

Automated Elementary Geometry Theorem Discovery via Inductive Diagram Manipulation

by

Lars Erik Johnson

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

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

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

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

Accepted by
Albert Meyer
Chairman, Masters of Engineering Thesis Committee

Automated Elementary Geometry Theorem Discovery via Inductive Diagram Manipulation

by

Lars Erik Johnson

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

Abstract

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

Thesis Supervisor: Gerald J. Sussman

Title: Panasonic Professor of Electrical Engineering

Acknowledgments

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

Contents

1	Introduction	11
1.1	Document Structure	12
2	Motivation and Examples	15
2.1	Manipulating Diagrams “In the Mind’s Eye”	17
2.1.1	An Initial Example	17
2.1.2	Diagrams, Figures, and Constraints	18
2.2	Geometry Investigation	19
2.2.1	Vertical Angles	19
2.2.2	Elementary Results	20
2.2.3	Nine Point Circle and Euler Segment	21
3	Demonstration	23
3.1	Imperative Figure Construction	23
3.2	Perception and Observation	27
3.3	Mechanism-based Declarative Constraint Solver	29
3.3.1	Bars and Joints	30
3.3.2	Geometry Examples	31
3.4	Learning Module	38
4	System Overview	45
4.1	Goals	45
4.2	Diagram Representations	46

4.2.1	Modules	48
4.3	Sample Interaction	49
4.3.1	Interpreting Construction Steps	49
4.3.2	Creating Figures	49
4.3.3	Noticing Interesting Properties	50
4.3.4	Simplifying Definitions and Known Facts	50
4.3.5	Reporting Findings	50
4.4	Example Interaction	50
5	Imperative Construction System	51
5.1	Overview	51
5.2	Construction Interface	51
5.3	Basic Structures	51
5.3.1	Points	52
5.3.2	Linear Elements	53
5.3.3	Angles	53
5.3.4	Math Support	53
5.4	Higher-level structures	54
5.4.1	Polygons	54
5.4.2	Figures	54
5.5	Construction Operations	54
5.5.1	Traditional constructions	54
5.5.2	Intersections	55
5.5.3	Measurement-based operations	55
5.5.4	Transformations	55
5.6	Randomness	55
5.6.1	Random Choices	55
5.6.2	Remembering choices	56
5.6.3	Backtracking	56
5.6.4	Avoiding almost-degenerate points	56

5.6.5	Animating choices	56
5.7	Dependencies	56
5.7.1	Implementation	56
5.7.2	Naming	56
5.7.3	Forcing higher-level random dependences	56
5.7.4	Dependency-less diagrams	57
5.8	Construction Language	57
5.8.1	Macros	57
5.8.2	Multiple Assignment	57
5.9	Graphics	57
6	Perception Module	59
6.1	Overview	59
6.1.1	Extracting segments and angles	60
6.1.2	What is Interesting?	60
6.1.3	Removing Obvious Properties	61
6.2	Representations	61
7	Declarative Geometry Constraint Solver	63
7.1	Overview	63
7.2	Mechanical Analogies	63
7.3	Example	64
7.3.1	Bar and Joint Linkages	64
7.3.2	Mechanism	65
7.4	Partial Information	65
7.4.1	Regions	65
7.4.2	Direction Intervals	65
7.5	Propagator Constraints	66
7.5.1	Basic Linkage Constraints	66
7.5.2	Higher Order Constraints	66
7.6	Solving: Specification Ordering	67

7.6.1	Anchored vs. Specified vs. Initialized	67
7.7	Backtracking	67
7.8	Interfacing with existing diagrams	67
7.9	Specification Interface	68
8	Learning Module	69
8.1	Overview	69
8.2	Interactions with Learning Module	70
8.2.1	Querying	70
8.2.2	Learning Definitions	70
8.2.3	Applying Learned Properties	71
8.2.4	Simplifying Definitions	72
8.2.5	Performing Investigations	73
8.3	Representing Discoveries	73
8.3.1	Placement of discoveries	73
8.3.2	Ordering of discoveries	73
8.3.3	Pattern Matching against existing conjectures	74
9	Related Work	75
9.1	Dynamic Geometry	75
9.2	Software	76
9.3	Automated Proof and Discovery	76
10	Conclusion	77
10.1	Overview	77
10.2	Extensions	77
A	Code Listings	79

Chapter 1

Introduction

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

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

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

To evaluate its recognition of such concepts, my system provides means for a user to extract the observations and apply its findings to new scenarios. Through a series of simple investigations similar to an introductory course in geometry, the system has

been able to propose and learn a few dozen standard geometry theorems. Furthermore, through more self-directed explorations, it has discovered several interesting properties and theorems not typically covered in standard mathematics courses.

1.1 Document Structure

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

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

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

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

Chapter 5 describes the implementation and function of the imperative construction module that enables the system to carry out constructions.

Chapter 6 describes the implementation and function of the perception module focused on observing interesting properties in diagrams. A key question involves determining “what is interesting”.

Chapter 7 describes the implementation and function of the propagator-based declarative geometry constraint solver that builds instances of diagrams satisfying declarative constraints.

Chapter 8 describes the analyzer module which integrates results from the other systems to create new discoveries. Main features include filtering out obvious or known results to focus on the most interesting discoveries, the persistence and storage of definitions and theorems, and an interface to apply these findings to new situations.

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

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

Chapter 2

Motivation and Examples

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

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

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

As an example, a curious student might notice that in a certain drawing of a

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

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

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

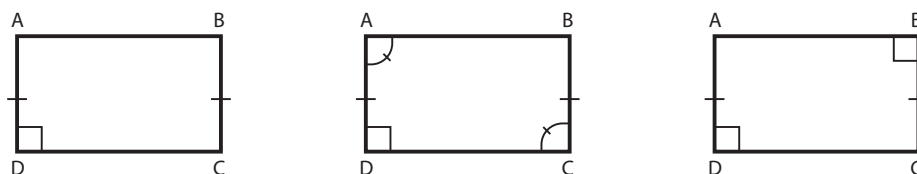
To evaluate its recognition of such concepts, the interactive system provide means for a user to extract the observations and apply such findings to new scenarios. In addition to the automated reasoning and symbolic artificial intelligence aspects of a system that can learn and reason inductively about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

and proof systems.

2.1 Manipulating Diagrams “In the Mind’s Eye”

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

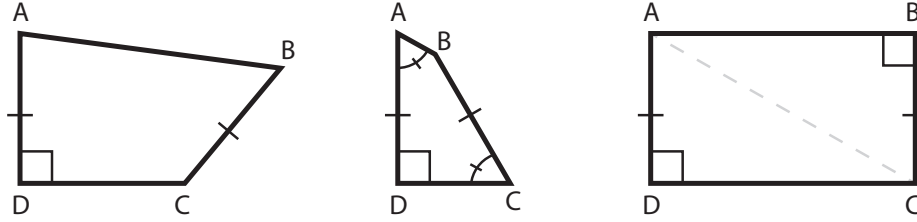
2.1.1 An Initial Example



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

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

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



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

2.1.2 Diagrams, Figures, and Constraints

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

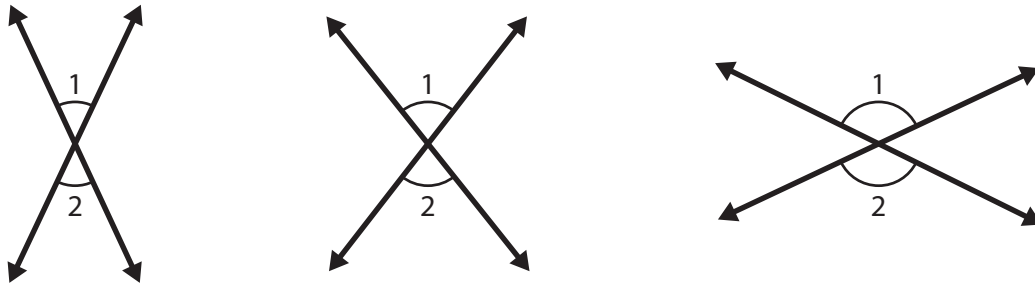
An annotated figure presented on a page is typically an instance of its corresponding diagram. However, it is certainly possible to add annotations to a figure that are not satisfied by that figure, yielding impossible diagrams. In such a case the diagram represents an empty set of satisfying figures.

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

2.2 Geometry Investigation

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

2.2.1 Vertical Angles

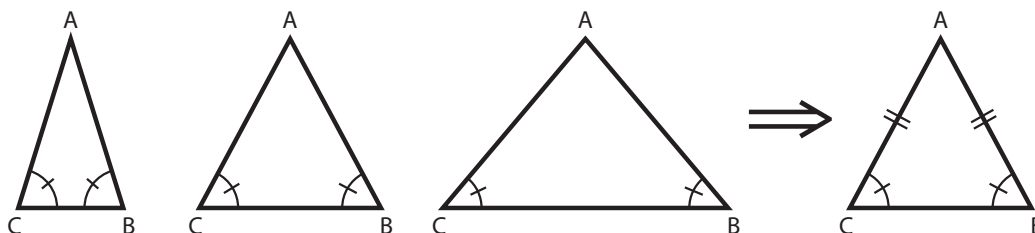


Investigation 1: Construct a pair of vertical angles. Notice anything interesting?

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

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

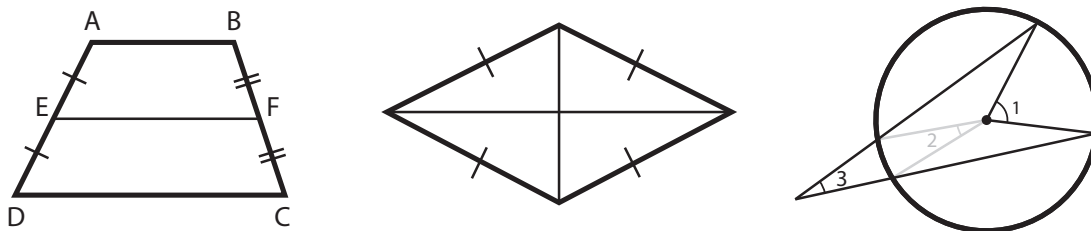
2.2.2 Elementary Results



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

A slightly more involved example includes discovering that if a triangle has two congruent angles, it is 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.

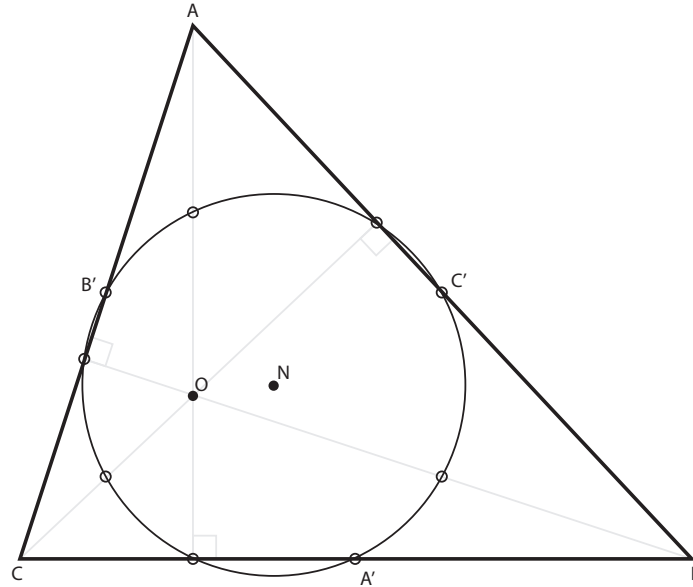
In this and further examples, an important question becomes what properties are considered “interesting” and worth investigating in further instances of the diagram, as discussed in section 4.3.3. As suggested by the examples in Investigation 3, this can include relations between segment and angle lengths,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

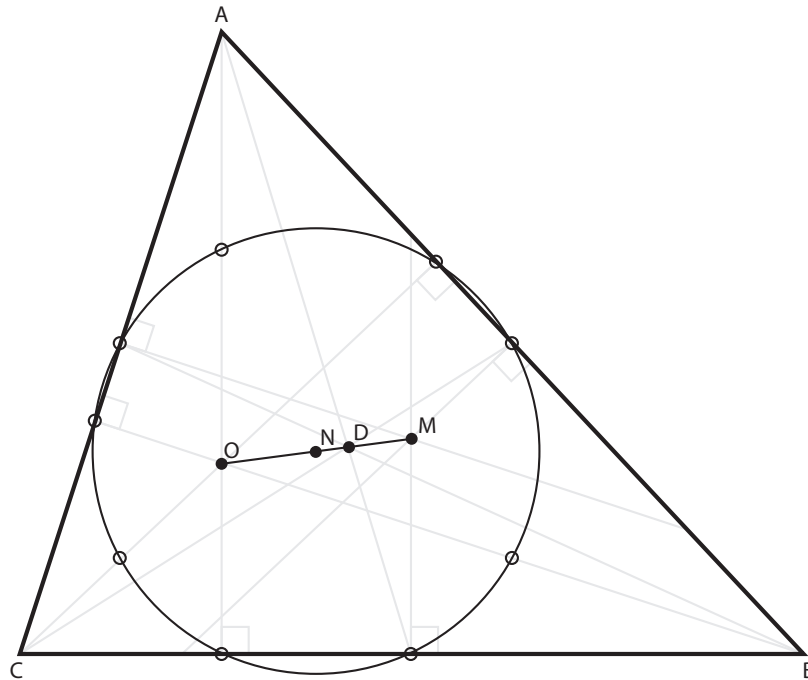
Finally, this technique can be used to explore and discover conjectures well beyond the scope of what one can visualize in his or her head:



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 (below), further “interesting properties” continue to appear as one constructs the centroid and circumcenter: All four of these special points (O , N , D , and M) are collinear on what is called the *Euler Segment*, and the ratios $ON : ND : DM$ of $3 : 1 : 2$ hold for any triangle.



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

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

Chapter 3

Demonstration

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

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

3.1 Imperative Figure Construction

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

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

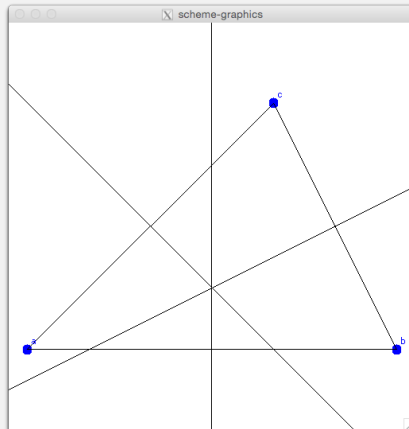
Code Example 3.1: Basic Figure Example

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

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

Interaction Example 3.2: Rendering the Basic Figure

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



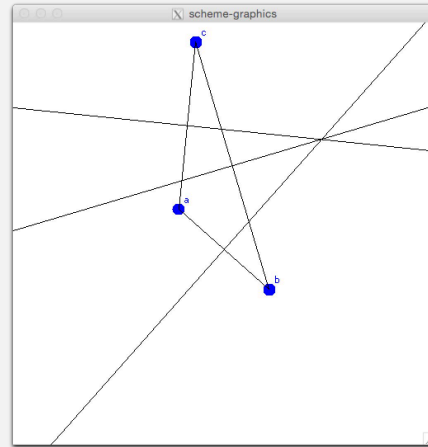
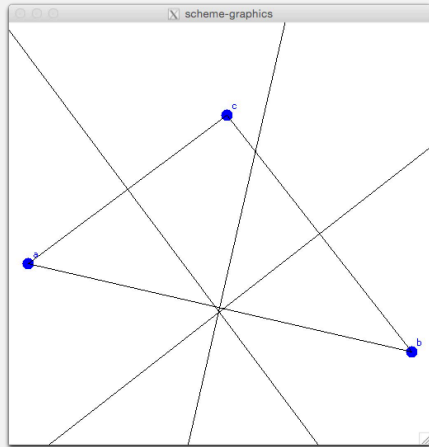
In the first figure, the coordinates of the point were explicitly specified yielding a deterministic instance of the figure. However, as geometry figures often involve arbitrary choices, the construction abstractions support random choices. Figure 3.3 demonstrates the creation of a figure involving an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

tension provided by `let-geo*` that shortens the common pattern of accessing and naming the components of a random object.

