* 'unknown)
(define (unknown? x)
  (eq? x *unknown*))
```

```scheme

(define (what-is term)
  (pprint (lookup term)))

(define *explain* #f)

(define (is-a? term obj)
  (let ((def (lookup term)))
    (if (unknown? def)
        `(,term unknown)
        (fluid-let ((*explain* #t))
          ((definition-predicate def) obj)))))

(define (internal-is-a? term obj)
  (let ((def (lookup term)))
    (if (unknown? def)
        `(,term unknown)
        ((definition-predicate def) obj))))

(define (show-me term)
  (let ((def (lookup term)))
    (if (unknown? def)
        `(,term unknown)
        (show-element ((definition-generator def))))))

(define (examine object)
  (show-element object)
  (let ((applicable-terms
         (filter (lambda (term)
                   (internal-is-a? term object))
                 (all-known-terms))))
    applicable-terms))

(define (all-known-terms)
  (hash-table/key-list
   (student-definitions *current-student*)))

;;;;;;;;;;;;;;;;;;;;;; Simplifying base terms ;;;;;;;;;;;;;;;;;;;;;;

(define (simplify-base-terms terms)
  (let ((parent-terms (append-map
                       (lambda (t) (definition-classifications (lookup
                            t)))
                           terms)))
    (filter (lambda (t) (not (memq t parent-terms)))
            terms)))

;;;;;;;;;;;;;;;;;;;;;;; Graphics Interfaces ;;;;;;;;;;;;;;;;;;;;;;;

(define (show-element element)
  (if (polygon? element)
      (name-polygon element))
  (show-figure (figure element)))

(define (show-figure figure)
  (draw-figure figure c))

;;;;;;;;;;;;;;;;;;;;;;;; Analyzing Elements ;;;;;;;;;;;;;;;;;;;;;;;;

(define (analyze-element element)
  (if (polygon? element)
      (name-polygon element))
  (let ((fig (figure (with-dependency '<premise> element))))
    (show-figure fig)
    (let ((obs-list (analyze-figure fig)))
      (map observation-with-premises obs-list))))

;;;;;;;;;;;;;;;;;;;;;;; Initializing Student ;;;;;;;;;;;;;;;;;;;;;;;

(define (initialize-student)
  (let ((s (make-student)))
    (provide-core-knowledge s)
    (set! *current-student* s)))


(define (learn-term term object-generator)
  (let ((v (lookup term)))
    (if (not (eq? v 'unknown))
        (pprint `(already-known ,term))
        (let ((example (name-polygon (object-generator))))
          (let* ((base-terms (examine example))
                 (simple-base-terms (simplify-base-terms base-terms))
                 (base-definitions (map lookup base-terms))
                 (base-conjectures (flatten (map definition-conjectures
                                                 base-definitions)))
                 (fig (figure (with-dependency '<premise> example)))
                 (observations (analyze-figure fig))
                 (conjectures (map conjecture-from-observation
                      observations))
                 (simplified-conjectures
                  (simplify-conjectures conjectures base-conjectures)))
            (pprint conjectures)
            (let ((new-def
                   (make-restrictions-definition
                    term
                    simple-base-terms
                    simplified-conjectures
                    object-generator)))
              (add-definition! *current-student* new-def)
              'done))))))

(define (get-simple-definitions term)
  (let ((def (lookup term)))
    (if (unknown? def)
        (error "Unknown term" term))
    (let* ((object ((definition-generator def)))
           (observations
            (filter
            observation->constraint
            (all-observations
             (figure (name-polygon object))))))
      (map
       (lambda (obs-subset)
         (pprint obs-subset)
         (let* ((topology (topology-for-object object))
                (new-figure
                 (observations->figure topology obs-subset)))
           (if new-figure
               (let ((new-polygon
                      (polygon-from-figure new-figure)))
                 (pprint new-polygon)
                 (if (is-a? term new-polygon)
                     (list 'valid-definition
                           obs-subset)
                     (list 'invalid-definition
                           obs-subset)))
               (list 'unknown-definition
                     obs-subset))))
       (all-subsets observations)))))
```

Listing A.42: learning/walkthrough.scm

