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

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

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 could be rigorously proved using external proof assistants, but are also 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.

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

Acknowledgments

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

I'd specifically like to thank Dan Butler, my geometry teacher who taught our class using Michael Serra's text, *Discovering Geometry: An Investigative Approach.* Such experience with an investigative methodology served as an important inspiration for pursuing this project.

I appreciate Nyan Lounge at Simmons Hall's patience with my efforts and assurance that I had plenty of fun along the way. With a great group of friends, I always had a reason to smile each day. I also thank my parents, sister, grandparents, and extended family for their unfailing love, open ears, and reassuring guidance over the years, providing me with a strong foundation for my current and future endeavors.

Finally, I could not have completed this project without my thesis advisor, Gerald Jay Sussman: Thank you for emphasizing the importance of seeking out and working on problems one finds personally interesting, and for providing insightful discussions, stories, and advice along the way.

Contents

Chapter 1

Introduction

I developed and analyzed 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 limited knowledge set regarding basic definitions in geometry and providing a means for a user interacting with the system to "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 language to build constructions, an observation-based perception module to notice interesting properties, a declarative geometry constraint solver to solve and test specifications, 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.

1.1 Document Structure

Following this introduction,

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

Chapter 3 Demonstration provides several sample interactions with the system showing the results for this project.

Chapter 4 System Overview presents several concepts used in the system, introduces the four main modules, and discusses how they work together in the discovery of new definitions and theorems.

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

Chapter 5 Imperative Construction describes the construction module that enables the system to represent, perform, and display figure constructions.

Chapter 6 Perception describes the perception module that focuses on observing interesting properties in diagrams. A key question involves determining "what is interesting?".

Chapter 7 Declarative Constraint Solver describes the propagator-based declarative geometry constraint solver that builds instances of diagrams satisfying declarative constraints.

Chapter 8 Learning Module describes the learning module that integrates results from the other systems to create new discoveries. Main features include filtering out obvious and known during investigations, representing and storing newly discovered definitions and theorems, providing an interface to apply these findings to new situations, and simplifying the resulting definitions.

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

Chapter 10 Conclusion evaluates the strengths and weaknesses of the system and discusses future work and extensions.

Appendix A Code Listings provides full listings for code used in the system and explains an external dependency on a propagator system.

Appendix B Bibliography lists works referenced in the document.

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 two-dimensional 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 sides are concurrent, and that a circle constructed with center at the point of concurrence through one triangle vertex intersects the other two triangle vertices. 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. This 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.

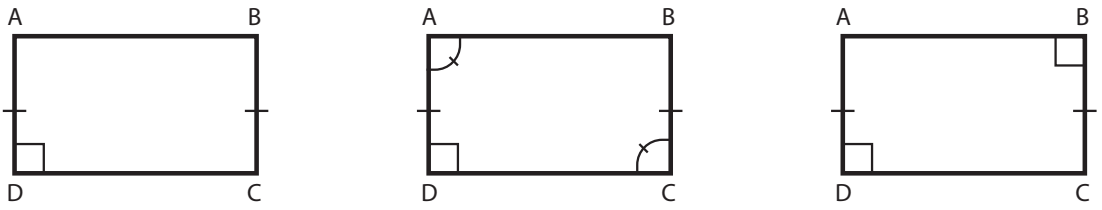
My interactive computer system emulates the curious student described above, and is capable of exploring geometric concepts through inductive investigation. The system begins with a limited set of factual knowledge regarding basic definitions in geometry and provides means by which a user interacting with the system can "teach" it 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 provides means for a user to extract its 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:

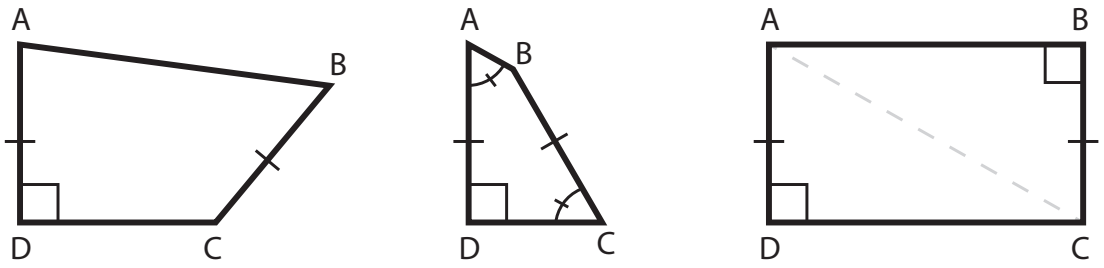
2.1.1 An Initial Example



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) placed on objects form a *diagram*, which is the abstract representation of the entire space of figure *instances* that satisfy the constraints.

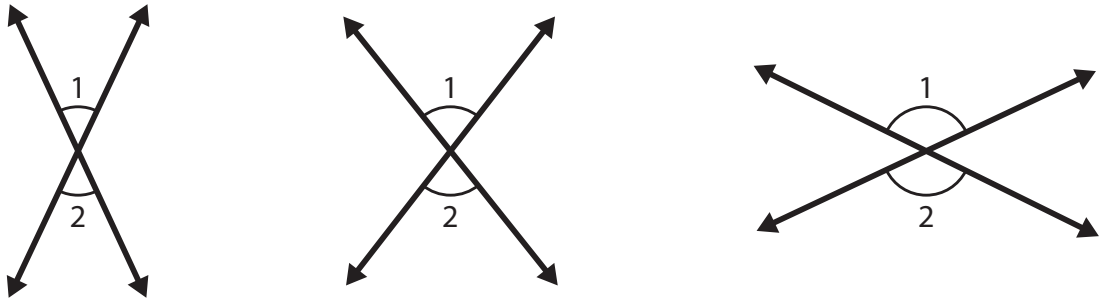
An annotated figure presented on a page is typically an instance of its corresponding diagram. However, it is certainly possible to add annotations to a figure that are not satisfied by that figure, yielding impossible diagrams. In such a case the diagram represents an empty set of satisfying figures.

In the initial example above, the three quadrilateral figures are drawn as rectangles. It is true that all quadrilateral figures in the space represented by the third diagram are rectangles. However, the spaces of quadrilaterals represented by the first two diagrams include instances that are not rectangles, as shown above. At this time, the system only works with diagrams whose constraints can be satisfied in some figure. Detecting and explaining impossible diagrams, purely from their set of constraints would 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

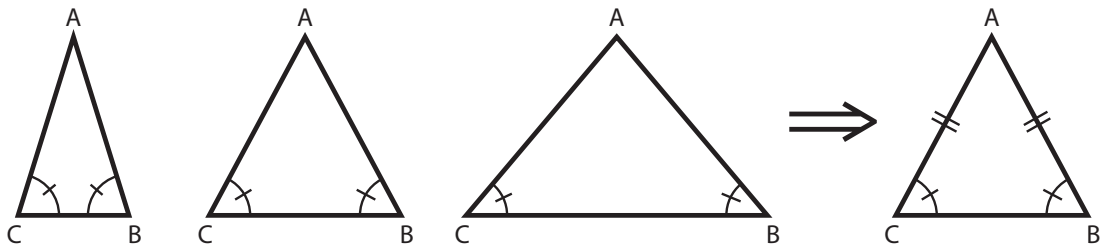


Investigation 1: Construct a pair of vertical angles. Notice anything interesting?

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 argument. However, the inductive manipulations are more reflective of the initial, intuitive process one typically takes when first presented with understanding a problem.

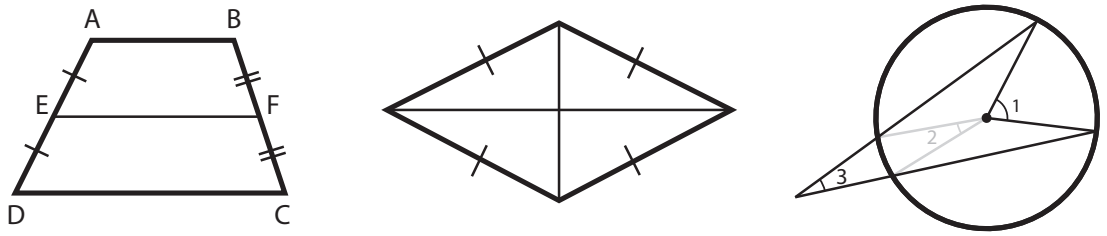
2.2.2 Elementary Results



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 isosceles. 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.

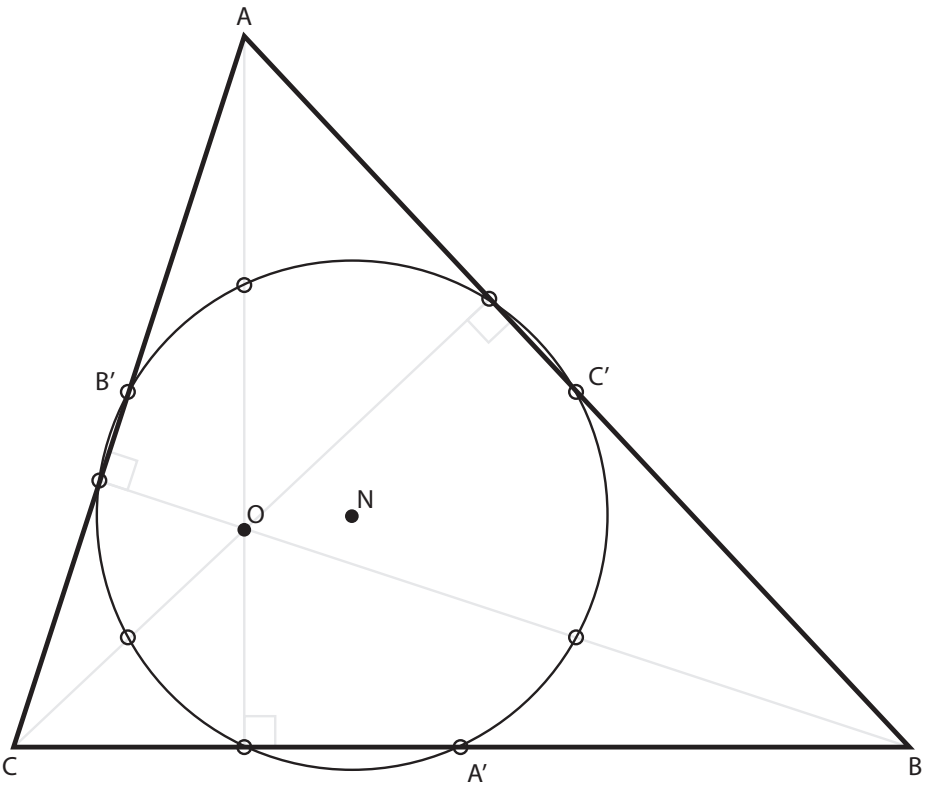
In this and further examples, an important question becomes what properties are considered "interesting" and worth investigating in further instances of the diagram, as discussed in Section 4.3.3. As suggested by the examples in Investigation 3, this can include relations between segment and angle lengths, 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

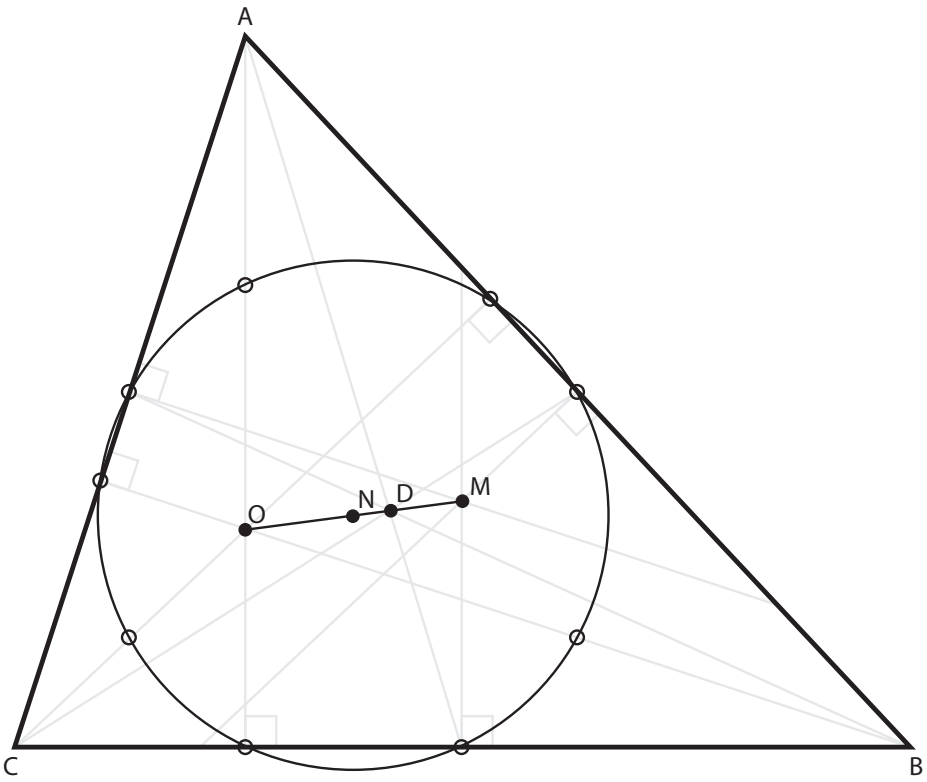
Finally, this technique can be used to explore and discover conjectures well beyond the scope of what one can visualize in his or her head:



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 (next page), 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 median concurrence) and circumcenter M (from perpendicular bisector concurrence). Notice anything interesting?

2.3 Discussion

As the examples and investigations in this chapter demonstrate, mental manipulation of figures to observe interesting relationships that are invariant across diagram instances is a useful reasoning skill. Such relationships can be generalized as conjectures or theorems and used in further analysis.

The following chapters present an interactive computer system that emulates this process. Similar to the process of making, manipulating, and observing pictures "in the mind's eye", the system constructs several examples of figures under investigation and extracts interesting relationships. A learning module aggregates and applies the results to new investigations.

Chapter 3

Demonstration

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

The overall goal of the system is to emulate a student learning geometry via an investigative approach. To accomplish this, the system is divided into four main modules: an imperative construction system, a perception-based analyzer, a declarative constraint solver, and a synthesizing learning module. The following examples will explore interactions with these modules in increasing complexity, building up to a full demonstration of the system achieving its learning goals in Sections 3.4 and 3.5.

3.1 Imperative Figure Construction

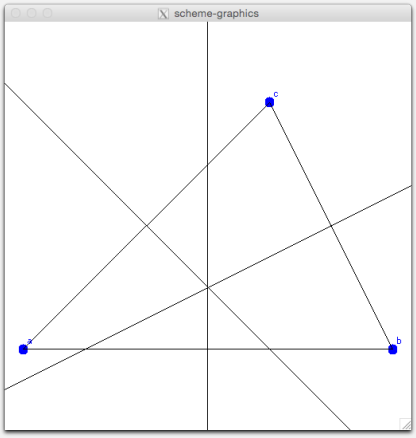
At its foundation, the system provides a language and engine for performing geometry constructions and building figures. Example 3.1 presents a simple specification of a figure. Primitives of points, lines, segments, rays, and circles can be combined into polygons and figures, and complicated constructions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

Code Example 3.1: Basic Figure Example

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

Interaction Example 3.2: Rendering the Basic Figure

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



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

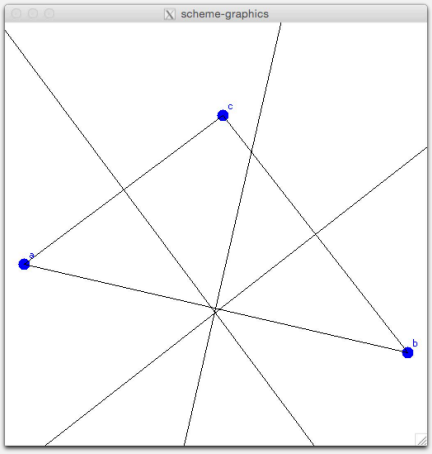
In the first example, the coordinates of the points were explicitly specified yielding a deterministic instance of the figure. However, to represent entire spaces of diagram instances, the construction abstractions support random choices. Example 3.3 demonstrates the creation of a figure involving an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

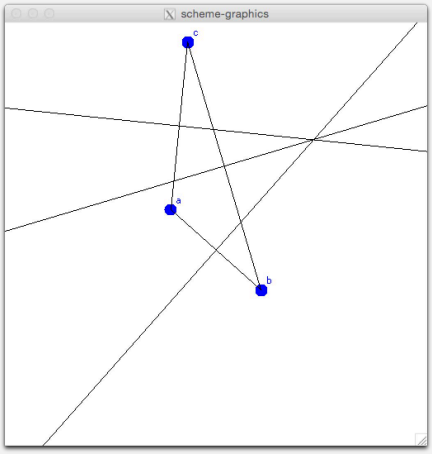
The second formulation, `(simple-random-triangle-perp-bisectors)`, is equivalent to the first. It displays a syntax extension provided by `let-geo*` that shortens

the common pattern of accessing and naming the components of an object. In this case, `((t (a b c)) (random-triangle))` will assign to the variable `t` the resulting random triangle, and to the variables `a`, `b`, and `c` the resulting triangle's vertices.

Interaction Example 3.3: Introducing Randomness

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

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

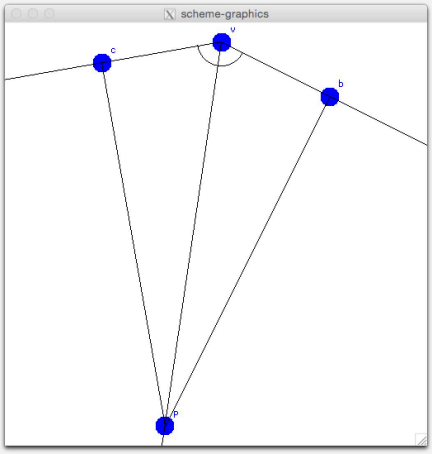


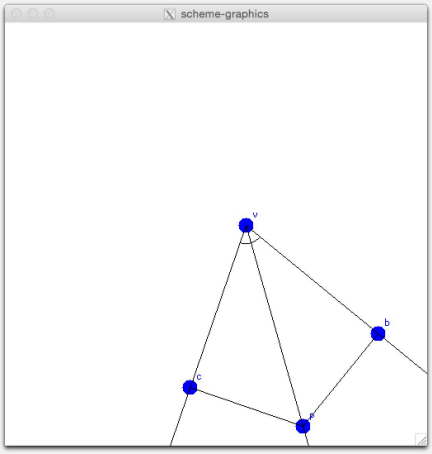


As examples of more involved constructions, Examples 3.4 and 3.5 demonstrate working with other objects (angles, rays, circles) and construction procedures. Notice that, in the angle bisector example, the pattern matching syntax extracts the components of an angle (ray, vertex, ray) and segment (endpoints),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

Interaction Example 3.4: Angle Bisector Distance

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

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

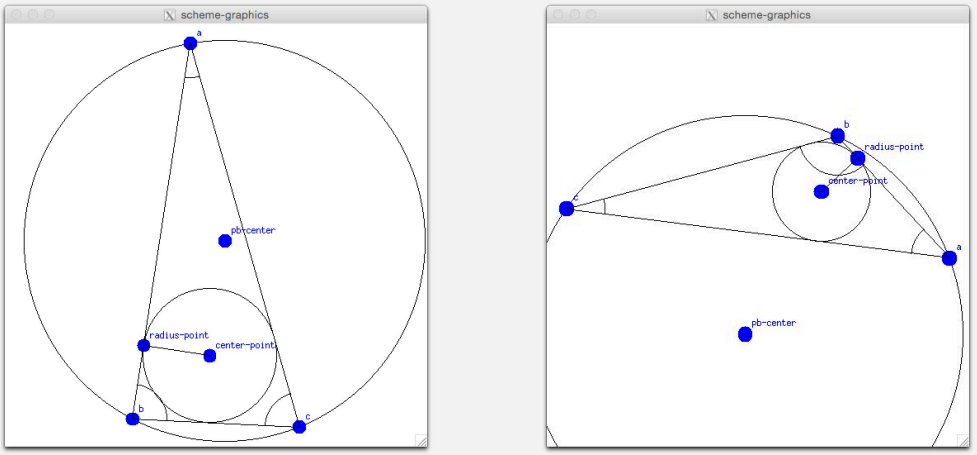




Interaction Example 3.5: Inscribed and Circumscribed Circles

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

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



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

3.2 Perception and Observation

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

Example 3.6 demonstrates the interface for obtaining observations from a figure. An observation is a structure that associates a relationship (such as concurrent, equal length, or parallel) with objects in the figure that satisfy the relationship. Relationships are represented as predicates over typed n-tuples and are checked against all such n-tuples found in the figure under analysis. For example, the perpendicular relationship is checked against all pairs of linear elements in the figure.

The observation objects returned are compound structures that maintain properties of the underlying relationships and references to the original objects under consideration. Dependency information about how these original objects were construction will be later used to generalize these observations into conjectures. For now, my custom printer `print-observations` can use the name information of the objects to display the observations in a more human-readable format.

Interaction Example 3.6: Simple Analysis

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

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

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

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

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

Interaction Example 3.7: Interesting Analysis

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

((concurrent pb1 pb2 pb3))
```

For an example with more relationships, Example 3.8 demonstrates the observations and relationships found in a figure with a random parallelogram. These analysis results will be used again later when I demonstrate the system learning definitions for

polygons and simplifying such definitions to minimal sufficient constraint sets. Note that although the segments, angles, and points were not explicitly listed in the figure, they are extracted from the polygon that is listed. Extensions to the observation model can extract additional points and segments not explicitly listed in the original figure.

Interaction Example 3.8: Parallelogram Analysis

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

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

((equal-length (segment a b) (segment c d))
 (equal-length (segment b c) (segment d a))
 (equal-angle (angle a) (angle c))
 (equal-angle (angle b) (angle d))
 (supplementary (angle a) (angle b))
 (supplementary (angle a) (angle d))
 (supplementary (angle b) (angle c))
 (supplementary (angle c) (angle d))
 (parallel (segment a b) (segment c d))
 (parallel (segment b c) (segment d a)))
```

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

3.3 Mechanism-based Declarative Constraint Solver

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

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

Thus, the third module is a declarative constraint solver. To model the physical concept of building and wiggling components until constraints are satisfied, the system is formulated around solving mechanisms built from bars and joints that must satisfy certain constraints. Such constraint solving is implemented by extending the Propagator Model created by Alexey Radul and Gerald Jay Sussman [22] to handle partial information and constraints about geometry positions. Chapter 7 discusses further implementation details.

3.3.1 Bars and Joints

Example 3.9 demonstrates the specification of a very simple mechanism. Unlike the sequential, Scheme variable based `let-geo*` specification of constructions in the imperative construction system, to specify mechanisms, `m:mechanism` is applied to a list of bar, joint, and constraint declarations containing symbolic identifiers. This example mechanism is composed of two bars with one joint between them, along with a constraint that the joint is a right angle.

Code Example 3.9: Very Simple Mechanism

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

Assembling a mechanism involves first adjoining the bars and joints together so that the named points are identified with one another. Initially, each bar has unknown length and direction, each joint has an unknown angle, and each endpoint has

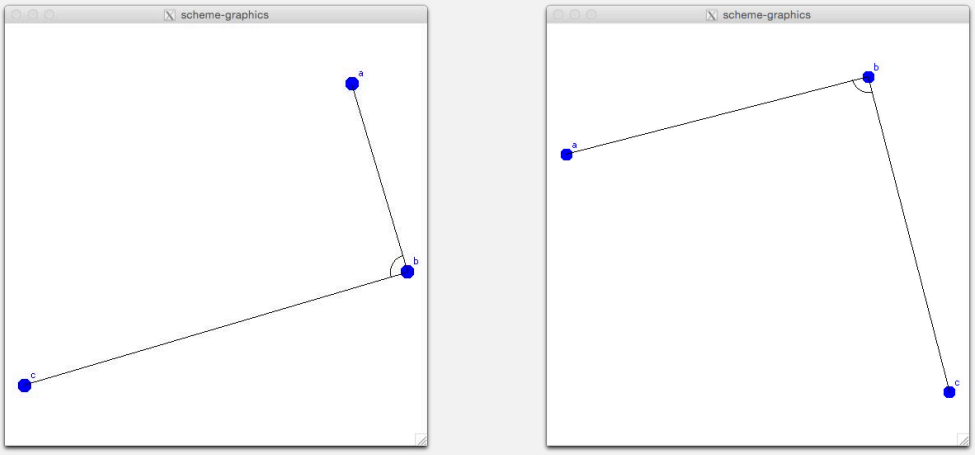
unknown position. Constraints for the bar and joint properties are then introduced alongside any explicitly specified constraints.

Solving a mechanism involves repeatedly selecting position, lengths, angles, and directions that are not yet determined and selecting values within the domain of that element's current partial information content. As values are specified, the wiring of the propagator model propagates partial information updates to neighboring cells.

The printed statements in Example 3.10 demonstrate that solving the simple mechanism above involves specifying the location of point `a`, then specifying the length of bar `a-b` and the direction from `a` that the bar extends. After those specifications, the joint angle is constrained to be a right angle and the location of point `b` is known by propagating information about point `a` and bar `a-b`'s position and length. Thus, point `c` is known to be on a ray extending outwards from `b` and the only remaining property needed to fully specify the figure is the length of bar `b-c`. The command `m:run-mechanism` builds and solves the mechanism, then converts the result into an analytic figure and displays it.