Interaction Example 3.3: Introducing Randomness

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

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

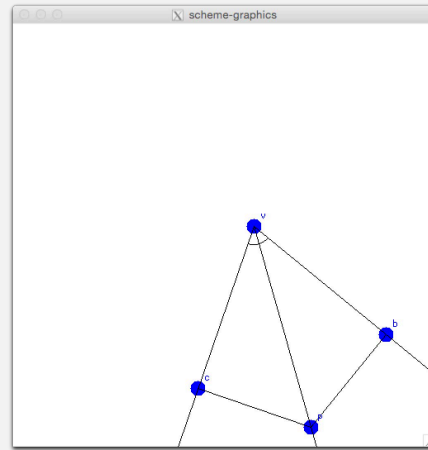
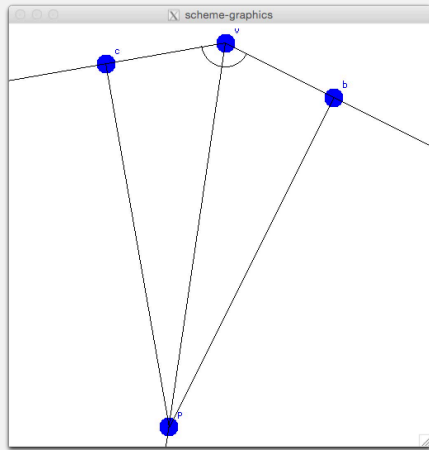


Finally, as examples of more involved constructions, Examples 3.4 and 3.5 demonstrate working with other objects (angles, rays, circles) and construction procedures. Notice that in the angle bisector example the pattern matching syntax extracts the components of an angle (ray, vertex, ray) and segment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

Interaction Example 3.4: Angle Bisector Distance

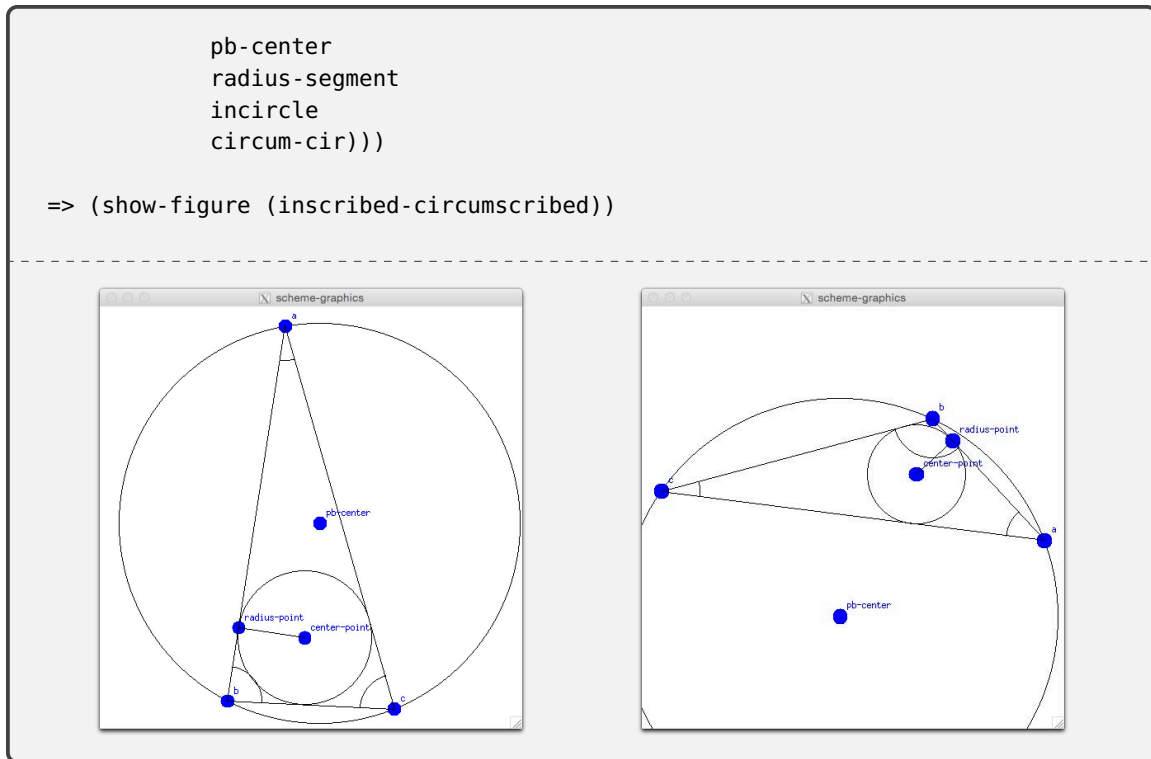
```
(define (angle-bisector-distance)
  (let-geo* (((a (r-1 v r-2)) (random-angle))
             (ab (angle-bisector a))
             (p (random-point-on-ray ab))
             ((s-1 (p b)) (perpendicular-to r-1 p))
             ((s-2 (p c)) (perpendicular-to r-2 p)))
    (figure a r-1 r-2 ab p s-1 s-2)))

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



Interaction Example 3.5: Inscribed and Circumscribed Circles

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



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

3.2 Perception and Observation

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

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

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

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

Interaction Example 3.6: Simple Analysis

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

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

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

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

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

Interaction Example 3.7: Interesting Analysis

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

((concurrent pb1 pb2 pb3))
```

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

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

Interaction Example 3.8: Parallelogram Analysis

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

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

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

3.3 Mechanism-based Declarative Constraint Solver

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

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

Thus, the third module is a declarative constraint solver. To model the physical

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

3.3.1 Bars and Joints

Example 3.9 demonstrates the specification of a very simple mechanism. Mechanisms are created by specifying the bars and joints involved as well as any additional constraints that must be satisfied. This example mechanism is composed of two bars with one joint between them that is constrained to be a right angle.

Code Example 3.9: Very Simple Mechanism

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

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

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

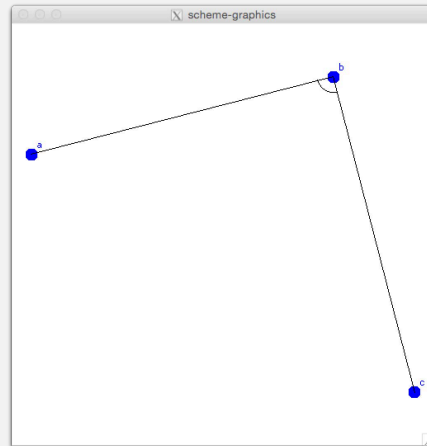
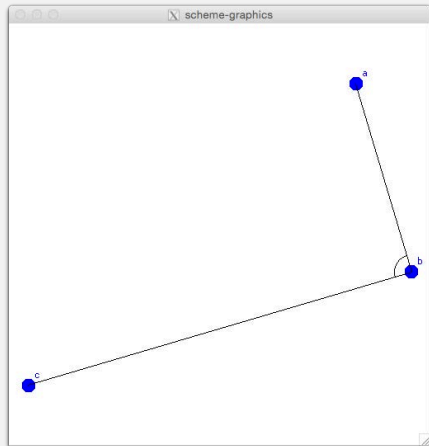
The printed statements in Example 3.10 demonstrate that solving the simple mechanism above involves specifying the location of point a, then specifying the length

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

Interaction Example 3.10: Solving the Very Simple Mechanism

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

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



3.3.2 Geometry Examples

These bar and linkage mechanisms can be used to represent the topologies of several geometry figures. Bars correspond to segments and joints correspond to angles. Example 3.11 demonstrates the set of linkages necessary to specify the topology of a triangle. The `m:establish-polygon-topology` procedure simplifies the specification of a closed polygon of joints.

Code Example 3.11: Describing an Arbitrary Triangle

```
1 (define (arbitrary-triangle)
2   (m:mechanism
3     (m:make-named-bar 'a 'b)
4     (m:make-named-bar 'b 'c)
5     (m:make-named-bar 'c 'a)
6     (m:make-named-joint 'a 'b 'c)
7     (m:make-named-joint 'b 'c 'a)
8     (m:make-named-joint 'c 'a 'b)))
9
10 (define (simpler-arbitrary-triangle)
11   (m:mechanism
12     (m:establish-polygon-topology 'a 'b 'c)))
```

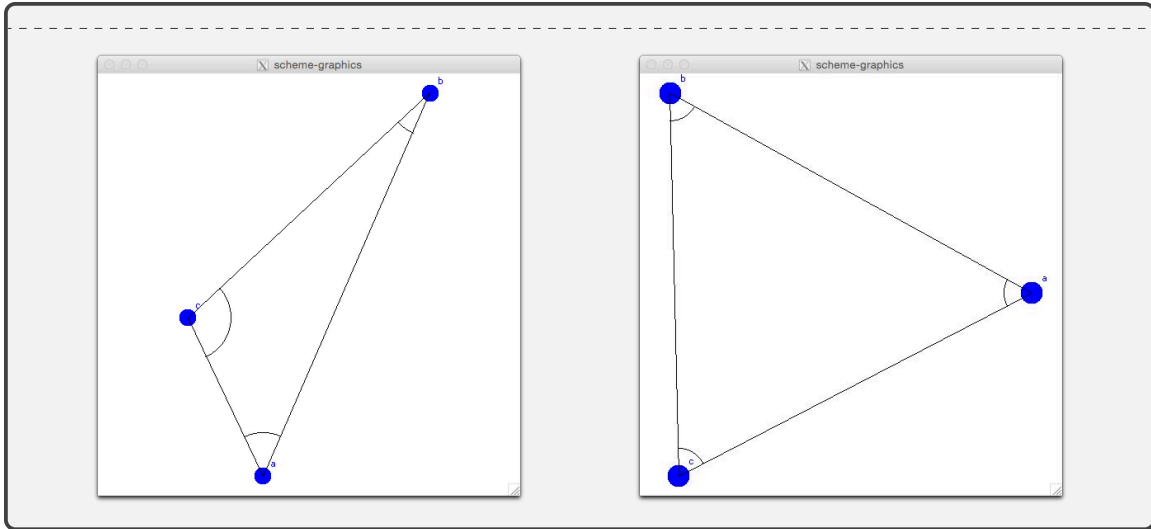
In example 3.12, Once joints b and c have had their angles specified, propagation sets the angle of joint a to a unique value. The only parameter to specify is the length of one of the bars. The two **initializing-** steps don't affect the resulting shape but determine its position and orientation on the canvas.

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

Interaction Example 3.12: Solving the Triangle

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

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

Example 3.13 shows the solving steps involved in solving an 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.

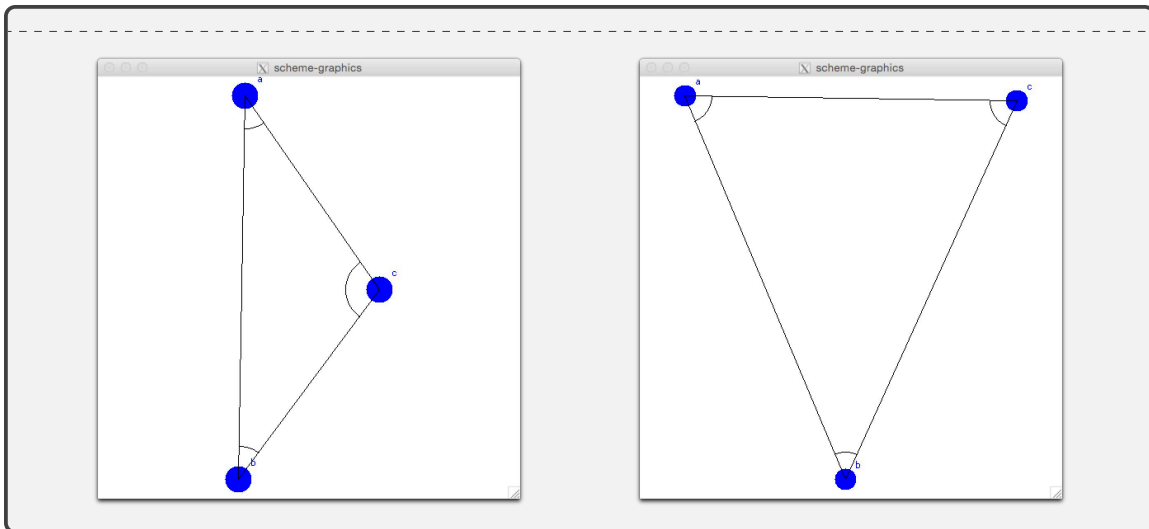
The values used in the propagation involves representing the partial information of where points and angles can be. A specified angle constrains a point to a ray and a specified length constrains a point to be on a circle. As information about a point is merged from several sources, intersecting these rays and circles yields unique solutions for where the points must exist. Although not as dynamic, these representations correspond to physically wiggling and extending the bars until they reach one another.

Interaction Example 3.13: Constraint Solving for 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)
```



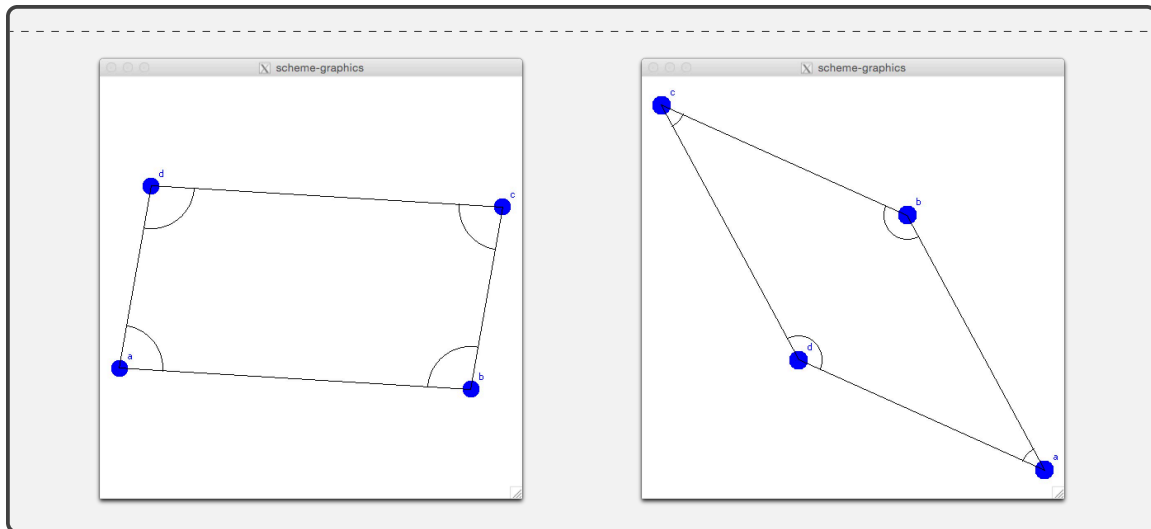
Example 3.14 continues our analysis of properties of the parallelogram. In this case, our constraint solver is able to build figures given the fact that its opposite angles are equal. The fact that these all happen to be parallelograms will be used by the learning module to produce a simpler definition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

Interaction Example 3.14: Constraint Solving for Parallelogram

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

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

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



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

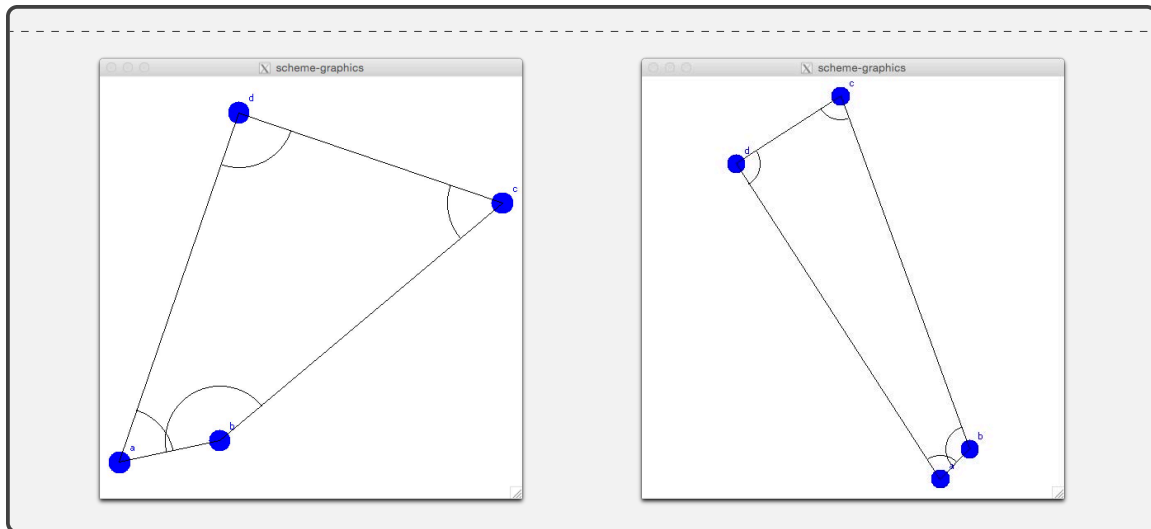
Code Example 3.15: Rectangle Constraints Example

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

Interaction Example 3.16: Solved Constraints

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

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



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

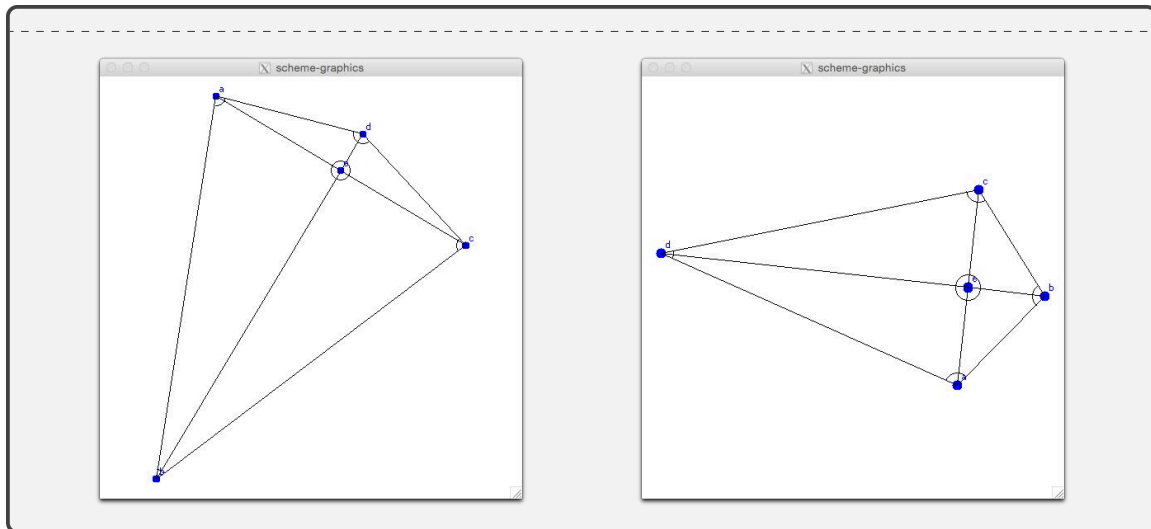
Code Example 3.17: More Involved Topologies for Constraint Solving

```
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))))