```
;;; Sample:

;;;;;;;;;;;;;;;;;;;;;;;;; Looking up terms ;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Starts with limited knowledge

(what-is 'square)

(what-is 'rhombus)

;;; Knows primitive objects

(what-is 'line)

(what-is 'point)

(what-is 'polygon)

;;; And some built-in non-primitives

(what-is 'triangle)

(what-is 'quadrilateral)

;;;;;;;;;;; Can idenitfy whether elements satisfy these ;;;;;;;;;;;;

(show-element (random-parallelogram))

(is-a? 'polygon (random-square))

(is-a? 'quadrilateral (random-square))

(is-a? 'triangle (random-square))

(is-a? 'segment (random-square))

(is-a? 'line (random-line))

;;;;;;;;;;;;;;;;; Can learn and explain new terms ;;;;;;;;;;;;;;;;;;

(what-is 'isoc-t)

(learn-term 'isoc-t random-isoceles-triangle)

(what-is 'isoc-t)

(is-a? 'isoc-t (random-isoceles-triangle))

(is-a? 'isoc-t (random-equilateral-triangle))

(is-a? 'isoc-t (random-triangle))

(learn-term 'equi-t random-equilateral-triangle)

(what-is 'equi-t)

(is-a? 'equi-t (random-isoceles-triangle))

(is-a? 'equi-t (random-equilateral-triangle))

;;;;;;;;;;;;;; Let's learn some basic quadrilaterals ;;;;;;;;;;;;;;;

(learn-term 'pl random-parallelogram)

(what-is 'pl)

(learn-term 'kite random-kite)

(what-is 'kite)

(learn-term 'rh random-rhombus)

(what-is 'rh)

(learn-term 'rectangle random-rectangle)

(what-is 'rectangle)

(learn-term 'sq random-square)

(what-is 'sq)
```

Listing A.43: content/load.scm

```
;;; load.scm -- Load learning module
(for-each (lambda (f) (load f))
          '("investigations"
            "thesis-demos"))
```

Listing A.44: content/thesis-demos.scm

```
;;; thesis-demos.scm -- Examples for thesis demonstration chapter

;;; Code

;;;;;;;;;;;;;;;;;;;;;;;; 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)))

(define (demo-figure-0)
  (let-geo* (((s (a b)) (random-segment))
             (pb (perpendicular-bisector s))
             (p (random-point-on-line pb)))
    (figure s pb
            (make-segment a p)
            (make-segment b p))))

(define (incircle-circumcircle)
  (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)))


(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))))

(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 (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)))

(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)))

(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))))

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

(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)))

(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))))

(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))))

(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))))
```

Listing A.45: core/load.scm

```
;;; load.scm -- Load core
(for-each (lambda (f) (load f))
          '("utils"
            "macros"
            "print"
            "animation"))
```

Listing A.46: core/animation.scm

```
;;; animation.scm --- Animating and persisting values in figure
     constructions

;;; Commentary:

;; Ideas:
;; - Animate a range
;; - persist randomly chosen values across frames

;; Future:
;; - Backtracking, etc.
;; - Save continuations?

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;; Configurations ;;;;;;;;;;;;;;;;;;;;;;;;;

(define *animation-steps* 15)

;; ~30 Frames per second:
(define *animation-sleep* 30)

;;;;;;;;;;;;;;;;;;;;;;;; Internal Constants ;;;;;;;;;;;;;;;;;;;;;;;;
(define *is-animating?* #f)
(define *animation-value* 0)
(define *next-animation-index* 0)
(define *animating-index* 0)

(define (run-animation f-with-animations)
  (fluid-let ((*is-animating?* #t)
              (*persistent-values-table* (make-key-weak-eq-hash-table)))
   (let lp ((animate-index 0))
     (fluid-let
         ((*animating-index* animate-index))
       (let run-frame ((frame 0))
         (fluid-let ((*next-animation-index* 0)
                     (*next-value-index* 0)
                     (*animation-value*
                      (/ frame (* 1.0 *animation-steps*))))
           (f-with-animations)
           (sleep-current-thread *animation-sleep*)
           (if (< frame *animation-steps*)
               (run-frame (+ frame 1))
               (if (< *animating-index* (- *next-animation-index* 1))
                   (lp (+ animate-index 1)))))))))