Interaction Example 3.10: Solving the Very Simple Mechanism

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

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



3.3.2 Geometry Examples

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

Code Example 3.11: Describing an Arbitrary Triangle

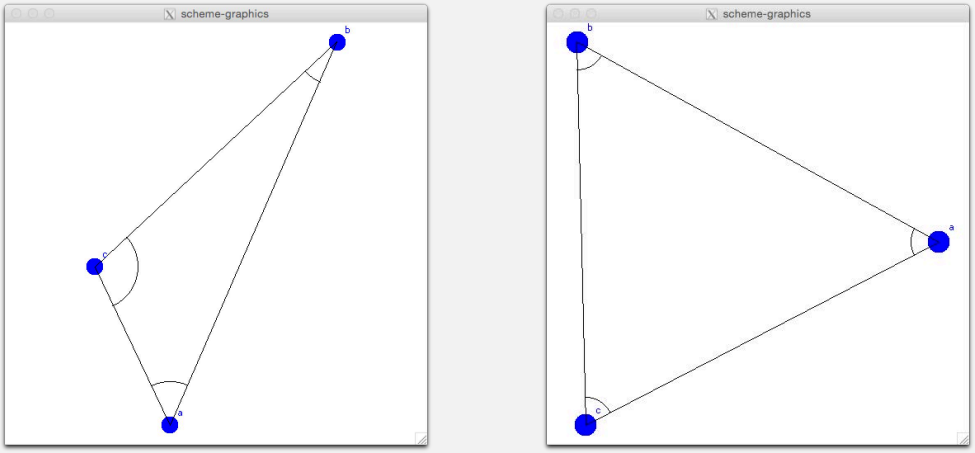
```
(define (arbitrary-triangle)
  (m:mechanism
   (m:make-named-bar 'a 'b)
   (m:make-named-bar 'b 'c)
   (m:make-named-bar 'c 'a)
   (m:make-named-joint 'a 'b 'c)
   (m:make-named-joint 'b 'c 'a)
   (m:make-named-joint 'c 'a 'b)))

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

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

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

Interaction Example 3.12: Solving the Triangle

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

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

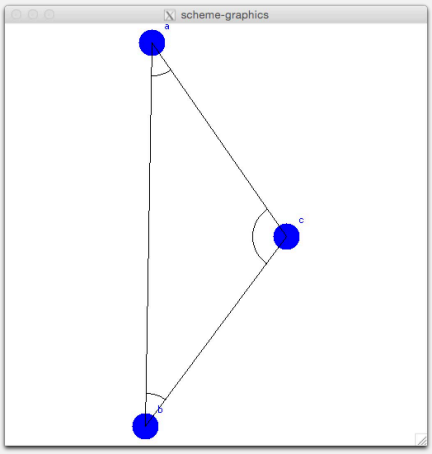


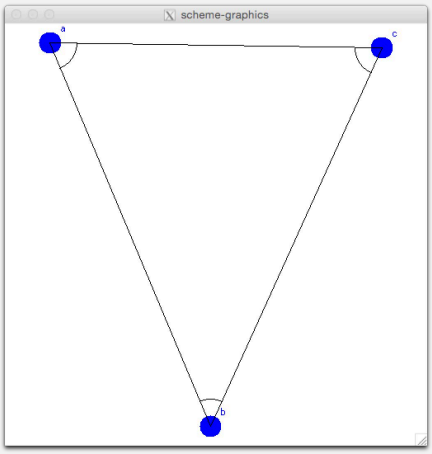
To include some user-specified constraints, Example 3.13 shows the steps involved in solving an isosceles triangle from the fact that its base angles are congruent. Notice that the only two values that must be specified are one joint angle and one bar length. The rest is handled by propagation.

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

Interaction Example 3.13: Constraint Solving for Isosceles Triangle

```
(define (isosceles-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  isosceles-triangle-by-angles)

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



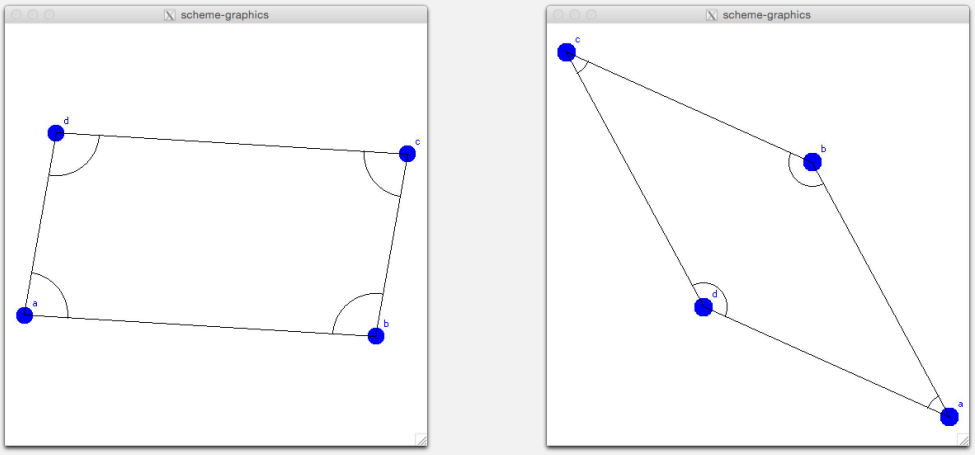


Example 3.14 continues the analysis of properties of the parallelogram. In this case, the constraint solver is able to build figures given the fact that its opposite angles are equal. The fact that these all happen to be parallelograms will be used by the learning module to produce a simpler definition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

Interaction Example 3.14: Constraint Solving for Parallelogram

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

```
=> (m:run-mechanism parallelogram-by-angles)
(specifying-joint-angle m:joint:c:b:a 1.6835699856637936)
(initializing-angle m:joint:c:b:a-dir-1 (direction 1.3978162819212452))
(initializing-point m:bar:a:b-p1 (0 0))
(specifying-bar-length m:bar:a:b .8152792207652096)
(specifying-bar-length m:bar:b:c .42887899934327023)
```



To demonstrate the constraint solving working on a more complicated example, Example 3.15 represents the constraints from the middle "Is this a rectangle?" question from Chapter 2 (page 17). This question asks whether a quadrilateral in which a pair of opposite sides is congruent, a pair of opposite angles is congruent, and one of the other angles is a right angle, is always a rectangle. Try working this constraint problem by hand or in your mind's eye.

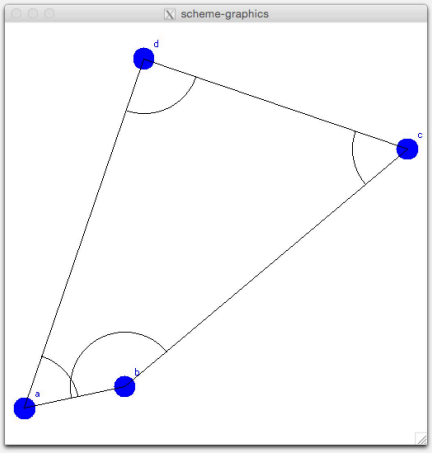
Code Example 3.15: Rectangle Constraints Example

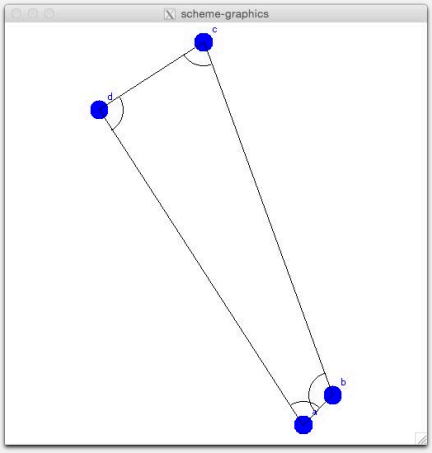
```
(define (is-this-a-rectangle-2)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'd) (m:bar 'b 'c))
   (m:c-right-angle (m:joint 'd))
   (m:c-angle-equal (m:joint 'a) (m:joint 'c))))
```

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

Interaction Example 3.16: Solved Constraints

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

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





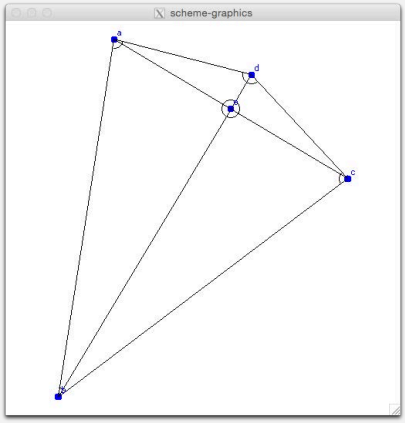
As a final mechanism example, in addition to solving constraints of the angles and sides for a *single* polygon, the mechanism system can support the creation of arbitrary topologies of bars and joints. In the following examples, by using several calls to the `m:establish-polygon-topology`, I build the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

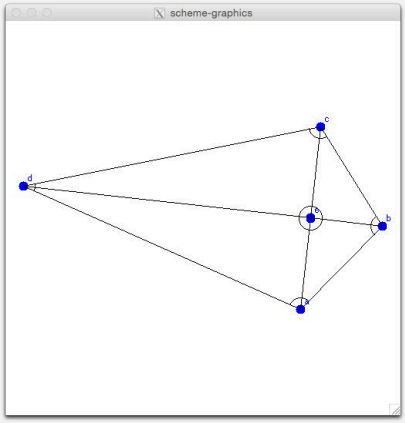
Code Example 3.17: More Involved Topologies for Constraint Solving

```
1 (define (m:quadrilateral-with-intersecting-diagonals a b c d e)
2   (list (m:establish-polygon-topology a b e)
3         (m:establish-polygon-topology b c e)
4         (m:establish-polygon-topology c d e)
5         (m:establish-polygon-topology d a e)
6         (m:c-line-order c e a)
7         (m:c-line-order b e d)))
```

Interaction Example 3.18: Kites from Diagonal Properties

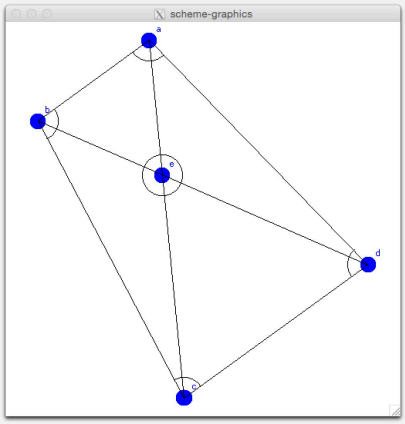
```
(define (kite-from-diagonals)
  (m:mechanism
   (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
   (m:c-right-angle (m:joint 'b 'e 'c)) ;; Right Angle in Center
   (m:c-length-equal (m:bar 'c 'e) (m:bar 'a 'e))))

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

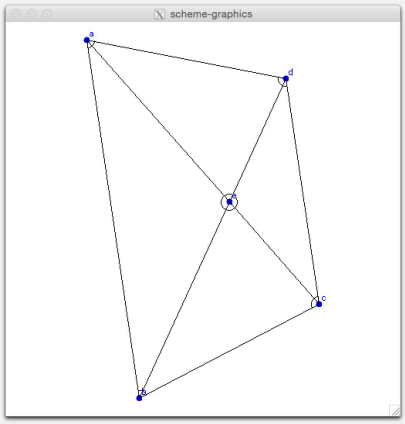




Interaction Example 3.19: Isosceles Trapezoids from Diagonals

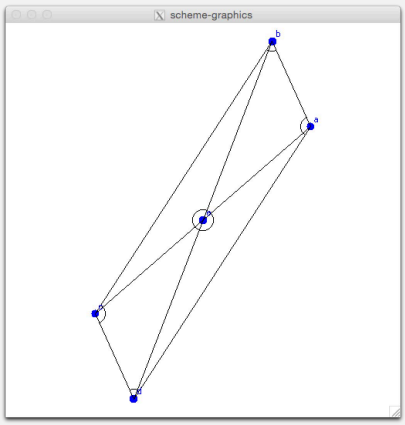
```
(define (isosceles-trapezoid-from-diagonals)
  (m:mechanism
   (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
   (m:c-length-equal (m:bar 'a 'e) (m:bar 'b 'e))
   (m:c-length-equal (m:bar 'c 'e) (m:bar 'd 'e))))

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

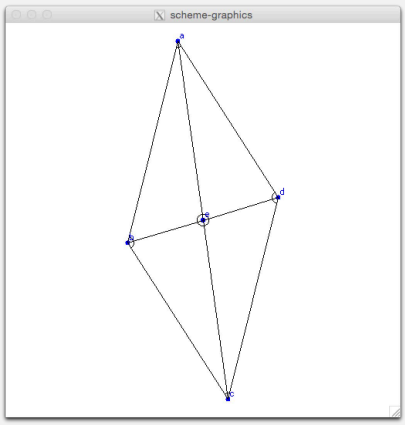




Interaction Example 3.20: Parallelograms from Diagonal Properties

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





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

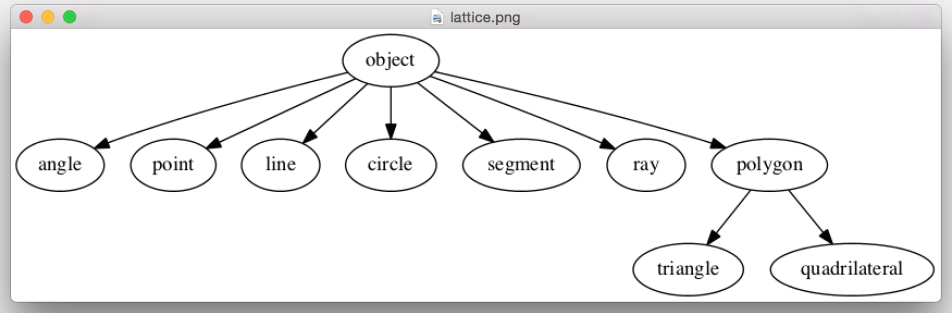
3.4 Learning Module

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

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

Interaction Example 3.21: Initial Lattice at Startup

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

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

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

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

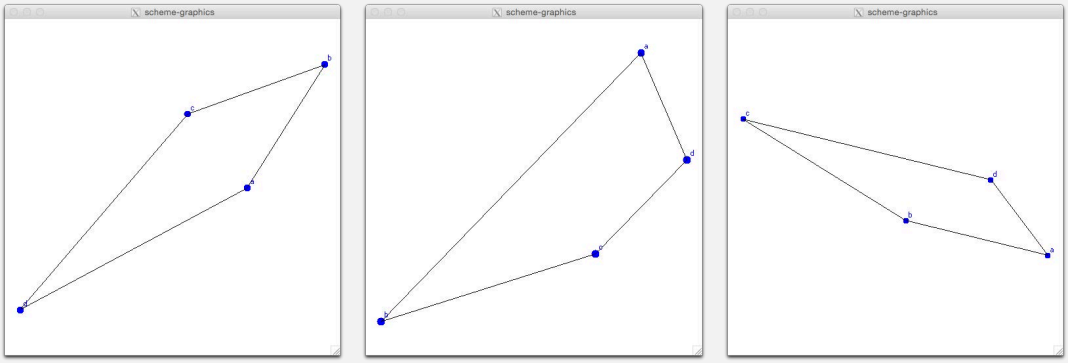


A user representing a "teacher" can interact with the system by creating investigations using these primitives. These investigations are typically steps to construct a diagram instance, but can include other specifications. The system will construct and examine the figure resulting from such investigations,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

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

Interaction Example 3.22: Random Figures

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



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

Interaction Example 3.23: Analyzing an Element

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

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

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

Interaction Example 3.24: Learning New Terms

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

=> (learn-term 'isosceles-triangle random-isosceles-triangle)
done
```

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

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

Interaction Example 3.25: Asking about Terms

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

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

Interaction Example 3.26: Testing Definitions

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

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

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

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

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

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

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

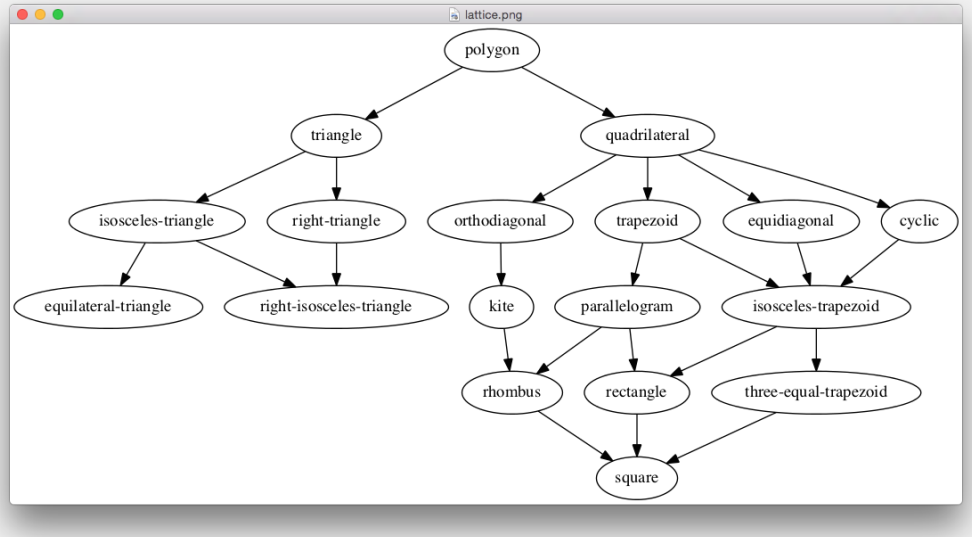
Interaction Example 3.27: Building on Definitions

```
=> (learn-term 'rhombus random-rhombus)
=> (learn-term 'kite random-kite)
=> (what-is 'rhombus)
(rhombus
 (parallelogram kite)
 ((equal-length (polygon-segment 0 1 <premise>)
                (polygon-segment 3 0 <premise>))
  (equal-length (polygon-segment 1 2 <premise>)
                (polygon-segment 2 3 <premise>))))

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

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

Interaction Example 3.28: Expanded Definition Lattice

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



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

Interaction Example 3.29: Learning Orthodiagonal Quadrilaterals

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

To handle such situations and to enable the learning module to capture more general theorems about its objects, the system allows users to specify investigations based on a premise. These investigations represent the English instructions of *"Given* `<premise objects>`*, construct* `<secondary constructions>`*. Notice anything interesting?"*. They also use the imperative construction `let-geo*` macros, but have a "dependency injected" premise argument to enable the learning module to control what is being investigated. By conditioning such constructions and analysis on the premise objects, the learning module is able to filter out observations based on previously-learned theorems and store new observations as theorems for future use.

The investigation in Example 3.30 takes a quadrilateral as its base premise and constructs a figure including the quadrilateral's diagonals.

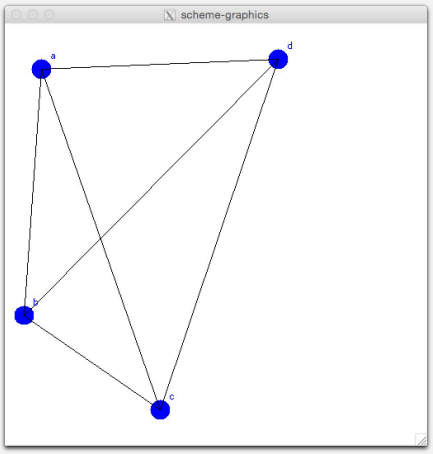
Code Example 3.30: Diagonals Investigation

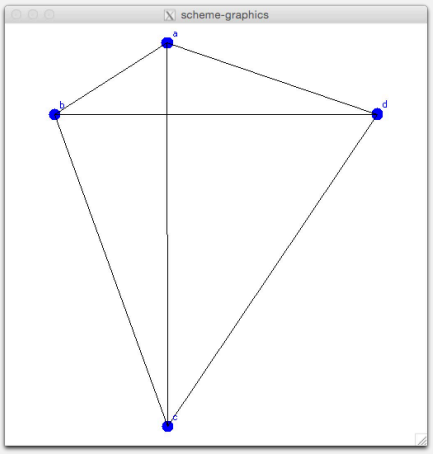
```
(define diagonal-investigation
  (make-investigation
   'quadrilateral
   (lambda (premise)
     (let-geo*
         ((((a b c d)) premise)
          (diag-1 (make-segment a c))
          (diag-2 (make-segment b d)))
       (figure premise diag-1 diag-2)))))
```

Investigations can be run either for a particular premise term or for an entire sublattice of descendants. Example 3.31 shows the results of the diagonals investigation being run for orthodiagonal and equidiagonal terms.

Interaction Example 3.31: Performing Investigations

```
=> (run-investigation-for-term diagonal-investigation 'equidiagonal)
((equal-length diag-1 diag-2))

=> (run-investigation-for-term diagonal-investigation 'orthodiagonal)
((perpendicular diag-1 diag-2))
```



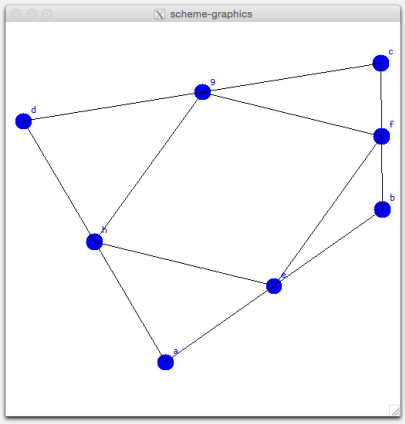


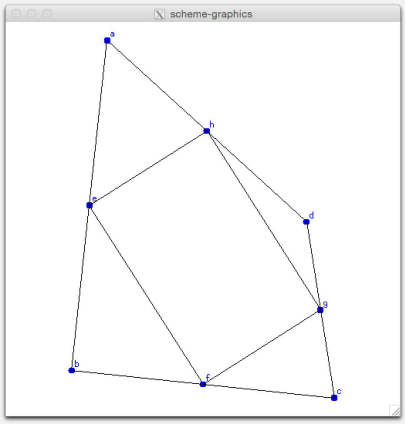
In addition to displaying the interesting new results the investigation yields, running an investigation stores discovered properties in the definition structures of the premises being checked. In the orthodiagonal and equidiagonal cases, performing this investigation correctly identifies and adds properties of the diagonals that can differentiate the shapes from arbitrary quadrilaterals and moves the terms to the correct locations in the lattice.

Running investigations on entire subtrees of related terms can often provide interesting information about where in the lattice particular properties change. Example 3.32 shows selected output from running a consecutive midpoint investigation that builds a inner quadrilateral from the midpoints of the sides of a given premise quadrilateral. The resulting interesting observations are simplified by expressing results in terms of known shapes.

Interaction Example 3.32: Consecutive Midpoint Investigation [Selected Output]

```
=> (run-investigation consecutive-midpoints-investigation)
(investigating quadrilateral)
   ((parallelogram inner-polygon))
...
(investigating equidiagonal)   ;; Left Image
   ((rhombus inner-polygon))
...
(investigating orthodiagonal)  ;; Right Image
   ((rectangle inner-polygon))
...
(investigating square)
   ((square inner-polygon))
```





Interestingly, these results show that, given any outer quadrilateral premise, the inner quadrilateral is a parallelogram, that the inner polygon for any equidiagonal quadrilateral is a rhombus, and the inner polygon for any orthodiagonal quadrilateral is a rectangle. Then, as reinforced by looking back to the full lattice in Example 3.28, it reports that the inner quadrilateral of a square (a descendent of both equidiagonal and orthodiagonal) is also a square (a descendent of both rhombus and rectangle).

Thus, user-specified investigations can represent broader explorations for the system to perform. Although not yet implemented, a similar process using a multi-element premise structure could explore relationships yielded by a applying a construction procedure to its arguments. In future executions, these observations could be marked as uninteresting and excluded from reporting.

3.5 Final Example: Simplifying Definitions

As properties accumulate from analysis and investigation, the need to satisfy all known properties for a shape over-constrains the resulting definitions. For example, satisfying some small subset of the known properties of a parallelogram is sufficient to determine whether an unknown object is a parallelogram without checking *every* property known about parallelograms.

Accordingly, the final, fun example that integrates all of these systems is the process of learning simpler definitions for geometry terms. In these examples, the procedure `get-simple-definitions` takes a known term, looks up the known properties for that term, and tests all reasonable subsets of those properties as constraints using 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 checked using with the `is-a?` procedure to see if all the *other* known properties of the original term still hold.

If so, the subset of properties is reported as a sufficient definition of the term, and if the resulting diagram fails some properties, the subset is reported as an insufficient set of constraints. These resulting sufficient definitions can be treated as equivalent, simpler definitions and used as the premises in new theorems about the objects.

In the Example 3.33, the initial necessary properties of an isosceles triangle are that it both has congruent legs *and* congruent base angles. After the definition simplification via constraint solving, we correctly discover that either of these constraints alone is sufficient: either a single pair of congruent base angles or a pair of congruent sides.

Interaction Example 3.33: Learning Simple Definitions

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

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

In the parallelogram Example 3.34, some subsets are marked as unknown because the constraint solver wasn't able to solve a diagram given those constraints. However, the results still show the interesting sufficient definitions of *either* pairs of opposite angles being equal as explored in Example 3.14 or pairs of opposite sides being equal, and correctly mark several sets of insufficient definitions as not being specific enough.

Interaction Example 3.34: Learning Simple Parallelogram Definitions

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

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

This simple definitions implementation is still a work in progress and has room for improvement. In the future I plan to use the knowledge about what properties are violated in an insufficient figure to add to the next constraint set to check, and improve how the solver handles difficult cases to construct. Further extensions could also involve generalizing `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 through 3.20.

Given this context of use cases for the modules, the remaining chapters will discuss additional representation and implementation details.

Chapter 4

System Overview

This chapter provides an overview of the system. It presents several concepts relating to input and output representations, introduces the four main modules, and discusses how they work together in the discovery of new definitions and theorems.

4.1 Goals

The end goal of the system is for it to notice and learn interesting concepts in Geometry from inductive explorations. Because these ideas are derived from inductive observation, I 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 can be grouped into three areas: properties, definitions, and theorems:

Properties Properties include all the facts derived from a single premise, such as "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). As seen in the demonstra-

tion, 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 involve relations among additional elements constructed from an initial premise. 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.

Given a repository of these conjectures about geometry, the system is 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 diagram representations. As discussed in the motivation chapter, we use the term "diagram" to represent the abstract geometric object represented by these means:

Construction Steps The main initial representation for most diagrams is a series of construction steps. These generally comprise 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 elements are built.

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

Symbolic Relationships Finally, the third representation of a diagram is as 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.