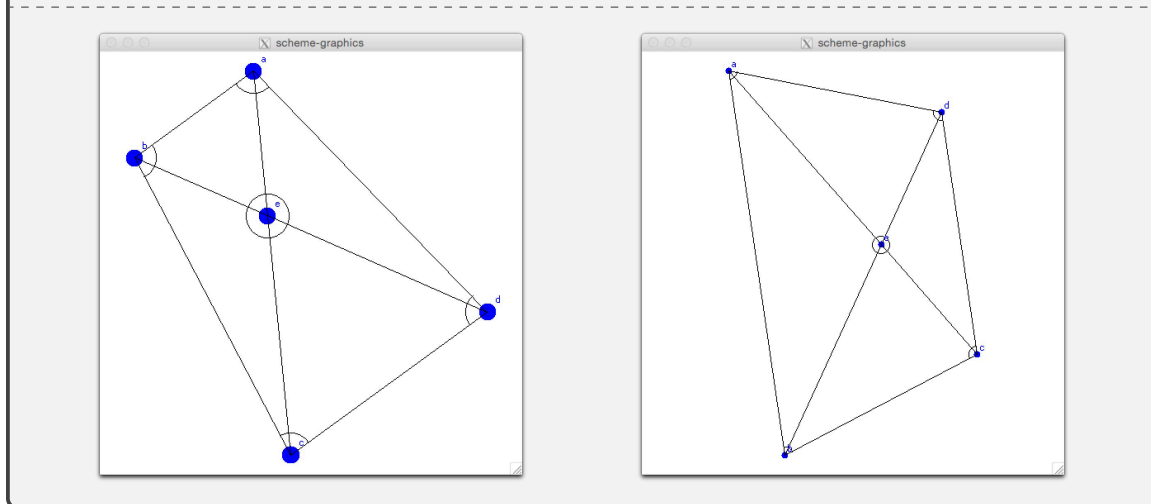
=> (m:run-mechanism kite-from-diagonals)
```



Interaction Example 3.19: Isoceles Trapezoids from Diagonals

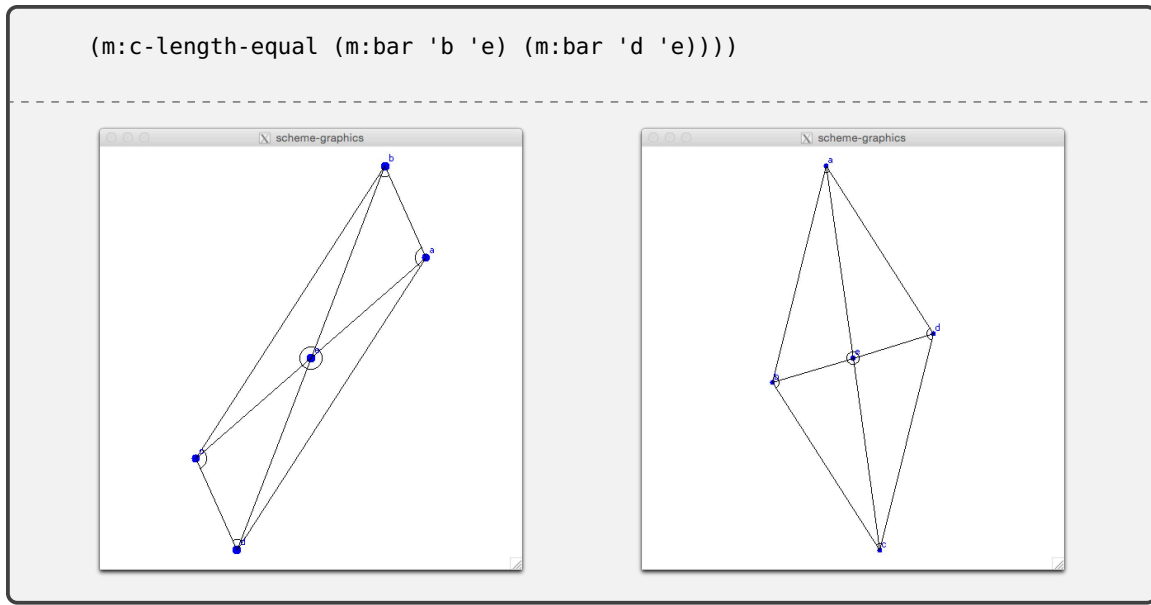
```
(define (isocetes-trapezoid-from-diagonals)
  (m:mechanism
    (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
    (m:c-length-equal (m:bar 'a 'e) (m:bar 'b 'e))
    (m:c-length-equal (m:bar 'c 'e) (m:bar 'd 'e))))
```

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



3.4 Learning Module

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

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

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

Interaction Example 3.21: Querying Terms

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

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

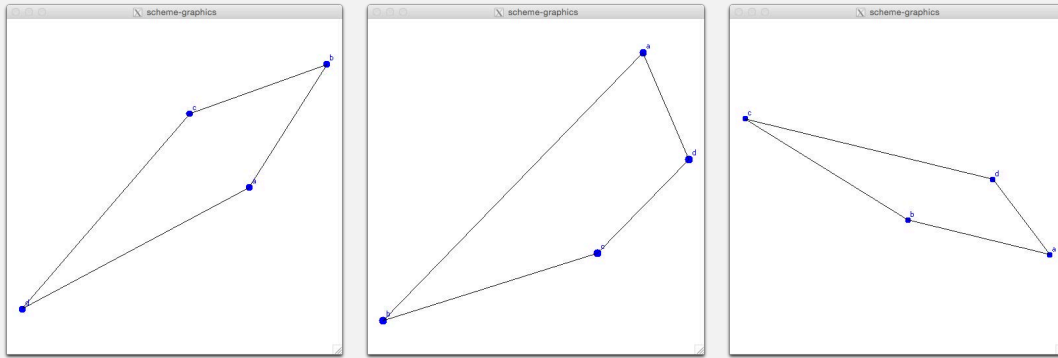
=> (what-is 'triangle)
```

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

A user can create an investigation to help the system learn a new definition by creating a procedure that creates random elements satisfying that definition. Example 3.22 shows the full range of trapezoids created via the `random-trapezoid` procedure.

Interaction Example 3.22: Random Figures

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



The learning module can interface with the perception module to obtain about the given element. In this case (3.23), we see the full dependencies of the elements under consideration instead of simply their names.

Interaction Example 3.23: Analyzing an Element

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

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

With these abilities, the system can be taught new definitions by providing a term ('pl for parallelogram) and a generator procedure that produces instances of that element. As shown in example 3.24, after being instructed to learn what a parallelogram

is from the `random-parallelogram` procedure, when queried for a definition, we're given the term, then the base definition of this element, then all properties known to be true of such objects.

Interaction Example 3.24: Learning Parallelogram Definition

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

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

To use such learned knowledge, we can use `is-a?` to test whether other elements are satisfy the definition of the term. As shown in example 3.25, results are correctly returned for any polygon that satisfies the observed properties. In cases where the properties are not satisfied, the system reports the failed conjectures or classifications (e.g. an 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 3.25: Testing Definitions

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

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

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

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

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

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

Learning individual definitions is nice, but cool properties arise when definitions build one another. When a new term is learned, the system checks other related terms for overlapping properties to determine where the new definition fits in the lattice of terms. In example 3.26, we see that after learning definitions of kites and rhombuses, the resulting definition of a rhombus is that it a parallelogram and kite that satisfies two additional properties. Later, after learning a rectangle, amazingly, the system shows us that the definition of a square is just a rhombus and rectangle with no additional properties.

Interaction Example 3.26: Building on Definitions

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

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

```

done

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

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

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

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

```

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

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

In the example 3.27, we see a trace of finding simple definitions for isoceles triangles and parallelograms. In the first example, the observed properties of an isoceles triangle are that its segments and angles are equal. Via the definitions simplification via constraint solving, we actually discover that the constraints of base angles equal or sides equal are sufficient.

Interaction Example 3.27: Learning Simple Definitions

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

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

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

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

In the parallelogram example 3.28, some subsets are omitted because the constraint solver wasn't able to solve a diagram given those constraints (to be improved / retried more gracefully in the future). However, the results still show some interesting valid definitions such as the pair of equal opposite angles as explored in Example 3.14 or equal length opposite sides and correctly mark several sets of invalid definitions as not being specific enough.

Interaction Example 3.28: Learning Simple Parallelogram Definitions

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

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

```

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

```

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

Chapter 4

System Overview

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

4.1 Goals

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

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

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

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

Definitions Definitions classify and differentiate an object from other objects. For instance “What is a rhombus?” yields the definition that it is a quadrilateral

(classification) with four equal sides (differentiation). For definitions, the system will attempt to simplify definition properties to more minimal sets, provide alternative formations, and use pre-existing definitions when possible: “A Square is a rhombus and a rectangle”

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

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

4.2 Diagram Representations

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

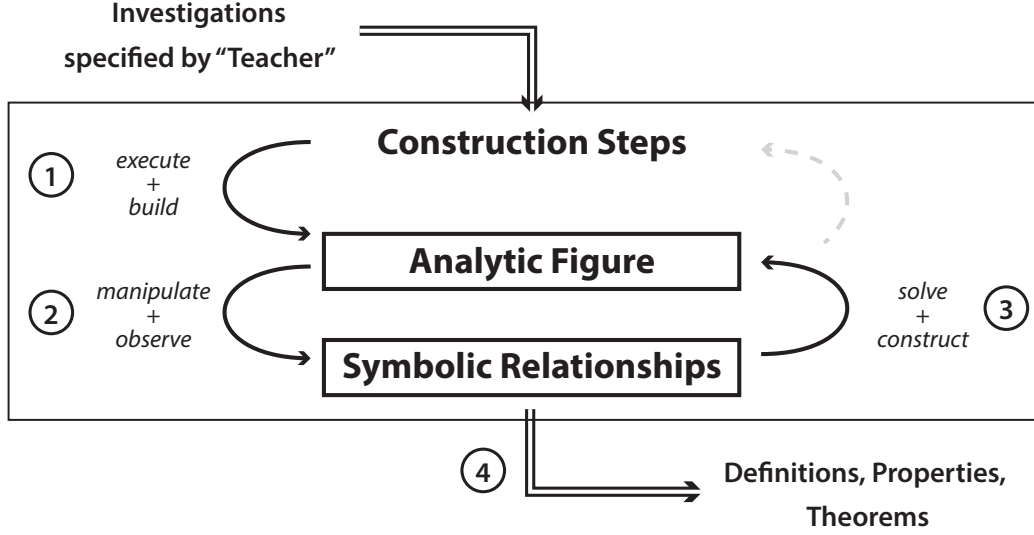
Construction Steps The main initial representation of most diagrams is a series of construction steps. These generally make up the input investigation from an external user trying to teach the system a concept. In some investigations, the actual construction steps are opaque to the system (as in a teacher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

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

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

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

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



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

4.2.1 Modules

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

4.3 Sample Interaction

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

4.3.1 Interpreting Construction Steps

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

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

4.3.2 Creating Figures

Given a language for expressing the constructions, the second phase of the system is to perform such constructions to yield an instance of the diagram. This process mimics “imagining” manipulations and results in an analytic representation of the figure with coordinates for each point. Arbitrary choices in the construction (“Let Q be some point not on the line.”) are chosen via a 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 Simplifying Definitions and Known Facts

4.3.5 Reporting Findings

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

4.4 Example Interaction

[For now see walkthrough in the “results” chapter. Will add a good, simple example here]

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 instances of figures.

I first discuss the basic structures and constructions that comprise figures, then describe the higher-order language used to specify construction steps.

5.2 Construction Interface

Code Example 5.1: Sample Construction

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

5.3 Basic Structures

The basic structures

Code Example 5.2: Basic Structures

```
1 (define-record-type <point>
2   (make-point x y)
3   point? "hi"
4   (x point-x)
5   (y point-y))
6
7 (define-record-type <segment>
8   (%segment p1 p2)
9   segment?
10  (p1 segment-endpoint-1)
11  (p2 segment-endpoint-2))
12
13 (define-record-type <line>
14   (%make-line point dir)
15   line?
16   (point line-point) ;; Point on the line
17   (dir line-direction))
```

Code Example 5.3: Angle and Circle Structures

```
1 (define-record-type <angle>
2   (make-angle dir1 vertex dir2)
3   angle?
4   (dir1 angle-arm-1)
5   (vertex angle-vertex)
6   (dir2 angle-arm-2))
7
8 (define-record-type <circle>
9   (make-circle center radius)
10  circle?
11  (center circle-center)
12  (radius circle-radius))
```

5.3.1 Points

Points form the basis of most elements. Throughout the system, points are labeled and used to identify other elements.

5.3.2 Linear Elements

The linear elements of Segments, Lines, and Rays are built upon points. Initially the internal representation of lines were that of two points, but to simplify manipulations,

To better specify angles (see below), all linear elements, including segments and lines are directioned. Thus, a line pointing. Predicates exist that compare lines for equality ignoring

5.3.3 Angles

Initially angles were represented as three points, now vertex + two directions. CCW orientation. Methods exist to determine them from various pairs of linear elements, uses directionality of linear elements to determine which “quadrant” of the angle is desired.

Given a figure, methods exist to extract angles from the diagrams in analysis rather than specifying each angle of interest while creating the diagram.

5.3.4 Math Support

Code Listing 5.4: Directions

```
1 (define (subtract-directions d2 d1)
2   (if (direction-equal? d1 d2)
3       0
4       (fix-angle-0-2pi (- (direction-theta d2)
5                             (direction-theta d1)))))
6
7 (define (direction-perpendicular? d1 d2)
8   (let ((difference (subtract-directions d1 d2)))
9     (or (close-enuf? difference (/ pi 2))
10        (close-enuf? difference (* 3 (/ pi 2)))))
```

Some “core” math structures to help these calculations: Direction represents a direction in $[0, 2\pi]$, fixes principal value $[0, 2\pi]$, and support various operations for direction intervals (basic intersection, adding, shifting, etc). Currently all represented by single theta value, could generalize via generics to dx, dy, or theta depending on

computation source.

5.4 Higher-level structures

In addition to the basic geometry structures, the system uses several grouping structures to combine and abstract the basic figure elements into higher-level figures elements.

For closure of combinators, all these higher level objects are also “Diagram objects”.

5.4.1 Polygons

Polygons are represented as groups of points.

5.4.2 Figures

Figures are currently groups of elements. In the creation of figures we extract additional information and build a graph out of adjacent components for use in the analysis stages.

5.5 Construction Operations

5.5.1 Traditional constructions

Code Listing 5.5: Perpendiculars

```
1 (define (perpendicular linear-element point)
2   (let* ((direction (->direction linear-element))
3         (rotated-direction (rotate-direction-90 direction)))
4     (make-line point rotated-direction)))
5
6 ;;; endpoint-1 is point, endpoint-2 is on linear-element
7 (define (perpendicular-to linear-element point)
8   (let ((pl (perpendicular linear-element point)))
9     (let ((i (intersect-linear-elements pl (->line linear-element))))
10      (make-segment point i))))
11
12 (define (angle-bisector a)
```

```

13  (let* ((d1 (angle-arm-1 a))
14         (d2 (angle-arm-2 a))
15         (vertex (angle-vertex a))
16         (radians (angle-measure a))
17         (half-angle (/ radians 2))
18         (new-direction (add-to-direction d2 half-angle)))
19    (make-ray vertex new-direction))

```

Midpoint, perpendicular line, bisectors

5.5.2 Intersections

Generic intersections, mathematically based at line/line or line/circle at the core. Other intersections also add the check that the resulting point(s) are on the elements.

5.5.3 Measurement-based operations

A “Ruler + Protractor” is generally not permitted in traditional construction problems. However, sometimes its nice to be able to use measurements to more quickly compute a result (e.g. angle bisector by halving angle) vs. going through the whole ray/circle based construction process.

5.5.4 Transformations

Currently, rotate about a point or translate by a vector. Also interfaces for by *random* point or vector.

5.6 Randomness

5.6.1 Random Choices

At the basis of all random

5.6.2 Remembering choices

5.6.3 Backtracking

Currently, the system does not backtrack based on random choices. However, there are plans to perform checks on randomly-generated elements that are too close to one another and to retry the random choice to avoid degenerate choices.

5.6.4 Avoiding almost-degenerate points

As discussed above, randomly making choices in

5.6.5 Animating choices

I animate over a small range within the specified random range. Top-level infrastructure determines frames, sleeping, etc. Constructions can request to animate functions of one arg $[0, 1]$. As the figure and animation is run, each call to randomize gets a call to random whenever their value is non-false.

5.7 Dependencies

5.7.1 Implementation

Eq-properties, etc.

5.7.2 Naming

Sometimes derived if unknown, figure out how name metadata relates to the dependencies.

5.7.3 Forcing higher-level random dependenceis

"Inverts" the dependency tree that would otherwise usually go down to points. set-dependency! as random-square. When given an element by the teacher, generally we

don't know how the construction was performed.

5.7.4 Dependency-less diagrams

In some cases, the dependency structure of a figure can be wiped.

5.8 Construction Language

Constructions and instruction-based investigations are specified by scheme procedures that return the desired figures.

5.8.1 Macros

I created a `let-geo*` special form that is similar to Scheme's `(let ...)` form, but sets the element names as specified so they can be more easily referred to later.

5.8.2 Multiple Assignment

In `let-geo*`, I also permit some constructions to optionally map to multiple assignments of names, such as the case in which you create a triangle and simultaneously want to store and name the triangle's vertex points.

5.9 Graphics

The system integrates with Scheme's graphics system for the X Window System to display the figures for the users. The graphical viewer can include labels and highlight specific elements, as well as display animations representing the “wiggling” of the diagram.

Code Listing 5.6: Drawing

```
1 (define (draw-figure figure canvas)
2   (set-coordinates-for-figure figure canvas)
3   (clear-canvas canvas))
```

```
4  (for-each
5    (lambda (element)
6      (canvas-set-color canvas (element-color element))
7      ((draw-element element) canvas))
8    (all-figure-elements figure))
9  (for-each
10   (lambda (element)
11     (canvas-set-color canvas (element-color element))
12     ((draw-label element) canvas))
13   (all-figure-elements figure))
14  (graphics-flush (canvas-g canvas)))
```

Chapter 6

Perception Module

6.1 Overview

Given a module that executes construction steps to build analytic figures, we need a way of “seeing” these figures in our mind’s eye. Thus, the perception module is primarily concerned with the task of examining the figure and observing interesting properties in figure.

Code Listing 6.1: Analyzer Routine

```
1 (define (analyze figure)
2   (let* ((points (figure-points figure))
3         (angles (figure-angles figure))
4         (linear-elements (figure-linear-elements figure))
5         (segments (figure-segments figure)))
6     (append
7       (extract-relationships points
8                             (list concurrent-points-relationship
9                                   concentric-relationship
10                                  concentric-with-center-relationship))
11      (extract-relationships segments
12                                (list equal-length-relationship))
13      (extract-relationships angles
14                                (list equal-angle-relationship
15                                      supplementary-angles-relationship
16                                      complementary-angles-relationship))
17      (extract-relationships linear-elements
18                                (list parallel-relationship
19                                      concurrent-relationship
20                                      perpendicular-relationship))))
```

Code Listing 6.2: (

```
1 define (extract-relationship elements relationship)
2   (map (lambda (tuple)
3     (make-observation relationship tuple))
4     (report-n-wise
5       (relationship-arity relationship)
6       (relationship-predicate relationship)
7       elements)))
8
9 (define (report-n-wise n predicate elements)
10  (let ((tuples (all-n-tuples n elements)))
11    (filter (nary-predicate n predicate) tuples)))
```

6.1.1 Extracting segments and angles

The observation module also builds and traverses a graph-representation of the object of connectedness and adjacencies to extract more segments and angles, or include intersections of elements in its investigation.

Auxillary Segments

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

6.1.2 What is Interesting?

Concurrent points, collinear points, equal angles, supplementary/complementary angles, parallel, perpendicular elements, concentric points, (future:) ratios between measurements, etc.

Code Listing 6.3: Relationships

```
1 (define-record-type <relationship>
2   (%make-relationship type arity predicate)
3   relationship?)
```

```

4  (type relationship-type)
5  (arity relationship-arity)
6  (predicate relationship-predicate))
7
8  (define equal-length-relationship
9    (%make-relationship 'equal-length 2 segment-equal-length?))
10
11 (define concurrent-relationship
12   (%make-relationship 'concurrent 3 concurrent?))

```

6.1.3 Removing Obvious Properties

This module makes use of available dependency information to eliminate some obvious properties. At this phase, the eliminations arise only from basic geometry knowledge “hard-coded” into the system, and not upon any specific prior-learned formula.

Trivial relations

Points being on lines, segments, circles directly dependent on that point.

Branch Relations

Other examples include “branch” relations. [REF: Chen, Song, etc.]. ABCD on a line with $AB = CD$ also means that $AC = BD$, for instance.

6.2 Representations

A “Relationship” object represents a type of relationship, a “Observation” object refers to a specific observation seen in a figure.

Chapter 7

Declarative Geometry Constraint Solver

7.1 Overview

The final module is a declarative geometric constraint solver. Given a user-specified topology of the diagram and various constraints, this system is able to solve those constraints and instantiate a diagram that satisfies them if possible.