;;;;;;;;;;;;;;;;;;;;;; Animating Functions ;;;;;;;;;;;;;;;;;;;;;;;;

;;; f should be a function of one float argument in [0, 1]
(define (animate f)
  (let ((my-index *next-animation-index*))
    (set! *next-animation-index* (+ *next-animation-index* 1))
    (f (cond ((< *animating-index* my-index) 0)
             ((= *animating-index* my-index) *animation-value*)
             ((> *animating-index* my-index) 1)))))

(define (animate-range min max)
  (animate (lambda (v)
             (+ min
                (* v (- max min))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;; Persistence ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *persistent-values-table* #f)
(define *next-value-index* 0)

(define (persist-value v)
  (if (not *is-animating?*)
      v
      (let* ((my-index *next-value-index*)
             (table-value (hash-table/get
                           *persistent-values-table*
                           my-index
                           #f)))
       (set! *next-value-index* (+ *next-value-index* 1))
       (or table-value
           (begin
             (hash-table/put! *persistent-values-table*
                               my-index
                               v)
             v)))))
```

Listing A.47: core/macros.scm

```
;;; macros.scm --- Macros for let-geo* to assign names and variables
;;; to elements

;;; Commentary:

;; Ideas:
;; - Basic naming
;; - Multiple assignment

;; Future:
;; - Warn about more errors
;; - More efficient multiple-assignment for lists

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;; Expanding Assignment ;;;;;;;;;;;;;;;;;;;;;;;;

(define *multiple-assignment-symbol* '*multiple-assignment-result*)

(define (expand-multiple-assignment lhs rhs)
  (expand-compound-assignment
   (list *multiple-assignment-symbol* lhs)
   rhs))

(define (make-component-assignments key-name component-names)
  (map (lambda (name i)
         (list name `(element-component ,key-name ,i)))
       component-names
       (iota (length component-names))))

(define (expand-compound-assignment lhs rhs)
  (if (not (= 2 (length lhs)))
      (error "Malformed compound assignment LHS (needs 2 elements): "
           lhs))
  (let ((key-name (car lhs))
        (component-names (cadr lhs)))
    (if (not (list? component-names))
        (error "Component names must be a list:" component-names))
    (let ((main-assignment (list key-name rhs))
          (component-assignments (make-component-assignments
                                  key-name
                                  component-names)))
      (cons main-assignment
            component-assignments))))

(define (expand-assignment assignment)
  (if (not (= 2 (length assignment)))
      (error "Assignment in letgeo* must be of length 2, found:"
           assignment))
  (let ((lhs (car assignment))
        (rhs (cadr assignment)))
    (if (list? lhs)
        (if (= (length lhs) 1)
            (expand-multiple-assignment (car lhs) rhs)
            (expand-compound-assignment lhs rhs))
        (list assignment))))

(define (expand-assignments assignments)
  (append-map expand-assignment assignments))

;;;;;;;;;;;;;;;;;;;;;; Extract Variable Names ;;;;;;;;;;;;;;;;;;;;;;

(define (variables-from-assignment assignment)
  (flatten (list (car assignment))))

(define (variables-from-assignments assignments)
  (append-map variables-from-assignment assignments))

(define (set-name-expressions symbols)
  (map (lambda (s)
         `(set-element-name! ,s (quote ,s)))
       symbols))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; let-geo* ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Syntax for setting names for geometry objects declared via let-geo
(define-syntax let-geo*
  (sc-macro-transformer
   (lambda (exp env)
     (let ((assignments (cadr exp))
           (body (caddr exp)))
       (let ((new-assignments (expand-assignments assignments))
             (variable-names (variables-from-assignments assignments)))
         (let ((result`(let*
                         ,new-assignments
                        ,@(set-name-expressions variable-names)
                        ,body)))
           result))))))
```

Listing A.48: core/print.scm

```

;;; print.scm --- Print things nicely

;;; Commentary:
;;; - Default printing is not very nice for many of our record structure

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Print ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define print
  (make-generic-operation 1 'print (lambda (x) x)))

(defhandler print
  (lambda (p) (cons (print (car p))
                    (print (cdr p))))
  pair?)