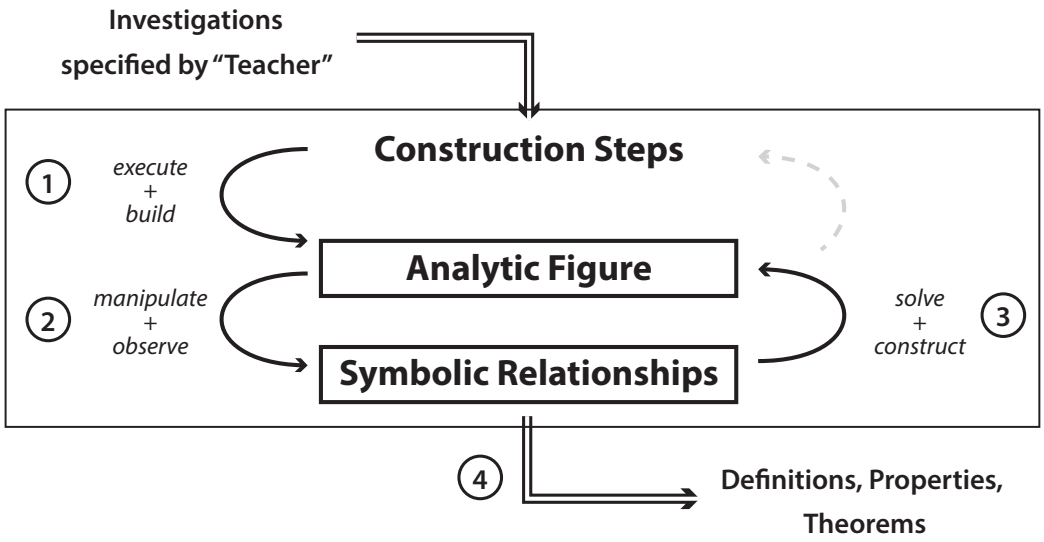
While construction steps are primarily used as input and to generate examples, as the system investigates a figure, the analytic figure and symbolic relationship models get increasingly intertwined. The "mind's eye" perception aspects of observing relationships in the analytic figure lead to new symbolic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

4.3 Steps in a Typical Interaction

The system overview figure on the next page depicts the typical process of interacting with the system and shows relationships between the four system modules.

These four modules are 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.



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 figure 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.

4.3.1 Interpreting Construction Instructions

The first step in an exploration is interpreting an input of the diagram to be investigated. The imperative construction module 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 that could be performed 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 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 are 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" images and results in an analytic representation of the figure with coordinates for each point. Arbitrary choices in the construction ("Let Q be some point 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 and Simplifying 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 initially 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. Finally, given several properties that are true of a figure, the learning module uses the constraint solver in an attempt to reconstruct a figure satisfying a subset of the constraints to determine which are essential in the original diagram.

Chapter 5

Imperative Construction System

5.1 Overview

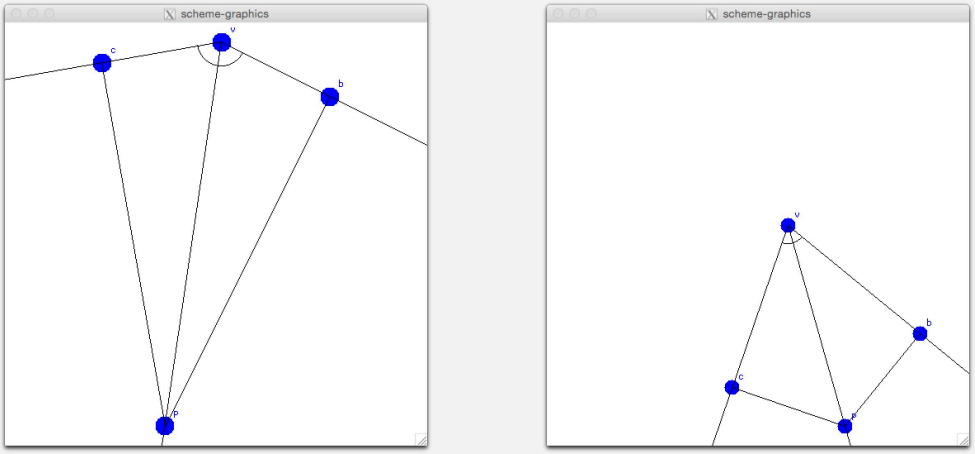
The first module is an imperative system for performing geometry constructions. This is the typical input method for generating coordinate-backed, analytic instances of diagrams.

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 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 Figure for Chapter

```
(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))
```



The sample construction in Example 5.1 constructs perpendiculars from an arbitrary point on an angle bisector to the ray extensions of the angle being bisected. It will be referenced several times in this chapter.

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, a segment is two ordered endpoints, a ray is an endpoint and a direction, and a line is a point on the line and the direction from that point the line extends. Thus, lines and segments are directioned, and the same geometric line and segment can have several different internal representations. Predicates exist to allow other procedures to work with or ignore these distinctions.

Code Listing 5.2: Basic Structures

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

(define-record-type &lt;segment&gt;
  (make-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))
```

As shown in Listing 5.3, angles are represented as a vertex point and two arm directions, and circles have a center point and radius length.

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.

Code Listing 5.4: Higher-order Constructors

```
(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?)
```

Listing 5.5 also demonstrates one of many places in the system where I use generic operations to add flexibility. Here, `angle-from-lines` is defined as the handler for the generic operation `angle-from` when both arguments are lines. Similar handlers exist for other combinations of linear elements.

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 demonstrates some utilities using directions. Similar abstractions exist for working with vectors.

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 chapter's sample construction.

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))))
```

Traditional constructions generally avoid using rulers and protractors. However, as shown in Listing 5.8, the internal implementation of the `angle-bisector` procedure uses measurements to simplify construction instead of repeatedly intersecting circle arcs to emulate compass sweeps. Although the internal implementations of some constructions use measured values, when providing the system with investigations, a user can still limit the construction steps used to ones that could be performed using only a compass and straight edge since the internal implementations of the constructions operations remain opaque to the learning module.

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

Polygon 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

Given these underlying objects and operations, to allow figures to represent general spaces of diagrams, random choices are needed when instantiating diagrams. The chapter's sample construction uses `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 that the figures stay reasonably legible for a human viewer, in `random-point-on-ray`, the procedure `extend-ray-to-max-segment` clips the 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

```
(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))))
```

Other random elements are created by combining these random choices, such as the random parallelogram in Listing 5.10. In `random-parallelogram`, a parallelogram is created by constructing two rays with an random angle between them, and selecting an arbitrary point on each. The final point is computed using vector arithmetic to "complete the parallelogram".

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 Listing 5.11 to avoid the edge cases of ranges, and some retry their local choices if the object they are returning has points too close to one another. However, further extensions could improve this system to periodically check for unintended relationships amongst all elements created previously in the figure and backtrack to select other values.

Code Listing 5.11: Safe Randomness

```
(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)))))
```

5.5 Construction Language Support

To simplify specification of figures, the module provides the `let-geo*` macro which allows for a multiple-assignment-like extraction of components from elements and automatically tags resulting elements with their variable names and dependencies. These dependencies are both symbolic for display and procedural so the system can generalize observations into conjectures that can be applied in other situations.

5.5.1 Multiple Component Assignment

Listing 5.12 shows the multiple component assignment expansion of a simple usage of `let-geo*`. In this case, `((a (r-1 v r-2)) (random-angle))` will assign to the variable `a` the resulting random angle, and to the variables `r-1`, `v`, and `r-2` the resulting angle's ray-1, vertex, and ray-2, respectively. If the specification was for a random quadrilateral, such as `((s (a b c d)) (random-square))`, the macro would assign to the variable `s` the resulting random square, and to the variables `a`, `b`, `c` and `d` the resulting square's vertices.

Interaction Example 5.12: Expansion of let-geo* macro

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

=> macro expands to:
(let* ((a (random-angle))
       (r-1 (element-component a 0))
       (v   (element-component a 1))
       (r-2 (element-component a 2)))
  (figure a r-1 r-2 ...))
```

To handle these varied cases, the macro expands to use the generic operation `element-component` to determine what components are extracted from an object during multiple component assignment. As shown in Listing 5.13, for polygons, the components are the point references directly, whereas angles and segments generate their handlers from a provided list of getters.

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?)

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

(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))))
```

Listing 5.14 demonstrates the multiple assignment portion of the `let-geo*` macro in which the user's specifications are expanded into the element-component expressions.

Code Listing 5.14: Multiple and Component Assignment Implementation

```
(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))))
```

5.5.2 Names and Dependencies

The other task the `let-geo*` macro handles is assigning names and dependencies to objects. As shown in Listing 5.15, these properties are attached to elements using the `eq-properties` methods. In this approach, a hash table is used to store mappings of elements to property values. Similar interfaces are provided for element dependencies and element sources.

Code Listing 5.15: Element Names

```
(define (element-name element)
  (or (eq-get element 'name)
      *unnamed*))

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

When an assignment is made in the `let-geo*` macro, three pieces of information are associated with the assigned object: (1) its `name`, taken from the variable used for the object in the let statement, (2) its symbolic `dependency` that stores the procedure name and arguments used to obtain the object, primarily stored for display purposes, and (3) a `source` procedure that allows the object to be recreated from a different starting premise. Example 5.16 shows the expansion of these dependencies in a very simple construction. These dependencies are attached after the multiple component assignments are expanded so will apply to all objects named in the form.

Interaction Example 5.16: Dependency and Name Assignment

```
(let-geo*
    ((s (make-segment a b)))
  (figure s))

=> macro expands to:
(let* ((s (make-segment a b)))
  (set-element-name! s 's)
  (set-source! s
   (lambda (p)
     (make-segment (from-new-premise p a) (from-new-premise p b))))
  (set-dependency! s (list 'make-segment a b))
  (figure s))
```

The decision to attach a procedure of a premise argument to an element as its `source` allows other starting premises to be injected during later explorations in the learning module. `from-new-premise` allows the system to recreate the corresponding object for a specified element given a different premise. Example 5.17 shows the implementation of `from-new-premise` and the interface for specifying an explicit premise dependency via `set-as-premise!`. To allow for multiple premises to be injected, the premise structure is represented as a list.

Code Example 5.17: Using sources with new premises

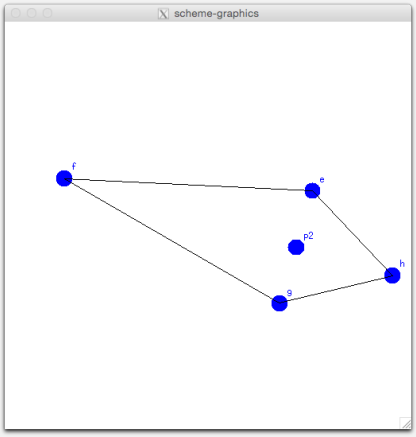
```
(define (from-new-premise new-premise element)
  ((element-source element) new-premise))

(define (set-as-premise! element i)
  (set-dependency! element (symbol '<premise- i '>))
  (set-source! element (lambda (p) (list-ref p i))))
```

These source and premise structures will be used more later in learning new terms, but Example 5.18 provides a concrete example of its use. The first definition creates a random square and obtains the intersection point of its two diagonals. `let-geo*` sets up the names and dependencies, and the square is marked as the initial premise. However, the intersection point is returned rather than a figure. The print statements (continued on the next page) show that while `diag-intersection-point` is a point structure with explicit coordinates it can produce information about how it was created via `print-dependencies`.

Interaction Example 5.18: Using from-new-premise

```
(define diag-intersection-point
  (let-geo*
      (((sq (a b c d)) (random-square))
       (diag-1 (make-segment a c))
       (diag-2 (make-segment b d))
       (p (intersect-linear-elements diag-1 diag-2)))
    (set-as-premise! sq 0)
    p))

=> (pp diag-intersection-point)
#[point 26] (x -.1071) (y -0.4464)
```

```
=> (print-dependencies (element-dependency diag-intersection-point))
(intersect-linear-elements
 (make-segment (element-component <premise-0> 0)
               (element-component <premise-0> 2))
 (make-segment (element-component <premise-0> 1)
               (element-component <premise-0> 3)))

(define new-figure
  (let-geo* (((k (e f g h)) (random-kite))
             (p2 (from-new-premise (list k) diag-intersection-point)))
    (figure k p2)))

=> (show-figure new-figure)
```



The second definition at the end of Example 5.18, `new-figure`, demonstrates using `from-new-premise` to apply source information from an existing object to a new premise. The specification of `new-figure` constructs a random kite and uses that object, `k`, as the new premise in creating point `p2` using the source information of the `diag-intersection-point`. As seen in the image, `from-new-premise` was able to correctly extract the construction steps about how `diag-intersection-point` was constructed and apply it to the new kite to specify `p2` as the intersection point of the *kite's* diagonals.

A similar process to this example will appear in an abstracted form later in the learning module as the system tests whether conjectures apply to new situations.

Listing 5.19 presents the implementation of the manipulations used to add dependency information to figures and Listing 5.20 presents the top-level definition for the `let-geo*` form.

Code Listing 5.19: Implementation of Dependency Expressions

```
(define (args-from-premise args)
  (map (lambda (arg)
         `(from-new-premise p ,arg))
       args))

(define (set-dependency-expressions assignments)
  (append-map
   (lambda (a)
     (let ((name (car a))
           (value (cadr a)))
       (if (list? value)
           (let ((proc (car value))
                 (args (cdr value)))
             `((set-source!
                ,name (lambda (p) (,proc ,@(args-from-premise args))))
               (set-dependency!
                ,name (list (quote ,proc) ,@args))))
           `((set-source! ,name (element-source ,value))
             (set-dependency! ,name (element-dependency ,value))))))
   assignments))
```

Code Listing 5.20: Full let-geo* Implementation

```
(define-syntax let-geo*
  (sc-macro-transformer
   (lambda (exp env)
     (let ((assignments (cadr exp))
           (body (cddr exp)))
       (let ((new-assignments (expand-assignments assignments))
             (variable-names (variables-from-assignments assignments)))
         (let ((result `(let*
                            ,new-assignments
                          ,@(set-name-expressions variable-names)
                          ,@(set-dependency-expressions new-assignments)
                          ,@body)))
           ;; (pp result) ;; To debug macro expansion
           (close-syntax result env)))))))
```

5.6 Graphics and Animation

Given the primitive objects and a language for specifying constructions, the final task of the imperative system is to display and animate figures. To do so, the system integrates with Scheme's graphics procedures for the X Window System. It 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 5.21 and 5.22.

Code Listing 5.21: Drawing Figures

```
(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)
     ((draw-label element) canvas))
   (all-figure-elements figure))
  (graphics-flush (canvas-g canvas)))
```

To support animation, 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 animate each successive random choice through its range of $[0, 1]$. `animate-range` provides an example where a user can specify a range to sweep over. The system uses this to "wiggle" random choices by sweeping over small ranges.

Code Listing 5.22: Animation

```
(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))))))
```

5.7 Discussion

In creating the imperative construction module, the main challenges involved settling on appropriate representations for geometry objects and properly yet effortlessly tracking dependencies. Initial efforts used over-specified object representations such as an angle consisting of three points and a line consisting of two points. Reducing these to nearly-minimal representations using directions helped simplify the creation of other construction utilities. In addition, the module initially had each individual construction procedure attach dependencies to the elements it produced. Automating this in the `let-geo*` macro helped simplify the annotation process and make the persistence of source procedures feasible.

Future extensions could provide additional construction procedures, particularly with added support for circle and arc-related operations, or improve the resilience of random choices. However, I believe the imperative module provides a sufficiently versatile library of components and procedures to enable users to specify interesting investigations. With this ability to construct and represent figures, the following chapters explain details of how the system is able to make, generalize, and learn from observations in user-specified constructions.

Chapter 6

Perception Module

6.1 Overview

The perception module focuses on "seeing" figures and simulating 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 uses several techniques to attempt to omit obvious properties, and combines with the learning module (Chapter 8) to filter already-learned discoveries and simplify results.

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 further ways to detect and remove obvious results and some attempted techniques used to extract auxiliary relationships from figures.

6.2 Relationships

Relationships are the primary structures defining what constitutes interesting properties 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

```
(define-record-type <relationship>
  (make-relationship type arity predicate equivalence-predicate) ...))

(define equal-length-relationship
  (make-relationship 'equal-length 2 segment-equal-length?
                     (set-equivalent-procedure segment-equivalent?)))

(define concurrent-relationship
  (make-relationship 'concurrent 3 concurrent?
                     (set-equivalent-procedure linear-element-equivalent?)))

(define concentric-relationship
  (make-relationship 'concentric 4 concentric?
                     (set-equivalent-procedure point-equal?)))
```

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 in Listing 6.2. `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 and Concentric Predicates

```
(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 (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))))
```

In addition to the type, arity, and predicate checked against arguments, the relationship structure also includes an equivalence predicate that is used in determining whether two observations using the relationship are equivalent, as will be discussed after explaining the observation structure in Section 6.3.

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
- sets of three concentric points with a fourth as its center.

These relationships covered most of the basic observations needed in my investigations, but further relationships can be easily added.

6.3 Observations

Observations are structures used to report the analyzer's findings. As seen in Listing 6.3, they combine the relevant relationship structure with a list of 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 6.3: Observations

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

It is important to know whether two arbitrary observations are equivalent. This enables the system to detect and avoid reporting redundant or uninteresting relationships. Listing 6.4 shows the implementation of `observation-equivalent?`. The procedure checks the observations are the same and then applies that observation's equivalence predicate to the two tuples of observation arguments.

Code Listing 6.4: Equivalent Observations

```
(define (observation-equivalent? obs1 obs2)
  (and (relationship-equal?
        (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))))
```

These equivalence predicates handle the various patterns in which objects may appear in observations. For example, in an observation that two segments have equal length, it does not matter which segment comes first or which order the endpoints are listed within each segment. Thus, as shown in Listing 6.5, the equivalence procedure ignores these ordering differences by comparing set equalities:

Code Listing 6.5: Equivalence of Equal Segment Length Observations

```
(set-equivalent-procedure segment-equivalent?)

(define (set-equivalent-procedure equality-predicate)
  (lambda (set1 set2)
    (set-equivalent? set1 set2 equality-predicate)))

(define (set-equivalent? set1 set2 equality-predicate)
  (and (subset? set1 set2 equality-predicate)
       (subset? set2 set1 equality-predicate)))

(define (segment-equivalent? s1 s2)
  (set-equivalent?
   (segment-endpoints s1)
   (segment-endpoints s2)
   point-equal?))

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

6.3.1 Numerical Accuracy

Throughout the system, numerical accuracy issues and floating point errors arise when comparing objects. As a result, the system uses custom equality operators for each data type, such as `point-equal?` shown in Listing 6.5. These use an underlying floating-point predicate `close-enuf?` taken from the MIT Scheme Mathematics Library [26] that estimates and sets a tolerance based on current machine's precision and handles small magnitude values intelligently. With this floating point tolerance in comparisons, floating point errors have been significantly less prevalent.

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 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)))

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

For the full `all-observations` procedure in Listing 6.8, the utility procedure `require-majority-animated` is used to generate random frames from wiggling the random choices in the provided figure procedure. It then only reports observations present in a majority of the frames. This corresponds to wiggling choices in a construction and observing invariant relationships.

Code Listing 6.8: All Observations from Wiggling Choices

```
(define (all-observations figure-proc)
  (require-majority-animated
   (lambda () (analyze-figure (figure-proc)))
   observation-equal?))
```

6.5 Focusing on Interesting Observations

The final task of the perception module involves filtering out obvious and previously discovered observations. Listing 6.9 shows the module's current state of accomplishing this task via the `interesting-observations` procedure. The procedure first extracts all observations from the figure and aggregates a list of obvious relations specified during the construction. It then uses the learning module to examine all polygons

found in the figure and determine the most specific definitions each satisfies. The procedure obtains all previously-discovered facts about such shapes to remove from the final result and adds new polygon observations in their place. Example 6.11 shows a concrete example of this.

Code Listing 6.9: Obtaining Interesting Observations

```
(define (interesting-observations figure-proc)
  (set! *obvious-observations* '())
  (let* ((fig (figure-proc))
         (all-obs (analyze-figure fig))
         (polygons (figure-polygons fig))
         (polygon-observations (polygon-observations polygons))
         (polygon-implied-observations
           (polygon-implied-observations polygons))
    (set-difference (append all-obs polygon-observations)
                    (append *obvious-observations*
                        polygon-implied-observations)
                    observation-equivalent?)))
```

Listing 6.10 shows the implementation of the `save-obvious-observation!` procedure. Construction procedures can use this to mark obvious relationships for the elements they create. For instance, the procedure that creates the perpendicular bisector of a segment creates and saves an observation that the line it is creating is perpendicular to the original segment before returning the bisector line.

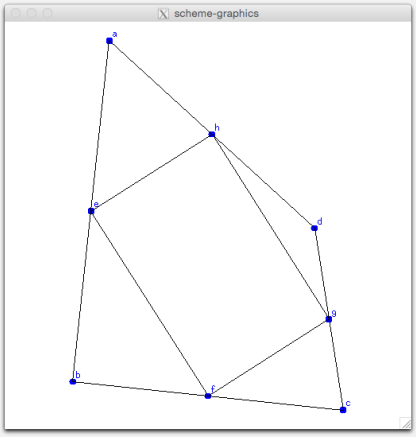
Code Listing 6.10: Marking Obvious Observations

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

Example 3.7 in the demonstration chapter demonstrated a simple example of a construction procedure that marked obvious properties of its results. Example 6.11, demonstrates the other, polygon definition-based technique of simplifying observations. Although there were 21 total observations found in the resulting figure, after examining the types of polygons in the figure and removing observations previously discovered about those elements, only two observations remain:

Interaction Example 6.11: Substituting Polygon Observations

```
(define (orthodiagonal-inner-polygon)
  (let-geo*
      (((oq (a b c d)) (random-orthodiagonal-quadrilateral))
       (e (midpoint a b))
       (f (midpoint b c))
       (g (midpoint c d))
       (h (midpoint d a))
       (inner-p (polygon-from-points e f g h)))
    (figure oq inner-p)))

=> (length (all-observations orthodiagonal-inner-polygon))
21

=> (pprint (interesting-observations orthodiagonal-inner-polygon)
((orthodiagonal oq) (rectangle inner-p))
```



6.6 Discussion and Extensions

Perfectly determining what observations are interesting or non-obvious is a large task, particularly since filtering out obvious relations often requires relationship-specific information:

As one example, imagine implementing a `collinear?` predicate that only reports non-obvious relations. Testing whether three arbitrary coordinate-based points are collinear is straightforward. However, it is not interesting that a random point on a line is collinear with the two points from which the line was defined. In order to

accurately know whether or not it is interesting that such points are collinear, the system would need to have access to a graph-like representation of which points were specified to be on which lines. Similar auxiliary structures can help filter other types of relationships.

The analysis routine was initially one large, complicated procedure in which individual checks were arbitrarily added. The restructuring to use relationships and observations has simplified the complexity and enabled better interactions with the learning module, but limited the ability for adding many relationship-specific optimizations.

Despite these limitations, the perception system has been sufficient to discover several relations via the learning model and use basic filtering of obvious relations to present intelligible results.

The examples below describe further efforts explored for improving the perception module. These involve 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, extracting angles, or merging results. These are areas for future work.

6.6.1 Auxiliary Segments

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

6.6.2 Extracting Angles

In addition, I briefly explored an implementation in which the construction module also maintains a graph-like representation of the connectedness and adjacencies in the figure. Such a representation could help with extracting angles not explicitly created in a figure. However, in addition to the complexity of determining which angles

to keep, keeping track of obvious relationships due to parallel lines and overlapping angles is quite a challenge.

6.6.3 Merging Related Observations

A final process I explored involved merging related observations into larger, combined results. For instance, when reporting segment length equality for a square, it is excessive to report all possible pairs of equal sides. I initially implemented a step to merge such observations to simply report that all four sides are congruent. However, as more relationships were added, the merge process became complicated as the arguments for all observations were not commutative and transitive. For example merging relationships about angles being supplementary to one another and merging sets of three concentric points with a fourth as its center would each require a unique merge procedure. Generalizing and adding such merge procedures would be an interesting extension.

Chapter 7

Declarative Geometry Constraint Solver

7.1 Overview

The third module is a declarative geometry constraint solver. Given a user-specified topology of a diagram and various constraints on segments and angles, this module attempts to solve the specification by instantiating 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 building and solving 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.

7.1.1 Mechanical Analogies

Mechanical analogies are often applied to mathematical problems to yield alternate, often more-intuitive solutions. Several texts such as [15] and [27] explore this and provide examples such as deriving the Pythagorean Theorem from a physical example dealing with water pressure in and torque on a rotating drum.

In this system, mechanical analogies are used to represent the physics simulation going on as one mentally manipulates a diagram "in the mind's eye". Often, given a diagram with constraints, one can imagine assembling a physical example of the figure out of bars and joints in one's head. Some bars can be sliding to make their lengths adjustable whereas others are constrained to be of equal length. As a person moves and wiggles these pieces to assemble satisfying mechanisms, they can examine whether the resulting mechanisms retain properties across instances and generalize such invariants into theorems.

This module simulates this process by assembling mechanisms of bars and joints, and using a propagator system to simulate incrementally selecting where bars and joints are positioned while maintaining local physical constraints.

7.1.2 Propagator System

The declarative geometry solver is built upon an existing propagator system created by Alexey Radul under the advisement of Gerald Jay Sussman [22]. The propagator system allows a user to create cells and connect them with propagator constraints. As content is added to cells, their neighbors are notified and updated with computations performed on the new information. Often, cells maintain a representation of partial information about their content and merge new information from several sources.

This module uses Radul's propagation system to handle the underlying propagation of data, but implements constraints, partial information types, specification protocols, and input formats particular to geometric figures.

7.2 Example of Solving Geometric Constraints

I begin by fully explaining an example. The geometry problem of inadequately constrained rectangles was introduced in the first example of Chapter 2 on page 17. The second proposed set of constraints in that problem was expressed as a mechanism in Example 3.15 in the demonstration (page 35), and is repeated here in Example 7.1. Example 7.2 shows the module's print messages as it solves the mechanism.