This system is implemented using propagators, involved the creation of new partial information about point regions and direction intervals, and focuses on a

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

7.2 Mechanical Analogies

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

7.3 Example

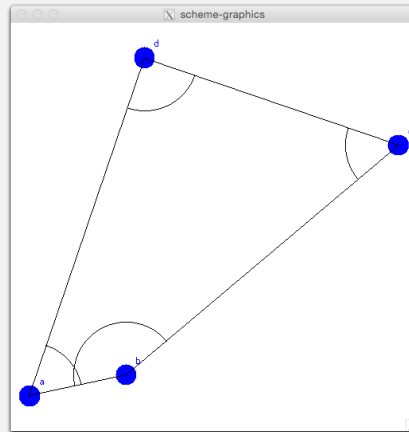
Code Example 7.1: Rectangle Constraints Example

```
1 {images/rect-demo-2.png}
2 (define (is-this-a-rectangle-2)
3   (m:mechanism
4     (m:establish-polygon-topology 'a 'b 'c 'd)
5     (m:c-length-equal (m:bar 'a 'd)
6                       (m:bar 'b 'c))
7     (m:c-right-angle (m:joint 'd))
8     (m:c-angle-equal (m:joint 'a)
9                     (m:joint 'c))))
```

Interaction Example 7.2: Solved Constraints

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

```
(m:run-mechanism rect-demo-2)
(specifying-bar m:bar:d:a .6742252545577186)
(initializing-direction m:bar:d:a-dir (direction 4.382829365403101))
(initializing-point m:bar:d:a-p1 (0 0))
(specifying-joint m:joint:c:b:a 2.65583669872538)
```



7.3.1 Bar and Joint Linkages

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

7.3.2 Mechanism

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

Code Listing 7.3: (

```
1 define (m:build-mechanism m)
2   (m:identify-vertices m)
3   (m:assemble-linkages (m:mechanism-bars m)
4                         (m:mechanism-joints m))
5   (m:apply-mechanism-constraints m)
6   (m:apply-slices m)
7
8 (define (m:solve-mechanism m)
9   (m:initialize-solve)
10  (let lp ()
11    (run)
12    (cond ((m:mechanism-contradictory? m)
13          (m:draw-mechanism m c)
14          #f)
15          ((not (m:mechanism-fully-specified? m))
16           (if (m:specify-something m)
17                (lp)
18                (error "Couldn't find anything to specify.")))
19          (else 'mechanism-built))))
```

7.4 Partial Information

7.4.1 Regions

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

7.4.2 Direction Intervals

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

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

7.5 Propagator Constraints

System uses propagators to solve these mechanism constraints.

7.5.1 Basic Linkage Constraints

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

Code Listing 7.4: Basic Bar Constraints

```
1 (define (m:make-bar bar-id)
2   (let ((bar-key (m:make-bar-name-key bar-id)))
3     (let ((p1 (m:make-point (symbol bar-key '-p1)))
4           (p2 (m:make-point (symbol bar-key '-p2))))
5       (name! p1 (symbol bar-key '-p1))
6       (name! p2 (symbol bar-key '-p2))
7       (let ((v (m:make-vec bar-key)))
8         (c:+ (m:point-x p1)
9              (m:vec-dx v)
10             (m:point-x p2))
11         (c:+ (m:point-y p1)
12              (m:vec-dy v)
13             (m:point-y p2))
14         (let ((bar (%m:make-bar p1 p2 v)))
15           (m:p1->p2-bar-propagator p1 p2 bar)
16           (m:p2->p1-bar-propagator p2 p1 bar)
17           bar))))))
```

7.5.2 Higher Order Constraints

Code Listing 7.5: User Constraints

```
1 (define-record-type <m:constraint>
2   (m:make-constraint type args constraint-procedure)
3   m:constraint?
4   (type m:constraint-type)
5   (args m:constraint-args)
6   (constraint-procedure m:constraint-procedure))
```

```

7
8 (define (m:c-length-equal bar-id-1 bar-id-2)
9   (m:make-constraint
10    'm:c-length-equal
11    (list bar-id-1 bar-id-2)
12    (lambda (m)
13      (let ((bar-1 (m:lookup m bar-id-1))
14            (bar-2 (m:lookup m bar-id-2)))
15        (c:id
16         (m:bar-length bar-1)
17         (m:bar-length bar-2))))))

```

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

7.6 Solving: Specification Ordering

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

7.6.1 Anchored vs. Specified vs. Initialized

7.7 Backtracking

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

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

7.8 Interfacing with existing diagrams

Converts between figures and symbolic relationships.

7.9 Specification Interface

Establish Polygon Topology

Nice techniques for establishing polygon topology.

Code Listing 7.6: Establishing Topology

```
1 (define (m:establish-polygon-topology . point-names)
2   (if (< (length point-names) 3)
3     (error "Min polygon size: 3"))
4   (let ((extended-point-names
5         (append point-names
6                 (list (car point-names) (cadr point-names)))))
7     (let ((bars
8           (map (lambda (p1-name p2-name)
9                 (m:make-named-bar p1-name p2-name))
10              point-names
11              (cdr extended-point-names)))
12          (joints
13            (map (lambda (p1-name vertex-name p2-name)
14                  (m:make-named-joint p1-name vertex-name p2-name))
15                 (cddr extended-point-names)
16                 (cdr extended-point-names)
17                 point-names)))
18      (append bars joints
19              (list (m:polygon-sum-slice
20                    (map m:joint-name joints))))))
```

Chapter 8

Learning Module

8.1 Overview

The Learning and learning module is one the core elements integrating information from the other components of the system. It maintains

Code Example 8.1: Using Definitions

```
1 (define (what-is term)
2   (pprint (lookup term)))
3
4 (define (is-a? term obj)
5   (let ((def (lookup term)))
6     (if (unknown? def)
7         `(.term unknown)
8         ((definition-predicate def) obj))))
9
10 (define (show-me term)
11   (let ((def (lookup term)))
12     (if (unknown? def)
13         `(.term unknown)
14         (show-element ((definition-generator def))))))
15
16 (define (examine object)
17   (show-element object)
18   (let ((applicable-terms
19         (filter (lambda (term)
20                   (is-a? term object))
21                 (all-known-terms))))
22     applicable-terms))
```

8.2 Interactions with Learning Module

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

8.2.1 Querying

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

```
(what-is 'rhombus)
```

8.2.2 Learning Definitions

Code Listing 8.2: Learning a new term

```
1 (define (learn-term term object-generator)
2   (let ((v (lookup term)))
3     (if (not (eq? v 'unknown))
4       (pprint `(already-known ,term))
5       (let ((example (name-polygon (object-generator))))
6         (let* ((base-terms (examine example))
7                (simple-base-terms (simplify-base-terms base-terms))
8                (base-definitions (map lookup base-terms))
9                (base-conjectures (flatten (map definition-conjectures
10                                                base-definitions)))
11                (fig (figure (with-dependency '<premise> example)))
12                (observations (analyze-figure fig))
13                (conjectures (map conjecture-from-observation observations))
14                (simplified-conjectures
15                 (simplify-conjectures conjectures base-conjectures)))
16           (pprint conjectures)
17           (let ((new-def
18                  (make-restrictions-definition
19                   term
20                   simple-base-terms
```

```

21         simplified-conjectures
22         object-generator)))
23     (add-definition! *current-student* new-def)
24     'done))))))

```

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 8.3: Building Predicates for Definitions

```

1 (define (build-predicate-for-definition s def)
2   (let ((classifications (definition-classifications def))
3         (conjectures (definition-conjectures def)))
4     (let ((classification-predicate
5           (lambda (obj)
6             (every
7              (lambda (classification)
8                (or ((definition-predicate (student-lookup s classification))
9                    obj)
10                 (begin (if *explain*
11                           (pprint `(failed-classification
12                                   ,classification)))
13                          #f)))
14              classifications))))
15       (lambda args
16         (and (apply classification-predicate args)
17              (every (lambda (o) (satisfies-conjecture o args))
18                     conjectures)))))

```

8.2.3 Applying Learned Properties

To apply the learned conjectures, the learning system can use its repository of geometry knowledge to tell you what a given figure is or to point out old vs. new properties seen in a specific figure:

8.2.4 Simplifying Definitions

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

Code Listing 8.4: Simplifying Definitions

```
1 (define (get-simple-definitions term)
2   (let ((def (lookup term)))
3     (if (unknown? def)
4         (error "Unknown term" term)
5         (let* ((object ((definition-generator def)))
6                 (observations
7                  (filter
8                   observation->constraint
9                   (all-observations
10                    (figure (name-polygon object))))))
11           (map
12            (lambda (obs-subset)
13              (pprint obs-subset)
14              (let* ((topology (topology-for-object object))
15                     (new-figure
16                      (observations->figure topology obs-subset)))
17                (if new-figure
18                    (let ((new-polygon
19                           (polygon-from-figure new-figure)))
20                      (pprint new-polygon)
21                      (if (is-a? term new-polygon)
22                          (list 'valid-definition
23                               obs-subset)
24                          (list 'invalid-definition
25                               obs-subset)))
26                    (list 'unknown-definition
27                           obs-subset))))
28            (all-subsets observations))))))
```

Code Listing 8.5: Converting Observations to a Figure

```
1 (define (observations->figure topology observations)
2   (initialize-scheduler)
3   (pprint (observations->constraints observations))
4   (let ((m (apply
5             m:mechanism
6             (list
7              topology
8              (observations->constraints observations))))
9       (m:build-mechanism m)
10      (if (not (m:solve-mechanism m))
```



```

11      (begin
12        (pp "Could not solve mechanism")
13        #f)
14      (let ((f (m:mechanism->figure m)))
15        (pp "Solved!")
16        (show-figure f)
17        f))))

```

8.2.5 Performing Investigations

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

8.3 Representing Discoveries

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

8.3.1 Placement of discoveries

Given this lattice structure, an interesting question when exploring new properties

8.3.2 Ordering of discoveries

An issue with this system is that often discoveries can be in slightly different formats. As such, for each relationship, we establish a consistent ordering of elements and use pattern matching to

For example, assertions about equality of segments $|AB| = |CD|$ are independent of the ordering of points within the elements.

8.3.3 Pattern Matching against existing conjectures

Based on dependencies, we replace the lowest-level random dependencies with arbitrary pattern elements (`? s1 ,segment`) for instance. Then, when new conjectures are being considered, we attempt to pattern match based on existing elements to see if there is a redundant observation. TODO.

Chapter 9

Related Work

[Need to update with a few more references I've found, and some relating to the mechanical simulation aspects]

The topics of automating geometric proofs and working with diagrams are areas of active research. Several examples of related work can be found in the proceedings of annual conferences such as *Automated Deduction in Geometry* [?] and *Diagrammatic Representation and Inference* [?]. 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 [?], and a paper by Seo and Hajishirzi describes using textual descriptions of problems to improve recognition of their accompanying figures [?].

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

9.1 Dynamic Geometry

From an education perspective, there are several texts that emphasize an investigative, conjecture-based approach to teaching such as *Discovering Geometry* by Michael Serra [?], the text I used to learn geometry. Some researchers praise these investiga-

tive methods [?] while others question whether it appropriately encourages deductive reasoning skills [?].

9.2 Software

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

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

9.3 Automated Proof and Discovery

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

Chapter 10

Conclusion

10.1 Overview

To be concluded . . .

10.2 Extensions

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

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

Appendix A

Code Listings

This chapter contains code listings. In addition to the code here, propagator system from GJS... cite other sources form GJS / scmutils (ghelper, eq-properties)...

Listings

A.1	load.scm	83
A.2	main.scm	84
A.3	figure/load.scm	87
A.4	figure/core.scm	88
A.5	figure/line.scm	89
A.6	figure/direction.scm	93
A.7	figure/vec.scm	95
A.8	figure/measurements.scm	97
A.9	figure/angle.scm	98
A.10	figure/bounds.scm	101
A.11	figure/circle.scm	104
A.12	figure/point.scm	105
A.13	figure/constructions.scm	106
A.14	figure/intersections.scm	109
A.15	figure/figure.scm	112
A.16	figure/math-utils.scm	113
A.17	figure/polygon.scm	114
A.18	figure/metadata.scm	117
A.19	figure/dependencies.scm	118
A.20	figure/randomness.scm	120
A.21	figure/transforms.scm	126
A.22	perception/load.scm	128
A.23	perception/observation.scm	129

A.24 perception/analyzer.scm	130
A.25 graphics/load.scm	134
A.26 graphics/appearance.scm	135
A.27 graphics/graphics.scm	136
A.28 manipulate/load.scm	139
A.29 manipulate/linkages.scm	140
A.30 manipulate/region.scm	154
A.31 manipulate/constraints.scm	160
A.32 manipulate/topology.scm	163
A.33 manipulate/mechanism.scm	164
A.34 manipulate/main.scm	169
A.35 learning/load.scm	173
A.36 learning/core-knowledge.scm	174
A.37 learning/definitions.scm	175
A.38 learning/conjecture.scm	176
A.39 learning/simplifier.scm	177
A.40 learning/student.scm	179
A.41 learning/walkthrough.scm	183
A.42 content/load.scm	185
A.43 content/thesis-demos.scm	186
A.44 core/load.scm	188
A.45 core/animation.scm	189
A.46 core/macros.scm	191
A.47 core/print.scm	193
A.48 core/utils.scm	194
A.49 lib/eq-properties.scm	197
A.50 lib/ghelper.scm	199

Listing A.1: load.scm

```

1  ;;; load.scm -- Load the system
2
3  ;;; Code:
4
5  ;;; Utilities ;;;;;;;;;;;;;;;;;;
6
7  (define (reset)
8    (ignore-errors (lambda () (close))))
9    (ge (make-top-level-environment))
10   (load "load"))
11
12 (define (load-module subdirectory)
13   (let ((cur-pwd (pwd)))
14     (cd subdirectory)
15     (load "load")
16     (cd cur-pwd)))
17
18 ;;; Load Modules ;;;;;;;;;;;;;;;;;;
19
20 (for-each (lambda (m) (load-module m))
21           '("lib"
22             "core"
23             "figure"
24             "graphics"
25             "manipulate"
26             "perception"
27             "learning"
28             "content"))
29 (load "main")
30
31 ;;; Initialize ;;;;;;;;;;;;;;;;;;
32
33 (set! *random-state* (fasload "a-random-state"))
34 (initialize-scheduler)
35 (initialize-student)
36
37 'done-loading

```

Listing A.2: main.scm

```

1 (define (i-t-figure)
2   (let-geo* (((t (a b c)) (random-isocetes-triangle)))
3     (figure t)))
4
5
6 (define (midpoint-figure)
7   (let-geo* (((s (a b)) (random-segment))
8     (m (segment-midpoint s)))
9     (figure s m)))
10
11 (define (random-rhombus-figure)
12   (let-geo* (((r (a b c d)) (random-rhombus)))
13     (figure r)))
14
15 ;;; Other Examples:
16
17 (define (debug-figure)
18   (let-geo* (((r (a b c d)) (random-parallelogram))
19     (m1 (midpoint a b))
20     (m2 (midpoint c d)))
21     (figure r m1 m2 (make-segment m1 m2)))
22
23 (define (demo-figure)
24   (let-geo* (((t (a b c)) (random-isocetes-triangle))
25     (d (midpoint a b))
26     (e (midpoint a c))
27     (f (midpoint b c))
28
29     (l1 (perpendicular (line-from-points a b) d))
30     (l2 (perpendicular (line-from-points a c) e))
31     (l3 (perpendicular (line-from-points b c) f))
32
33     (i1 (intersect-lines l1 l2))
34     (i2 (intersect-lines l1 l3))
35
36     (cir (circle-from-points i1 a)))
37
38   (figure
39     (make-segment a b)
40     (make-segment b c)
41     (make-segment a c)
42     a b c l1 l2 l3 cir
43     i1 i2))
44
45 (define (circle-line-intersect-test)
46   (let-geo* ((cir (random-circle))
47     ((rad (a b)) (random-circle-radius cir))
48     (p (random-point-on-segment rad))
49     (l (random-line-through-point p))
50     (cd (intersect-circle-line cir l))
51     (c (car cd))
52     (d (cadr cd)))
53     (figure cir rad p l c d))
54
55 (define (circle-test)
56   (let-geo* ((a (random-point))
57     (b (random-point))
58     (d (distance a b))
59     (r (rand-range
60       (* d 0.5)
61       (* d 1)))
62     (c1 (make-circle a r))
63     (c2 (make-circle b r))
64     (cd (intersect-circles c1 c2))
65     (c (car cd))
66     (d (cadr cd)))

```

```

67     (figure (polygon-from-points a c b d)))
68
69 (define (line-test)
70   (let-geo* ((a (random-point))
71             (b (random-point))
72             (c (random-point))
73             (d (random-point))
74             (l1 (line-from-points a b))
75             (l2 (line-from-points c d))
76             (e (intersect-lines l1 l2))
77             (f (random-point-on-line l1))
78             (cir (circle-from-points e f)))
79     (figure a b c d l1 l2 e f cir)))
80
81 (define (incircle-circumcircle)
82   (let-geo* (((t (a b c)) (random-triangle))
83             (((a-1 a-2 a-3)) (polygon-angles t))
84             (ab1 (angle-bisector a-1))
85             (ab2 (angle-bisector a-2))
86             ((radius-segment (center-point radius-point))
87              (perpendicular-to (make-segment a b)
88                                (intersect-linear-elements ab1 ab2)))
89             (incircle (circle-from-points
90                        center-point
91                        radius-point))
92             (pb1 (perpendicular-bisector
93                  (make-segment a b)))
94             (pb2 (perpendicular-bisector
95                  (make-segment b c)))
96             (pb-center (intersect-lines pb1 pb2))
97             (circum-cir (circle-from-points
98                          pb-center
99                          a)))
100   (figure t a-1 a-2 a-3
101         pb-center
102         radius-segment
103         incircle
104         circum-cir)))
105
106 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
107 ;; Run commands
108
109 (define current-figure demo-figure)
110
111 (define c
112   (if (environment-bound? (the-environment) 'c)
113       c
114       (canvas)))
115
116 (define (close)
117   (ignore-errors (lambda () (graphics-close (canvas-g c)))))
118
119 (define *num-inner-loop* 5)
120 (define *num-outer-loop* 5)
121
122
123 (define (run-figure current-figure-proc)
124   (let ((analysis-data (make-analysis-collector)))
125     (run-animation
126      (lambda ()
127        (let ((current-figure (current-figure-proc)))
128          (draw-figure current-figure c)
129          (let ((analysis-results (analyze-figure current-figure)))
130            (save-results (print analysis-results) analysis-data))
131          ))))
132   (display "--- Results ---\n")
133   (print-analysis-results analysis-data)))
134

```

```
135 (define interesting-figures
136   (list
137     debug-figure
138     parallel-lines-converse
139     perpendicular-bisector-equidistant
140     perpendicular-bisector-converse
141     demo-figure
142     linear-pair
143     vertical-angles
144     corresponding-angles
145     cyclic-quadrilateral))
146
147 (define (r)
148   (for-each (lambda (figure)
149               (run-figure figure))
150             interesting-figures)
151   'done)
152
153 ;(r)
```