(defhandler print
  (lambda (l) (map print l))
  list?)

(define (pprint x)
  (pp (print x))
  (display "\n"))
```

Listing A.49: core/utils.scm

```
;;; close-enuf? floating point comparison from scmutils
;;; Origin: Gerald Jay Sussman

(define *machine-epsilon*
  (let loop ((e 1.0))
    (if (= 1.0 (+ e 1.0))
        (* 2 e)
        (loop (/ e 2)))))

(define *sqrt-machine-epsilon*
  (sqrt *machine-epsilon*))

#|
 (define (close-enuf? h1 h2 tolerance)
   (<= (magnitude (- h1 h2))
       (* .5 (max tolerance *machine-epsilon*)
          (+ (magnitude h1) (magnitude h2) 2.0))))
|#

(define (close-enuf? h1 h2 #!optional tolerance scale)
  (if (default-object? tolerance)
      (set! tolerance (* 10 *machine-epsilon*)))
  (if (default-object? scale)
      (set! scale 1.0))
  (<= (magnitude (- h1 h2))
      (* tolerance
         (+ (* 0.5
               (+ (magnitude h1) (magnitude h2)))
            scale))))

(define (assert boolean error-message)
  (if (not boolean) (error error-message)))

(define (flatten list)
  (cond ((null? list) '())
        ((list? (car list))
         (append (flatten (car list))
                 (flatten (cdr list))))
        (else (cons (car list) (flatten (cdr list))))))

(define ((notp predicate) x)
  (not (predicate x)))

(define ((andp p1 p2) x)
  (and (p1 x)
       (p2 x)))

(define (true-proc . args) #t)
(define (false-proc . args) #f)

(define (identity x) x)

;;; ps1 \ ps2
(define (set-difference set1 set2 equality-predicate)
  (define delp (delete-member-procedure list-deletor equality-predicate))
  (let lp ((set1 set1)
           (set2 set2))
    (if (null? set2)
        (dedupe-by equality-predicate set1)
        (let ((e (car set2)))
          (lp (delp e set1)
              (cdr set2))))))

(define (subset? small-set big-set equality-predicate)
  (let ((sd (set-difference small-set big-set equality-predicate)))
    (null? sd)))

(define (eq-subset? small-set big-set)
  (subset? small-set big-set eq?))

(define (set-intersection set1 set2 member-predicate)
  (let lp ((set1 (dedupe member-predicate set1))
           (intersection '()))
    (if (null? set1)
        intersection
        (let ((e (car set1)))
          (lp (cdr set1)
              (if (member-predicate e set2)
                  (cons e intersection)
                  intersection))))))

(define (eq-append! element key val)
  (eq-put! element key
           (cons val
                 (or (eq-get element key) '()))))
```

```scheme

(define (sort-by-key l key)
  (sort l (lambda (v1 v2)
            (< (key v1)
               (key  v2)))))

(define (sort-by-key-2 l key)
  (let ((v (sort-by-key-2 l key)))
    (pprint (map (lambda (x) (cons (name x) (key x))) v))
    v))

(define ((negatep f) x)
  (- (f x)))

(define ((flip-args f) x y)
  (f y x))

(define (index-of el list equality-predicate)
  (let lp ((i 0)
           (l list))
    (cond ((null? l) #f)
          ((equality-predicate (car l) el)
           i)
          (else (lp (+ i 1) (cdr l))))))

;;; (nth-letter-symbol 1) => 'a , 2 => 'b, etc.
(define (nth-letter-symbol i)
  (symbol (make-char (+ 96 i) 0)))

(define (hash-table/append table key element)
  (hash-table/put! table
                   key
                   (cons element
                         (hash-table/get table key '()))))

(define (dedupe-by equality-predicate elements)
  (dedupe (member-procedure equality-predicate) elements))