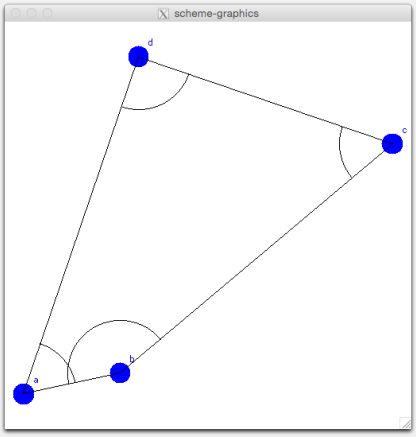
The illustrations in Explanation 7.3 and accompanying text on the following pages explain how propagation is used to solve this mechanism.

Code Example 7.1: 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))))
```

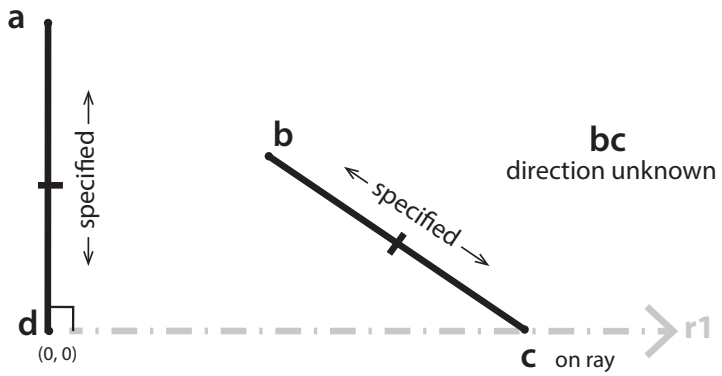
Interaction Example 7.2: Solved Constraints

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

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

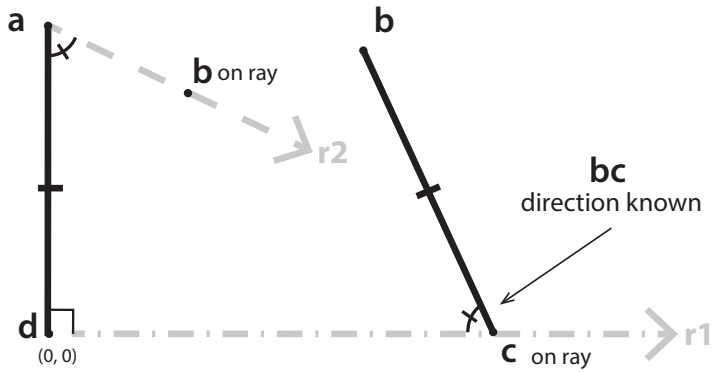


Solving a mechanism involves repeatedly selecting positions, lengths, angles, and directions that are not fully specified and selecting values within the domain of that element's current partial information. As values are specified, the wiring of the propagator model propagates further partial information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

Propagation Explanation 7.3: This series of illustrations depicts the propagation steps that occur to enable the system to solve the underconstrained rectangle from Example 7.1.

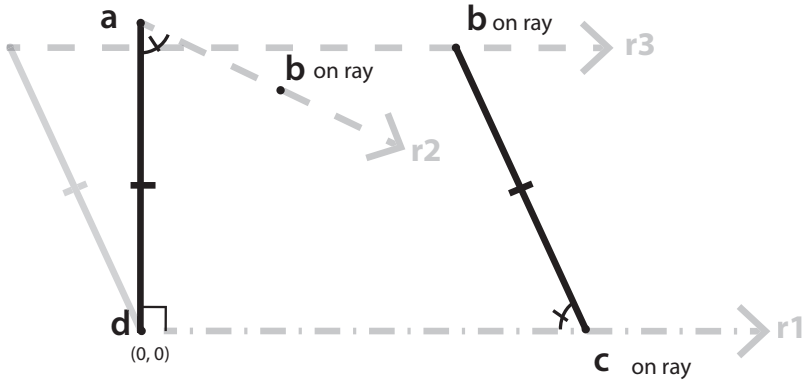


Step 1: The first value the module specifies is the length of bar `ad`. In doing so, it also initializes the bar's endpoint and direction to anchor it on the canvas. Because joint `d` is constrained to be a right angle, the system knows the direction but not length of bar `dc`. It propagates the partial information that point `c` is on the ray $r1$ extending out from `d` to the cell within point `c`. Furthermore, since bars `ad` and `bc` are constrained to have equal length, at this point, bar `bc` also knows its length but not direction. Next, the system specifies joint angle `b`:

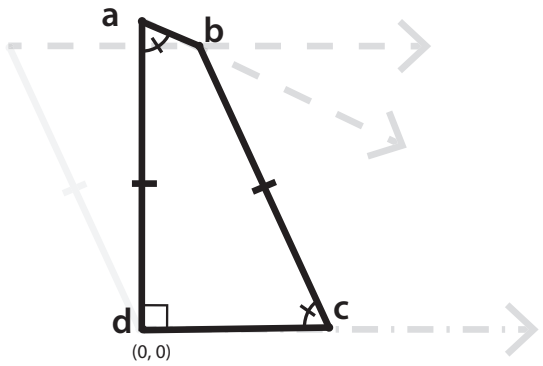


Step 2: Once the angle measure of `b` is specified, constraints using the sum of angles in the specified polygon and a "slice" constraint on the pair of constrained angles will set the angle measures of joints `a` and `c` to be half of the remaining total: $\texttt{a,c} \leftarrow \frac{2\pi - \texttt{b} - \texttt{d}}{2}$. With these angles specified, point `b` is informed that it is on the ray $r2$ and bar `bc` now knows both its length and direction.

Propagation Explanation 7.3 continued: This series of illustrations depicts the propagation steps that occur to enable the system to solve the underconstrained rectangle solved in Example 7.1.



Step 3: Since now both the length and direction of bar `bc` are known and point `c` is known to be on ray $r1$, the propagation constraints can translate this ray by the length and direction of `bc` and provide the information that point `b` must therefore also be on ray $r3$. This emulates the physical process of sliding bar `bc` along ray $r1$.



Step 4: The information about point `b` being on rays $r2$ and $r3$ is merged via ray intersection to fully determine the location of `b`. Then, once point `b` is specified, since the length and direction of bar `bc` is known, propagation sets the value and location of point `c`, yielding a fully-specified solution.

Similar steps allow propagation to solve specifications for many figures including isosceles triangles, parallelograms, and quadrilaterals from their diagonals. Several of these are shown in Section 3.3. In cases when bars have their length and one endpoint determined first, the propagators specify that the other endpoint is on an arc of a circle. The next sections describe the implementation of these partial information structures before explaining bar and joint structures and how mechanisms are built and solved.

7.3 Partial Information Structures

Radul's propagation system typically used numeric intervals for partial information. The declarative constraint solver uses some standard numeric intervals, but also uses its own module-specific partial information structures. These include `regions` and `direction-intervals`, described below:

7.3.1 Regions

Regions are the partial information structure for point locations and represent subsets of the plane where the points could be located. These could be arbitrarily complex regions of the plane, but the module currently implements point sets, rays, and arcs as shown in Listing 7.4. As new information about locations are provided, regions are merged by intersection. A contradiction region represents an empty region.

Code Listing 7.4: Region Structures

```
(define-record-type <m:point-set>
  (%m:make-point-set points) ...)

(define-record-type <m:ray>
  (%m:make-ray endpoint direction) ...)

(define-record-type <m:arc>
  (m:make-arc center-point radius dir-interval) ...)

(define-record-type <m:region-contradiction>
  (m:make-region-contradiction error-regions) ...)

(defhandler merge m:intersect-regions m:region? m:region?)
```

7.3.2 Direction Intervals

In addition, a module-specific direction interval structure is used for the partial information about directions. Several additional utilities were needed for working with and merging direction intervals since directions form a periodic range $[0, 2\pi)$. Currently, the subsystem treats an intersection of direction intervals that would yield multiple distinct direction intervals as providing no new information.

7.4 Bar and Joint Constraints

The solver uses bar and joint linkages to represent segments and angles. These structures are composed of propagator cells storing information about locations, lengths, directions, and angles. To assist with some of the propagation between these cells, the module uses substructures for points and vectors.

Point structures contain both numeric Cartesian coordinate cells and a cell containing region structures. The propagators `m:x-y->region` and `m:region->x,y` transform location information between these representations.

Code Listing 7.5: Points and Regions

```
(define (m:make-point)
  (let-cells (x y region)
    (p:m:x-y->region x y region)
    (p:m:region->x region x)
    (p:m:region->y region y)
    (p: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))
(propagatify m:region->x)
```

Vectors represent the difference between two points and bidirectionally constrain both rectangular and polar information.

Code Listing 7.6: Vectors

```
(define (m:make-vec)
  (let-cells (dx dy length direction)
    (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)
    (p:m:make-vec dx dy length direction)))
```

7.4.1 Bar Structure and Constraints

As seen in Listing 7.7, bar structure contains two `m:point`s and a `m:vec` representing the distance and direction between the points. The bar links these structures together using simple bidirectional constraints on the coordinates. These constrains will only propagate information when the bar's length and direction are fully specified. `m:p1->p2-bar-propagator` and its reverse handle the other cases.

Code Listing 7.7: Basic Bar Structure

```
(define (m:make-bar bar-id)
  (let ((p1 (m:make-point))
        (p2 (m:make-point))
        (v (m:make-vec)))
    (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)))
```

The propagators specified by `m:p1->p2-bar-propagator` shown in Listing 7.8 propagate partial information about point locations based on whether the bar's direction or length is determined. `m:x-y-length-di->region` handles the case where only the length of the bar is specifies and adds information to the other endpoint's region cell that it is on the arc formed from the bar's length and current direction interval. This implementation is seen in Listing 7.8.

Code Listing 7.8: Bar Region Propagator

```
(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: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))
```

7.4.2 Joint Structure and Constraints

Joints are represented by a vertex point, two directions, and an angle representing the measure between the directions. Propagators bidirectionally constrain the angle measure to reflect and update the ranges of the joint's directions. Special mechanism-specific operators adding and subtracting directions were created since both the direction and angle argument could be intervals. Creating a joint also initializes its measure to the range $[0, \pi]$, reflecting the maximum angle sweep.

Code Listing 7.9: Joint Constraints

```
(define (m:make-joint)
  (let ((vertex (m:make-point)))
    (let-cells (dir-1 dir-2 theta)
      (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))))
```

7.5 User-specified Constraints

In addition to constraints resulting from the bar and joint connections, users can specify additional constraints on the mechanism. Listing 7.10 shows the structure for a user constraint. These structures include a name, a list of bar or joint identifiers the constraint constrains, and a procedure used to apply the constraint.

Code Listing 7.10: User Constraints

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

Code Listing 7.11: Bar Length Equality

```
(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))))))
```

This constraint procedure takes the assembled mechanism as its argument. As shown in Listing 7.11, such procedures typically look up mechanism elements by bar or joint identifiers and introduce additional constraints. In `m:c-length-equal`, the lengths of the two bars are set to be identical to one another.

7.5.1 Slice Constraints

In addition to general user constraints, mechanisms are also support slice constraints. These slices are structured in the same manner as constraints but are applied after all other user constraints, and thus can use information about user constraints in adding their propagators. In particular, the system uses slices to determine the values of cells that are constrained as equal to one another within a sum, once the total of the sum and all other cells in the sum have been determined. This process is inspired by Gerald Jay Sussman's use of slices to represent local patterns and help determine values in propagation networks for circuit design [25].

7.6 Assembling Mechanisms

Mechanism structures in the declarative system are the analogs of figures from the imperative system. Here, instead of grouping geometry elements, the mechanism group linkages and constraints. As seen in Listing 7.12, `m:mechanism` will flatten and separate its arguments. Then, in addition to storing the components in a record structure, `m:make-mechanism` will also build hash tables for looking up bars and joints by their endpoint and vertex names.

Code Listing 7.12: Mechanism Structure

```
(define-record-type <m:mechanism>
    (%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))))
```

To assist with specifying the bars and joints for a closed polygon, the utility `m:establish-polygon-topology` is often used. The procedure takes n vertex names as its arguments and returns n bars and n joints. It uses the linkage constructors `m:make-named-`$*$ to attach names to the structures. Such names are later used to attach linkages to one another and to lookup elements in constraint procedures.

Code Listing 7.13: 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)))))))
```

Once specified, mechanisms can be assembled using `m:build-mechanism`. That procedure first identifies all joint vertices with the same names as being identical to one another to handle topologies in which multiple joints share vertices. Then it assembles bars and joints based on their names.

Code Listing 7.14: Building Mechanisms

```
(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))
```

When assembling the mechanism, bars are identified into or out of the arms of joints that share their names. Joints names refer to the three vertices they connect and bar names refer to their two endpoint vertices. `m:identify-into-arm-1` (7.15) demonstrates how bars and joints get attached to one another. Corresponding point locations and directions are constrained to be identical to one another via `c:id`. Identifying two points involves identifying all of its component properties.

Code Listing 7.15: Identifying points

```
(define (m:identify-into-arm-1 joint bar)
  (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-points p1 p2)
  (for-each (lambda (getter)
              (c:id (getter p1) (getter p2)))
            (list m:point-x m:point-y m:point-region)))
```

7.7 Solving Mechanisms

Once assembled, mechanisms can be solved via `m:solve-mechanism`. Solving a mechanism involves repeatedly selecting position, lengths, angles, and directions that are not fully specified and selecting values within the domain of that element's current partial information structure. As values are specified, the constraint wiring of the propagator model propagates updated partial information to other values.

Code Listing 7.16: Solving Mechanisms

```
(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))))
```

The ordering of what is specified is guided by a heuristic in `m:specify-something` (7.17). This heuristic was determined empirically and helps the majority of the examples I explored converge to solutions. It generally prefers specifying the most constrained values first. However, in some scenarios, specifying values in the wrong order can yield premature contradictions. Additionally, sometimes partial information about a value is incomplete and picking a value arbitrarily may fail. A planned extension will attempt to recover from such situations more gracefully by trying other values or orderings for specifying components.

Code Listing 7.17: Specifying and Instantiating Values

```
(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:initialize-bar-if m m:bar-length-specified?)
   ...)
```

The system uses `m:instantiate` to add content to cells. As seen in Listing 7.18, `m:instantiate` wraps the value in a truth maintenance system structure provided by Radul's propagator system. These structures maintain dependencies for values, can report which sets of premises are at odds with one another and allow individual choices to be removed and replaced with new values.

Code Listing 7.18: Instantiating Values with TMS

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

7.7.1 Interfacing with imperative diagrams

Finally, as shown in Listing 7.19, `m:mechanism->figure` can convert fully specified mechanisms into their corresponding figures so they can be observed and analyzed.

Code Listing 7.19: Converting to Figure

```
(define (m:mechanism->figure m)
  (let ((points (map m:joint->figure-point (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 (filter identity (append points segments angles)))))
```

7.8 Discussion and Extensions

The process of incrementally specifying values and propagating properties implied by constraints is able to solve many geometry constraint problems. Radul's propagator network framework helps with propagating local constraints, representing partial information, and merging updates.

Although the module successfully solves many useful mechanism configurations, adding propagation alone is not a magic wand. Even with selecting values based on updated partial information and heuristics for choosing items to specify, there are several instances in which the module can fail to solve a mechanism specification that actually has a solution. Because of such false negatives, the constraint solver is never able to report that a set of constraints is infeasible, just that it hasn't been able to produce a solution. This works for my main use cases as the module is typically used to explore the diversity represented by subsets of satisfiable constraints.

As an example with premature contradictions, imagine the system attempting to solve a specification that should yield an isosceles trapezoid and that the angle measures and non-parallel side lengths have already been determined. The remaining step to fully specify the polygon is to determine how long the parallel sides are. If the shorter parallel side is selected to be specified first, any length value chosen will yield a valid solution. However, if the longer parallel side is selected to be specified first, choosing too small of a length value yields a contradiction since the shorter parallel side must be additively shorter than the longer one.

The two main ways of alleviating this problem are to change the order of how elements are selected to be specified and to change how values are chosen. These are the focus of several proposed extensions to the system:

7.8.1 Backtracking

One approach to handling the fact that certain orders and value selections are better than others is to backtrack and retry previous choices when contradictions occur. This could involve both backtracking and retrying different values for a certain specification or choosing different orderings of bars and joints to specify.

I started implementing this ability but ended up focusing efforts elsewhere. The current system only has support for retrying an entire figure specification on failure. However, the module does already use a feature of the underlying propagator system that tracks dependency and contingency information. Thus, the task of identifying and replacing the choices that led to such contradictions should be rather straightforward. Deciding what values to try in a choice's place, possibly through a binary search-like process, is more complicated.

7.8.2 Improved Partial Information

In the isosceles trapezoid case, computing the minimum feasible length is possible given sufficient information. However adding such computations to the system would require measuring and representing distances between more complicated region struc-

ture representations. Although such extensions may fix some cases, it still does not solve the general problem of the system sometimes failing to solve some otherwise feasible constraint sets.

7.8.3 Basing Choices on Existing Values

A final idea for an extension to improve value selection is to base values chosen as slight variations on an already-satisfied solved instance of the constraint specification. Although this wouldn't help with solving general specifications, in the typical use case the learning module is testing declarative specifications for which it already has one solution instance to see what other instances exist. Choosing values in such a manner may limit the diversity of solutions found but could eliminate some extreme value choices made in the existing system that lead to contradictions.

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 defines 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 learns 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.

Sections 3.4 and 3.5 in the demonstration chapter included several use cases and examples of working with the learning module. As a result, this discussion will focus on structures and implementation rather than uses and applications. Refer to the demonstration for examples.

8.2 Learning Module Interface

As seen in the demonstration, the learning module defines 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 explore. Listing 8.1 shows the implementation for some of these methods.

Code Example 8.1: Learning System Interface Examples

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

(define (example-object term)
  ((definition-generator (lookup term))))

(define (show-example term)
    (show-element (example-object term))

(define (is-a? term obj)
  (let ((def (lookup term)))
    (definition-holds? def obj)))

(define (examine object)
  (let ((satisfying-terms
         (filter (lambda (term) (is-a? term object))
           (known-terms))))
    (remove-supplants more-specific? satisfying-terms)))
```

Explaining these interface implementations serves as a context for introducing the representations of definitions and conjectures.

8.3 Querying

Users can query the system's knowledge using `what-is`. When queried, the system uses `lookup` to find a definition from its dictionary. Printing this definition provides the classification (that a rhombus is a parallelogram) and a set of properties that differentiates that object from its classification. Further requests can present all known properties of the named object or generate a minimal set of properties needed to specify the object.

8.3.1 Student Structure

Internally, geometry knowledge is stored in a `student` object that maintains a definition `dictionary` mapping terms to definitions and a `term-lattice` representing how these definitions relate to one another. Listing 8.2 demonstrates how the interfaces above use a global `*current-student*` variable to access information. Although the system currently only ever instantiates one student, this architecture provides the flexibility to teach or compare multiple students in the future.

Code Listing 8.2: Student Structure

```
(define-record-type <student>
  (%make-student definition-dictionary term-lattice) ...)

(define (student-lookup-definition s name)
  (hash-table/get (student-dictionary s) name #f))

(define *current-student* (make-initialized-student))

(define (lookup-definition term)
  (student-lookup-definition *current-student* term))

(define (lookup term)
  (or (lookup-definition term) (error "Term Unknown:" term)))
```

8.3.2 Definition Structure

Code Listing 8.3: Definition Structure

```
(define-record-type <definition>
  (%make-definition name generator primitive-predicate
                    primitive?
                    all-conjectures
                    classifications specific-conjectures) ...)
```

Listing 8.3 shows the implementation of definition structures. Definitions combine the name and generator procedure provided when originally learning the definition with a list of all conjectures known about that class of object. `primitive?` is a boolean indicator of whether the definition is a primitive, built-in definition. In such

cases, `primitive-predicate` is an imperative system predicate that tests whether an object satisfies the definition. In non-primitive definitions, the `primitive-predicate` is that of the primitive that the definition is a specialization of. Storing and checking against this primitive predicate prevents inapplicable operations from being performed such as attempting to obtain the angles of a segment object.

The last two fields, `classifications` and `specific-conjectures`, are derived fields that are updated based on the definition's relation to other terms in the lattice. A definition's `classifications` are the next-least specific terms that its class of objects also satisfy and `specific-conjectures` are added conjectures that differentiate the definition from being the union of those classification definitions.

8.4 Testing Definitions

The learning module provides the `is-a?` procedure to test whether a given object satisfies a known term. As shown in Listing 8.4, testing whether a definition holds involves ensuring that it is the right type of primitive object by checking the underlying primitive predicate and then ensuring the relevant conjectures are satisfied.

In this nonrecursive version, the system checks that an object satisfies *all* known conjectures. A recursive version shown later first checks that it satisfies the parent classifications before checking definition-specific conjectures that differentiate it from its classifications.

Code Listing 8.4: Definition Checking

```
(define (definition-holds-nonrecursive? def obj)
  (let ((all-conjectures (definition-conjectures def)))
    (and ((definition-primitive-predicate def) obj)
         (every (lambda (conjecture)
                  (satisfies-conjecture? conjecture (list obj)))
                all-conjectures))))
```

8.4.1 Conjecture Structure

Conjectures are similar to observations in that they associate a perception relationship with information about what satisfies the relationship. However, instead of associating a relationship with actual elements that satisfy the relationship, conjectures abstract this observation by storing only the symbolic dependencies and source procedures of those arguments.

Similar to how Example 5.18 in the imperative system used the element source procedures to obtain constructed elements corresponding to those observed in an original diagram, satisfying a conjecture involves applying its source-procedures to a new premise structure to obtain new relationship arguments. These new arguments are then checked to see if they satisfy the underlying relationship. This process is shown in Listing 8.5. The interface procedure `is-a?` creates a list of the object in question to use as the new premise.

Code Listing 8.5: Conjecture Structure

```
(define-record-type <conjecture>
  (make-conjecture dependencies source-procedures relationship) ...)

(define (satisfies-conjecture? conj premise-instance)
  (or (true? (observation-from-conjecture conj premise-instance))
      (begin (if *explain* (pprint `(failed-conjecture ,conj)))
             #f)))

(define (observation-from-conjecture conj premise-instance)
  (let ((new-args
         (map (lambda (construction-proc)
                (construction-proc premise-instance))
          (conjecture-construction-procedures conj)))
        (rel (conjecture-relationship conj)))
    (and (relationship-holds rel new-args)
         (make-observation rel new-args))))
```

8.5 Examining Objects

Given these tests, `examine`, the last interface function shown in Listing 8.1 allows a user to provide a geometry object and ask the system to examine it and report what

it is. Its implementation (in Listing 8.1) first determines all terms that apply to an object and then removes terms that are supplanted by others in the list. It uses the procedure `more-specific?` to determine which terms supplant others. As shown in Listing 8.6, this procedure checks if an example object of the proposed less specific term satisfies the definition of the proposed more specific term.

Code Listing 8.6: Relations among terms

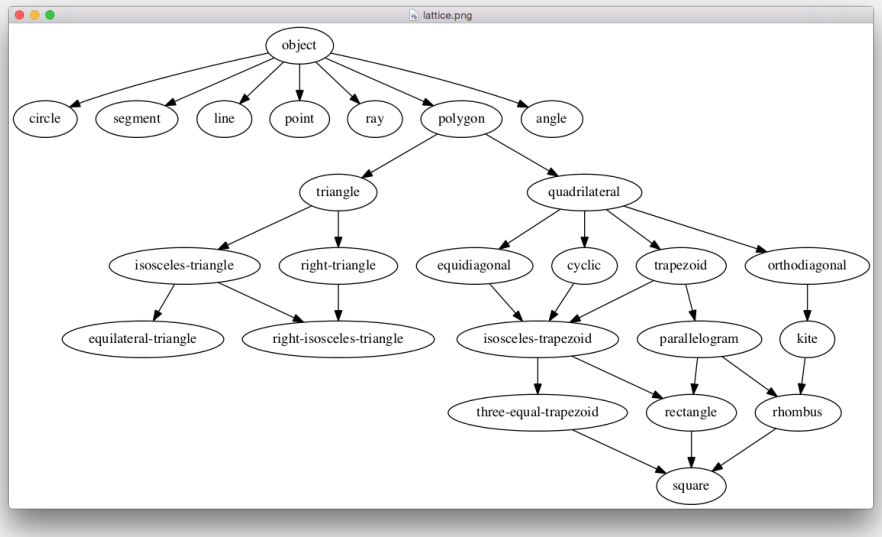
```
(define (more-specific? more-specific-term less-specific-term)
  (let ((more-specific-obj (example-object more-specific-term)))
    (is-a? less-specific-term more-specific-obj)))
```

8.5.1 Maintaining the Term Lattice

In addition to helping remove redundant information in results, this partial order on terms is used to build and maintain a lattice of terms in the student structure. This lattice can be rendered to a figure using dot/Graphviz as shown in Example 8.7.

Interaction Example 8.7: Full Definition Lattice

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



The definition lattice is implemented as a general lattice data structure I created that can be used with any partial order comparator. It correctly positions nodes and updates the relevant parent and child pointers as nodes are added and removed.

Information from the lattice is used to update the derived definition fields. As seen in Listing 8.8, after a new definition term is added to the lattice, it and its child terms (determined from lattice) are updated. The immediate parent nodes in the lattice become the definition's `classifications`. Then `definition-specific-conjectures` is updated to be the set difference of the definition's current conjectures and the conjectures known about its ancestors in the lattice.

Code Listing 8.8: Updating Terms from Lattice

```
(define (add-definition-lattice-node! term)
  (add-lattice-node (definition-lattice) (make-lattice-node term term))
  (update-definitions-from-lattice (cons term (child-terms term))))

(define (update-definition-from-lattice term)
  (let* ((def (lookup term))
         (current-conjectures (definition-conjectures def))
         (ancestor-terms (ancestor-terms term))
         (ancestor-defs (map lookup ancestor-terms))
         (ancestor-conjectures
          (append-map definition-conjectures ancestor-defs))
         (new-conjectures
          (set-difference current-conjectures
                          ancestor-conjectures
                          conjecture-equivalent?)))
    (set-definition-classifications! def (parent-terms term))
    (set-definition-specific-conjectures! def new-conjectures)))
```

This lattice structure allows terms definitions to build off of one another and allows definitions to report only definition-specific conjectures. These updated classification and definition-specific properties are also used in the full version of checking when a definition holds as shown in Listing 8.9. This version checks that a definition satisfies all parent classifications first before checking the definition-specific conjectures that differentiate it from those classifications.

Code Listing 8.9: Recursive Definition Holds