Listing A.3: figure/load.scm

```
1 ;;; load.scm -- Load figure
2 (for-each (lambda (f) (load f))
3           '("core"
4             "metadata"
5             "line"
6             "direction"
7             "direction-interval"
8             "vec"
9             "measurements"
10            "angle"
11            "bounds"
12            "circle"
13            "point"
14            "constructions"
15            "intersections"
16            "figure"
17            "math-utils"
18            "polygon"
19            "dependencies"
20            "randomness"
21            "transforms"))
```

Listing A.4: figure/core.scm

```

1  ;; core.scm --- Core definitions used throughout the figure elements
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Some generic handlers used in figure elements
7
8  ;; Future:
9  ;; - figure-element?, e.g.
10
11  ;; Code:
12
13  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Element Component ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
14
15  (define element-component
16    (make-generic-operation
17      2 'element-component
18      (lambda (el i)
19        (error "No component procedure for element" el))))
20
21  (define (component-procedure-from-getters . getters)
22    (let ((num-getters (length getters)))
23      (lambda (el i)
24        (if (not (<= 0 i (- num-getters 1)))
25            (error "Index out of range for component procedure: " i))
26            ((list-ref getters i)
27              el))))
28
29  (define (declare-element-component-handler handler type)
30    (defhandler element-component handler type number?))
31
32  (declare-element-component-handler list-ref list?)
33
34  #|
35  Example Usage:
36
37  (declare-element-component-handler
38    (component-procedure-from-getters car cdr)
39    pair?)
40
41  (declare-element-component-handler vector-ref vector?)
42
43  (element-component '(3 . 4) 1)
44  ;Value: 4
45
46  (element-component #(1 2 3) 2)
47  ;Value: 3
48 |#

```


Listing A.5: figure/line.scm

```

1  ;; line.scm --- Line
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Linear Elements: Segments, Lines, Rays
7  ;; - All have direction
8  ;; - Conversions to directions, extending.
9  ;; - Lines are point + direction, but hard to access point
10 ;; - Means to override dependencies for random segments
11
12 ;; Future:
13 ;; - Simplify direction requirements
14 ;; - Improve some predicates, more tests
15 ;; - Fill out more dependency information
16
17 ;; Code:
18
19 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Segments ;;;;;;;;;;;;;;;;;;
20
21 (define-record-type <segment>
22   (%segment p1 p2)
23   segment?
24   (p1 segment-endpoint-1)
25   (p2 segment-endpoint-2))
26
27 (define (set-segment-dependency! segment dependency)
28   (set-dependency! segment dependency)
29   (set-dependency!
30     (segment-endpoint-1 segment)
31     `(segment-endpoint-1 segment))
32   (set-dependency!
33     (segment-endpoint-2 segment)
34     `(segment-endpoint-2 segment)))
35
36 (defhandler print
37   element-name
38   segment?)
39
40 ;; Alternate, helper constructors
41
42 (define (make-segment p1 p2)
43   (let ((seg (%segment p1 p2)))
44     (with-dependency
45       `(segment ,p1 ,p2)
46       seg)))
47
48 (define (make-auxiliary-segment p1 p2)
49   (with-dependency
50     `(aux-segment ,p1 ,p2)
51     (make-segment p1 p2)))
52
53 (declare-element-component-handler
54   (component-procedure-from-getters segment-endpoint-1
55                                       segment-endpoint-2)
56   segment?)
57
58 (defhandler generic-element-name
59   (lambda (seg)
60     `(segment ,(element-name (segment-endpoint-1 seg))
61               ,(element-name (segment-endpoint-2 seg))))
62   segment?)
63
64 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Lines ;;;;;;;;;;;;;;;;;;
65
66 (define-record-type <line>

```

```

67  (%make-line point dir)
68  line?
69  (point line-point) ;; Point on the line
70  (dir line-direction)
71
72  (defhandler print
73    element-name
74    line?)
75
76  (define make-line %make-line)
77
78  (define (line-from-points p1 p2)
79    (make-line p1 (direction-from-points p1 p2)))
80
81  (define (line-from-point-direction p dir)
82    (make-line p dir))
83
84  ;;; TODO, use for equality tests?
85  (define (line-offset line)
86    (let ((direction (direction-from-points p1 p2))
87          (x1 (point-x p1))
88          (y1 (point-y p1))
89          (x2 (point-x p2))
90          (y2 (point-y p2)))
91      (let ((offset (/ (- (* x2 y1)
92                          (* y2 x1))
93                      (distance p1 p2))))
94        (%make-line direction offset))))
95
96  ;;; TODO: Figure out dependencies for these
97  (define (two-points-on-line line)
98    (let ((point-1 (line-point line))
99          (let ((point-2 (add-to-point
100                        point-1
101                        (unit-vec-from-direction (line-direction line))))
102          (list point-1 point-2))))
103
104  (define (line-p1 line)
105    (car (two-points-on-line line)))
106
107  (define (line-p2 line)
108    (cadr (two-points-on-line line)))
109
110
111  ;;; Rays ;;;
112
113  (define-record-type <ray>
114    (make-ray initial-point direction)
115    ray?
116    (initial-point ray-endpoint)
117    (direction ray-direction))
118
119  (define (ray-from-point-direction p dir)
120    (make-ray p dir))
121
122  (define (ray-from-points endpoint p1)
123    (make-ray endpoint (direction-from-points endpoint p1)))
124
125  (define (shorten-ray-from-point r p)
126    (if (not (on-ray? p r))
127        (error "Can only shorten rays from points on the ray"))
128    (ray-from-point-direction p (ray-direction r)))
129
130  ;;; Constructors from angles ;;;
131
132  (define (ray-from-arm-1 a)
133    (let ((v (angle-vertex a))
134          (dir (angle-arm-1 a)))

```

```

135     (make-ray v dir)))
136
137 (define (ray-from-arm-2 a)
138   (ray-from-arm-1 (reverse-angle a)))
139
140 (define (line-from-arm-1 a)
141   (ray->line (ray-from-arm-1 a)))
142
143 (define (line-from-arm-2 a)
144   (ray->line (ray-from-arm-2 a)))
145
146 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Transforms ;;;;;;;;;;;;;;;;;
147
148 (define flip (make-generic-operation 1 'flip))
149
150 (define (flip-line line)
151   (make-line
152     (line-point line)
153     (reverse-direction (line-direction line))))
154 (defhandler flip flip-line line?)
155
156 (define (flip-segment s)
157   (make-segment (segment-endpoint-2 s) (segment-endpoint-1 s)))
158 (defhandler flip flip-segment segment?)
159
160 (define (reverse-ray r)
161   (make-ray (ray-endpoint r)
162     (reverse-direction (ray-direction r))))
163
164 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Operations ;;;;;;;;;;;;;;;;;
165
166 (define (segment-length seg)
167   (distance (segment-endpoint-1 seg)
168     (segment-endpoint-2 seg)))
169
170 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;;;;
171
172 (define (linear-element? x)
173   (or (line? x)
174     (segment? x)
175     (ray? x)))
176
177 (define (parallel? a b)
178   (direction-parallel? (->direction a)
179     (->direction b)))
180
181 (define (perpendicular? a b)
182   (direction-perpendicular? (->direction a)
183     (->direction b)))
184
185 (define (segment-equal? s1 s2)
186   (and
187     (point-equal? (segment-endpoint-1 s1)
188       (segment-endpoint-1 s2))
189     (point-equal? (segment-endpoint-2 s1)
190       (segment-endpoint-2 s2))))
191
192 (define (segment-equal-ignore-direction? s1 s2)
193   (or (segment-equal? s1 s2)
194     (segment-equal? s1 (flip-segment s2))))
195
196 (define (segment-equal-length? seg-1 seg-2)
197   (close-enuf? (segment-length seg-1)
198     (segment-length seg-2)))
199
200 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Conversions ;;;;;;;;;;;;;;;;;
201
202 ;; Ray shares point p1

```

```

203 (define (segment->ray segment)
204   (make-ray (segment-endpoint-1 segment)
205             (direction-from-points
206               (segment-endpoint-1 segment)
207               (segment-endpoint-2 segment))))
208
209 (define (ray->line ray)
210   (make-line (ray-endpoint ray)
211              (ray-direction ray)))
212
213 (define (segment->line segment)
214   (ray->line (segment->ray segment)))
215
216 (define (line->direction l)
217   (line-direction l))
218
219 (define (ray->direction r)
220   (ray-direction r))
221
222 (define (segment->direction s)
223   (direction-from-points
224     (segment-endpoint-1 s)
225     (segment-endpoint-2 s)))
226
227 (define (segment->vec s)
228   (sub-points
229     (segment-endpoint-2 s)
230     (segment-endpoint-1 s)))
231
232 (define ->direction (make-generic-operation 1 '->direction))
233 (defhandler ->direction line->direction line?)
234 (defhandler ->direction ray->direction ray?)
235 (defhandler ->direction segment->direction segment?)
236
237 (define ->line (make-generic-operation 1 '->line))
238 (defhandler ->line identity line?)
239 (defhandler ->line segment->line segment?)
240 (defhandler ->line ray->line ray?)

```

Listing A.6: figure/direction.scm

```

1  ;; direction.scm --- Low-level direction structure
2
3  ;; Commentary:
4
5  ;; A Direction is equivalent to a unit vector pointing in some direction.
6
7  ;; Ideas:
8  ;; - Ensures range [0, 2pi]
9
10 ;; Future:
11 ;; - Could generalize to dx, dy or theta
12
13 ;; Code:
14
15 ;;;;;;;;;;;;;; Direction Structure ;;;;;;;;;;;;;;
16
17 (define-record-type <direction>
18   (%direction theta)
19   direction?
20   (theta direction-theta))
21
22 (define (make-direction theta)
23   (%direction (fix-angle-0-2pi theta)))
24
25 (define (print-direction dir)
26   `(direction ,(direction-theta dir)))
27 (defhandler print print-direction direction?)
28
29 ;;;;;;;;;;;;;; Arithmetic ;;;;;;;;;;;;;;
30
31 (define (add-to-direction dir radians)
32   (make-direction (+ (direction-theta dir)
33                      radians)))
34 ;; D2 - D1
35 (define (subtract-directions d2 d1)
36   (if (direction-equal? d1 d2)
37       0
38       (fix-angle-0-2pi (- (direction-theta d2)
39                           (direction-theta d1)))))
40
41 ;;;;;;;;;;;;;; Operations ;;;;;;;;;;;;;;
42
43 ;; CCW
44 (define (rotate-direction-90 dir)
45   (add-to-direction dir (/ pi 2)))
46
47 (define (reverse-direction dir)
48   (add-to-direction dir pi))
49
50 ;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;
51
52 (define (direction-equal? d1 d2)
53   (or (close-enuf? (direction-theta d1)
54                    (direction-theta d2))
55       (close-enuf? (direction-theta (reverse-direction d1))
56                    (direction-theta (reverse-direction d2)))))
57
58 (define (direction-opposite? d1 d2)
59   (close-enuf? (direction-theta d1)
60                (direction-theta (reverse-direction d2))))
61
62 (define (direction-perpendicular? d1 d2)
63   (let ((difference (subtract-directions d1 d2)))
64     (or (close-enuf? difference (/ pi 2))
65         (close-enuf? difference (* 3 (/ pi 2))))))
65
66

```

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

Listing A.7: figure/vec.scm

```

1  ;;; vec.scm --- Low-level vector structures
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Simplifies lots of computation, cartesian coordinates
7  ;; - Currently 2D, could extend
8
9  ;; Future:
10 ;; - Could generalize to allow for polar vs. cartesian vectors
11
12 ;;; Code:
13
14 ;;; Vector Structure ;;;
15
16 (define-record-type <vec>
17   (make-vec dx dy)
18   vec?
19   (dx vec-x)
20   (dy vec-y))
21
22 ;;; Transformations of Vectors
23 (define (vec-magnitude v)
24   (let ((dx (vec-x v))
25         (dy (vec-y v)))
26     (sqrt (+ (square dx) (square dy)))))
27
28 ;;; Alternate Constructors ;;;
29
30 (define (unit-vec-from-direction direction)
31   (let ((theta (direction-theta direction)))
32     (make-vec (cos theta) (sin theta))))
33
34 (define (vec-from-direction-distance direction distance)
35   (scale-vec (unit-vec-from-direction direction) distance))
36
37 ;;; Conversions ;;;
38
39 (define (vec->direction v)
40   (let ((dx (vec-x v))
41         (dy (vec-y v)))
42     (make-direction (atan dy dx))))
43
44 ;;; Operations ;;;
45
46 ;;; Returns new vecs
47
48 (define (rotate-vec v radians)
49   (let ((dx (vec-x v))
50         (dy (vec-y v))
51         (c (cos radians))
52         (s (sin radians)))
53     (make-vec (+ (* c dx) (- (* s dy)))
54               (+ (* s dx) (* c dy)))))
55
56 (define (scale-vec v c)
57   (let ((dx (vec-x v))
58         (dy (vec-y v)))
59     (make-vec (* c dx) (* c dy))))
60
61 (define (scale-vec-to-dist v dist)
62   (scale-vec (unit-vec v) dist))
63
64 (define (reverse-vec v)
65   (make-vec (- (vec-x v))
66             (- (vec-y v))))

```

```

67
68 (define (rotate-vec-90 v)
69   (let ((dx (vec-x v))
70         (dy (vec-y v)))
71     (make-vec (- dy) dx)))
72
73 (define (unit-vec v)
74   (scale-vec v (/ (vec-magnitude v))))
75
76 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
77
78 (define (vec-equal? v1 v2)
79   (and (close-enuf? (vec-x v1) (vec-x v2))
80        (close-enuf? (vec-y v1) (vec-y v2))))
81
82 (define (vec-direction-equal? v1 v2)
83   (direction-equal?
84    (vec->direction v1)
85    (vec->direction v2)))
86
87 (define (vec-perpendicular? v1 v2)
88   (close-enuf?
89    (* (vec-x v1) (vec-x v2))
90    (* (vec-y v1) (vec-y (reverse-vec v2)))))

```


Listing A.8: figure/measurements.scm

```

1  ;;; measurements.scm
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Measurements primarily for analysis
7  ;; - Occasionally used for easily duplicating angles or segments
8
9  ;; Future:
10 ;; - Arc Measure
11
12 ;;; Code:
13
14 ;;;;;;;;;;;;;; Distance ;;;;;;;;;;;;;;
15
16 (define (distance p1 p2)
17   (sqrt (+ (square (- (point-x p1)
18                       (point-x p2)))
19           (square (- (point-y p1)
20                       (point-y p2))))))
21
22 ;;; Sign of distance is positive if the point is to the left of
23 ;;; the line direction and negative if to the right.
24 (define (signed-distance-to-line point line)
25   (let ((p1 (line-p1 line))
26         (p2 (line-p2 line))
27         (x0 (point-x point))
28         (y0 (point-y point))
29         (x1 (point-x p1))
30         (y1 (point-y p1))
31         (x2 (point-x p2))
32         (y2 (point-y p2)))
33     (/ (+ (- (* x0 (- y2 y1)))
34          (* y0 (- x2 x1))
35          (- (* x2 y1))
36          (* y2 x1))
37        (* 1.0
38           (sqrt (+ (square (- y2 y1))
39                   (square (- x2 x1)))))))
40
41 (define (distance-to-line point line)
42   (abs (signed-distance-to-line point line)))
43
44 ;;;;;;;;;;;;;; Angles ;;;;;;;;;;;;;;
45
46 (define (angle-measure a)
47   (let* ((d1 (angle-arm-1 a))
48         (d2 (angle-arm-2 a))
49         (subtract-directions d1 d2)))
50
51 ;;;;;;;;;;;;;; Measured Elements ;;;;;;;;;;;;;;
52
53 (define (measured-point-on-ray r dist)
54   (let* ((p1 (ray-p1 r))
55         (p2 (ray-p2 r))
56         (v (sub-points p1 p2))
57         (scaled-v (scale-vec-to-dist v dist)))
58     (add-to-point p1 scaled-v)))
59
60 (define (measured-angle-ccw p1 vertex radians)
61   (let* ((v1 (sub-points p1 vertex))
62         (v-rotated (rotate-vec v (- radians))))
63     (angle v1 vertex v-rotated)))
64
65 (define (measured-angle-cw p1 vertex radians)
66   (reverse-angle (measured-angle-ccw p1 vertex (- radians))))

```

Listing A.9: figure/angle.scm

```

1  ;;; angle.scm --- Angles
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Initially three points, now vertex + two directions
7  ;; - Counter-clockwise orientation
8  ;; - Uniquely determining from elements forces directions
9  ;; - naming of "arms" vs. "directions"
10
11 ;; Future Ideas:
12 ;; - Automatically discover angles from diagrams (e.g. from a pile of
13 ;;   points and segments)
14 ;; - Angle intersections
15
16 ;;; Code:
17
18 ;;;;;;;;;;;;;;;;;;;;;;;;; Angles ;;;;;;;;;;;;;;;;;;;;;;;;;
19
20 ;;; dir1 and dir2 are directions of the angle arms
21 ;;; The angle sweeps from dir2 *counter clockwise* to dir1
22 (define-record-type <angle>
23   (make-angle dir1 vertex dir2)
24   angle?
25   (dir1 angle-arm-1)
26   (vertex angle-vertex)
27   (dir2 angle-arm-2))
28
29 (declare-element-component-handler
30   (component-procedure-from-getters
31     ray-from-arm-1
32     angle-vertex
33     ray-from-arm-2)
34   angle?)
35
36 (defhandler generic-element-name
37   (lambda (angle)
38     `(angle ,(element-name (angle-vertex angle))))
39   angle?)
40
41 (defhandler print
42   element-name
43   angle?)
44
45 ;;;;;;;;;;;;;;;;;;;;;;;;; Transformations on Angles ;;;;;;;;;;;;;;;;;;;;;;;;;
46
47 (define (reverse-angle a)
48   (let ((d1 (angle-arm-1 a))
49         (v (angle-vertex a))
50         (d2 (angle-arm-2 a)))
51     (make-angle d2 v d1)))
52
53 (define (smallest-angle a)
54   (if (> (angle-measure a) pi)
55       (reverse-angle a)
56       a))
57
58 ;;;;;;;;;;;;;;;;;;;;;;;;; Alternate Constructors ;;;;;;;;;;;;;;;;;;;;;;;;;
59
60 (define (angle-from-points p1 vertex p2)
61   (let ((arm1 (direction-from-points vertex p1))
62         (arm2 (direction-from-points vertex p2)))
63     (make-angle arm1 vertex arm2)))
64
65 (define (smallest-angle-from-points p1 vertex p2)
66   (smallest-angle (angle-from-points p1 vertex p2)))