(define (dedupe member-predicate elements)
  (cond ((null? elements) '())
        (else
         (let ((b1 (car elements)))
           (if (member-predicate b1 (cdr elements))
               (dedupe member-predicate (cdr elements))
               (cons b1 (dedupe member-predicate (cdr elements))))))))

;;; supplanted-by-prediate takes two args: an element under consideration
;;; and an existing element in the list. If true, the first element
;;; will be removed from the list.
(define (remove-supplants supplanted-by-predicate elements)
  (define member-predicate (member-procedure
                            supplanted-by-predicate))
  (let lp ((elements-tail elements)
           (elements-head '()))
    (if (null? elements-tail)
        elements-head
        (let ((el (car elements-tail))
              (new-tail (cdr elements-tail)))
          (lp new-tail
              (if (or (member-predicate el new-tail)
                      (member-predicate el elements-head))
                  elements-head
                  (cons el elements-head)))))))

(define (partition-into-equivalence-classes elements
     equivalence-predicate)
  (let lp ((equivalence-classes '())
           (remaining-elements elements))
    (if (null? remaining-elements)
        equivalence-classes
        (lp
         (add-to-equivalence-classes
          equivalence-classes
          (car remaining-elements)
          (member-procedure equivalence-predicate))
         (cdr remaining-elements)))))

(define (add-to-equivalence-classes classes element memp)
  (if (null? classes)
      (list (list element))
      (let ((first-class (car classes))
            (remaining-classes (cdr classes)))
        (if (memp element first-class)
            (cons (cons element first-class)
                  remaining-classes)
            (cons first-class
                  (add-to-equivalence-classes remaining-classes
                                              element
                                              memp))))))

(define (all-subsets elements)
  (append-map
   (lambda (n)
     (all-n-tuples n elements))
   (iota (+ (length elements) 1))))
```

Listing A.50: lib/eq-properties.scm

```scheme
;;;; Traditional LISP property lists
;;; extended to work on any kind of eq? data structure.

(declare (usual-integrations))

;;; Property lists are a way of creating data that looks like a record
;;; structure without commiting to the fields that will be used until
;;; run time.  The use of such flexible structures is frowned upon by
;;; most computer scientists, because it is hard to statically
;;; determine the bounds of the behavior of a program written using
;;; this stuff.  But it makes it easy to write programs that confuse
;;; such computer scientists.  I personally find it difficult to write
;;; without such crutches.  -- GJS


(define eq-properties (make-eq-hash-table))

(define (eq-put! node property value)
  (let ((plist (hash-table/get eq-properties node #f)))
    (if plist
        (let ((vcell (assq property (cdr plist))))
          (if vcell
              (set-cdr! vcell value)
              (set-cdr! plist
                        (cons (cons property value)
                              (cdr plist)))))
        (hash-table/put! eq-properties node
                         (list node (cons property value)))))
  'done)

(define (eq-adjoin! node property new)
  (eq-put! node property
           (eq-set/adjoin new
                          (or (eq-get node property) '())))
  'done)

(define (eq-rem! node property)
  (let ((plist (hash-table/get eq-properties node #f)))
    (if plist
        (let ((vcell (assq property (cdr plist))))
          (if vcell
              (hash-table/put! eq-properties node (delq! vcell
                   plist))))))
  'done)


(define (eq-get node property)
  (let ((plist (hash-table/get eq-properties node #f)))
    (if plist
        (let ((vcell (assq property (cdr plist))))
          (if vcell
              (cdr vcell)
              #f))
        #f)))

(define (eq-plist node)
  (hash-table/get eq-properties node #f))



(define (eq-path path)
  (define (lp node)
    (if node
        (if (pair? path)
            (eq-get ((eq-path (cdr path)) node)
                    (car path))
            node)
        #f))
  lp)
```

Listing A.51: lib/ghelper.scm

```

(define make-generic-operation make-generic-operator)

#|
;;;;           Most General Generic-Operator Dispatch
(declare (usual-integrations))          ; for compiler

;;; Generic-operator dispatch is implemented here by a
;;; discrimination list (a "trie", invented by Ed Fredkin),
;;; where the arguments passed to the operator are examined
;;; by predicates that are supplied at the point of
;;; attachment of a handler.  (Handlers are attached by
;;; ASSIGN-OPERATION alias DEFHANDLER).