```
(define (definition-holds? def obj)
  (let ((classifications (definition-classifications def))
        (specific-conjectures (definition-specific-conjectures def)))
    (and ((definition-predicate def) obj)
         (every (lambda (classification-term)
                  (is-a? classification-term obj))
                classifications)
         (every (lambda (conjecture)
                  (satisfies-conjecture? conjecture (list obj)))
                specific-conjectures))))
```

8.5.2 Core Knowledge

To initialize the system, the student structure is provided with several primitive definitions at startup as shown in Listing 8.10.

Code Listing 8.10: Introducing Core Knowledge

```
(define (provide-core-knowledge)
  (for-each add-definition! primitive-definitions))

(define primitive-definitions
  (list
   (make-primitive-definition 'object true-proc true-proc)
   (make-primitive-definition 'point point? random-point)
   (make-primitive-definition 'line line? random-line)
   ...
   (make-primitive-definition 'triangle triangle? random-triangle))
```

8.6 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 8.11: Learning a new term

```
(define (learn-term term object-generator)
  (if (term-known? term) (error "Term already known:" term))
  (let ((term-example (name-polygon (object-generator))))
    (let* ((primitive-predicate (get-primitive-predicate term-example))
           (fig (figure (as-premise term-example 0)))
           (observations (analyze-figure fig))
           (conjectures (map conjecture-from-observation observations)))
      (pprint conjectures)
      (let ((new-def
             (make-definition term object-generator
                primitive-predicate conjectures)))
        (add-definition! new-def)
        (check-new-def new-def)
        'done))))

(define (conjecture-from-observation obs)
  (make-conjecture
   (map element-dependencies->list (observation-args obs))
   (map element-source (observation-args obs))
   (observation-relationship obs)))
```

Listing 8.11 shows the implementation of the `learn-term` procedure. It uses the provided generator procedure to produce an example object for the term, creates a figure with that object as its premise and obtains observations. These observations are converted to conjectures via `conjecture-from-observation` and the resulting definition is added to the student dictionary and term lattice.

8.6.1 Performing Investigations

As demonstrated in Example 3.30 (page 44), the learning module also supports investigations to learn conjectures based on elements constructed from base premises. Performing investigations are similar to learning terms except that, rather than providing a procedure that just generates an example of the term in consideration, an investigation uses a procedure which takes an instance of the premise (polygon in these cases) and constructs an entire figure to analyze. In addition to reporting the interesting observations of such investigations, conjectures for new observations derived by that investigation are added to the definition for the term under investigation.

8.7 Simplifying Definitions

As properties accumulate from analysis and investigation, the need to satisfy all known properties for a shape overconstraints the resulting definitions. Thus, the final role of the learning module is to simplify term definitions by checking declarative constraints.

As seen in Listing 8.12, `get-simple-definitions` takes a known term, looks up the known properties for that term, and tests all reasonable subsets of those properties as constraints using 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 checked using with the `is-a?` procedure to see if all the other known properties of the original term still hold.

If so, the subset of properties is reported as a sufficient definition of the term, and if the resulting diagram fails some properties, the subset is reported as an insufficient set of constraints. These resulting sufficient definitions can be treated as equivalent, simpler definitions and used as the premises in new theorems about the objects.

Code Listing 8.12: Simplifying Definitions

```
(define (get-simple-definitions term)
  (let ((def (lookup term))
        (simple-def-result (make-simple-definitions-result)))
    (let* ((object ((definition-generator def)))
           (fig (figure (as-premise (name-polygon object) 0)))
           (all-observations (analyze-figure fig))
           (eligible-observations
            (filter observation->constraint all-observations)))
      (for-each
       (lambda (obs-subset)
         (if (simple-def-should-test? simple-def-result obs-subset)
             (let ((polygon
                    (polygon-from-object-observations object obs-subset)))
               ((cond ((false? polygon) mark-unknown-simple-def!)
                      ((is-a? term polygon) mark-sufficient-simple-def!)
                      (else mark-insufficient-simple-def!))
                simple-def-result obs-subset)
               (simplify-definitions-result! simple-def-result))
             (pprint `(skipping ,obs-subset))))
       (shuffle (all-subsets eligible-observations)))
      simple-def-result)))
```

The `simple-definitions-result` structure maintains information about what subsets are known to sufficient or insufficient as the analysis proceeds and provides the predicate `simple-def-should-test?` to skip over subsets where the result is already known.

The main workhorse in this definition simplification process is the procedure `polygon-from-object-observations`. It interfaces with the constraint solver via `observations->figure` to convert observations back into a figure. Its implementation is shown below in Listing 8.13. The object provided is used to determine the topology and names of bars and linkages in the mechanism and the observation structures are used to add the necessary mechanism constraints. If the declarative system can solve the mechanism, it once again uses the element names to extract and return the resulting object.

Code Listing 8.13: Converting Observations to a Figure

```
(define (polygon-from-object-observations object obs-subset)
  (let* ((topology (topology-for-object object))
         (new-figure (observations->figure topology obs-subset)))
    (and new-figure (object-from-new-figure object new-figure))))

(define (establish-polygon-topology-for-polygon polygon)
  (let* ((points (polygon-points polygon))
         (vertex-names (map element-name points)))
    (apply m:establish-polygon-topology vertex-names)))

(define (observations->figure-one-trial topology observations)
  (initialize-scheduler)
  (let* ((constraints (observations->constraints observations))
         (m (m:mechanism topology constraints)))
    (m:build-mechanism m)
    (and (m:solve-mechanism m)
         (let ((fig (m:mechanism->figure m)))
           (show-figure fig)
           fig))
```

8.8 Discussion

The learning module has been able to successfully integrate with the other system modules to discover and learn dozens of simple elementary geometry terms and theorems through its investigations. These include simple properties such as "the base angles in an isosceles triangle are congruent," derived properties such as "the diagonals of a rhombus are orthogonal and bisect one another" or "the polygon found by connecting consecutive side midpoints of an orthodiagonal quadrilateral is always a rectangle," and simplified definitions such as "a quadrilateral with two pairs of congruent opposite angles is a parallelogram."

The current system has focused on discoveries related to polygons. Further extensions of the module could explore ideas related to other object types (segments, lines, circles) or derive conjectures that depend on several arbitrary choices. Finally, an interesting extension of the learning module would be to investigate properties about constructions. This would be similar to a teacher instructing a student "this is how you create a perpendicular bisector..." The student could then independently explore creating perpendicular bisectors of various elements so that the system could infer what interesting properties such constructions yield and omit those observations when that construction is used.

Chapter 9

Related Work

The topics of working with geometry theorems and diagrams have rich histories yet are still areas of active research.

As a seminal paper in the field, in the early 1960s, Herbert Gelernter created a "Geometry Theorem Proving Machine" [8]. His machine focused on a deductive process to search for proofs and used a formal system based on strings of characters. In addition to purely logic-based inference rules, the system also asks the user requesting a proof to provide a coordinate-backed diagram against which the system checks various subgoals it is considering in a proof.

Despite this long history, several examples of related work are still found in the proceedings of annual conferences such as *Automated Deduction in Geometry* [29] 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 [23].

The main areas of work related to my thesis are automated geometry theorem proof, automated geometry theorem discovery, and mechanical analogs of geometry concepts. After explaining some systems in these areas, I will discuss further related work including descriptions of the educational impacts of dynamic geometry

approaches and some software to explore geometric diagrams and proofs.

Some systems use techniques similar to those in this system's modules, but most approaches focus on deductive proof or complicated algebraic reformulations rather than inductive reasoning and exploration.

9.1 Automated Geometry Proof

As opposed to my system which focuses on modeling a student's investigations and *discoveries* about geometry, the main focus of historic Artificial Intelligence efforts related to geometry was obtaining *proofs* for theorems given by a user. Projects explored both algebraic and synthetic approaches, some of which involved using diagrams in addition to purely symbolic manipulations [3], [9], [18]. Texts such as [11] include a more detailed history and description of such systems. These systems are reasonably powerful but generally produce long proofs.

9.2 Automated Geometry Discovery

Several papers also describe automated *discovery* in geometry. However, most of these use alternate, more algebraic methods to find and later prove theorems. These approaches include an area method [20], Wu's Method involving systems of polynomial equations [6], and a system based on Gröbner Bases [16]. Some papers discuss reasoning systems including the construction and application of a deductive database of geometric theorems [4]. However, all of these methods focused on either deductive reasoning or complex algebraic reformulations.

The effort closest to my system's approach is Chen, Song and Wang's "Automated Generation of Geometric Theorems from Images of Diagrams" [2]. This paper includes an initial section with several image processing algorithms for detecting points and segments from images. It then applies a series of heuristic strategies to determine which elements are particularly relevant and propose candidate theorems. These strategies generally involved assigning weights to points to determine which are

"characteristic points" or "points of attraction." By doing so, their system successfully proposed several nontrivial theorems that the original image could have been illustrating. Integrating some of these strategies into my system would be an interesting extension.

9.3 Geometry Constraint Solving and Mechanics

Ideas about solving geometry diagram constraints are related to the fields of kinematic mechanisms and computer-aided design. Glenn Kramar provides a system for solving geometry constraints in mechanisms [14], but focuses on several practical three-dimensional case studies with complicated joints. Summaries such as [12] provide more information about other graph-based, logic-based, and algebraic methods for solving 2D geometry constraints. My system builds on a propagator system by Alexey Radul and Gerald Jay Sussman [21] and applies it to simple geometry constraints.

9.4 Dynamic Geometry

From an education perspective, there are several texts that emphasize an investigative, conjecture-based approach to teaching. These include *Discovering Geometry* by Michael Serra [24], the text I used to learn geometry and that served as an inspiration to this thesis project. Some researchers praise these investigative methods [19] while others question whether they appropriately encourage deductive reasoning skills [13].

9.5 Software

Some of these teaching methods include accompanying software such as Cabri Geometry [7] and the Geometer's Sketchpad [10] designed to enable students to explore constructions interactively. These programs occasionally provide scripting tools, but have no theorem or proof-related automation.

A few more academic analogs of these programs introduce some proof features. For instance, GeoProof [17] integrates diagram construction with verified proofs using a number of symbolic methods carried out by the Coq Proof Assistant, and Geometry Explorer [28] uses a full-angle method of chasing angle relations to check assertions requested by the user. However, none of the software described simulates or automates the exploratory, inductive investigation process used by students first discovering new conjectures.

One interesting piece of software is Geometer [5] created by Tom Davis. Like the other programs, Geometer is primarily a user interface for accurately constructing diagrams. It does not attempt to produce or prove theorems, but does have a "Test Diagram" mode. When this mode is activated, the user can wiggle elements in the diagram as they please. When "End Test" is selected, the program lists all features that were maintained during the users' manipulations. The creator claims that these observations can be useful pieces for a user attempting to deductively prove a theorem about the figure they are drawing. This is a similar to the observations and manipulations in my system but requires the user to manually manipulate elements in the figure rather than automatically arbitrary choices in a specified construction.

Chapter 10

Conclusion

10.1 Overview

The system presented in this thesis provides a versatile framework for building, exploring, and analyzing geometry diagrams. As shown in the demonstrations, the modules can both be used independently to construct and analyze interesting properties in geometric figures, and combined with one another to discover new geometry concepts. By constructing and examining figures, generalizing observations, solving constraints, and aggregating results, the system has been able to discover, learn, and simplify dozens of elementary geometry properties and theorems.

In doing so, the process modeled and emulated the human-like process of imagining and manipulating instance of problems "in the mind's eye" to better understand new concepts. By focusing on noticing interesting invariants in externally specified investigations, it simulates the effectiveness of an investigative-based approach to learning and discovering geometry concepts.

Although the architecture of the four interrelated imperative construction building, perceiving, declarative constraint solving, and learning modules serves as a proof of concept of and foundation for exploring such a learning approach, it has room for further improvement and extension. Several chapters conclude with a discussion section including ideas for future extensions and improvements.

In addition, while the techniques developed in this system generally reflect my

own approach to visualizing and thinking about geometry and background in learning geometry via an investigative approach, there is room to integrate the *discovery* ideas in this system with some of the techniques from the rich history of automated geometry theorem *proving*.

10.2 Limitations

Despite its successes, there are certainly limitations to the system's current abilities. Reasoning about geometry concepts is a very broad domain, and it becomes difficult to develop general techniques that can apply in a wide variety of circumstances. Chapters 6 and 7 discuss how this challenge arises when trying to filter more categories of obvious observations and when deciding the ideal method for specifying values in the constraint solver. There are also some sizable limitations to the system's purely-investigative approach that restrict what it is able to discover:

10.2.1 Probabilistic Approach

One challenge is that its approach is inherently probabilistic. As with any numerical-based system, an important issue with using a coordinate-based, inductive technique for discovering concepts is dealing with numerical inaccuracies. Although techniques were used to lessen some of the effects of floating point errors, such techniques also emphasize the probabilistic nature of the system. Without using deductive reasoning, the system cannot ever fully confirm its findings are correct and may occasionally report false properties due to uncertainty. However, reporting likely results is sufficient for encouraging discovery as results in question could be further explored and checked using external approaches.

10.2.2 Negative Relations and Definitions

In addition to only providing probabilistic confidence for its findings, there are some relations and definitions that are hard to notice via a purely inductive, random-

sampling based approach. For instance, negative definitions such as learning that `scalene` triangles are ones with *no* equal sides would require the system to handle more complicated logical combinations of relationships.

10.2.3 Generality of Theorems

Finally, the full space of theorems about geometry is quite broad. Some of these statements require a richer set of tools than provided in this system. For instance, noticing the fact that that "the shortest distance from a point to a line is along the perpendicular to the line" would require the current system to be testing and searching for maxima and minima in its manipulations. The current system is limited to discovering conjectures regarding simple relationships among objects that are constructed from some initial premises.

10.3 System-level Extensions

In addition to improvements to individual modules to reduce the effects of randomness, filter out additional obvious properties, and support more declarative constraints, there are several interesting larger-scale extensions that could integrate with the system.

10.3.1 Deductive Proof Systems

One of the main extensions is to integrate the results from the system with an automated, deductive geometry prover. Although such provers often use less human-like approaches when verifying statements, having access to such a system could increase this system's confidence in the properties and conjectures it finds as it continues to explore new concepts.

10.3.2 Learning Constructions

In addition to generating formal, deductive proofs about the properties and theorems resulting from the system's explorations, another interesting extension would be for the system to learn from the *process* it uses in generating its results. For example, the sequence and dependencies for how values were determined in solving a set of declarative constraints might be able to be abstracted into a sequence of more typical construction procedures that produce the same diagram.

10.3.3 Self-directed Explorations

A final exciting addition to the system is to build a self-directed mode of operation in which the system proposes its own constructions and diagrams to investigate rather than being prompted from an outside user. As the system expands its repository of knowledge about constructions and conjectures, it could use these findings to direct further explorations. This would provide some full circle closure to the discovery process and could even lead to the system creatively devising interesting exercises or exam questions that test the knowledge it has acquired.

Appendix A

Code Listings

This appendix contains full code listings for the system, implemented using MIT/GNU Scheme 9.2. In addition to the code provided here, the system is dependent on the propagator system used in Alexey Radul and Gerald Jay Sussman's *Revised Report on the Propagator Model* available at `http://groups.csail.mit.edu/mac/users/gjs/propagators/`.

The three files in `lib/` are code used with permission from external sources, and include excerpts of code created by Gerald Sussman and others from the MIT Scheme Mechanics Library [26]. These excerpts handle numeric accuracy, generic operations, and hash-table based eq-properties. All other code in these listings is written solely by me for use in this thesis.

List of Listings

Example Content:

Core Components and Utilities:

External Library Procedures:

Listing A.1: load.scm

```
;;; 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"
            "solver"
            "perception"
            "learning"
            "content"))

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

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

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


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

'done-loading
```

Listing A.2: 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.3: figure/linear.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 &lt;segment&gt;
  (make-segment p1 p2)
  segment?
  (p1 segment-endpoint-1)
  (p2 segment-endpoint-2))

(defhandler print
  (lambda (s)
    (if (named? s)
        (element-name s)
        `(*segment* ,(print (segment-endpoint-1 s))
                    ,(print (segment-endpoint-2 s)))))
  segment?)

(define (segment-endpoints s)
  (list (segment-endpoint-1 s)
        (segment-endpoint-2 s)))

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

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

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

(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))))

;;; Regardless of ordering or point naming, refers to the same pair of
;;; point locations.
(define (segment-equivalent? s1 s2)
  (set-equivalent?
   (segment-endpoints s1)
   (segment-endpoints s2)
   point-equal?))

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

(define (ray-equal? r1 r2)
  (and (point-equal?
        (ray-endpoint r1)
        (ray-endpoint r2))
       (direction-equal?
        (ray-direction r1)
        (ray-direction r2))))

;;; Ignores line point and direction
(define (line-equivalent? l1 l2)
  (and (or (on-line? (line-point l1) l2)
           (on-line? (line-point l2) l1))
       (or
        (direction-equal?
         (line-direction l1)
         (line-direction l2))
        (direction-opposite?
         (line-direction l1)
         (line-direction l2)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; 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?)

(define linear-element-equivalent?
  (make-generic-operation 2 'linear-element-equivalent?
                          false-proc))

(defhandler linear-element-equivalent?
  segment-equivalent?
  segment? segment?)

(defhandler linear-element-equivalent?
  ray-equal?
  ray? ray?)

(defhandler linear-element-equivalent?
  line-equivalent?
  line? line?)
```

Listing A.4: 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.5: figure/vec.scm

```scheme
;;; 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)
  (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.6: 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-endpoint r))
         (dir (ray-direction r))
         (v (vec-from-direction-distance
             dir dist)))
    (add-to-point p1 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.7: figure/angle.scm

```
;;; 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?)

(define (angle-equivalent? a1 a2)
  (and (point-equal?
        (angle-vertex a1)
        (angle-vertex a2))
       (set-equivalent?
        (list (angle-arm-1 a1) (angle-arm-2 a1))
        (list (angle-arm-1 a2) (angle-arm-2 a2))
        direction-equal?)))

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

(defhandler print
  (lambda (a)
    (if (named? a)
```

```
        (element-name a)
        `(*angle* ,(print (angle-vertex a)))))
  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)))

;;;;;;;;;;;;;;;;;;;;;; Ideas for Definitions ;;;;;;;;;;;;;;;;;;;;;;

;;; Not currently used, but could be learned later?

(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.8: 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.9: 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))))

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

(define (circle-equivalent? c1 c2)
  (and (point-equal?
        (circle-center c1)
        (circle-center c2))
       (close-enuf?
        (circle-radius c1)
        (circle-radius c2))))

(define (on-circle? p c)
  (close-enuf?
   (distance p (circle-center c))
   (circle-radius c)))
```

Listing A.10: 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)
    (if (named? p)
        (element-name 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))))

(define (points-non-overlapping? points)
  (= (length points)
     (length (dedupe-by point-equal? points))))
```

Listing A.11: 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)))))
    (save-obvious-observation!
     (make-observation equal-length-relationship
                       (list
                        (make-segment p1 newpoint)
                        (make-segment p2 newpoint))))
    newpoint))

(define (segment-midpoint s)
  (let ((p1 (segment-endpoint-1 s))
        (p2 (segment-endpoint-2 s)))
    (midpoint p1 p2)))

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

(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))
         (new-line (make-line point rotated-direction)))
    (save-obvious-observation!
     (make-observation
      perpendicular-relationship
      (list linear-element new-line)))
    new-line))

;;; 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))))
      (let ((seg (make-segment point i)))
        (save-obvious-observation!
         (make-observation
          perpendicular-relationship
          (list seg linear-element)))
        seg))))

(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 segment pb)))
      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)))
    (save-obvious-observation!
     (make-observation
      equal-angle-relationship
      (list (make-angle d2 vertex new-direction)
            (make-angle new-direction vertex d1))))
    (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 (distinct? (list p1 p2 p3 p4) point-equal?)
       (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 (collinear? p1 p2 p3)
  (and (distinct? (list p1 p2 p3) point-equal?)
       (on-line? p3 (line-from-points p1 p2))))

(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.12: 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 ;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; 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)))))))

(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
(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?)
(defhandler on-element? on-circle? point? circle?)

;;;;;;;;;;;;;;;;;;;;;;; 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.13: 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-polygons figure)
  (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.14: 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 ;;;;;;;;;;;;;;;;;;;;;;;;;

(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.15: figure/polygon.scm

```
;;; 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))))

(defhandler print
  (lambda (p)
    (if (named? p)
        (element-name p)
        `(*polygon* ,@(map print (polygon-points p)))))
  polygon?)

;;;;;;;;;;;;;;;;;;;;;;;;; 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)
  (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)))
       segment)))))

(define (polygon-segments polygon)
  (let ((n-points (polygon-n-points polygon)))
    (map (lambda (i)
           (let ((j (modulo (+ i 1) n-points)))
            (with-dependency-if-unknown
             `(polygon-segment ,polygon ,i ,j)
             (with-source
              (lambda (p)
                (polygon-segment (from-new-premise p polygon)
                                 i j))
              (polygon-segment polygon i j)))))
         (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)))
        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)
         (with-dependency
          `(polygon-angle ,polygon ,i)
          (with-source
           (lambda (p)
             (polygon-angle-by-index
              (from-new-premise p polygon) i))
           (polygon-angle-by-index polygon i))))
       (iota (polygon-n-points polygon))))
```

Listing A.16: 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)
  (if (and (named? element)
           (not (eq? (element-name element)
                     name)))
      (error "Reassining element name:"
             (list element (element-name element) name)))
  (eq-put! element 'name name)
  element)

(define (element-name element)
  (or (eq-get element 'name)
      *unnamed*))

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

(define generic-element-name
```

```
  (make-generic-operation 1 'generic-element-name
                          element-name))

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

Listing A.17: 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 (memoize-function source))
  element)

(define (element-source element)
  (or (eq-get element 'source)
      (lambda (p) element)))

(define (from-new-premise new-premise element)
  ((element-source element)
   new-premise))

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

(define (set-dependency! element dependency)
  (if (not (number? element))
      (eq-put! element 'dependency dependency)))

(define (set-dependency-if-unknown! element dependency)
  (if (dependency-unknown? element)
      (set-dependency! element 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))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Premises ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (set-as-premise! element i)
  (set-dependency! element (symbol '<premise- i '>))
  (set-source! element (lambda (p) (list-ref p i))))

(define (as-premise element i)
  (set-as-premise! element i)
  element)

;;;;;;;;;;;;;;;;;;;;;;;; Unknown Dependencies ;;;;;;;;;;;;;;;;;;;;;;;;

(define (dependency-known? element)
  (eq-get element 'dependency))

(define dependency-unknown? (notp dependency-known?))