```

```

67
68 ;;;;;;;;;;;;;; Angle from pairs of elements ;;;;;;;;;;;;;;
69
70 (define angle-from (make-generic-operation 2 'angle-from))
71
72 (define (angle-from-lines l1 l2)
73   (let ((d1 (line->direction l1))
74         (d2 (line->direction l2))
75         (p (intersect-lines l1 l2)))
76     (make-angle d1 p d2)))
77 (defhandler angle-from angle-from-lines line? line?)
78
79 (define (angle-from-line-ray l r)
80   (let ((vertex (ray-endpoint r)))
81     (assert (on-line? vertex l)
82             "Angle-from-line-ray: Vertex of ray not on line"))
83   (let ((d1 (line->direction l))
84         (d2 (ray->direction r)))
85     (make-angle d1 vertex d2)))
86 (defhandler angle-from angle-from-line-ray line? ray?)
87
88 (define (angle-from-ray-line r l)
89   (reverse-angle (angle-from-line-ray l r)))
90 (defhandler angle-from angle-from-ray-line ray? line?)
91
92 (define (angle-from-segment-segment s1 s2)
93   (define (angle-from-segment-internal s1 s2)
94     (let ((vertex (segment-endpoint-1 s1)))
95       (let ((d1 (segment->direction s1))
96             (d2 (segment->direction s2)))
97         (make-angle d1 vertex d2))))
98   (cond ((point-equal? (segment-endpoint-1 s1)
99                        (segment-endpoint-1 s2))
100         (angle-from-segment-internal s1 s2))
101         ((point-equal? (segment-endpoint-2 s1)
102                        (segment-endpoint-1 s2))
103         (angle-from-segment-internal (flip s1) s2))
104         ((point-equal? (segment-endpoint-1 s1)
105                        (segment-endpoint-2 s2))
106         (angle-from-segment-internal s1 (flip s2)))
107         ((point-equal? (segment-endpoint-2 s1)
108                        (segment-endpoint-2 s2))
109         (angle-from-segment-internal (flip s1) (flip s2)))
110         (else (error "Angle-from-segment-segment must share vertex"))))
111 (defhandler angle-from angle-from-segment-segment segment? segment?)
112
113 (define (smallest-angle-from a b)
114   (smallest-angle (angle-from a b)))
115
116 ;;;;;;;;;;;;;; Predicates on Angles ;;;;;;;;;;;;;;
117
118 (define (angle-measure-equal? a1 a2)
119   (close-enuf? (angle-measure a1)
120                (angle-measure a2)))
121
122 (define (supplementary-angles? a1 a2)
123   (close-enuf? (+ (angle-measure a1)
124                    (angle-measure a2))
125                pi))
126
127 (define (complementary-angles? a1 a2)
128   (close-enuf? (+ (angle-measure a1)
129                    (angle-measure a2))
130                (/ pi 2.0)))
131
132 ;;;;;;;;;;;;;; Definitions ;;;;;;;;;;;;;;
133
134 ;;; TODO? Consider learning or putting elsewhere

```

```

135 (define (linear-pair? a1 a2)
136   (define (linear-pair-internal? a1 a2)
137     (and (point-equal? (angle-vertex a1)
138                        (angle-vertex a2))
139          (direction-equal? (angle-arm-2 a1)
140                           (angle-arm-1 a2))
141          (direction-opposite? (angle-arm-1 a1)
142                               (angle-arm-2 a2))))
143   (or (linear-pair-internal? a1 a2)
144       (linear-pair-internal? a2 a1)))
145
146 (define (vertical-angles? a1 a2)
147   (and (point-equal? (angle-vertex a1)
148                      (angle-vertex a2))
149        (direction-opposite? (angle-arm-1 a1)
150                             (angle-arm-1 a2))
151        (direction-opposite? (angle-arm-2 a1)
152                             (angle-arm-2 a2))))

```

Listing A.10: figure/bounds.scm

```

1  ;;; bounds.scm --- Graphics Bounds
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Logic to extend segments to graphics bounds so they can be drawn.
7
8  ;; Future:
9  ;; - Separate logical bounds of figures from graphics bounds
10 ;; - Combine logic for line and ray (one vs. two directions)
11 ;; - Should these be a part of "figure" vs. "graphics"
12 ;; - Remapping of entire figures to different canvas dimensions
13
14 ;;; Code:
15
16 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Bounds ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
17
18 (define-record-type <bounds>
19   (make-bounds x-interval y-interval)
20   bounds?
21   (x-interval bounds-x-interval)
22   (y-interval bounds-y-interval))
23
24 (define (bounds-xmin b) (interval-low (bounds-x-interval b)))
25 (define (bounds-xmax b) (interval-high (bounds-x-interval b)))
26 (define (bounds-ymin b) (interval-low (bounds-y-interval b)))
27 (define (bounds-ymax b) (interval-high (bounds-y-interval b)))
28
29 (define (print-bounds b)
30   `(bounds ,(bounds-xmin b)
31            ,(bounds-xmax b)
32            ,(bounds-ymin b)
33            ,(bounds-ymax b)))
34 (defhandler print print-bounds bounds?)
35
36 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Bounds Constants ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
37
38 ;;; Max bounds of the graphics window
39
40 (define *g-min-x* -2)
41 (define *g-max-x* 2)
42 (define *g-min-y* -2)
43 (define *g-max-y* 2)
44
45 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Conversion to segments for Graphics ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
46
47 (define (extend-to-max-segment p1 p2)
48   (let ((x1 (point-x p1))
49         (y1 (point-y p1))
50         (x2 (point-x p2))
51         (y2 (point-y p2)))
52     (let ((dx (- x2 x1))
53           (dy (- y2 y1)))
54       (cond
55         ((= 0 dx) (make-segment
56                     (make-point x1 *g-min-y*)
57                     (make-point x1 *g-max-y*)))
58         ((= 0 dy) (make-segment
59                     (make-point *g-min-x* y1)
60                     (make-point *g-max-x* y1)))
61         (else
62          (let ((t-xmin (/ (- *g-min-x* x1) dx))
63                (t-xmax (/ (- *g-max-x* x1) dx))
64                (t-ymin (/ (- *g-min-y* y1) dy))
65                (t-ymax (/ (- *g-max-y* y1) dy)))
66            (let* ((sorted (sort (list t-xmin t-xmax t-ymin t-ymax) <)))

```

```

67         (min-t (cadr sorted))
68         (max-t (caddr sorted))
69         (min-x (+ x1 (* min-t dx)))
70         (min-y (+ y1 (* min-t dy)))
71         (max-x (+ x1 (* max-t dx)))
72         (max-y (+ y1 (* max-t dy))))
73     (make-segment (make-point min-x min-y)
74                   (make-point max-x max-y))))))
75
76 (define (ray-extend-to-max-segment p1 p2)
77   (let ((x1 (point-x p1))
78         (y1 (point-y p1))
79         (x2 (point-x p2))
80         (y2 (point-y p2)))
81     (let ((dx (- x2 x1))
82           (dy (- y2 y1)))
83       (cond
84         ((= 0 dx) (make-segment
85                     (make-point x1 *g-min-y*)
86                     (make-point x1 *g-max-y*)))
87         ((= 0 dy) (make-segment
88                     (make-point *g-min-x* y1)
89                     (make-point *g-min-y* y1)))
90         (else
91          (let ((t-xmin (/ (- *g-min-x* x1) dx))
92                (t-xmax (/ (- *g-max-x* x1) dx))
93                (t-ymin (/ (- *g-min-y* y1) dy))
94                (t-ymax (/ (- *g-max-y* y1) dy)))
95              (let* ((sorted (sort (list t-xmin t-xmax t-ymin t-ymax) <))
96                    (min-t (cadr sorted))
97                    (max-t (caddr sorted))
98                    (min-x (+ x1 (* min-t dx)))
99                    (min-y (+ y1 (* min-t dy)))
100                   (max-x (+ x1 (* max-t dx)))
101                   (max-y (+ y1 (* max-t dy))))
102                (make-segment p1
103                              (make-point max-x max-y)))))))))
104
105 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Rescale Figure ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
106
107 (define empty-bounds (make-bounds (make-interval 0 0)
108                                   (make-interval 0 0)))
109
110 (define (extend-interval i new-value)
111   (let ((low (interval-low i))
112         (high (interval-high i)))
113     (make-interval (min low new-value)
114                   (max high new-value))))
115
116 (define (interval-length i)
117   (- (interval-high i)
118      (interval-low i)))
119
120 (define (extend-bounds bounds point)
121   (let ((px (point-x point))
122         (py (point-y point)))
123     (make-bounds
124      (extend-interval (bounds-x-interval bounds)
125                       px)
126      (extend-interval (bounds-y-interval bounds)
127                       py))))
128
129 (define (bounds-width bounds)
130   (interval-length (bounds-x-interval bounds)))
131
132 (define (bounds-height bounds)
133   (interval-length (bounds-y-interval bounds)))
134

```

```

135 (define (bounds->square bounds)
136   (let ((new-side-length
137         (max (bounds-width bounds)
138              (bounds-height bounds))))
139     (recenter-bounds bounds
140                      new-side-length
141                      new-side-length)))
142
143 (define (recenter-interval i new-length)
144   (let* ((min (interval-low i))
145         (max (interval-high i))
146         (old-half-length (/ (- max min) 2))
147         (new-half-length (/ new-length 2)))
148     (make-interval (- (+ min old-half-length) new-half-length)
149                   (+ (- max old-half-length) new-half-length))))
150
151 (define (recenter-bounds bounds new-width new-height)
152   (make-bounds
153     (recenter-interval (bounds-x-interval bounds) new-width)
154     (recenter-interval (bounds-y-interval bounds) new-height)))
155
156 (define (scale-bounds bounds scale-factor)
157   (recenter-bounds
158     bounds
159     (* (bounds-width bounds) scale-factor)
160     (* (bounds-height bounds) scale-factor)))
161
162 (define (extract-bounds figure)
163   (let ((all-points (figure-points figure)))
164     (let lp ((bounds empty-bounds)
165              (points all-points))
166       (if (null? points)
167           bounds
168           (extend-bounds (lp bounds (cdr points))
                          (car points))))))
169

```

Listing A.11: figure/circle.scm

```

1  ;; circle.scm --- Circles
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Currently rather limited support for circles
7
8  ;; Future:
9  ;; - Arcs, tangents, etc.
10
11  ;; Code:
12
13  ;;;;;;;;;;;;;;;;; Circle structure ;;;;;;;;;;;;;;;;;
14
15  (define-record-type <circle>
16    (make-circle center radius)
17    circle?
18    (center circle-center)
19    (radius circle-radius))
20
21  ;;;;;;;;;;;;;;;;; Alternate Constructions ;;;;;;;;;;;;;;;;;
22
23  (define (circle-from-points center radius-point)
24    (make-circle center
25                  (distance center radius-point)))
26
27  ;;;;;;;;;;;;;;;;; Points on circle ;;;;;;;;;;;;;;;;;
28
29  (define (point-on-circle-in-direction cir dir)
30    (let ((center (circle-center cir))
31          (radius (circle-radius cir)))
32      (add-to-point
33        center
34        (vec-from-direction-distance
35          dir radius))))

```


Listing A.12: figure/point.scm

```

1  ;;; point.scm --- Point
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Points are the basis for most elements
7
8  ;; Future:
9  ;; - Transform to different canvases
10 ;; - Have points know what elements they are on.
11
12 ;;; Code:
13
14 ;;; Point Structure ;;;
15
16 (define-record-type <point>
17   (make-point x y)
18   point?
19   (x point-x)
20   (y point-y))
21
22 (define (print-point p)
23   `(point ,(point-x p) ,(point-y p)))
24
25 (defhandler print
26   print-point point?)
27
28 ;;; Predicates ;;;
29
30 (define (point-equal? p1 p2)
31   (and (close-enuf? (point-x p1)
32                     (point-x p2))
33        (close-enuf? (point-y p1)
34                     (point-y p2))))
35
36 ;;; Operations ;;;
37
38 ;;; P2 - P1
39 (define (sub-points p2 p1)
40   (let ((x1 (point-x p1))
41         (x2 (point-x p2))
42         (y2 (point-y p2))
43         (y1 (point-y p1)))
44     (make-vec (- x2 x1)
45               (- y2 y1))))
46
47 ;;; Direction from p1 to p2
48 (define (direction-from-points p1 p2)
49   (vec->direction (sub-points p2 p1)))
50
51 (define (add-to-point p vec)
52   (let ((x (point-x p))
53         (y (point-y p))
54         (dx (vec-x vec))
55         (dy (vec-y vec)))
56     (make-point (+ x dx)
57                 (+ y dy))))

```

Listing A.13: figure/constructions.scm

```

1  ;; constructions.scm --- Constructions
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Various logical constructions that can be performed on elements
7  ;; - Some higher-level constructions...
8
9  ;; Future:
10 ;; - More constructions?
11 ;; - Separation between compass/straightedge and compound?
12 ;; - Experiment with higher-level vs. learned constructions
13
14 ;; Code:
15
16 ;;;;;;;;;;;;;;;;;;;;;;;;;; Segment Constructions ;;;;;;;;;;;;;;;;;;;;;;;;;;
17
18 (define (midpoint p1 p2)
19   (let ((newpoint
20         (make-point (avg (point-x p1)
21                          (point-x p2))
22                      (avg (point-y p1)
23                          (point-y p2)))))
24     (with-dependency
25       `(midpoint ,(element-dependency p1) ,(element-dependency p2))
26       (with-source (lambda (premise)
27                     (midpoint
28                       ((element-source p1) premise)
29                       ((element-source p1) premise)))
30                     newpoint))))
31
32 (define (segment-midpoint s)
33   (let ((p1 (segment-endpoint-1 s))
34         (p2 (segment-endpoint-2 s)))
35     (with-dependency
36       `(segment-midpoint ,s)
37       (with-source (lambda (premise)
38                     (segment-midpoint
39                       ((element-source s) premise))
40                     (midpoint p1 p2))))))
41
42 ;;;;;;;;;;;;;;;;;;;;;;;;;; Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;
43
44 ;; TODO: Where to put these?
45 (define (on-segment? p seg)
46   (let ((seg-start (segment-endpoint-1 seg))
47         (seg-end (segment-endpoint-2 seg)))
48     (or (point-equal? seg-start p)
49         (point-equal? seg-end p)
50         (let ((seg-length (distance seg-start seg-end))
51               (p-length (distance seg-start p))
52               (dir-1 (direction-from-points seg-start p))
53               (dir-2 (direction-from-points seg-start seg-end)))
54           (and (direction-equal? dir-1 dir-2)
55                 (< p-length seg-length))))))
56
57 (define (on-line? p l)
58   (let ((line-pt (line-point l))
59         (line-dir (line-direction l)))
60     (or (point-equal? p line-pt)
61         (let ((dir-to-p (direction-from-points p line-pt)))
62           (or (direction-equal? line-dir dir-to-p)
63               (direction-equal? line-dir (reverse-direction dir-to-p))))))
64
65 (define (on-ray? p r)
66   (let ((ray-endpt (ray-endpoint r))

```

```

67     (ray-dir (ray-direction r)))
68   (or (point-equal? ray-endpt p)
69       (let ((dir-to-p (direction-from-points ray-endpt p)))
70         (direction-equal? dir-to-p ray-dir))))
71
72 ;;;;;;;;;;;;;;;;;;;;;;;;;; Construction of lines ;;;;;;;;;;;;;;;;;;;;;;;;;;
73
74 (define (perpendicular linear-element point)
75   (let* ((direction (->direction linear-element))
76          (rotated-direction (rotate-direction-90 direction))
77          (make-line point rotated-direction)))
78
79 ;;; endpoint-1 is point, endpoint-2 is on linear-element
80 (define (perpendicular-to linear-element point)
81   (let ((pl (perpendicular linear-element point)))
82     (let ((i (intersect-linear-elements pl (->line linear-element))))
83       (make-segment point i))))
84
85 (define (perpendicular-line-to linear-element point)
86   (let ((pl (perpendicular linear-element point)))
87     pl))
88
89 (define (perpendicular-bisector segment)
90   (let ((midpt (segment-midpoint segment)))
91     (let ((pb (perpendicular (segment->line segment)
92                               midpt)))
93       (save-obvious-observation!
94         (make-observation perpendicular-relationship
95                           (list pb segment)))
96       pb)))
97
98 (define (angle-bisector a)
99   (let* ((d1 (angle-arm-1 a))
100          (d2 (angle-arm-2 a))
101          (vertex (angle-vertex a))
102          (radians (angle-measure a))
103          (half-angle (/ radians 2))
104          (new-direction (add-to-direction d2 half-angle)))
105     (make-ray vertex new-direction)))
106
107 (define (polygon-angle-bisector polygon vertex-angle)
108   (angle-bisector (polygon-angle polygon vertex-angle)))
109
110 ;;;;;;;;;;;;;;;;;;;;;;;;;; Higher-order constructions ;;;;;;;;;;;;;;;;;;;;;;;;;;
111
112 (define (circumcenter t)
113   (let ((p1 (polygon-point-ref t 0))
114         (p2 (polygon-point-ref t 1))
115         (p3 (polygon-point-ref t 2)))
116     (let ((l1 (perpendicular-bisector (make-segment p1 p2)))
117           (l2 (perpendicular-bisector (make-segment p1 p3))))
118       (intersect-linear-elements l1 l2)))
119
120 ;;;;;;;;;;;;;;;;;;;;;;;;;; Concurrent Linear Elements ;;;;;;;;;;;;;;;;;;;;;;;;;;
121
122 (define (concurrent? l1 l2 l3)
123   (let ((i-point (intersect-linear-elements-no-endpoints l1 l2)))
124     (and i-point
125          (on-element? i-point l3)
126          (not (element-endpoint? i-point l3)))))
127
128 (define (concentric? p1 p2 p3 p4)
129   (and (not (point-equal? p1 p2))
130        (not (point-equal? p1 p3))
131        (not (point-equal? p1 p4))
132        (not (point-equal? p2 p3))
133        (not (point-equal? p2 p4))
134        (not (point-equal? p3 p4))))

```

```

135      (let ((pb-1 (perpendicular-bisector
136                  (make-segment p1 p2)))
137            (pb-2 (perpendicular-bisector
138                  (make-segment p2 p3)))
139            (pb-3 (perpendicular-bisector
140                  (make-segment p3 p4))))
141        (concurrent? pb-1 pb-2 pb-3)))
142
143 (define (concentric-with-center? center p1 p2 p3)
144   (let ((d1 (distance center p1))
145         (d2 (distance center p2))
146         (d3 (distance center p3)))
147     (and (close-enuf? d1 d2)
148          (close-enuf? d1 d3))))