;;; The discrimination list has the following structure: it
;;; is an improper alist whose "keys" are the predicates
;;; that are applicable to the first argument.  If a
;;; predicate matches the first argument, the cdr of that
;;; alist entry is a discrimination list for handling the
;;; rest of the arguments.  Each discrimination list is
;;; improper: the cdr at the end of the backbone of the
;;; alist is the default handler to apply (all remaining
;;; arguments are implicitly accepted).

;;; A successful match of an argument continues the search
;;; on the next argument.  To be the correct handler all
;;; arguments must be accepted by the branch predicates, so
;;; this makes it necessary to backtrack to find another
;;; branch where the first argument is accepted if the
;;; second argument is rejected.  Here backtracking is
;;; implemented using #f as a failure return, requiring
;;; further search.


#| ;;; For example.
(define foo (make-generic-operator 2 'foo))

(defhandler foo + number? number?)

(define (symbolic? x)
  (or (symbol? x)
      (and (pair? x) (symbolic? (car x)) (list? (cdr x)))))

(define (+:symb x y) (list '+ x y))

```

```
(defhandler foo +:symb number? symbolic?)
(defhandler foo +:symb symbolic? number?)
(defhandler foo +:symb symbolic? symbolic?)

(foo 1 2)
;Value: 3

(foo 1 'a)
;Value: (+ 1 a)

(foo 'a 1)
;Value: (+ a 1)

(foo '(+ a 1) '(+ 1 a))
;Value: (+ (+ a 1) (+ 1 a))
|#


(define (make-generic-operator arity
                   #!optional name default-operation)
  (let ((record (make-operator-record arity)))

    (define (operator . arguments)
      (if (not (acceptable-arglist? arguments arity))
          (error:wrong-number-of-arguments
           (if (default-object? name) operator name)
           arity arguments))
      (apply (find-handler (operator-record-tree record)
                           arguments)
             arguments))

    (set-operator-record! operator record)

    (set! default-operation
      (if (default-object? default-operation)
          (named-lambda (no-handler . arguments)
            (error "Generic operator inapplicable:"
                   (if (default-object? name) operator name)
                   arguments))
          default-operation))
    (if (not (default-object? name))    ; Operation by name
        (set-operator-record! name record))

    (assign-operation operator default-operation)
    operator))


#|
;;; To illustrate the structure we populate the
;;; operator table with quoted symbols rather
;;; than actual procedures.

(define blend
  (make-generic-operator 2 'blend 'blend-default))

(pp (get-operator-record blend))
(2 . blend-default)

(defhandler blend 'b+b 'blue?  'blue?)
(defhandler blend 'g+b 'green? 'blue?)
(defhandler blend 'b+g 'blue?  'green?)
(defhandler blend 'g+g 'green? 'green?)

(pp (get-operator-record blend))
(2 (green? (green? . g+g) (blue? . g+b))
   (blue? (green? . b+g) (blue? . b+b))
   .
   blend-default)
|#


#|
;;; Backtracking

;;; An operator satisfies bleen?
;;; if it satisfies either blue? or green?

(defhandler blend 'e+r 'bleen? 'red?)
(defhandler blend 'e+u 'bleen? 'grue?)

(pp (get-operator-record blend))
(2 (bleen? (grue? . e+u) (red? . e+r))
   (green? (green? . g+g) (blue? . g+b))
   (blue? (green? . b+g) (blue? . b+b))
   .
   blend-default)

;;; Consider what happens if we invoke
;;; (blend <bleen> <blue>)
|#


;;; This is the essence of the search.