(define (clear-dependency! element)
  (set-dependency! element #f))

;;;;;;;;;;;;;;;;;;;;;;;; Accessing Dependencies ;;;;;;;;;;;;;;;;;;;;;;;;

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

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

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

(define (print-dependencies object)
```

```
  (pprint (element-dependencies->list object)))
```

Listing A.18: 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

;;; 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
;;; 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-acute-angle-measure)
  (rand-range (* pi 0.05) (* .45 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))))
        (add-to-point
         (make-point x y)
         (vec-from-direction-distance dir *point-wiggle-radius*))))))

;;; 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))))

(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)))
    (random-line-through-point p)))

(define (random-segment)
  (let ((p1 (random-point))
        (p2 (random-point)))
    (let ((seg (make-segment p1 p2)))
      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)))

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

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

(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))))

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

(define (random-quadrilateral)
  (random-n-gon 4))


;;; More in content/random-polygons.scm
```


Listing A.19: 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 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Rotates counterclockwise
(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 seg)
  (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.20: figure/direction-interval.scm

```
;;; direction-interval.scm --- Direction Intervals

;;; Commentary:

;; Ideas:
;; - Structure for representing ranges of directions
;; - Also interface for propagating partial information about angles
;; - Full circle intervals

;; Future:
;; - Multi-segment direction intervals
;; - Include direction? as direction-interval?
;; - Migrate additional direction/interval code from linkages.scm
;; - Deal with adding intervals to directions
;; - Clean up direction-interval intersection
;;   (subtract to start-1, e.g.)

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;; Direction Intervals ;;;;;;;;;;;;;;;;;;;;;;;

;;; "arc" of the circle from start-dir CCW to end-dir
;;; "invalid" allows for "impossible" intervals
(define-record-type <standard-direction-interval>
  (%make-standard-direction-interval start-dir end-dir)
  standard-direction-interval?
  (start-dir direction-interval-start)
  (end-dir direction-interval-end))

```

```
(define (make-direction-interval start-dir end-dir)
  (if (direction-equal? start-dir end-dir)
      (error "Cannot make direction-interval with no range:
 use direction or full interval"))
  (%make-standard-direction-interval start-dir end-dir))

(define (print-direction-interval di)
  `(direction-interval ,(direction-theta (direction-interval-start di))
                       ,(direction-theta (direction-interval-end di))))

(define (direction-interval-center di)
  (add-to-direction
   (direction-interval-start di)
   (/ (direction-interval-size di) 2.0)))

(defhandler print print-direction-interval standard-direction-interval?)

;;;;;;;;;;;;;;;;;;; Invalid Direction Intervals ;;;;;;;;;;;;;;;;;;;;

(define-record-type <invalid-direction-interval>
  (%make-invalid-direction-interval)
  invalid-direction-interval?)

(define (make-invalid-direction-interval)
  (%make-invalid-direction-interval))

(define (print-invalid-direction-interval di)
  `(invalid-direction-interval))
(defhandler print print-invalid-direction-interval
    invalid-direction-interval?)

;;;;;;;;;;;;;;;;;;;; Full Direction Intervals ;;;;;;;;;;;;;;;;;;;;

(define-record-type <full-circle-direction-interval>
  (%make-full-circle-direction-interval)
  full-circle-direction-interval?)

(define (make-full-circle-direction-interval)
  (%make-full-circle-direction-interval))

(define (print-full-circle-direction-interval di)
  `(full-circle-direction-interval))
(defhandler print print-full-circle-direction-interval
  full-circle-direction-interval?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;; All Types ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (direction-interval? x)
  (or (standard-direction-interval? x)
      (invalid-direction-interval? x)
      (full-circle-direction-interval? x)))

;;;;;;;;;;;;;;;;;;;;;;;; More Constructors ;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-semi-circle-direction-interval start-dir)
  (make-direction-interval start-dir
                           (add-to-direction start-dir pi)))

(define (make-direction-interval-from-start-dir-and-size start-dir
    radians)
  (cond  ((or (close-enuf? radians (* 2 pi))
              (>= radians (* 2 pi)))
          (make-full-circle-direction-interval))
         ((close-enuf? radians 0)
          (error "cannot have interval of size 0: use direction"))
         ((< radians 0)
          (make-invalid-direction-interval))
         (else
          (make-direction-interval
           start-dir
           (add-to-direction start-dir radians)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Equality ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define direction-interval-equal?
  (make-generic-operation 2 'direction-interval-equal?))

(define (standard-direction-interval-equal? di1 di2)
  (and (direction-equal?
        (direction-interval-start di1)
        (direction-interval-start di2))
       (direction-equal?
        (direction-interval-end di1)
        (direction-interval-end di2))))

(defhandler direction-interval-equal?
  false-proc direction-interval? direction-interval?)

(defhandler direction-interval-equal?
  true-proc full-circle-direction-interval?
       full-circle-direction-interval?)

(defhandler direction-interval-equal?
  true-proc invalid-direction-interval? invalid-direction-interval?)

(defhandler direction-interval-equal?
  standard-direction-interval-equal?
  standard-direction-interval?
  standard-direction-interval?)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Inclusion ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define within-direction-interval?
  (make-generic-operation 2 'within-direction-interval?))

(define (within-standard-direction-interval? dir dir-interval)
  (let ((dir-start (direction-interval-start dir-interval))
        (dir-end (direction-interval-end dir-interval)))
```

```
    (or (direction-equal? dir dir-start)
        (direction-equal? dir dir-end)
        (< (subtract-directions dir dir-start)
           (subtract-directions dir-end dir-start)))))

(defhandler within-direction-interval?
  within-standard-direction-interval?
  direction?
  standard-direction-interval?)

(defhandler within-direction-interval?
  true-proc direction? full-circle-direction-interval?)

(defhandler within-direction-interval?
  false-proc direction? invalid-direction-interval?)

(define within-direction-interval-non-inclusive?
  (make-generic-operation 2 'within-direction-interval-non-inclusive?))

(define (within-standard-direction-interval-non-inclusive? dir
    dir-interval)
  (let ((dir-start (direction-interval-start dir-interval))
        (dir-end (direction-interval-end dir-interval)))
    (and (not (direction-equal? dir dir-start))
         (not (direction-equal? dir dir-end))
         (< (subtract-directions dir dir-start)
            (subtract-directions dir-end dir-start)))))

(defhandler within-direction-interval-non-inclusive?
  within-standard-direction-interval-non-inclusive?
  direction?
  standard-direction-interval?)

(defhandler within-direction-interval-non-inclusive?
  true-proc direction? full-circle-direction-interval?)

(defhandler within-direction-interval-non-inclusive?
  false-proc direction? invalid-direction-interval?)

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

(define reverse-direction-interval
  (make-generic-operation 1 'reverse-direction-interval))

(define (reverse-standard-direction-interval di)
  (make-direction-interval
   (reverse-direction (direction-interval-start di))
   (reverse-direction (direction-interval-end di))))

(defhandler reverse-direction-interval
  reverse-standard-direction-interval
  standard-direction-interval?)

(defhandler reverse-direction-interval identity
  full-circle-direction-interval?)

(define direction-interval-size
  (make-generic-operation 1 'direction-interval-size))

(define (standard-direction-interval-size di)
  (subtract-directions (direction-interval-end di)
                       (direction-interval-start di)))

(defhandler direction-interval-size
  standard-direction-interval-size
  standard-direction-interval?)

(defhandler direction-interval-size
  (lambda (di) (* 2 pi))
  full-circle-direction-interval?)

;;; Rotate CCW by radians
(define shift-direction-interval
  (make-generic-operation 2 'shift-direction-interval))

(define (shift-standard-direction-interval di radians)
  (make-direction-interval
   (add-to-direction (direction-interval-start di) radians)
   (add-to-direction (direction-interval-end di) radians)))

(defhandler shift-direction-interval
  shift-standard-direction-interval
  standard-direction-interval? number?)

(defhandler shift-direction-interval
  (lambda (fcdi r) fcdi) full-circle-direction-interval? number?)

;;;;;;;;;;;;;;;;;; Direction interval intersection ;;;;;;;;;;;;;;;;;;

(define intersect-direction-intervals
  (make-generic-operation 2 'intersect-direction-intervals))

(define (test-intersect-standard-dir-intervals di-1 di-2)
  (let ((result (internal-intersect-standard-dir-intervals di-1 di-2)))
    (let ((r-start (direction-interval-start result))
          (r-center (direction-interval-center result))
          (r-end (direction-interval-start result)))
      (if (not (and (within-direction-interval? r-start di-1)
                    (within-direction-interval? r-end di-1)
                    (within-direction-interval? r-center di-1)
                    (within-direction-interval? r-start di-2)
                    (within-direction-interval? r-center di-2)
                    (within-direction-interval? r-end di-2)))
          (error "Dir Intersection fail!"
                 (print (list di-1 di-2 result))))
      result)))

(define (intersect-standard-dir-intervals di-1 di-2)
```

```
  (let ((start-1 (direction-interval-start di-1))
        (end-1 (direction-interval-end di-1))
        (start-2 (direction-interval-start di-2))
        (end-2 (direction-interval-end di-2)))
    (if (> (direction-theta start-1)
           (direction-theta start-2))
        (intersect-standard-dir-intervals di-2 di-1)
        (cond
         ((or (direction-equal? start-2 start-1)
              (within-direction-interval-non-inclusive? start-2 di-1))
          ;; case 1: di-2 starts within di-1
          (if (within-direction-interval? end-1 di-2)
              (cond ((direction-equal? end-1 end-2)
                     (make-direction-interval start-2 end-2))
                    ;; Exclude the case where it loops around end ends
                    ;; within the start of di-1 again
                    ((within-direction-interval-non-inclusive? end-2
                         di-1)
                     nothing)
                    (else
                     (make-direction-interval start-2 end-1)))
              (make-direction-interval start-2 end-2)))
         ;; case 2: di-2 starts after di-1 and ends within di-1
         ((within-direction-interval? end-2 di-1)
          (make-direction-interval start-1 end-2))
         ;; case 3: di-2 starts after di-1 and ends beyond di-1
         ((or (within-direction-interval? end-1 di-2)
              (direction-equal? end-1 end-2))
          (make-direction-interval start-1 end-1))
         ;; Case 4: di-2 starts after di-1 and ends before di-1 starts
             again
         (else
          (pp (print (list di-1 di-2)))
          (error "No intersection")
          (make-invalid-direction-interval))))))

#|
;; Test cases
(define d0 (make-direction 0.))
(define d1 (make-direction 1.))
(define d2 (make-direction 2.))
(define d3 (make-direction 3.))
(define d4 (make-direction 4.))
(define d5 (make-direction 5.))
(define d6 (make-direction 6.))         ; almost all the way around

(define (test s1 e1 s2 e2)
  (print (intersect-standard-dir-intervals
    (make-direction-interval s1 e1)
    (make-direction-interval s2 e2))))

(test d0 d1 d0 d1)

|#
```

```

(defhandler intersect-direction-intervals
  (lambda (di idi) idi)
  direction-interval? invalid-direction-interval?)

(defhandler intersect-direction-intervals
  (lambda (idi di) idi)
  invalid-direction-interval? direction-interval?)

(defhandler intersect-direction-intervals
  (lambda (fcdi di) di)
  full-circle-direction-interval? direction-interval?)

(defhandler intersect-direction-intervals
  (lambda (di fcdi) di)
  direction-interval? full-circle-direction-interval?)

(defhandler intersect-direction-intervals
  intersect-standard-dir-intervals
  standard-direction-interval? standard-direction-interval?)


(define (intersect-direction-with-interval dir dir-interval)
  (if (within-direction-interval? dir dir-interval)
      dir
      (make-invalid-direction-interval)))

#|
 (define a (make-direction 0))
 (define b (make-direction (/ pi 4)))
 (define c (make-direction (/ pi 2)))
 (define d (make-direction pi))
 (define e (make-direction (* 3 (/ pi 2))))
 (define f (make-direction (* 7 (/ pi 4))))

 (within-direction-interval? b
  (make-direction-interval f c))
 ;Value: #t

 (within-direction-interval? b
  (make-direction-interval c f))
 ;Value: #f

 (print-direction-interval
  (intersect-direction-intervals
   (make-direction-interval b d)
   (make-direction-interval f c)))
 ;Value: (dir-interval .7853981633974483 1.5707963267948966)

 etc.
|#

;;;;;;;;;;;;;; Direction Intervals as propagator values ;;;;;;;;;;;;;;

```

```
(defhandler equivalent? direction-interval-equal?
  direction-interval? direction-interval?)
(defhandler equivalent? (lambda (a b) #f)
  direction-interval? direction?)
(defhandler equivalent? (lambda (a b) #f)
  direction? direction-interval?)

(defhandler merge intersect-direction-intervals
  direction-interval? direction-interval?)
(defhandler merge intersect-direction-with-interval
  direction? direction-interval?)
(defhandler merge
  (lambda (di d)
    (intersect-direction-with-interval d di))
  direction-interval? direction?)
(defhandler merge
  (lambda (d1 d2)
    (if (direction-equal? d1 d2)
        d1
        (make-invalid-direction-interval)))
  direction? direction?)

(defhandler contradictory? invalid-direction-interval?
  direction-interval?)

;;;;;; Propagator generic operations on directions / intervals ;;;;;;;

(propagatify make-direction)

(define direction-sin (make-generic-operator 1 'direction-sin))

(defhandler direction-sin
  (lambda (d) nothing)
  direction-interval?)

(defhandler direction-sin
  (lambda (d) (sin (direction-theta d)))
  direction?)

(define direction-cos (make-generic-operator 1 'direction-cos))

(defhandler direction-cos
  (lambda (d) nothing)
  direction-interval?)

(defhandler direction-cos
  (lambda (d) (cos (direction-theta d)))
  direction?)

(propagatify direction-sin)

(propagatify direction-cos)
```


Listing A.21: perception/relationship.scm

```
;;; relationship.scm -- relationships among element-list

;;; Commentary

;; Ideas:
;; - Include with relationship types predicates for how to use them.

;; Future:
;; - Think about procedures / dependencies to obtain arguments

;;; Code:

;;;;;;;;;;;;;;;;;;;;;; Relationship Structure ;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <relationship>
  (make-relationship name arity predicate equivalence-predicate)
  relationship?
  (name relationship-name)
  (arity relationship-arity)
  (predicate relationship-predicate)
  (equivalence-predicate relationship-equivalence-predicate))

(define (relationship-equivalent? r1 r2)
  (eq? (relationship-name r1)
       (relationship-name r2)))

(define print-relationship relationship-name)

(defhandler print print-relationship relationship?)

;;;;;;;;;;;;;;;;;;;;;; Checking relationships ;;;;;;;;;;;;;;;;;;;;;;

(define (relationship-holds r element-list)
  (apply (relationship-predicate r) element-list))

;;;;;;;;;;;;;;;;;;;;;; Basic relationship ;;;;;;;;;;;;;;;;;;;;;;;;

;;; Segments:

(define equal-length-relationship
  (make-relationship 'equal-length 2 segment-equal-length?
                     (set-equivalent-procedure segment-equivalent?)))

;;; Angles:
(define equal-angle-relationship
  (make-relationship 'equal-angle 2 angle-measure-equal?
                     (set-equivalent-procedure angle-equivalent?)))

(define supplementary-angles-relationship
  (make-relationship 'supplementary 2 supplementary-angles?
                     (set-equivalent-procedure angle-equivalent?)))

```

```
(define complementary-angles-relationship
  (make-relationship 'complementary 2 complementary-angles?
                     (set-equivalent-procedure angle-equivalent?)))

;;; Linear elements:
(define perpendicular-relationship
  (make-relationship 'perpendicular 2 perpendicular?
                     (set-equivalent-procedure
                          linear-element-equivalent?)))

(define parallel-relationship
  (make-relationship 'parallel 2 parallel?
                     (set-equivalent-procedure
                          linear-element-equivalent?)))

(define concurrent-relationship
  (make-relationship 'concurrent 3 concurrent?
                     (set-equivalent-procedure
                          linear-element-equivalent?)))

;;; Points:
(define concurrent-points-relationship
  (make-relationship 'concurrent 2 point-equal?
                     (set-equivalent-procedure point-equal?)))

(define concentric-relationship
  (make-relationship 'concentric 4 concentric?
                     (set-equivalent-procedure point-equal?)))

(define concentric-with-center-relationship
  (make-relationship 'concentric-with-center
                     4 concentric-with-center?
                     (set-equivalent-procedure point-equal?)))

;;; Polygons:
(define (make-polygon-n-sides-relationship n)
  (make-relationship (symbol 'n-sides- n)
                     1 (ngon-predicate n)
                     eq?))

(define (make-polygon-term-relationship polygon-term)
  (make-relationship polygon-term
                     1
                     (lambda (obj) (is-a? polygon-term obj))
                     eq?))
```

Listing A.22: perception/observation.scm

```
;;; 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 (observation-equivalent? obs1 obs2)
  (and (relationship-equivalent?
        (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.23: 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-proc)
  (require-majority
   (lambda () (analyze-figure (figure-proc)))
   observation-equal?))

;;; Given a figure, report what's interesting
(define (analyze-figure 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 (polygon-observations polygons)
  (append-map (lambda (poly)
                (map (lambda (term)
                       (make-observation
                        (make-polygon-term-relationship term)
                        (list poly)))
                     (examine poly)))
              polygons))

(define (polygon-implied-observations polygons)
  (append-map
   (lambda (poly)
     (append-map (lambda (term)
                   (observations-implied-by-term term poly))
                 (examine poly)))
   polygons))

(define (interesting-observations figure-proc)
  (set! *obvious-observations* '())
  (let* ((fig (figure-proc))
         (all-obs (analyze-figure fig))
         (a (pp "Done extracting all Observations"))
         (polygons (figure-polygons fig))
         (polygon-observations
          (polygon-observations polygons))
         (a (pp "Done determining polygon Observatsions"))
         (polygon-implied-observations
          (polygon-implied-observations polygons))
         (b (pp "Done determining implied Observatsions")))
    (set-difference (append all-obs
                            polygon-observations)
                    (append *obvious-observations*
                            polygon-implied-observations)
                    observation-equivalent?)))

(define *obvious-observations* #f)

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

;;;;;;;;;;;;;;;;;;;;; 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)))))
               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))
```

```
225
226 (define (save-results results data-table)
227   (hash-table/put! data-table results
228                    (+ 1 (hash-table/get data-table results 0))))
229
230 (define (print-analysis-results data-table)
231   (hash-table/for-each
232    data-table
233    (lambda (k v)
234      (pprint (list v (cons 'discovered k))))))
```

Listing A.24: graphics/appearance.scm

```
 1 (define (with-color color element)
 2   (eq-put! element 'color color)
 3   element)
 4
 5 (define default-element-color
 6   (make-generic-operation 1
 7                           'default-element-color
 8                           (lambda (e) "black")))
 9
10 (defhandler default-element-color (lambda (e) "blue") point?)
11 (defhandler default-element-color (lambda (e) "black") segment?)
12
13 (define (element-color element)
14   (or (eq-get element 'color)
15       (default-element-color element)))
```

Listing A.25: graphics/graphics.scm

```
 1 ;;; graphics.scm -- Graphics Commands
 2
 3 ;;;;;;;;;;;;;;;;;;;;;;;;; Main Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;
 4
 5 (define (draw-figure figure canvas)
 6   (set-coordinates-for-figure figure canvas)
 7   (clear-canvas canvas)
 8   (for-each
 9    (lambda (element)
10      (canvas-set-color canvas (element-color element))
11      ((draw-element element) canvas))
12    (all-figure-elements figure))
13   (for-each
14    (lambda (element)
15      (canvas-set-color canvas (element-color element))
16      ((draw-label element) canvas))
17    (all-figure-elements figure))
18   (graphics-flush (canvas-g canvas)))
19
20 (define (set-coordinates-for-figure figure canvas)
21   (let* ((bounds (scale-bounds (bounds->square (extract-bounds figure))
22                                1.1)))
23     (graphics-set-coordinate-limits
24      (canvas-g canvas)
25      (bounds-xmin bounds)
26      (bounds-ymin bounds)
27      (bounds-xmax bounds)
28      (bounds-ymax bounds))))
29
30 (define draw-element
31   (make-generic-operation 1 'draw-element
32                           (lambda (e) (lambda (c) 'done))))
33
34 (define draw-label
35   (make-generic-operation 1 'draw-label (lambda (e) (lambda (c)'done))))
36
37 (define (add-to-draw-element! predicate handler)
38   (defhandler draw-element
39     (lambda (element)
40       (lambda (canvas)
41         (handler canvas element)))
42     predicate))
43
44 (define (add-to-draw-label! predicate handler)
45   (defhandler draw-label
46     (lambda (element)
47       (lambda (canvas)
48         (handler canvas element)))
49     predicate))
50
51
52 (define *point-radius* 0.02)
53 (define (draw-point canvas point)
54   (canvas-fill-circle canvas
55               (point-x point)
56               (point-y point)
57               *point-radius*))
58 (define (draw-point-label canvas point)
59   (canvas-draw-text canvas
60                     (+ (point-x point) *point-radius*)
61                     (+ (point-y point) *point-radius*)
62                     (symbol->string (element-name point))))
63
64 (define (draw-segment canvas segment)
65   (let ((p1 (segment-endpoint-1 segment))
66         (p2 (segment-endpoint-2 segment)))
67     (canvas-draw-line canvas
68                       (point-x p1)
69                       (point-y p1)
70                       (point-x p2)
71                       (point-y p2))))
72 (define (draw-segment-label canvas segment)
73   (let ((v (vec-from-direction-distance (rotate-direction-90
74                                          (segment->direction segment))
75                                         (* 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.26: solver/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))
```

```
106
107 (defhandler m:add-to-direction
108   m:add-interval-to-direction direction? interval?)
109
110 (defhandler m:add-to-direction
111   add-to-direction direction? number?)
112
113 (defhandler m:add-to-direction
114   m:add-interval-to-standard-direction-interval
115   standard-direction-interval? interval?)
116
117 (defhandler m:add-to-direction
118   m:add-interval-to-full-circle-direction-interval
119   full-circle-direction-interval? interval?)
120
121 (defhandler m:add-to-direction
122   m:add-interval-to-invalid-direction-interval
123   invalid-direction-interval? interval?)
124
125 (defhandler m:add-to-direction
126   shift-direction-interval direction-interval? number?)
127
128 (propagatify m:add-to-direction)
129
130 ;;;;;;;;;;;;;;;;;;;;;; Subtracting directions ;;;;;;;;;;;;;;;;;;;;;;
131
132 (defhandler generic-negate
133   (lambda (i) (mul-interval i -1)) %interval?)
134
135 (define (m:standard-direction-interval-minus-direction di d)
136   (if (within-direction-interval? d di)
137       (make-interval
138        0
139        (subtract-directions (direction-interval-end di) d))
140       (make-interval
141        (subtract-directions (direction-interval-start di) d)
142        (subtract-directions (direction-interval-end di) d))))
143
144 (define (m:full-circle-direction-interval-minus-direction di d)
145   (make-interval
146    0 (* 2 pi)))
147
148 (define (m:direction-minus-standard-direction-interval d di)
149   (if (within-direction-interval? d di)
150       (make-interval
151        0
152        (subtract-directions d (direction-interval-start di)))
153       (make-interval
154        (subtract-directions d (direction-interval-end di))
155        (subtract-directions d (direction-interval-start di)))))
156
157 (define (m:direction-minus-full-circle-direction-interval d di)
158   (make-interval
159    0 (* 2 pi)))
160
161 (define m:subtract-directions
162   (make-generic-operation 2 'm:subtract-directions))
163
164 (defhandler m:subtract-directions
165   subtract-directions direction? direction?)
166
167 (defhandler m:subtract-directions
168   (lambda (di1 di2)
169     nothing)
170   direction-interval? direction-interval?)
171
172 (defhandler m:subtract-directions
173   m:standard-direction-interval-minus-direction
174   standard-direction-interval? direction?)
175
176 (defhandler m:subtract-directions
177   m:full-circle-direction-interval-minus-direction
178   full-circle-direction-interval? direction?)
179
180 (defhandler m:subtract-directions
181   m:direction-minus-standard-direction-interval
182   direction? standard-direction-interval?)
183
184 (defhandler m:subtract-directions
185   m:direction-minus-full-circle-direction-interval
186   direction? full-circle-direction-interval?)
187
188 (propagatify m:subtract-directions)
189
190 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Vec ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
191 (define-record-type <m:vec>
192   (%m:make-vec dx dy length direction)
193   m:vec?
194   (dx m:vec-dx)
195   (dy m:vec-dy)
196   (length m:vec-length)
197   (direction m:vec-direction))
198
199
200 ;;; Allocate and wire up the cells in a vec
201 (define (m:make-vec vec-id)
202   (let-cells (dx dy length direction)
203     (name! dx (symbol vec-id '-dx))
204     (name! dy (symbol vec-id '-dy))
205     (name! length (symbol vec-id '-len))
206     (name! direction (symbol vec-id '-dir))
207
208     (p:make-direction
209      (e:atan2 dy dx) direction)
210     (p:sqrt (e:+ (e:square dx)
211                  (e:square dy))
212             length)
213     (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>
  (%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)))))

(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?)

(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 ;;;;;;;;;;;;;;;;;;

(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.27: solver/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))

;;;  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)
              (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))
                      (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)
                           (and (m:on-arc? p arc1)
                                (m:on-arc? p arc2)))
                         intersections)))
            (if (> (length points) 0)
                (m:make-point-set points)
                (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)
            (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)
        (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?)
```

```
340
341 #|
342  Simple Examples
343  (pp (let-cells (c)
344     (add-content c (m:make-arc (make-point 1 0) (sqrt 2)
345                                (make-direction-interval
346                                 (make-direction (/ pi 8))
347                                 (make-direction (* 7 (/ pi 8))))))
348
349     (add-content c (m:make-ray (make-point -3 1) (make-direction 0)))
350     (add-content c (m:make-ray (make-point 1 2)
351                   (make-direction (* 7 (/ pi 4)))))
352     (content c)))
353
354  (let ((a (make-point 0 0))
355        (b (make-point 1 0))
356        (c (make-point 0 1))
357        (d (make-point 1 1)))
358     (let-cells (cell)
359       (add-content cell
360                    (make-tms
361                     (contingent (m:make-point-set (list a b c))
362                                 '(a))))
363       (add-content cell
364                    (make-tms
365                     (contingent (m:make-point-set (list a d))
366                                 '(a))))
367       (pp (tms-query (content cell)))))
368 |#
369 ;;;;;;;;;;;;;;;;;;;;;;;; To Figure elements ;;;;;;;;;;;;;;;;;;;;;;;;
370
371 (define m:region->figure-elements
372   (make-generic-operation 1 'm:region->figure-elements (lambda (r) #f )))
373
374 (defhandler m:region->figure-elements
375   m:ray->figure-ray
376   m:ray?)
377
378 (defhandler m:region->figure-elements
379   m:region-contradiction->figure-elements
380   m:region-contradiction?)
```

Listing A.28: solver/constraints.scm

```
1 ;;; constraints.scm --- Constraints for mechanisms
2
3 ;;; Commentary:
4
5 ;; Ideas:
6 ;; - Abstraction for specifying constraints
7 ;; - Length, angle equality
8 ;; - Perpendicular / Parellel
9
10 ;; Future:
11 ;; - Constraints for other linkages?
12
13 ;;; Code:
14
15 ;;;;;;;;;;;;;;;;;;;;;;; Constraint Structure ;;;;;;;;;;;;;;;;;;;;;;;;
16
17 (define-record-type <m:constraint>
18   (m:make-constraint type args constraint-procedure)
19   m:constraint?
20   (type m:constraint-type)
21   (args m:constraint-args)
22   (constraint-procedure m:constraint-procedure))
23
24 ;;;;;;;;;;;;;;;;;;;;;;;; Constraint Types ;;;;;;;;;;;;;;;;;;;;;;;;
25
26 (define (m:c-length-equal bar-id-1 bar-id-2)
27   (m:make-constraint
28    'm:c-length-equal
29    (list bar-id-1 bar-id-2)
30    (lambda (m)
31      (let ((bar-1 (m:lookup m bar-id-1))
32            (bar-2 (m:lookup m bar-id-2)))
33        (c:id (m:bar-length bar-1)
34              (m:bar-length bar-2))))))
35
36 (define (m:c-angle-equal joint-id-1 joint-id-2)
37   (m:make-constraint
38    'm:c-angle-equal
39    (list joint-id-1 joint-id-2)
40    (lambda (m)
41      (let ((joint-1 (m:lookup m joint-id-1))
42            (joint-2 (m:lookup m joint-id-2)))
43        (c:id (m:joint-theta joint-1)
44              (m:joint-theta joint-2))))))
45
46 (define (m:c-right-angle joint-id)
47   (m:make-constraint
48    'm:right-angle
49    (list joint-id)
50    (lambda (m)
51      (let ((joint (m:lookup m joint-id)))
52        (c:id
53         (m:joint-theta joint)
54         (/ pi 2))))))
55
56 ;;; p2 between p1 p3 in a line
57 (define (m:c-line-order p1-id p2-id p3-id)
58   (list
59    (m:make-named-bar p1-id p2-id)
60    (m:make-named-bar p2-id p3-id)
61    (m:make-named-joint p1-id p2-id p3-id)
62    (m:c-full-angle (m:joint p1-id p2-id p3-id))))
63
```

```
(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 after 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))
         (nonsingular-classes (filter nonsingular? equivalence-classes))
         (all-cells (map cell-transformer elements)))
    (cons (c:id total-sum (ce:multi+ all-cells))
          (map (lambda (equiv-class)
                 (m:equal-values-in-sum
                  (map cell-transformer equiv-class) all-cells
                       total-sum))
               equivalence-classes))))
```

```
(define (nonsingular? equivalence-class)
  (> (length equivalence-class) 1))

(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.29: solver/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.30: solver/mechanism.scm

```
;;; 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:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Debug ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *m:debug* #f)

(define (m:pp msg) (if *m:debug* (pp msg)))

;;;;;;;;;;;;;;;;;;;;;;;; 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))))

(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)
        (m: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)
        (m: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)))
    (m: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)))
    (m: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:joint->figure-point joint)
  (m:point->figure-point (m:joint-vertex joint)))


(define (m:mechanism->figure m)
  (let ((points (map m:joint->figure-point (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 identity (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))))
    (m: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)
      (m:pp (analyze-figure f)))))
|#
```

Listing A.31: solver/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 (isosceles-triangle)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'b 'c))))

(define (isosceles-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 (isosceles-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.32: learning/interface.scm

```
;;; interface -- Main interface for learning module

;;; Discussion:

;; Ideas:
;; - "What is"

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;; Explanations ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *explain* #f)

(define (with-explanation thunk)
  (fluid-let ((*explain* #t))
    (thunk)))


;;;;;;;;;;;;;;;;;;;;;;;;;;; Current Student ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (lookup term)
  (or (lookup-definition term)
      (error "Term Unknown:" term)))

(define (example-object term)
  ((definition-generator (lookup term))))

(define (more-specific? more-specific-term less-specific-term )
  (let ((more-specific-obj (example-object more-specific-term))
        (less-specific-obj (example-object less-specific-term)))
    (is-a? less-specific-term more-specific-obj)))

(define less-specific? (flip-args more-specific?))