```

Listing A.14: figure/intersections.scm

```

1  ;;; intersections.scm --- Intersections
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Unified intersections
7  ;; - Separation of core computations
8
9  ;; Future:
10 ;; - Amb-like selection of multiple intersections, or list?
11 ;; - Deal with elements that are exactly the same
12
13 ;;; Code:
14
15 ;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Computations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
16
17 ;;; http://en.wikipedia.org/wiki/Line%E2%80%93line\_intersection
18 ;;; line 1 through p1, p2 with line 2 through p3, p4
19 (define (intersect-lines-by-points p1 p2 p3 p4)
20   (let ((x1 (point-x p1))
21         (y1 (point-y p1))
22         (x2 (point-x p2))
23         (y2 (point-y p2))
24         (x3 (point-x p3))
25         (y3 (point-y p3))
26         (x4 (point-x p4))
27         (y4 (point-y p4)))
28     (let* ((denom
29             (det (det x1 1 x2 1)
30                  (det y1 1 y2 1)
31                  (det x3 1 x4 1)
32                  (det y3 1 y4 1)))
33            (num-x
34             (det (det x1 y1 x2 y2)
35                  (det x1 1 x2 1)
36                  (det x3 y3 x4 y4)
37                  (det x3 1 x4 1)))
38            (num-y
39             (det (det x1 y1 x2 y2)
40                  (det y1 1 y2 1)
41                  (det x3 y3 x4 y4)
42                  (det y3 1 y4 1))))
43       (if (= denom 0)
44           '()
45           (let
46             ((px (/ num-x denom))
47              (py (/ num-y denom)))
48             (list (make-point px py)))))))
49
50 ;;; http://mathforum.org/library/drmath/view/51836.html
51 (define (intersect-circles-by-centers-radii c1 r1 c2 r2)
52   (let* ((a (point-x c1))
53          (b (point-y c1))
54          (c (point-x c2))
55          (d (point-y c2))
56          (e (- c a))
57          (f (- d b))
58          (p (+ (square e)
59                (square f)))
60          (k (/ (- (+ (square p) (square r1))
61                    (square r2))
62                (* 2 p))))
63     (if (> k r1)
64         (error "Circle's don't intersect")
65         (let* ((t (sqrt (- (square r1)
66                             (square k))))
67                (square k))))
66

```

```

67         (x1 (+ a (/ (* e k) p)))
68         (y1 (+ b (/ (* f k) p)))
69         (dx (/ (* f t) p))
70         (dy (- (/ (* e t) p))))
71     (list (make-point (+ x1 dx)
72                     (+ y1 dy))
73           (make-point (- x1 dx)
74                     (- y1 dy))))))
75
76 ;;; Intersect circle centered at c with radius r and line through
77 ;;; points p1, p2
78 ;;; http://mathworld.wolfram.com/Circle-LineIntersection.html
79 (define (intersect-circle-line-by-points c r p1 p2)
80   (let ((offset (sub-points (make-point 0 0) c)))
81     (let ((p1-shifted (add-to-point p1 offset))
82           (p2-shifted (add-to-point p2 offset)))
83       (let ((x1 (point-x p1-shifted))
84             (y1 (point-y p1-shifted))
85             (x2 (point-x p2-shifted))
86             (y2 (point-y p2-shifted)))
87         (let* ((dx (- x2 x1))
88                (dy (- y2 y1))
89                (dr (sqrt (+ (square dx) (square dy))))
90                (d (det x1 x2 y1 y2))
91                (disc (- (* (square r) (square dr)) (square d))))
92           (if (< disc 0)
93               (list)
94               (let ((x-a (* d dy))
95                     (x-b (* (sgn dy) dx (sqrt disc)))
96                     (y-a (- (* d dx)))
97                     (y-b (* (abs dy) (sqrt disc))))
98                 (let ((ip1 (make-point
99                         (/ (+ x-a x-b) (square dr))
100                        (/ (+ y-a y-b) (square dr))))
101                   (ip2 (make-point
102                        (/ (- x-a x-b) (square dr))
103                        (/ (- y-a y-b) (square dr))))))
104                 (if (close-enuf? 0 disc) ;; Tangent
105                     (list (add-to-point ip1 (reverse-vec offset)))
106                     (list (add-to-point ip1 (reverse-vec offset))
107                           (add-to-point ip2 (reverse-vec offset))))))))))
108
109 ;;; Basic Intersections ;;;
110
111 (define (intersect-lines-to-list line1 line2)
112   (let ((p1 (line-p1 line1))
113         (p2 (line-p2 line1))
114         (p3 (line-p1 line2))
115         (p4 (line-p2 line2)))
116     (intersect-lines-by-points p1 p2 p3 p4)))
117
118 (define (intersect-lines line1 line2)
119   (let ((i-list (intersect-lines-to-list line1 line2)))
120     (if (null? i-list)
121         (error "Lines don't intersect")
122         (car i-list))))
123
124 (define (intersect-circles cir1 cir2)
125   (let ((c1 (circle-center cir1))
126         (c2 (circle-center cir2))
127         (r1 (circle-radius cir1))
128         (r2 (circle-radius cir2)))
129     (intersect-circles-by-centers-radii c1 r1 c2 r2)))
130
131 (define (intersect-circle-line cir line)
132   (let ((center (circle-center cir))
133         (radius (circle-radius cir))
134         (p1 (line-p1 line))

```

```

135         (p2 (line-p2 line)))
136     (intersect-circle-line-by-points center radius p1 p2)))
137
138 (define standard-intersect
139   (make-generic-operation 2 'standard-intersect))
140
141 (defhandler standard-intersect
142   intersect-lines-to-list line? line?)
143
144 (defhandler standard-intersect
145   intersect-circles circle? circle?)
146
147 (defhandler standard-intersect
148   intersect-circle-line circle? line?)
149
150 (defhandler standard-intersect
151   (flip-args intersect-circle-line) line? circle?)
152
153 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Generic intersection ;;;;;;;;;;;;;;;;;
154
155 (define (intersect-linear-elements el-1 el-2)
156   (let ((i-list (standard-intersect (->line el-1)
157                                     (->line el-2))))
158     (if (null? i-list)
159         #f
160         (let ((i (car i-list)))
161           (if (or (not (on-element? i el-1))
162                 (not (on-element? i el-2)))
163               #f
164               i))))))
165
166 (define (intersect-linear-elements-no-endpoints el-1 el-2)
167   (let ((i (intersect-linear-elements el-1 el-2)))
168     (and (or i
169              (element-endpoint? i el-1)
170              (element-endpoint? i el-2))
171          i)))
172
173 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; On Elements ;;;;;;;;;;;;;;;;;
174
175 (define on-element? (make-generic-operation 2 'on-element?))
176
177 (defhandler on-element? on-segment? point? segment?)
178 (defhandler on-element? on-line? point? line?)
179 (defhandler on-element? on-ray? point? ray?)
180
181 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Element Endpoint Test ;;;;;;;;;;;;;;;;;
182
183 (define element-endpoint? (make-generic-operation 2 'on-endpoint?
184                                                    (lambda (p el) #f)))
185
186 (define (segment-endpoint? p seg)
187   (or (point-equal? p (segment-endpoint-1 seg))
188       (point-equal? p (segment-endpoint-2 seg))))
189 (defhandler element-endpoint? segment-endpoint? point? segment?)
190
191 (define (ray-endpoint? p ray)
192   (point-equal? p (ray-endpoint seg)))
193 (defhandler element-endpoint? ray-endpoint? point? ray?)

```

Listing A.15: figure/figure.scm

```

1  ;;; figure.scm --- Figure
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Gathers elements that are part of a figure
7  ;; - Helpers to extract relevant elements
8
9  ;; Future:
10 ;; - Convert to record type like other structures
11 ;; - Extract points automatically?
12
13 ;;; Code:
14
15 ;;; Figure Structure ;;;
16
17 (define (figure . elements)
18   (cons 'figure elements))
19 (define (figure-elements figure)
20   (cdr figure))
21
22 (define (all-figure-elements figure)
23   (append (figure-elements figure)
24           (figure-points figure)
25           (figure-linear-elements figure)))
26
27 (define (figure? x)
28   (and (pair? x)
29        (eq? (car x 'figure))))
30
31 ;;; Getters ;;;
32
33 (define (figure-filter predicate figure)
34   (filter predicate (figure-elements figure)))
35
36 (define (figure-points figure)
37   (dedupe-by point-equal?
38              (append (figure-filter point? figure)
39                      (append-map (lambda (polygon) (polygon-points polygon))
40                                  (figure-filter polygon? figure))
41                      (append-map (lambda (s)
42                                   (list (segment-endpoint-1 s)
43                                         (segment-endpoint-2 s)))
44                                  (figure-filter segment? figure))
45                      (map (lambda (a)
46                           (angle-vertex a))
47                          (figure-filter angle? figure)))))
48
49 (define (figure-angles figure)
50   (append (figure-filter angle? figure)
51           (append-map (lambda (polygon) (polygon-angles polygon))
52                       (figure-filter polygon? figure))))
53
54 (define (figure-segments figure)
55   (append (figure-filter segment? figure)
56           (append-map (lambda (polygon) (polygon-segments polygon))
57                       (figure-filter polygon? figure))))
58
59 (define (figure-linear-elements figure)
60   (append (figure-filter linear-element? figure)
61           (append-map (lambda (polygon) (polygon-segments polygon))
62                       (figure-filter polygon? figure))))

```


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

```

1  ;;; math-utils.scm --- Math Helpers
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - All angles are [0, 2pi]
7  ;; - Other helpers
8
9  ;; Future:
10 ;; - Add more as needed, integrate with scmutils-basic
11
12 ;;; Code:
13
14 ;;; Angles ;;;
15
16 (define pi (* 4 (atan 1)))
17
18 (define (fix-angle-0-2pi a)
19   (float-mod a (* 2 pi)))
20
21 (define (rad->deg rad)
22   (* (/ rad (* 2 pi)) 360))
23
24 ;;; Modular ;;;
25
26 (define (float-mod num mod)
27   (- num
28      (* (floor (/ num mod))
29          mod)))
30
31 ;;; Basic Operators ;;;
32
33 (define (avg a b)
34   (/ (+ a b) 2))
35
36 (define (sgn x)
37   (if (< x 0) -1 1))
38
39 ;;; Linear Algebra ;;;
40
41 (define (det a11 a12 a21 a22)
42   (- (* a11 a22) (* a12 a21)))
43
44 ;;; Extensions of Max/Min ;;;
45
46 (define (min-positive . args)
47   (min (filter (lambda (x) (>= x 0)) args)))
48
49 (define (max-negative . args)
50   (min (filter (lambda (x) (<= x 0)) args)))

```

Listing A.17: figure/polygon.scm

```

1  ;;; polygon.scm --- Polygons
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Points and (derived) segments define polygon
7
8  ;; Future
9  ;; - Figure out dependencies better
10 ;; - Other operations, angles? diagonals? etc.
11
12 ;;; Code:
13
14 ;;; Polygon Structure ;;;
15
16 ;;; Data structure for a polygon, implemented as a list of
17 ;;; points in counter-clockwise order.
18 ;;; Drawing a polygon will draw all of its points and segments.
19 (define-record-type <polygon>
20   (%polygon n-points points)
21   polygon?
22   (n-points polygon-n-points)
23   (points %polygon-points))
24
25 (define (polygon-from-points . points)
26   (let ((n-points (length points)))
27     (%polygon n-points points)))
28
29 (define ((ngon-predicate n) obj)
30   (and (polygon? obj)
31        (= n (polygon-n-points obj))))
32
33 ;;; Polygon Points ;;;
34
35 ;;; Internal reference for polygon points
36 (define (polygon-point-ref polygon i)
37   (if (not (<= 0 i (- (polygon-n-points polygon) 1)))
38       (error "polygon point index not in range"))
39   (list-ref (%polygon-points polygon) i))
40
41 (define (polygon-points polygon)
42   (map (lambda (i) (polygon-point-ref polygon i))
43        (iota (polygon-n-points polygon))))
44
45 ;;; External polygon points including dependencies
46 (define (polygon-point polygon i)
47   ;;; TODO: Handle situations where polygon isn't terminal dependency
48   (with-dependency ;;-if-unknown
49     `(polygon-point ,i ,(element-dependency polygon))
50     (with-source
51       (lambda (p) (polygon-point (car p) i))
52       (polygon-point-ref polygon i))))
53
54 (declare-element-component-handler
55   polygon-point
56   polygon?)
57
58 (define (polygon-index-from-point polygon point)
59   (index-of
60    point
61    (%polygon-points polygon)
62    point-equal?))
63
64 (define (name-polygon polygon)
65   (for-each (lambda (i)
66               (set-element-name! (polygon-point-ref polygon i)

```

```

67         (nth-letter-symbol (+ i 1))))
68     (iota (polygon-n-points polygon)))
69 polygon)
70
71 ;;;;;;;;;;;;;; Polygon Segments ;;;;;;;;;;;;;;
72
73 ;;; i and j are indices of adjacent points
74 (define (polygon-segment polygon i j)
75   (let ((n-points (polygon-n-points polygon)))
76     (cond
77       ((not (or (= i (modulo (+ j 1) n-points))
78                 (= j (modulo (+ i 1) n-points)))))
79       (error "polygon-segment must be called with adjacent indices"))
80     ((or (>= i n-points)
81          (>= j n-points))
82      (error "polygon-segment point index out of range"))
83     (else
84      (let* ((p1 (polygon-point-ref polygon i))
85              (p2 (polygon-point-ref polygon j))
86              (segment (make-segment p1 p2)))
87        ;;; TODO: Handle situations where polygon isn't terminal dependency
88        (with-dependency
89          `(polygon-segment ,i ,j ,polygon)
90          (with-source
91            (lambda (p) (polygon-segment (car p) i j))
92            segment))))))
93
94 (define (polygon-segments polygon)
95   (let ((n-points (polygon-n-points polygon)))
96     (map (lambda (i)
97            (polygon-segment polygon i (modulo (+ i 1) n-points)))
98          (iota n-points))))
99
100 ;;;;;;;;;;;;;; Polygon Angles ;;;;;;;;;;;;;;
101
102 (define polygon-angle
103   (make-generic-operation 2 'polygon-angle))
104
105 (define (polygon-angle-by-index polygon i)
106   (let ((n-points (polygon-n-points polygon)))
107     (cond
108       ((not (<= 0 i (- n-points 1)))
109        (error "polygon-angle point index out of range"))
110     (else
111      (let* ((v (polygon-point-ref polygon i))
112              (a1p (polygon-point-ref polygon
113                    (modulo (- i 1)
114                          n-points)))
115              (a2p (polygon-point-ref polygon
116                    (modulo (+ i 1)
117                          n-points)))
118              (angle (angle-from-points a1p v a2p)))
119        (with-dependency
120          `(polygon-angle ,i ,polygon)
121          (with-source
122            (lambda (p) (polygon-angle-by-index (car p) i))
123            angle))))))
124
125 (defhandler polygon-angle
126   polygon-angle-by-index
127   polygon? number?)
128
129 (define (polygon-angle-by-point polygon p)
130   (let ((i (polygon-index-from-point polygon p)))
131     (if (not i)
132         (error "Point not in polygon" (list p polygon)))
133     (polygon-angle-by-index polygon i)))
134

```

```
135 (defhandler polygon-angle
136   polygon-angle-by-point
137   polygon? point?)
138
139 (define (polygon-angles polygon)
140   (map (lambda (i) (polygon-angle-by-index polygon i))
141        (iota (polygon-n-points polygon))))
```

Listing A.18: figure/metadata.scm

```

1  ;; metadata.scm - Element metadata
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Currently, names
7  ;; - Dependencies grew here, but are now separate
8
9  ;; Future:
10 ;; - Point/Linear/Circle adjacency - walk like graph
11
12 ;; Code:
13
14 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Names ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
15
16 (define (set-element-name! element name)
17   (eq-put! element 'name name)
18   element)
19
20 (define (element-name element)
21   (or (eq-get element 'name)
22       (generic-element-name element)))
23
24 (define *unnamed* (list 'unnamed))
25 (define (is-unnamed? x) (eq? *unnamed* x))
26
27 (define generic-element-name
28   (make-generic-operation 1 'generic-element-name
29                           (lambda (el) *unnamed*)))
30
31 (define (named? element)
32   (not (is-unnamed? (element-name element))))

```

Listing A.19: figure/dependencies.scm

```

1  ;; dependencies.scm --- Dependencies of figure elements
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Use eq-properties to set dependencies of elements
7  ;; - Some random elements are given external/random dependencies
8  ;; - For some figures, override dependencies of intermediate elements
9
10 ;; Future:
11 ;; - Expand to full dependencies
12 ;; - Start "learning" and generalizing
13
14 ;; Code:
15
16 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Sources ;;;;;;;;;;;;;;;;;
17
18 (define (set-source! element source)
19   (eq-put! element 'source source))
20
21 (define (with-source source element)
22   (set-source! element source)
23   element)
24
25 (define (element-source element)
26   (or (eq-get element 'source)
27       '*unknown-source*))
28
29 ;;;;;;;;;;;;;;;;; Setitng Dependencies ;;;;;;;;;;;;;;;;;
30
31 (define (set-dependency! element dependency)
32   (eq-put! element 'dependency dependency))
33
34 (define (with-dependency dependency element)
35   (set-dependency! element dependency)
36   element)
37
38
39 (define (with-dependency-if-unknown dependency element)
40   (if (dependency-unknown? element)
41       (with-dependency dependency element)
42       element))
43 ;;;;;;;;;;;;;;;;; Unknown Dependencies ;;;;;;;;;;;;;;;;;
44
45 (define *unknown-dependency* (list '*unknown-dependency*))
46 (define (unknown-dependency? x)
47   (eq? x *unknown-dependency*))
48
49 (define (dependency-unknown? element)
50   (unknown-dependency? (element-dependency element)))
51
52 (define (dependency-known? (notp dependency-unknown?))
53 ;;;;;;;;;;;;;;;;; Accessing Dependencies ;;;;;;;;;;;;;;;;;
54
55 (define (element-dependency element)
56   (or (eq-get element 'dependency)
57       *unknown-dependency*))
58
59 ;;;;;;;;;;;;;;;;; Random Dependencies ;;;;;;;;;;;;;;;;;
60 (define (make-random-dependency tag)
61   (%make-random-dependency tag 0))
62
63 (define-record-type <random-dependency>
64   (%make-random-dependency tag num)
65   random-dependency?
66   (tag random-dependency-tag)