(define (find-handler tree args)
  (if (null? args)
      tree
      (find-branch tree
                   (car args)
                   (lambda (result)
                     (find-handler result
                                   (cdr args))))))

(define (find-branch tree arg next)
  (let loop ((tree tree))
    (cond ((pair? tree)
           (or (and ((caar tree) arg)
                    (next (cdar tree)))
```

```
               (loop (cdr tree))))
          ((null? tree) #f)
          (else tree))))

(define (assign-operation operator handler
                          . argument-predicates)
  (let ((record (get-operator-record operator))
        (arity (length argument-predicates)))
    (if record
        (begin
          (if (not (<= arity
                       (procedure-arity-min
                        (operator-record-arity record))))
              (error "Incorrect operator arity:" operator))
          (bind-in-tree argument-predicates
                        handler
                        (operator-record-tree record)
                        (lambda (new)
                          (set-operator-record-tree! record
                                                     new))))
        (error "Undefined generic operator" operator)))
  operator)

(define defhandler assign-operation)

(define (bind-in-tree keys handler tree replace!)
  (let loop ((keys keys) (tree tree) (replace! replace!))
    (cond ((pair? keys)   ; more argument-predicates
           (let find-key ((tree* tree))
             (if (pair? tree*)
                 (if (eq? (caar tree*) (car keys))
                     ;; There is already some discrimination
                     ;; list keyed by this predicate: adjust it
                     ;; according to the remaining keys
                     (loop (cdr keys)
                           (cdar tree*)
                           (lambda (new)
                             (set-cdr! (car tree*) new)))
                     (find-key (cdr tree*)))
                 (let ((better-tree
                        (cons (cons (car keys) '()) tree)))
                   ;; There was no entry for the key I was
                   ;; looking for.  Create it at the head of
                   ;; the alist and try again.
                   (replace! better-tree)
                   (loop keys better-tree replace!)))))
          ;; cond continues on next page.

          ((pair? tree)  ; no more argument predicates.
            ;; There is more discrimination list here,
            ;; because my predicate list is a proper prefix
            ;; of the predicate list of some previous
            ;; assign-operation.  Insert the handler at the
            ;; end, causing it to implicitly accept any
            ;; arguments that fail all available tests.
           (let ((p (last-pair tree)))
             (if (not (null? (cdr p)))
                 (warn "Replacing a default handler:"
                       (cdr p) handler))
             (set-cdr! p handler)))
          (else
           ;; There is no discrimination list here.  This
           ;; handler becomes the discrimination list,
           ;; accepting further arguments if any.
           (if (not (null? tree))
               (warn "Replacing a handler:" tree handler))
           (replace! handler)))))

(define *generic-operator-table* (make-eq-hash-table))

(define (get-operator-record operator)
  (hash-table/get *generic-operator-table* operator #f))

(define (set-operator-record! operator record)
  (hash-table/put! *generic-operator-table* operator
                   record))

(define (make-operator-record arity) (cons arity '()))
(define (operator-record-arity record) (car record))
(define (operator-record-tree record) (cdr record))
(define (set-operator-record-tree! record tree)
  (set-cdr! record tree))

(define (acceptable-arglist? lst arity)
  (let ((len (length lst)))
    (and (fix:<= (procedure-arity-min arity) len)
         (or (not (procedure-arity-max arity))
             (fix:>= (procedure-arity-max arity) len)))))

#|
;;; Demonstration of handler tree structure.
;;; Note: symbols were used instead of procedures

(define foo (make-generic-operator 3 'foo 'foo-default))

(pp (get-operator-record foo))
(3 . foo-default)

(defhandler foo 'two-arg-a-b 'a 'b)
(pp (get-operator-record foo))
(3 (a (b . two-arg-a-b)) . foo-default)

```

```
(defhandler foo 'two-arg-a-c 'a 'c)
(pp (get-operator-record foo))
(3 (a (c . two-arg-a-c) (b . two-arg-a-b)) . foo-default)

(defhandler foo 'two-arg-b-c 'b 'c)
(pp (get-operator-record foo))
(3 (b (c . two-arg-b-c))
   (a (c . two-arg-a-c) (b . two-arg-a-b))
   . foo-default)
|#


#|
(defhandler foo 'one-arg-b 'b)
(pp (get-operator-record foo))
(3 (b (c . two-arg-b-c) . one-arg-b)
   (a (c . two-arg-a-c) (b . two-arg-a-b))
   . foo-default)

(defhandler foo 'one-arg-a 'a)
(pp (get-operator-record foo))
(3 (b (c . two-arg-b-c) . one-arg-b)
   (a (c . two-arg-a-c) (b . two-arg-a-b) . one-arg-a)
   .
   foo-default)