(define (more-specific-nonrecursive?
         more-specific-term less-specific-term )
  (let ((more-specific-obj (example-object more-specific-term))
        (less-specific-obj (example-object less-specific-term)))
    (is-a-nonrecursive? less-specific-term more-specific-obj)))

(define less-specific-nonrecursive?
  (flip-args more-specific-nonrecursive?))


;;;;;;;;;;;;;;;;;;;;;;; Definitions Interface ;;;;;;;;;;;;;;;;;;;;;;;;

(define (what-is term)
  (if (not (term-known? term))
      (pprint 'unknown)
      (pprint (lookup term))))

(define (show-example term)
  (let ((def (lookup term)))
    (show-element ((definition-generator def)))))

(define (examine object)
  (let ((satisfying-terms
         (filter
          (lambda (term)
            (is-a? term object))
          (known-terms))))
    (remove-supplanted more-specific? satisfying-terms)))

(define (examine-primitive object)
  (let ((satisfying-terms
         (filter
          (lambda (term)
```

```
            (and (primitive-definition? (lookup term))
                 (is-a? term object)))
          (known-terms))))
    (pp satisfying-terms)
    (remove-supplanted more-specific? satisfying-terms)))

(define (show-definition-lattice)
  (show-lattice (definition-lattice)))

(define (show-definition-sublattice term)
  (show-lattice-from-key (definition-lattice) term))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Applying ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(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))))

;;;;;;;;;;;;;;;;;;;;;;;;; Graphics Interfaces ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (show-element element)
  (if (polygon? element)
      (name-polygon element))
  (show-figure (figure element)))

(define (show-figure figure)
  (draw-figure figure c))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Initial Setup ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (initialize-student)
  (let ((s (make-student)))
    (set! *current-student* s)
    (provide-core-knowledge)))
```


Listing A.33: 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
   (lambda (child)
           (add-lattice-node-child! node child))
   children-nodes))

(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 node-index)
  lattice?
  (partial-order-proc lattice-partial-order-proc)
  (root lattice-root)
  (node-index lattice-node-index))

(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)))
  (let ((node-index (make-key-weak-eq-hash-table)))
    (hash-table/put! node-index
                     (lattice-node-key root)
                     root)
    (%make-lattice node-partial-order-proc root
                   node-index)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Index by Key ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (lattice-node-by-key lattice key)
  (hash-table/get
   (lattice-node-index lattice)
   key
   #f))

(define (lattice-keys lattice)
  (hash-table/key-list
   (lattice-node-index lattice)))

(define (lattice-nodes lattice)
  (hash-table/datum-list
   (lattice-node-index lattice)))

;;;;;;;;;;;;;;;;;;;;;;;; Querying Lattice ;;;;;;;;;;;;;;;;;;;;;;;;

;;; Sublattice downwards from node
(define (sublattice-nodes lattice start-key)
  (sublattice-nodes-from-key-with-getter
   lattice start-key lattice-node-children))

(define (sublattice-nodes-upwards lattice start-key)
  (sublattice-nodes-from-key-with-getter
   lattice start-key lattice-node-parents))

(define (sublattice-nodes-from-key-with-getter
         lattice start-key next-nodes-getter)
  (let ((visited '())
        (start-node (lattice-node-by-key lattice start-key)))
    (define (visited? node)
      (memq (lattice-node-key node) visited))
    (define (mark-visited node)
      (set! visited (cons (lattice-node-key node) visited)))
    (define (get-unvisited nodes)
      (let ((unvisited-nodes
             (filter (notp visited?)
                     nodes)))
        (for-each mark-visited unvisited-nodes)
        unvisited-nodes))
    (mark-visited start-node)
    (let lp ((agenda (list start-node))
             (sublattice-nodes (list start-node)))
      (if (null? agenda)
          sublattice-nodes
          (let ((node (car agenda)))
            (let ((unvisited-nodes
                   (get-unvisited (next-nodes-getter node))))
              (lp (append (cdr agenda) unvisited-nodes)
                  (append sublattice-nodes unvisited-nodes))))))))

;;;;;;;;;;;;;;;;;;;;;;;; Adding to Lattice ;;;;;;;;;;;;;;;;;;;;;;;;


(define (add-lattice-node lattice new-node)
  (if (lattice-node-by-key lattice (lattice-node-key new-node))
      'done
      (let ((visited '()))
        (hash-table/put!
         (lattice-node-index lattice)
         (lattice-node-key new-node)
         new-node)
        (define (visited? node)
          (memq (lattice-node-key node) visited))
        (define (mark-visited node)
          (set! visited (cons (lattice-node-key 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 nodes)
          (let ((unvisited-nodes
                 (filter (notp visited?) nodes)))
            (for-each mark-visited unvisited-nodes)
            unvisited-nodes))
        (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 ((children (lattice-node-children node)))
                    (let ((descendent-children
                           (filter descendent-of-new-node?
                                   children))
                          (nondescendent-children
                           (filter (notp descendent-of-new-node?)
                                   children)))
                      (add-lattice-node-children!
                       new-node descendent-children)
                      (lp (append (cdr agenda)
                                  (get-unvisited
                                   nondescendent-children)))))))))
        (let lp ((agenda (list (lattice-root lattice))))
          (if (null? agenda)
              (update-parent-child-pointers lattice new-node)
              (let ((node (car agenda)))
                (let ((children (lattice-node-children node)))
                  (let ((ancestor-children
                         (filter ancestor-of-new-node?
                                 children)))
                    (if (null? ancestor-children)
                        (begin (save-as-parent node)
                               (lp (cdr agenda)))
                        (lp (append (cdr agenda)
                                    (get-unvisited
                                     ancestor-children)))))))))))

(define (clean-children lattice node)
```

```
  (let ((children (dedupe-by eq? (lattice-node-children node))))
    (set-lattice-node-children!
     node
     (remove-supplanted
      (lattice-partial-order-proc lattice)
      children))))

(define (clean-parents lattice node)
  (let ((parents (dedupe-by eq? (lattice-node-parents node))))
    (set-lattice-node-parents!
     node
     (remove-supplanted
      (flip-args (lattice-partial-order-proc lattice))
      parents))))

(define (update-parent-child-pointers lattice new-node)
  (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 lattice 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 lattice child-node))
              children-of-new-node)
    (clean-children lattice new-node)
    (clean-parents lattice new-node)))

(define (remove-lattice-node lattice node-key)
  (let* ((node-to-remove (lattice-node-by-key lattice node-key))
         (children-of-removed-node
          (lattice-node-children node-to-remove))
         (parents-of-removed-node
          (lattice-node-parents node-to-remove)))
    (hash-table/remove! (lattice-node-index lattice)
                        node-key)
    (for-each (lambda (parent-node)
                (set-lattice-node-children!
                 parent-node
                 (append
                  (delq node-to-remove
                        (lattice-node-children parent-node))
                  children-of-removed-node))
                (clean-children lattice parent-node))
              parents-of-removed-node)
    (for-each (lambda (child-node)
                (set-lattice-node-parents!
                 child-node
                 (append
                  (delq node-to-remove
                        (lattice-node-parents child-node))
                  parents-of-removed-node))
                (clean-parents lattice child-node))
              children-of-removed-node)))

;;;;;;;;;;;;;;;;;;;;;;;; Dot Visualization ;;;;;;;;;;;;;;;;;;;;;;;;

;;; Replace - with _
(define (dot-encode-symbol symbol)
  (list->string
   (map (lambda (char)
          (if (char=? char #\-)
              #\_
              char))
        (string->list (symbol->string symbol)))))

(define (lattice-node->string node)
  (let ((key (lattice-node-key node))
        (content (lattice-node-content node)))
    (string-append
     (symbol->string key)
     (if (not (eq? key content))
         (with-output-to-string
           (lambda ()
             (write-string "\n")
             (write (print content))))
         ""))))

(define (lattice-nodes->dot-string lattice-nodes)
  (string-append
   "digraph G {"
   (apply
    string-append
    (append-map
     (lambda (node)
       (let ((node-key (lattice-node-key node)))
         (cons
          (string-append
           (dot-encode-symbol node-key)
           "[label=\"" (lattice-node->string node) "\"];\n")
          (map (lambda (child-node)
                 (string-append
                  (dot-encode-symbol node-key)
                  " -> "
                  (dot-encode-symbol (lattice-node-key child-node))
                  ";\n"))
               (lattice-node-children node)))))
```

```
     lattice-nodes))
   "}\n"))

(define (show-lattice-nodes lattice-nodes)
  (let ((dot-string (lattice-nodes->dot-string lattice-nodes)))
    (call-with-output-file "/tmp/lattice.dot"
      (lambda (dot-file)
        (write-string dot-string dot-file)))
    (run-shell-command "rm /tmp/lattice.png")
    (run-shell-command "dot -Tpng -o /tmp/lattice.png /tmp/lattice.dot")
    (run-shell-command "open /tmp/lattice.png")))

(define (show-lattice lattice)
  (show-lattice-nodes (lattice-nodes lattice)))

(define (show-lattice-from-key lattice key)
  (show-lattice-nodes
   (sublattice-nodes lattice key)))

;;; 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-lattice-node lattice root)
  (add-lattice-node lattice c)
  (add-lattice-node lattice h)
  (add-lattice-node lattice f)
  (add-lattice-node lattice e)
  (add-lattice-node lattice g)
  (add-lattice-node lattice a)
  (add-lattice-node lattice d)
  (add-lattice-node lattice b)
  (pprint root)
  (pprint a)
  (pprint b)
  (pprint c)
  (pprint d)
  (pprint e)
  (pprint f)
  (pprint g)
  (pprint h)
  (remove-lattice-node lattice 'd)
  (show-lattice-from-key lattice 'root))

; ->
(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.34: 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
                    generator
                    predicate
                    primitive?
                    all-conjectures
                    classifications
                    specific-conjectures)
  definition?
  (name definition-name)
  (generator definition-generator)
  (predicate definition-predicate set-definition-predicate!)
  (primitive? definition-primitive?)
  (all-conjectures definition-conjectures set-definition-conjectures!)
  (classifications definition-classifications
                   set-definition-classifications!)
  (specific-conjectures definition-specific-conjectures
                        set-definition-specific-conjectures!))

(define (make-primitive-definition name predicate generator)
  (%make-definition name generator predicate #t '() '() '()))

(define (primitive-definition? def)
  (and (definition? def)
       (definition-primitive? def)))

```

```
;;;;;;;;;;;;;;;;;;;;;;;; Using Definitions ;;;;;;;;;;;;;;;;;;;;;;;;

(define (definition-holds? def obj)
  (let ((classifications (definition-classifications def))
        (specific-conjectures (definition-specific-conjectures def)))
    (and ((definition-predicate def) obj)
         (every (lambda (classification-term)
                  (is-a? classification-term obj))
                classifications)
         (every (lambda (conjecture)
                  (satisfies-conjecture? conjecture (list obj)))
                specific-conjectures))))

(define (definition-holds-nonrecursive? def obj)
  (let ((all-conjectures (definition-conjectures def)))
    (and ((definition-predicate def) obj)
         (every (lambda (conjecture)
                  (satisfies-conjecture? conjecture (list obj)))
                all-conjectures))))

;;;;;;;;;;;;;;;;;;;;; Higher-order Definitions ;;;;;;;;;;;;;;;;;;;;;

(define (make-definition
         name
         generator
         primitive-predicate
         conjectures)
  (%make-definition name
                    generator
                    primitive-predicate
                    #f
                    conjectures
                    '()
                    '()))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Formatting ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (print-definition def)
  (list (definition-name def)
        (definition-classifications def)
        (map print (definition-specific-conjectures def))))

(defhandler print print-definition
  definition?)

(define (print-primitive-definition def)
  `(primitive-definition ,(definition-name def)))

(defhandler print print-primitive-definition
  primitive-definition?)
```

Listing A.35: 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 construction-dependencies
                   construction-source-procedures
                   relationship)
  conjecture?
  (construction-dependencies conjecture-constructions)
  (construction-source-procedures conjecture-construction-procedures)
  (relationship conjecture-relationship))

(define (print-conjecture conj)
  (cons
   (print (conjecture-relationship conj))
   (conjecture-constructions conj)))

(defhandler print print-conjecture conjecture?)

(define (conjecture-equal? conj1 conj2)
  (equal? (print conj1)
          (print conj2)))

(define conjecture-equivalent? conjecture-equal?)

;;; Whether


(define (satisfies-conjecture? conj premise-instance)
  (or (true? (observation-from-conjecture conj premise-instance))
      (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)))

(define (observation-from-conjecture conj premise-instance)
  (let ((new-args
```

```
         (map (lambda (construction-proc)
                (construction-proc premise-instance))
              (conjecture-construction-procedures conj)))
        (rel (conjecture-relationship conj)))
    (and (relationship-holds rel new-args)
         (make-observation rel new-args))))

;;; 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.36: learning/simplifier.scm

```
;;; 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-name 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-polygon polygon)
  (let* ((points (polygon-points polygon))
         (vertex-names (map element-name points)))
    (apply m:establish-polygon-topology vertex-names)))

(define *num->figure-trials* 20)

(define (observations->figure topology observations)
  (pprint (list 'testing observations))
  (let lp ((trials-left *num->figure-trials*))
    (if (zero? trials-left)
        #f
        (or (observations->figure-one-trial topology observations)
            (lp (- trials-left 1))))))

(define (observations->figure-one-trial topology observations)
  (initialize-scheduler)
  (let* ((constraints (observations->constraints observations))
         (m (m:mechanism topology constraints)))
    (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-polygon
       obj)
      (error "Object isn't a polygon")))

(define (polygon-from-new-figure point-names figure)
  (let* ((all-points (figure-points figure))
         (polygon-points
          (map
           (lambda (point-name)
             (find (lambda (p) (eq? (element-name p)
                                    point-name))
                   all-points))
           point-names)))
```

```
    (apply polygon-from-points polygon-points)))

(define (object-from-new-figure old-object figure)
  (if (polygon? old-object)
      (polygon-from-new-figure
       (map element-name (polygon-points old-object))
       figure)
      (error "Object isn't a polygon")))

;;;;;;;;;;;;;;;;;;;; Simple Definitions Result ;;;;;;;;;;;;;;;;;;;;

(define-record-type <simple-definitions-result>
  (%make-simple-definitions-result sufficient insufficient unknown)
  simple-definitions-result?
  (sufficient simple-def-result-sufficient
        set-simple-def-result-sufficient!)
  (insufficient simple-def-result-insufficient
          set-simple-def-result-insufficient!)
  (unknown simple-def-result-unknown
          set-simple-def-result-unknown!))

(define (make-simple-definitions-result)
  (%make-simple-definitions-result '() '() '()))

(define (mark-unknown-simple-def! def-result obs-subset)
  (set-simple-def-result-unknown! def-result
   (cons obs-subset  (simple-def-result-unknown def-result))))

(define (mark-insufficient-simple-def! def-result obs-subset)
  (set-simple-def-result-insufficient! def-result
   (cons obs-subset (simple-def-result-insufficient def-result))))

(define (mark-sufficient-simple-def! def-result obs-subset)
  (set-simple-def-result-sufficient! def-result
   (cons obs-subset (simple-def-result-sufficient def-result))))

(define (simplify-definitions-result! def-result)
  (set-simple-def-result-sufficient! def-result
   (remove-supplanted eq-subset?
                      (simple-def-result-sufficient def-result)))
  (set-simple-def-result-insufficient! def-result
   (remove-supplanted (flip-args eq-subset?)
                      (simple-def-result-insufficient def-result)))
  ;; Subsets of any insufficient ones are insufficient
  (set-simple-def-result-unknown! def-result
   (set-difference (simple-def-result-unknown def-result)
                   (simple-def-result-insufficient def-result)
                   eq-subset?))
  (set-simple-def-result-unknown! def-result
   (set-difference (simple-def-result-unknown def-result)
                   (simple-def-result-sufficient def-result)
                   (flip-args eq-subset?))))

(define (print-simple-def-result def-result)
  (list (list 'sufficient
              (map print (simple-def-result-sufficient def-result)))
        (list 'insufficient
              (map print (simple-def-result-insufficient def-result)))
        (list 'unknown
              (map print (simple-def-result-unknown def-result)))))

(define (superset-of-known-sufficient? def-result obs-subset)
  ((member-procedure eq-subset?)
   obs-subset
   (simple-def-result-sufficient def-result)))

(define (subset-of-known-insufficient? def-result obs-subset)
  ((member-procedure (flip-args eq-subset?))
   obs-subset
   (simple-def-result-insufficient def-result)))

(define (simple-def-should-test? def-result obs-subset)
  (and (not (superset-of-known-sufficient? def-result obs-subset))
       (not (subset-of-known-insufficient? def-result obs-subset))))


(defhandler print
  print-simple-def-result
  simple-definitions-result?)
```

Listing A.37: learning/student.scm

```
;;; student.scm -- base model of a student's knowlege

;;; Commentary:

;; Ideas:
;; - Definitions, constructions, theorems

;; Future:
;; - Simplifiers of redudant / uninsteresting info
;; - Propose own investigations?

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;; Student Structure ;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <student>
  (%make-student definition-dictionary
                 definition-lattice)
  student?
  (definition-dictionary student-definitions)
  (definition-lattice student-definition-lattice))

(define (make-student)
  (%make-student (make-key-weak-eq-hash-table)
                 (make-student-lattice)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Lattice ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-student-lattice)
  (make-lattice less-specific-nonrecursive?
                (make-lattice-node 'object 'object)))

;;;;;;;;;;;;;;;; Procedures using student directly ;;;;;;;;;;;;;;;;;;

(define (student-lookup-definition s name)
  (hash-table/get (student-definitions s) name #f))

(define (student-save-definition! s def)
  (hash-table/put! (student-definitions s)
                   (definition-name def)
                   def))

(define (student-known-terms s)
  (hash-table/key-list
   (student-definitions s)))

;;;;;;;;;;;;;;;;;;;; Public Versionns of student ;;;;;;;;;;;;;;;;;;;;

(define *current-student* #f)

(define (lookup-definition term)
  (student-lookup-definition *current-student* term))

(define (save-definition! def)
  (student-save-definition! *current-student* def))

(define (definition-lattice)
  (student-definition-lattice *current-student*))

(define (known-terms)
  (student-known-terms *current-student*))

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

(define (add-definition-lattice-node! term)
  (add-lattice-node
   (definition-lattice) (make-lattice-node term term))
  (update-definitions-from-lattice
   (cons term (child-terms term))))

(define (remove-definition-lattice-node! term)
  (let ((old-parent-terms (parent-terms term))
        (old-child-terms (child-terms term)))
    (remove-lattice-node
     (definition-lattice) term)
    (update-definitions-from-lattice old-parent-terms)
    (update-definitions-from-lattice old-child-terms)))

(define (add-definition! def)
  (let ((term (definition-name def)))
    (if (lookup-definition name)
        (error "Definition already exists for" term))
    (save-definition! def)
    (add-definition-lattice-node! term)))

;;;;;;;;;;;;;;;;;;;;;;;; Student Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (term-known? term)
  (lookup-definition term))

(define (is-a? term obj)
  (let ((def (lookup term)))
    (definition-holds? def obj)))

(define (is-a-nonrecursive? term obj)
  (let ((def (lookup term)))
    (definition-holds-nonrecursive? def obj)))

;;;;;;;;;;;;;;;;;;;;;;;; Learning Terms ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (learn-term term object-generator)
  (if (term-known? term)
      (error "Term already known:" term))
  (let ((example (name-polygon (object-generator))))
    (let* ((primitive-predicate (get-primitive-predicate example))
           (fig (figure (as-premise example 0)))
           (observations (analyze-figure fig))
           (conjectures (map conjecture-from-observation observations)))
      (pprint conjectures)
      (let ((new-def
             (make-definition term object-generator
                primitive-predicate conjectures)))
        (add-definition! new-def)
        (check-new-def new-def)
        'done))))

(define (get-primitive-predicate object)
  (let ((primitives (examine-primitive object)))
    (definition-predicate (lookup (car primitives)))))

(define (check-new-def new-def)
  (if (and (= 1 (length (definition-classifications new-def)))
           (null? (definition-specific-conjectures new-def)))
      (pp (string-append
           "Warning: No new known properties for term: "
           (symbol->string (definition-name new-def))
           ". Appears same as "
           (symbol->string (car (definition-classifications
                new-def)))))))

(define (all-conjectures-for-term term)
  (let* ((ancestor-terms (ancestor-terms term))
         (ancestor-defs (map lookup ancestor-terms))
```

```
         (ancestor-conjectures
          (append-map definition-specific-conjectures ancestor-defs)))
    (append (definition-specific-conjectures (lookup term))
            ancestor-conjectures)))

(define (update-definitions-from-lattice terms)
  (for-each update-definition-from-lattice terms))

(define (update-definition-from-lattice term)
  (let* ((def (lookup term))
         (current-conjectures (definition-conjectures def))
         (parent-terms (parent-terms term))
         (ancestor-terms (ancestor-terms term))
         (ancestor-defs (map lookup ancestor-terms))
         (ancestor-conjectures
          (append-map definition-conjectures ancestor-defs))
         (new-conjectures
          (set-difference current-conjectures
                          ancestor-conjectures
                          conjecture-equal?)))
    (set-definition-classifications!
     def
     parent-terms)
    (set-definition-specific-conjectures!
     def
     new-conjectures)))

(define (lattice-node-for-term term)
  (lattice-node-by-key (definition-lattice) term))

(define (child-terms term)
  (let* ((lattice-node (lattice-node-for-term term))
         (child-nodes (lattice-node-children lattice-node)))
    (map lattice-node-key child-nodes)))

(define (parent-terms term)
  (let* ((lattice-node (lattice-node-for-term term))
         (parent-nodes (lattice-node-parents lattice-node)))
    (map lattice-node-key parent-nodes)))

(define (ancestor-terms term)
  (let ((ancestor-nodes (sublattice-nodes-upwards
                         (definition-lattice)
                         term)))
    (delq term (map lattice-node-key ancestor-nodes))))

(define (descendent-terms term)
  (let ((descendent-nodes (sublattice-nodes
                           (definition-lattice)
                           term)))
    (delq term (map lattice-node-key descendent-nodes))))

;;;;;;;;;;;;;;;;;;; Getting Implied Observations ;;;;;;;;;;;;;;;;;;;

(define (observations-implied-by-term term object)
  (let ((conjectures (all-conjectures-for-term term)))
    (map (lambda (conjecture)
           (observation-from-conjecture conjecture (list object)))
         conjectures)))

;;;;;;;;;;;;;;;;;;;; Performing Investigations ;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;; Simplifying Definitions ;;;;;;;;;;;;;;;;;;;;

(define (polygon-from-object-observations object obs-subset)
  (let* ((topology (topology-for-object object))
         (new-figure (observations->figure topology obs-subset)))
    (and new-figure (object-from-new-figure object new-figure))))

(define (get-simple-definitions term)
  (let ((def (lookup term))
        (simple-def-result (make-simple-definitions-result)))
    (let* ((object ((definition-generator def)))
           (fig (figure (as-premise (name-polygon object) 0)))
           (all-observations (analyze-figure fig))
           (eligible-observations
            (filter observation->constraint all-observations)))
      (for-each
       (lambda (obs-subset)
         (if (simple-def-should-test? simple-def-result obs-subset)
             (let ((polygon
                    (polygon-from-object-observations object
                         obs-subset)))
               ((cond ((false? polygon) mark-unknown-simple-def!)
                      ((is-a? term polygon)
                       (begin (pp "=> Sufficient")
                              mark-sufficient-simple-def!))
                      (else (begin (pp "=> Insufficient")
                                   mark-insufficient-simple-def!)))
                simple-def-result obs-subset)
               (simplify-definitions-result! simple-def-result))
             (pprint `(skipping ,obs-subset))))
       (shuffle (all-subsets eligible-observations)))
      (pprint simple-def-result)
      simple-def-result)))


(define (get-simple-definitions term)
  (let ((def (lookup term))
        (simple-def-result (make-simple-definitions-result)))
    (let* ((object ((definition-generator def)))
           (fig (figure (as-premise (name-polygon object) 0)))
           (all-observations (analyze-figure fig))
           (eligible-observations
            (filter observation->constraint all-observations)))
      (for-each
       (lambda (obs-subset)
         (if (simple-def-should-test? simple-def-result obs-subset)
```

```
            (let ((polygon
                   (polygon-from-object-observations object
                        obs-subset)))
              ((cond ((false? polygon) mark-unknown-simple-def!)
                     ((is-a? term polygon) mark-sufficient-simple-def!)
                     (else mark-insufficient-simple-def!))
               simple-def-result obs-subset)
              (simplify-definitions-result! simple-def-result))
            (pprint `(skipping ,obs-subset))))
       (shuffle (all-subsets eligible-observations)))
      (pprint simple-def-result)
      simple-def-result)))
```