(defhandler foo 'one-arg-a-prime 'a)
;Warning: Replacing a default handler:
;         one-arg-a one-arg-a-prime

(defhandler foo 'two-arg-a-b-prime 'a 'b)
;Warning: Replacing a handler:
;         two-arg-a-b two-arg-a-b-prime

(defhandler foo 'three-arg-x-y-z 'x 'y 'z)
(pp (get-operator-record foo))
(3 (x (y (z . three-arg-x-y-z)))
   (b (c . two-arg-b-c) . one-arg-b)
   (a (c . two-arg-a-c)
      (b . two-arg-a-b-prime)
      .
      one-arg-a-prime)
   .
   foo-default)
|#


;;; Compatibility with previous extensible generics

(define make-generic-operation make-generic-operator)

(define (add-to-generic-operation! operator
                                   applicability
                                   handler)
  ;; An applicability is a list representing a
  ;; disjunction of lists, each representing a
  ;; conjunction of predicates.

  (for-each (lambda (conj)
              (apply assign-operation
                     operator
                     handler
                     conj))
            applicability))

|#
```

Bibliography

[1] Dave Barker-Plummer, Richard Cox, and Nik Swoboda, editors. *Diagrammatic Representation and Inference*, volume 4045 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006.

[2] Xiaoyu Chen, Dan Song, and Dongming Wang. Automated generation of geometric theorems from images of diagrams. *CoRR*, abs/1406.1638, 2014.

[3] Shang-Ching Chou, Xiao-Shan Gao, and Jing-Zhong Zhang. A deductive database approach to automated geometry theorem proving and discovering. *Journal of Automated Reasoning*, 25(3):219–246, 2000.

[4] Joran Elias. Automated geometric theorem proving: WuâĂŹs method. *The Montana Mathematics Enthusiast*, 3(1):3–50, 2006.

[5] Anne Berit Fuglestad. *Discovering geometry with a computer: using Cabri-géomètre*. Chartwell-Yorke, 114 High Street, Belmont, Bolton, Lancashire, BL7 8AL, England, 1994.

[6] R Nicholas Jackiw and William F Finzer. The geometer's sketchpad: programming by geometry. In *Watch what I do*, pages 293–307. MIT Press, 1993.

[7] Keith Jones. Providing a foundation for deductive reasoning: Students' interpretations when using dynamic geometry software and their evolving mathematical explanations. *Educational Studies in Mathematics*, 44(1-2):55–85, 2000.

[8] Antonio Montes and Tomás Recio. Automatic discovery of geometry theorems using minimal canonical comprehensive gröbner systems. In *Automated Deduction in Geometry*, pages 113–138. Springer, 2007.

[9] Julien Narboux. A graphical user interface for formal proofs in geometry. *Journal of Automated Reasoning*, 39(2):161–180, 2007.

[10] Stavroula Patsiomitou and Anastassios Emvalotis. Developing geometric thinking skills through dynamic diagram transformations. In *6th Mediterranean Conference on Mathematics Education*, pages 249–258, 2009.

[11] Pavel Pech. Deriving geometry theorems by automated tools. In *Proceedings of the Sixteenth Asian Technology Conference in Mathematics*. Mathematics and Technology, LLC, 2011.

[12] Alexey Radul and Gerald Jay Sussman. The art of the propagator. In *CSAIL Technical Report*, 2009.

[13] Min Joon Seo, Hannaneh Hajishirzi, Ali Farhadi, and Oren Etzioni. Diagram understanding in geometry questions. In *Proceedings of the Twenty-eighth AAAI Conference on Artificial Intelligence*, 2014.

[14] Michael Serra. *Discovering geometry: An investigative approach*, volume 4. Key Curriculum Press, 2003.

[15] Sean Wilson and Jacques D. Fleuriot. Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs. In *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS) Satellite Workshop on User Interfaces for Theorem Provers (UITP)*. Springer, 2005.

[16] Franz Winkler, editor. *Automated Deduction in Geometry*, volume 2930 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004.