Listing A.38: learning/core-knowledge.scm

```
;;; core-knowledge.scm -- Core knowledge of a student

;;; Commentary:

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;; Adding to student ;;;;;;;;;;;;;;;;;;;;;;;;

(define (provide-core-knowledge)
  (for-each add-definition! primitive-definitions))

;;;;;;;;;;;;;;;;;;;;;;;; Primitive definitions ;;;;;;;;;;;;;;;;;;;;;;;;

(define triangle? (ngon-predicate 3))
(define quadrilateral? (ngon-predicate 4))

(define primitive-definitions
  (list
   (make-primitive-definition 'object true-proc true-proc)
   (make-primitive-definition 'point point? random-point)
   (make-primitive-definition 'line line? random-line)
   (make-primitive-definition 'ray ray? random-ray)
   (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)
   (make-primitive-definition 'triangle triangle?
                              random-triangle)
   (make-primitive-definition 'quadrilateral quadrilateral?
                              random-quadrilateral)))

(define primitive-terms (map definition-name primitive-definitions))
```

Listing A.39: learning/investigation.scm

```
;;; investigation.scm --- Investigation

;;; Code:

;;; Investigation Type

(define-record-type <investigation>
  (make-investigation starting-premise figure-proc)
  investigation?
  (starting-premise investigation-starting-premise)
  (figure-proc investigation-figure-procedure))


#|
Example:
|#

(define (diagonal-investigation)
  (make-investigation
   'quadrilateral
   (lambda (premise)
     (let-geo*
         ((((a b c d)) premise)
          (diag-1 (make-segment a c))
          (diag-2 (make-segment b d)))
       (figure premise diag-1 diag-2)))))

(define (midsegment-investigation)
  (make-investigation
   'quadrilateral
   (lambda (premise)
     (let-geo*
         ((((a b c d)) premise)
          (e (midpoint a b))
          (f (midpoint b c))
          (g (midpoint c d))
          (h (midpoint d a))
          (midsegment-1 (make-segment e g))
          (midsegment-2 (make-segment f h)))
       (figure premise midsegment-1 midsegment-2)))))

(define (consecutive-midpoints-investigation)
  (make-investigation
   'quadrilateral
   (lambda (premise)
     (let-geo*
         ((((a b c d)) premise)
          (e (midpoint a b))
          (f (midpoint b c))
          (g (midpoint c d))
          (h (midpoint d a))
          (p (polygon-from-points e f g h)))
       (figure premise p)))))

(define (run-investigation investigation)
  (let* ((starting-term
```

```
          (investigation-starting-premise investigation)))
    (for-each (lambda (descendent-term)
                (run-investigation-for-term
                 investigation descendent-term))
              (cons starting-term
                    (descendent-terms starting-term)))))

(define (run-investigation-for-term investigation premise-term)
  (pprint `(investigating ,premise-term))
  (let* ((figure-proc
          (investigation-figure-procedure investigation))
         (premise-def (lookup premise-term))
         (example (example-object premise-term)))
    (set-as-premise! example 0)
    (let* ((all-obs (all-observations (lambda () (figure-proc example))))
           (interesting-obs (interesting-observations (lambda ()
                (figure-proc example))))
           (investigation-conjectures
            (map conjecture-from-observation all-obs))
           (orig-conjectures (all-conjectures-for-term premise-term))
           (new-conjectures (set-difference
                             investigation-conjectures
                             orig-conjectures
                             conjecture-equivalent?))
           (new-interesting-observations
            (set-difference
             interesting-obs
             (list
              (make-observation
               (make-polygon-term-relationship premise-term)
               (list example)))
             observation-equivalent?)))
      (pprint (make-observation
               (make-polygon-term-relationship premise-term)
               (list example)))
      (set-definition-conjectures!
       premise-def
       (dedupe-by conjecture-equivalent?
                  (append orig-conjectures
                          investigation-conjectures)))
      (show-figure (figure-proc example))
      (if (not (memq premise-term primitive-terms))
          (begin (remove-definition-lattice-node! premise-term)
                 (add-definition-lattice-node! premise-term)))
      (pprint
       new-interesting-observations))))
```

Listing A.40: content/random-polygons.scm

```
;;; random-polygons.scm --- Random creation of polygons

;;; Commentary:

;; Ideas:
;; - Separated out polygons from other system-centric random proceudres
;; - These can be thought of as "user-provided" instead of system
     provided.

;; Future:
;; - More polygon types

;;; Code:

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

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

(define (random-right-triangle)
  (let* ((r1 (random-ray))
         (r2 (rotate-about (ray-endpoint r1)
                           (/ pi 2)
                           r1))
         (p1 (random-point-on-ray r1))
         (p2 (random-point-on-ray r2)))
    (polygon-from-points
     (ray-endpoint r1) p1 p2)))

(define (random-isosceles-triangle)
  (let* ((s1 (random-segment))
         (base-angle (rand-angle-measure))
         (s2 (rotate-about (segment-endpoint-1 s1)
                           base-angle
                           s1)))
    (polygon-from-points
     (segment-endpoint-1 s1)
     (segment-endpoint-2 s1)
     (segment-endpoint-2 s2))))

(define (random-right-isosceles-triangle)
  (let* ((s1 (random-segment))
         (s2 (rotate-about (segment-endpoint-1 s1)
                           (/ pi 2)
                           s1)))
    (polygon-from-points
     (segment-endpoint-1 s1)
     (segment-endpoint-2 s1)
     (segment-endpoint-2 s2))))
;;;;;;;;;;;;;;;;;;;;;;;; Random Quadrilaterals ;;;;;;;;;;;;;;;;;;;;;;;;

```

```
(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)))
    (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))))
    (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))))
    (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)))
    (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))))
    (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)))
    (polygon-from-points p1 p2 p3 p4)))

(define (random-orthodiagonal-quadrilateral)
  (let* ((r1 (random-ray))
         (r2 (rotate-about
              (ray-endpoint r1)
              (/ pi 2)
              r1))
         (r3 (reverse-ray r1))
         (r4 (reverse-ray r2))
         (a (random-point-on-ray r1))
         (b (random-point-on-ray r2))
         (c (random-point-on-ray r3))
         (d (random-point-on-ray r4)))
    (polygon-from-points a b c d)))

(define (random-cyclic-quadrilateral)
  (let ((cir (random-circle)))
    (let lp ()
      (let ((points (n-random-points-on-circle-ccw cir 4)))
        (if (points-non-overlapping? points)
            (apply polygon-from-points points)
            (lp))))))

(define (random-equidiagonal-quadrilateral)
  (let* ((s (random-segment))
         (p1 (random-point-on-segment s))
         (s-rotated (rotate-randomly-about p1 s))
         (p2 (random-point-on-segment s-rotated))
         (s2 (translate-by
              (sub-points p1 p2)
              s-rotated)))
    (polygon-from-points (segment-endpoint-1 s)
                         (segment-endpoint-1 s2)
                         (segment-endpoint-2 s)
                         (segment-endpoint-2 s2))))

(define (random-isosceles-trapezoid)
  (let* ((a1 (random-obtuse-angle))
         (p1 (angle-vertex a1))
```

```
        (r1 (ray-from-arm-1 a1))
        (r2 (ray-from-arm-2 a1))
        (p4 (random-point-on-ray r2))
        (p2 (random-point-on-ray r1))
        (s (make-segment p1 p2))
        (pb (perpendicular-bisector s))
        (p3 (reflect-about-line pb p4)))
    (polygon-from-points p1 p2 p3 p4)))

(define (random-3-equal-trapezoid)
  (let* ((a1 (random-obtuse-angle))
        (p1 (angle-vertex a1))
        (r1 (ray-from-arm-1 a1))
        (r2 (ray-from-arm-2 a1))
        (p2 (random-point-on-ray r1))
        (p4 (measured-point-on-ray
             r2 (distance p1 p2)))
        (s (make-segment p1 p2))
        (pb (perpendicular-bisector s))
        (p3 (reflect-about-line pb p4)))
    (polygon-from-points p1 p2 p3 p4)))
```

Listing A.41: 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 (isosceles-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.42: content/walkthrough.scm

```
;;; Sample walkthrough, also used as a sort of "system test"

;;;;;;;;;;;;;;;;;;;;;;;;; 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 'isosceles-triangle)
(learn-term 'isosceles-triangle random-isosceles-triangle)
(what-is 'isosceles-triangle)
(is-a? 'isosceles-triangle (random-isosceles-triangle))
(is-a? 'isosceles-triangle (random-equilateral-triangle))
(is-a? 'isosceles-triangle (random-triangle))

(learn-term 'equilateral-triangle random-equilateral-triangle)
(what-is 'equilateral-triangle)
(is-a? 'equilateral-triangle (random-isosceles-triangle))
(is-a? 'equilateral-triangle (random-equilateral-triangle))

(learn-term 'right-isosceles-triangle random-right-isosceles-triangle)
(learn-term 'right-triangle random-right-triangle)

;;;;;;;;;;;;;;; Let's learn some basic quadrilaterals ;;;;;;;;;;;;;;;

;;; Notice Random Ordering:

(learn-term 'kite random-kite)
(what-is 'kite)

(learn-term 'rectangle random-rectangle)
(what-is 'rectangle)

(learn-term 'trapezoid random-trapezoid)
(what-is 'trapezoid)

(learn-term 'square random-square)
(what-is 'square)

(learn-term 'orthodiagonal random-orthodiagonal-quadrilateral)
(what-is 'orthodiagonal)

```

```
(learn-term 'parallelogram random-parallelogram)
(what-is 'parallelogram)

(learn-term 'rhombus random-rhombus)
(what-is 'rhombus)

(learn-term 'equidiagonal random-equidiagonal-quadrilateral)
(what-is 'equidiagonal)

(learn-term 'cyclic random-cyclic-quadrilateral)
(what-is 'cyclic)

(learn-term 'isosceles-trapezoid random-isosceles-trapezoid)
(what-is 'isosceles-trapezoid)

(learn-term 'three-equal-trapezoid random-3-equal-trapezoid)
(what-is 'three-equal-trapezoid)


;;; Invetigations to disambiguate equidiagonal / orthodiagonal

(run-investigation-for-term (diagonal-investigation) 'equidiagonal)
(run-investigation-for-term (diagonal-investigation) 'orthodiagonal)

;;;;;;;;;;;;;;;;;;;;;;; More Investigations ;;;;;;;;;;;;;;;;;;;;;;;;
;; (run-investigation (diagonal-investigation))
;; (run-investigation (midsegment-investigation))

;;;;;;;;;;;;;;;;;;;;; Check definition-lattice ;;;;;;;;;;;;;;;;;;;;;
(show-definition-lattice)

;;;;;;;;;;;;;;;;;;;;;;;; Check Simple Terms ;;;;;;;;;;;;;;;;;;;;;;;;
(get-simple-definitions 'isosceles-triangle)
```

Listing A.43: content/investigations.scm

```
;;; investigations.scm -- Some sample investigations and ideas that
;;; could be persued

;;; Linear Pair Conjecture
;;; Givens: Angles a-1 and a-2 form a linear pair
;;; Goal: m(a-1) + m(a-2) = 180 degrees
(define (linear-pair)
  (let-geo* ((a (random-point))
             (l1 (random-line-through-point a))
             (r (random-ray-from-point a))
             (a-1 (smallest-angle-from l1 r))
             (a-2 (smallest-angle-from r (flip l1))))
    (figure a l1 r a-1 a-2)))

;;; Vertical Angles Conjecture
;;; Givens: Angles a-1 and a-2 are vertical angles
;;; Goal: m(a-1) = m(a-2)
(define (vertical-angles)
  (let-geo* ((l1 (random-line))
             (c (random-point-on-line l1))
             (l2 (rotate-randomly-about c l1))
             (a-1 (smallest-angle-from l1 l2))
             (a-2 (smallest-angle-from (flip l1) (flip l2))))
    (figure l1 c l2 a-1 a-2)))

;;; Corresponding Angles Conjecture
;;; Givens: - Lines l1 and l2 are parallel
;;;         - Line l3 is a transversal
;;;         - a-1 and a-2 are resulting corresponding angles
;;; Goal: m(a-1) = m(a-2)
(define (corresponding-angles)
  (let-geo* ((l1 (random-line))
             (l2 (translate-randomly l1))
             (a (random-point-on-line l1))
             (b (random-point-on-line l2))
             (l3 (line-from-points a b))
             (a-1 (smallest-angle-from l3 l2))
             (a-2 (smallest-angle-from l3 l1)))
            (figure l1 l2 a b l3 a-1 a-2)))

;;; Interior / alternate interior: ordering of angles and

;;; Converse of Parallel lines
;;; Givens: - m(a-1) = m(a-2)
;;;         - a-1, a-2, are either CA, AIA, AEA, etc. of Lines l1, l2
;;; Goal: lines l1 and l2 are parallel
(define (parallel-lines-converse)
  (let-geo* ((a-1 (random-angle))
             (l3 (line-from-arm-1 a-1))
             (a-2 (translate-randomly-along-line l3 a-1))
             (l1 (line-from-arm-2 a-1))
             (l2 (line-from-arm-2 a-2)))
    (figure a-1 a-2 l1 l2 l3)))

;;; Perpendicular bisector conjecture
;;; Givens: - p is a point on perpendicular bisector of segment (a, b)
;;; Goal: p is equidistant from a and b
(define (perpendicular-bisector-equidistant)
  (let-geo* (((s (a b)) (random-segment))
             (l1 (perpendicular-bisector s))
             (p (random-point-on-line l1)))
            (figure s l1 p)))

;;; Converse of perpendicular bisector conjecture
;;; Given: - a and b are equidistant from point p
;;; Goal: p is on the perpendicular bisector of a, b
(define (perpendicular-bisector-converse)
  (let-geo* ((p (random-point))
             (a (random-point))
             (b (rotate-randomly-about p a))
             (s (make-segment a b))
```

```
            (pb (perpendicular-bisector s)))
           (figure p a b s pb)))

;;; Angle bisector conjecture
;;; Given: angle a-1 of rays r-1, r-2, point a on angle-bisector l1
;;; Goal: Distnace from a to r-1 = distance a to r-2

(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)))
;;; Interesting, dependent on "shortest distance" from prior conjecture

;;; Angle bisector concurrency
;;; Given: Triangle abc with angle-bisectors l1, l2, l3
;;; Goal: l1, l2, l3 are concurrent
(define (angle-bisector-concurrency)
  (let-geo* (((t1 (a b c)) (random-triangle))
             (((a-1 a-2 a-3)) (polygon-angles t1))
             (l1 (polygon-angle-bisector t1 a))
             (l2 (polygon-angle-bisector t1 b))
             (l3 (polygon-angle-bisector t1 c)))
    (figure t1 l1 l2 l3)))

;;; Perpendicular Bisector Concurrency
;;; Given: Triangle ABC with sides s1, s2, s3, perpendicular bisectors
;;; l1, l2, l3
;;; Goal: l1, l2, l3 are concurrent
(define (perpendicular-bisector-concurrency)
  (let-geo* (((t (a b c)) (random-triangle))
             (l1 (perpendicular-bisector (make-segment a b)))
             (l2 (perpendicular-bisector (make-segment b c)))
             (l3 (perpendicular-bisector (make-segment c a))))
    (figure t l1 l2 l3)))

;;; Altitude Concurrency
;;; Given: Triangle ABC with altituds alt-1, alt2, alt-3
;;; Goal: alt-1, alt-2, alt-3 are concurrent
(define (altitude-concurrency)
  (let-geo* (((t (a b c)) (random-triangle))
             (alt-1 (perpendicular-line-to (make-segment b c) a))
             (alt-2 (perpendicular-line-to (make-segment a c) b))
             (alt-3 (perpendicular-line-to (make-segment a b) c)))
           (figure t alt-1 alt-2 alt-3)))
```

Listing A.44: content/initial-demo.scm

```
;;; Initial System Demo, Early Spring 2015

(define (i-t-figure)
  (let-geo* (((t (a b c)) (random-isosceles-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-isosceles-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)))

;;;;;;;;;;;;;;;;;;;;;; Original 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 (run-initial-demo)
  (for-each (lambda (figure)
              (run-figure figure))
            interesting-figures)
  'done)
```

Listing A.45: 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)

(define *animate-value-only* #f)

;;;;;;;;;;;;;;;;;;;;;;;; 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)
  (if *animate-value-only*
      (f (random 1.0))
      (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))))))

;;;;;;;;;;;;;;;;;;;; Selected Animation Frames ;;;;;;;;;;;;;;;;;;;;

(define (n-random-frames n f)
  (fluid-let ((*animate-value-only* #t)
              (*is-animating?* #t)
              (*persistent-values-table* (make-key-weak-eq-hash-table))
              (*animation-value* 0))
    (map (lambda (x) (fluid-let ((*next-value-index* 0)) (f))) (iota
         n))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; 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.46: 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))

;;;;;;;;;;;;;;;;;;;;;; Setting Dependencies ;;;;;;;;;;;;;;;;;;;;;;

(define (args-from-premise args)
  (map (lambda (arg)
         `(from-new-premise p ,arg))
       args))

(define (set-dependency-expressions assignments)
  (append-map
   (lambda (a)
     (let ((name (car a))
           (value (cadr a)))
       (if (list? value)
           (let ((proc (car value))
                 (args (cdr value)))
             `((set-source!
                ,name (lambda (p) (,proc ,@(args-from-premise args))))
               (set-dependency!
                ,name (list (quote ,proc) ,@args))))
           `((set-source! ,name (element-source ,value))
             (set-dependency! ,name (element-dependency ,value))))))
   assignments))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; 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 (cddr exp)))
       (let ((new-assignments (expand-assignments assignments))
             (variable-names (variables-from-assignments assignments)))
         (let ((result `(let*
                            ,new-assignments
                          ,@(set-name-expressions variable-names)
                          ,@(set-dependency-expressions new-assignments)
                          ,@body)))
           ;; (pp result) ;; Uncomment to debug macro expansion
           (close-syntax result env)))))))
```

Listing A.47: 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.48: core/utils.scm

```
(define (assert boolean error-message)
  (if (not boolean) (error error-message)))

;;;;;;;;;;;;;;;;;;;;;;;;;; List Utilities ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (sort-by-key l key)
  (sort l (lambda (v1 v2)
            (< (key v1)
               (key  v2)))))

(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))))))

;;; Swaps the elements at indices i and j in the vector
(define (swap vec i j)
  (let ((tmp (vector-ref vec i)))
    (vector-set! vec i (vector-ref vec j))
    (vector-set! vec j tmp)))

(define (shuffle alts)
  (let ((alts-vec (list->vector alts))
        (num-alts (length alts)))
    (if (= num-alts 0)
        alts
        (let lp ((to-index (- num-alts 1)))
          (cond
           ((= to-index 0) (vector->list alts-vec))
           (else (let ((from-index
                        (random (+ 1 to-index))))
                   (swap alts-vec from-index to-index)
                   (lp (- to-index 1)))))))))

(define (flatten list)
  (cond ((null? list) '())
        ((list? (car list))
         (append (flatten (car list))
                 (flatten (cdr list))))
        (else (cons (car list) (flatten (cdr list))))))

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

(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)

(define (true? x)
  (if x #t #f))

;;;;;;;;;;;;;;;;;;;;;;;;;; Set Operations ;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; 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 (set-equivalent? set1 set2 equality-predicate)
  (and (subset? set1 set2 equality-predicate)
       (subset? set2 set1 equality-predicate)))

(define (set-equivalent-procedure equality-predicate)
  (lambda (set1 set2)
    (set-equivalent? set1 set2 equality-predicate)))

(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 (distinct? elements equality-predicate)
  (= (length elements)
     (length (set-intersection
              elements elements
              (member-procedure equality-predicate)))))

```

```
(define (dedupe-eq elements)
  (dedupe-by eq? elements))

(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-supplanted supplants-predicate elements)
  (define member-predicate (member-procedure
                            supplants-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 (all-subsets elements)
  (append-map
   (lambda (n)
     (all-n-tuples n elements))
   (iota (+ (length elements) 1))))

;;;;;;;;;;;;;;;;;;;;;;; Equivalence Classes ;;;;;;;;;;;;;;;;;;;;;;;;

(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))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Majorities ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Runs procedure on random animation frames and checks that results
;;; appear in a majority of frames.

(define *majority-trials-total* 3)
(define *majority-trials-required* 2)

(define (require-majority f equality-predicate)
  (require-enough f *majority-trials-total* *majority-trials-required*
                  equality-predicate))

(define (require-enough f total-trials num-required equality-predicate)
  (let ((all-executions (n-random-frames total-trials f)))
    (check-enough all-executions num-required equality-predicate)))

(define (check-enough execution-results num-required equality-predicate)
  (let ((hash-table ((weak-hash-table/constructor
                      (lambda (a b) 1) equality-predicate))))
    (for-each (lambda (execution-result)
                (for-each (lambda (element)
                            (hash-table/append hash-table
                             element element))
                          execution-result))
              execution-results)
    (filter identity
            (map (lambda (a-pair)
                   (and (>= (length (cdr a-pair)) num-required)
                        (car a-pair)))
                 (hash-table->alist hash-table)))))

;;;;;;;;;;;;;;;;;;;;;;;; Function Utilities ;;;;;;;;;;;;;;;;;;;;;;;;

(define ((negatep f) x)
  (- (f x)))

(define ((flip-args f) x y)
  (f y x))

(define (memoize-function f)
  (let ((cache (make-key-weak-eq-hash-table)))
    (lambda (arg)
```

```
      (hash-table/intern!
       cache
       arg
       (lambda () (f arg))))))

;;;;;;;;;;;;;;;;;;;;;;;;; Other Utilities ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (eq-append! element key val)
  (eq-put! element key
           (cons val
                 (or (eq-get element key) '()))))

;;; (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 '()))))
```

Listing A.49: lib/close-enuf.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))))
```



```

;;; end GJS
```

Listing A.50: lib/eq-properties.scm

```
;;;; 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)
;;; Propagators also provide this. The above makes the below a
;;; compatible extension of that version
#|
;;;;           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.


(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))


;;; 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)))))
|#
```

Appendix B

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. *Mechanical geometry theorem proving*, volume 41. Springer Science & Business Media, 1988.

[4] 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.

[5] Tom Davis. Geometer dynamic geometry program. Software available at http://www.geometer.org/geometer/index.html, 2009.

[6] Joran Elias. Automated geometric theorem proving: Wu's method. *The Montana Mathematics Enthusiast*, 3(1):3–50, 2006.

[7] Anne Berit Fuglestad. *Discovering geometry with a computer: using Cabri-géomètre*. Chartwell-Yorke, 114 High Street, Belmont, Bolton, Lancashire, BL7 8AL, England, 1994.

[8] Herbert Gelernter. Realization of a geometry theorem proving machine. In *Computers and Thought*, pages 134–152, 1963.

[9] Ira Goldstein. Elementary geometry theorem proving. *AI Memo 280, Massachusetts Institute of Technology*, 1973.

[10] 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.

[11] Mateja Jamnik. *Mathematical Reasoning with Diagrams*. University of Chicago Press, 2001.

[12] Robert Joan-Arinyo. Basics on geometric constraint solving. *Proceedings of 13th Encuentros de Geometrfa Computacional (EGC09), Zaragoza (Spain)*, 2009.

[13] 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.

[14] Glenn A Kramer. *Solving geometric constraint systems: a case study in kinematics*. MIT press, 1992.

[15] Mark Levi. *The mathematical mechanic: using physical reasoning to solve problems*. Princeton University Press, 2009.

[16] 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.

[17] Julien Narboux. A graphical user interface for formal proofs in geometry. *Journal of Automated Reasoning*, 39(2):161–180, 2007.

[18] Arthur J Nevins. Plane geometry theorem proving using forward chaining. *Artificial Intelligence*, 6(1):1–23, 1975.

[19] Stavroula Patsiomitou and Anastassios Emvalotis. Developing geometric thinking skills through dynamic diagram transformations. In *6th Mediterranean Conference on Mathematics Education*, pages 249–258, 2009.

[20] Pavel Pech. Deriving geometry theorems by automated tools. In *Proceedings of the Sixteenth Asian Technology Conference in Mathematics*. Mathematics and Technology, LLC, 2011.

[21] Alexey Radul. *Propagation networks: A flexible and expressive substrate for computation*. PhD thesis, Massachusetts Institute of Technology, 2009.

[22] Alexey Radul and Gerald Jay Sussman. The art of the propagator. Technical report, Massachusetts Institute of Technology, 2009.

[23] 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.

[24] Michael Serra. *Discovering geometry: An investigative approach*, volume 4. Key Curriculum Press, 2003.

[25] Gerald Jay Sussman. Slices: At the boundary between analysis and synthesis. *Massachusetts Institute of Technology AI Memo*, 1977.

[26] Gerald Jay Sussman et al. Scmutils library. MIT Scheme Mechanics Mathematics Library, http://groups.csail.mit.edu/mac/users/gjs/6946/linux-install.htm, 2014.

[27] Vladimir Andreevich Uspenskii, Halina Moss, and Ian N Sneddon. *Some applications of mechanics to mathematics*. Pergamon Press Oxford-London-New York-Paris, 1961.

[28] 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.

[29] Franz Winkler, editor. *Automated Deduction in Geometry*, volume 2930 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004.