```

```

67 (num % random-dependency-num set-random-dependency-num!))
68
69 (define (random-dependency-num rd)
70   (let ((v (% random-dependency-num rd)))
71     (if (= v 0)
72         0
73         v)))
74
75 (define (print-random-dependency rd)
76   (list (random-dependency-tag rd)
77         (random-dependency-num rd)))
78 (defhandler print print-random-dependency random-dependency?)
79
80 (define (number-figure-random-dependencies! figure)
81   (define *random-dependency-num* 1)
82   (map (lambda (el)
83         (let ((dep (element-dependency el)))
84           (cond ((random-dependency? dep)
85                 (set-random-dependency-num!
86                  dep
87                  *random-dependency-num*)
88                 (set! *random-dependency-num*
89                       (+ *random-dependency-num* 1))))))
90       (figure-elements figure))
91   'done)
92
93 (define element-dependencies->list
94   (make-generic-operation
95    1 'element-dependencies->list
96    (lambda (x) x)))
97
98 (define (element-dependency->list el)
99   (element-dependencies->list
100    (element-dependency el)))
101
102 (defhandler element-dependencies->list
103   element-dependency->list
104   dependency-known?)
105
106 (defhandler element-dependencies->list
107   print-random-dependency
108   random-dependency?)
109
110 (defhandler element-dependencies->list
111   (lambda (l)
112     (map element-dependencies->list l))
113   list?)
114
115
116
117
118 ;;;;;;;;;;;;;; Formatting Dependencies ;;;;;;;;;;;;;;
119
120 (define (format-dependencies object)
121   (element-dependencies->list object))

```

Listing A.20: figure/randomness.scm

```

1  ;; randomness.scm --- Random creation of elements
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Random points, segments, etc. essential to system
7  ;; - Separated out animation / persistence across frames
8
9  ;; Future:
10 ;; - Better random support
11 ;; - Maybe separating out "definitions" (random square, etc.)
12
13 ;; Code:
14
15 ;;;;;;;;;;;;;;;;;; Base: Random Scalars ;;;;;;;;;;;;;;;;;;
16
17 (define (internal-internal-range min-v max-v)
18   (if (close-enuf? min-v max-v)
19       (error "range is too close for rand-range"
20             (list min-v max-v))
21       (let ((interval-size (max *machine-epsilon* (- max-v min-v))))
22         (persist-value (+ min-v (random (* 1.0 interval-size)))))))
23
24 (define (safe-internal-range min-v max-v)
25   (let ((interval-size (max 0 (- max-v min-v))))
26     (internal-internal-range
27      (+ min-v (* 0.1 interval-size))
28      (+ min-v (* 0.9 interval-size)))))
29
30 ;;;;;;;;;;;;;;;;;; Animated Ranges ;;;;;;;;;;;;;;;;;;
31
32 (define *wiggle-ratio* 0.15)
33
34 ;; Will return floats even if passed integers
35 ;; TODO: Rename to animated?
36 (define (rand-range min max)
37   (let* ((range-size (- max min))
38          (wiggle-amount (* range-size *wiggle-ratio*))
39          (v (internal-internal-range min (- max wiggle-amount))))
40     (animate-range v (+ v wiggle-amount))))
41
42 ;; Random Values - distances, angles
43
44 (define (rand-theta)
45   (rand-range 0 (* 2 pi)))
46
47 (define (rand-angle-measure)
48   (rand-range (* pi 0.05) (* .95 pi)))
49
50 (define (rand-obtuse-angle-measure)
51   (rand-range (* pi 0.55) (* .95 pi)))
52
53 (define (random-direction)
54   (let ((theta (rand-theta)))
55     (make-direction theta)))
56
57 ;;;;;;;;;;;;;;;;;; Random Points ;;;;;;;;;;;;;;;;;;
58
59 (define *point-wiggle-radius* 0.05)
60 (define (random-point)
61   (let ((x (internal-internal-range -0.8 0.8))
62         (y (internal-internal-range -0.8 0.8)))
63     (random-point-around (make-point x y))))
64
65 (define (random-point-around p)
66   (let ((x (point-x p))

```



```

67     (y (point-y p)))
68   (let ((theta (internal-rand-range 0 (* 2 pi)))
69         (d-theta (animate-range 0 (* 2 pi))))
70     (let ((dir (make-direction (+ theta d-theta))))
71       (with-dependency
72         (make-random-dependency 'random-point)
73         (add-to-point
74           (make-point x y)
75           (vec-from-direction-distance dir *point-wiggle-radius*))))))
76
77   ;;; TODO: Maybe separate out reflection about line?
78   (define (random-point-left-of-line line)
79     (let* ((p (random-point))
80            (d (signed-distance-to-line p line))
81            (v (rotate-vec-90
82                (unit-vec-from-direction
83                  (line-direction line)))))
84       (if (> d 0)
85         p
86         (add-to-point p (scale-vec v (* 2 (- d)))))))
87
88   (define (random-point-between-rays r1 r2)
89     (let ((offset-vec (sub-points (ray-endpoint r2)
90                                   (ray-endpoint r1))))
91       (let ((d1 (ray-direction r1))
92             (d2 (ray-direction r2)))
93         (let ((dir-difference (subtract-directions d2 d1))
94               (new-dir (add-to-direction
95                           d1
96                           (internal-rand-range 0.05 dir-difference))))
97           (random-point-around
98             (add-to-point
99               (add-to-point (ray-endpoint r1)
100                            (vec-from-direction-distance
101                              new-dir
102                              (internal-rand-range 0.05 0.9)))
103               (scale-vec offset-vec
104                (internal-rand-range 0.05 0.9)))))))))
105
106   (define (random-point-on-segment seg)
107     (let* ((p1 (segment-endpoint-1 seg))
108            (p2 (segment-endpoint-2 seg))
109            (t (rand-range 0.0 1.0))
110            (v (sub-points p2 p1)))
111       (add-to-point p1 (scale-vec v t)))
112
113   ;;; TODO: Fix this for new construction
114   (define (random-point-on-line l)
115     (let* ((p1 (line-p1 l))
116            (p2 (line-p2 l))
117            (seg (extend-to-max-segment p1 p2))
118            (sp1 (segment-endpoint-1 seg))
119            (sp2 (segment-endpoint-2 seg))
120            (t (rand-range 0.0 1.0))
121            (v (sub-points sp2 sp1)))
122       (add-to-point sp1 (scale-vec v t)))
123
124   (define (random-point-on-ray r)
125     (let* ((p1 (ray-endpoint r))
126            (dir (ray-direction r))
127            (p2 (add-to-point p1 (unit-vec-from-direction dir)))
128            (seg (ray-extend-to-max-segment p1 p2))
129            (sp1 (segment-endpoint-1 seg))
130            (sp2 (segment-endpoint-2 seg))
131            (t (rand-range 0.05 1.0))
132            (v (sub-points sp2 sp1)))
133       (add-to-point sp1 (scale-vec v t)))
134

```

```

135 (define (random-point-on-circle c)
136   (let ((dir (random-direction)))
137     (point-on-circle-in-direction c dir)))
138
139 (define (n-random-points-on-circle-ccw c n)
140   (let* ((thetas
141          (sort
142           (make-initialized-list n (lambda (i) (rand-theta)))
143           <)))
144     (map (lambda (theta)
145           (point-on-circle-in-direction c
146                                           (make-direction theta)))
147          thetas)))
148
149 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Random Linear Elements ;;;;;;;;;;;;;;;;;;
150
151 (define (random-line)
152   (let ((p (random-point)))
153     (with-dependency
154      (make-random-dependency 'random-line)
155      (random-line-through-point p))))
156
157 (define (random-segment)
158   (let ((p1 (random-point))
159         (p2 (random-point)))
160     (let ((seg (make-segment p1 p2)))
161       (set-segment-dependency!
162        seg
163        (make-random-dependency 'random-segment))
164       seg)))
165
166 (define (random-ray)
167   (let ((p (random-point)))
168     (random-ray-from-point p)))
169
170 (define (random-line-through-point p)
171   (let ((v (random-direction)))
172     (line-from-point-direction p v)))
173
174 (define (random-ray-from-point p)
175   (let ((v (random-direction)))
176     (ray-from-point-direction p v)))
177
178 (define (random-horizontal-line)
179   (let ((p (random-point))
180         (v (make-vec 1 0)))
181     (line-from-point-vec p v)))
182
183 (define (random-vertical-line)
184   (let ((p (random-point))
185         (v (make-vec 0 1)))
186     (line-from-point-vec p v)))
187
188 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Random Circle Elements ;;;;;;;;;;;;;;;;;;
189
190 (define (random-circle-radius circle)
191   (let ((center (circle-center circle))
192         (radius (circle-radius circle))
193         (angle (random-direction)))
194     (let ((radius-vec
195           (scale-vec (unit-vec-from-direction
196                      (random-direction))
197                     radius)))
198       (let ((radius-point (add-to-point center radius-vec)))
199         (make-segment center radius-point))))))
200
201 (define (random-circle)
202   (let ((pr1 (random-point))

```

```

203      (pr2 (random-point)))
204      (circle-from-points (midpoint pr1 pr2) pr1)))
205
206 (define (random-angle)
207   (let* ((v (random-point))
208          (d1 (random-direction))
209          (d2 (add-to-direction
210               d1
211               (rand-angle-measure))))
212     (smallest-angle (make-angle d1 v d2))))
213
214 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Random Polygons ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
215
216 (define (random-n-gon n)
217   (if (< n 3)
218       (error "n must be > 3")
219       (let* ((p1 (random-point))
220              (p2 (random-point))
221              (let ((ray2 (reverse-ray (ray-from-points p1 p2))))
222                  (let lp ((n-remaining (- n 2))
223                           (points (list p2 p1)))
224                    (if (= n-remaining 0)
225                        (apply polygon-from-points (reverse points))
226                        (lp (- n-remaining 1)
227                            (cons (random-point-between-rays
228                                   (reverse-ray (ray-from-points (car points)
229                                                                    (cadr points)))
230                                   ray2)
231                                points)))))))
232
233 (define (random-polygon)
234   (random-n-gon (+ 3 (random 5))))
235
236 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Random Triangles ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
237
238 (define (random-triangle)
239   (let* ((p1 (random-point))
240          (p2 (random-point))
241          (p3 (random-point-left-of-line (line-from-points p1 p2))))
242     (with-dependency
243       (make-random-dependency 'random-triangle)
244       (polygon-from-points p1 p2 p3))))
245
246 (define (random-equilateral-triangle)
247   (let* ((s1 (random-segment))
248          (s2 (rotate-about (segment-endpoint-1 s1)
249                             (/ pi 3)
250                             s1)))
251     (with-dependency
252       (make-random-dependency 'random-equilateral-triangle)
253       (polygon-from-points
254         (segment-endpoint-1 s1)
255         (segment-endpoint-2 s1)
256         (segment-endpoint-2 s2)))))
257
258 (define (random-isocetes-triangle)
259   (let* ((s1 (random-segment))
260          (base-angle (rand-angle-measure))
261          (s2 (rotate-about (segment-endpoint-1 s1)
262                             base-angle
263                             s1)))
264     (with-dependency
265       (make-random-dependency 'random-isocetes-triangle)
266       (polygon-from-points
267         (segment-endpoint-1 s1)
268         (segment-endpoint-2 s1)
269         (segment-endpoint-2 s2)))))
270

```

```

271 ;;;;;;;;;;;;;;; Random Quadrilaterals ;;;;;;;;;;;;;;;
272
273 (define (random-quadrilateral)
274   (with-dependency
275     (make-random-dependency 'random-quadrilateral)
276     (random-n-gon 4)))
277
278 (define (random-square)
279   (let* ((s1 (random-segment))
280          (p1 (segment-endpoint-1 s1))
281          (p2 (segment-endpoint-2 s1))
282          (p3 (rotate-about p2
283                            (- (/ pi 2))
284                            p1))
285          (p4 (rotate-about p1
286                            (/ pi 2)
287                            p2)))
288     (with-dependency
289       (make-random-dependency 'random-square)
290       (polygon-from-points p1 p2 p3 p4))))
291
292 (define (random-rectangle)
293   (let* ((r1 (random-ray))
294          (p1 (ray-endpoint r1))
295          (r2 (rotate-about (ray-endpoint r1)
296                            (/ pi 2)
297                            r1))
298          (p2 (random-point-on-ray r1))
299          (p4 (random-point-on-ray r2))
300          (p3 (add-to-point
301                p2
302                (sub-points p4 p1))))
303     (with-dependency
304       (make-random-dependency 'random-rectangle)
305       (polygon-from-points
306         p1 p2 p3 p4))))
307
308 (define (random-parallelogram)
309   (let* ((r1 (random-ray))
310          (p1 (ray-endpoint r1))
311          (r2 (rotate-about (ray-endpoint r1)
312                            (rand-angle-measure)
313                            r1))
314          (p2 (random-point-on-ray r1))
315          (p4 (random-point-on-ray r2))
316          (p3 (add-to-point
317                p2
318                (sub-points p4 p1))))
319     (with-dependency
320       (make-random-dependency 'random-parallelogram)
321       (polygon-from-points p1 p2 p3 p4))))
322
323 (define (random-kite)
324   (let* ((r1 (random-ray))
325          (p1 (ray-endpoint r1))
326          (r2 (rotate-about (ray-endpoint r1)
327                            (rand-obtuse-angle-measure)
328                            r1))
329          (p2 (random-point-on-ray r1))
330          (p4 (random-point-on-ray r2))
331          (p3 (reflect-about-line
332                (line-from-points p2 p4)
333                p1)))
334     (with-dependency
335       (make-random-dependency 'random-parallelogram)
336       (polygon-from-points p1 p2 p3 p4))))
337
338 (define (random-rhombus)

```

```

339 (let* ((s1 (random-segment))
340        (p1 (segment-endpoint-1 s1))
341        (p2 (segment-endpoint-2 s1))
342        (p4 (rotate-about p1
343                          (rand-angle-measure)
344                          p2))
345        (p3 (add-to-point
346             p2
347             (sub-points p4 p1))))
348 (with-dependency
349   (make-random-dependency 'random-rhombus)
350   (polygon-from-points p1 p2 p3 p4)))
351
352 (define (random-trapezoid)
353   (let* ((r1 (random-ray))
354          (r2 (translate-randomly r1))
355          (p1 (ray-endpoint r1))
356          (p2 (random-point-on-ray r1))
357          (p3 (random-point-on-ray r2))
358          (p4 (ray-endpoint r2)))
359     (with-dependency
360       (make-random-dependency 'random-trapezoid)
361       (polygon-from-points p1 p2 p3 p4))))

```

Listing A.21: figure/transforms.scm

```

1  ;; transforms.scm --- Transforms on Elements
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Generic transforms - rotation and translation
7  ;; - None mutate points, just return new copies.
8
9  ;; Future:
10 ;; - Translation or rotation to match something
11 ;; - Consider mutations?
12 ;; - Reflections?
13
14 ;; Code:
15
16 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Rotations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
17
18 (define (rotate-point-about rot-origin radians point)
19   (let ((v (sub-points point rot-origin)))
20     (let ((rotated-v (rotate-vec v radians)))
21       (add-to-point rot-origin rotated-v))))
22
23 (define (rotate-segment-about rot-origin radians seg)
24   (define (rotate-point p) (rotate-point-about rot-origin radians p))
25   (make-segment (rotate-point (segment-endpoint-1 seg))
26                 (rotate-point (segment-endpoint-2 seg))))
27
28 (define (rotate-ray-about rot-origin radians r)
29   (define (rotate-point p) (rotate-point-about rot-origin radians p))
30   (make-ray (rotate-point-about rot-origin radians (ray-endpoint r))
31             (add-to-direction (ray-direction r) radians)))
32
33 (define (rotate-line-about rot-origin radians l)
34   (make-line (rotate-point-about rot-origin radians (line-point l))
35             (add-to-direction (line-direction l) radians)))
36
37 (define rotate-about (make-generic-operation 3 'rotate-about))
38 (defhandler rotate-about rotate-point-about point? number? point?)
39 (defhandler rotate-about rotate-ray-about point? number? ray?)
40 (defhandler rotate-about rotate-segment-about point? number? segment?)
41 (defhandler rotate-about rotate-line-about point? number? line?)
42
43 (define (rotate-randomly-about p elt)
44   (let ((radians (rand-angle-measure)))
45     (rotate-about p radians elt)))
46
47 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Translations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
48
49 (define (translate-point-by vec point)
50   (add-to-point point vec))
51
52 (define (translate-segment-by vec segment)
53   (define (translate-point p) (translate-point-by vec p))
54   (make-segment (translate-point (segment-endpoint-1 seg))
55                 (translate-point (segment-endpoint-2 seg))))
56
57 (define (translate-ray-by vec r)
58   (make-ray (translate-point-by vec (ray-endpoint r))
59             (ray-direction r)))
60
61 (define (translate-line-by vec l)
62   (make-line (translate-point-by vec (line-point l))
63             (line-direction l)))
64
65 (define (translate-angle-by vec a)
66   (define (translate-point p) (translate-point-by vec p))

```

```

67 (make-angle (angle-arm-1 a)
68             (translate-point (angle-vertex a))
69             (angle-arm-2 a)))
70
71 (define translate-by (make-generic-operation 2 'rotate-about))
72 (defhandler translate-by translate-point-by vec? point?)
73 (defhandler translate-by translate-ray-by vec? ray?)
74 (defhandler translate-by translate-segment-by vec? segment?)
75 (defhandler translate-by translate-line-by vec? line?)
76 (defhandler translate-by translate-angle-by vec? angle?)
77
78 ;;; Reflections
79
80 (define (reflect-about-line line p)
81   (if (on-line? p line)
82       p
83       (let ((s (perpendicular-to line p)))
84         (let ((v (segment->vec s)))
85           (add-to-point
86             p
87             (scale-vec v 2))))))
88
89 ;;;;;;;;;;;;;; Random Translation ;;;;;;;;;;;;;;
90
91 (define (translate-randomly-along-line l elt)
92   (let* ((vec (unit-vec-from-direction (line->direction l)))
93         (scaled-vec (scale-vec vec (rand-range 0.5 1.5))))
94     (translate-by vec elt)))
95
96 (define (translate-randomly elt)
97   (let ((vec (rand-translation-vec-for elt)))
98     (translate-by vec elt)))
99
100 (define (rand-translation-vec-for-point p1)
101   (let ((p2 (random-point)))
102     (sub-points p2 p1)))
103
104 (define (rand-translation-vec-for-segment seg)
105   (rand-translation-vec-for-point (segment-endpoint-1 seg)))
106
107 (define (rand-translation-vec-for-ray r)
108   (rand-translation-vec-for-point (ray-endpoint r)))
109
110 (define (rand-translation-vec-for-line l)
111   (rand-translation-vec-for-point (line-point l)))
112
113 (define rand-translation-vec-for
114   (make-generic-operation 1 'rand-translation-vec-for))
115 (defhandler rand-translation-vec-for
116   rand-translation-vec-for-point point?)
117 (defhandler rand-translation-vec-for
118   rand-translation-vec-for-segment segment?)
119 (defhandler rand-translation-vec-for
120   rand-translation-vec-for-ray ray?)
121 (defhandler rand-translation-vec-for
122   rand-translation-vec-for-line line?)

```

Listing A.22: perception/load.scm

```
1  ;; load.scm -- Load perception
2  (for-each (lambda (f) (load f))
3            '("relationship"
4              "observation"
5              "analyzer"))
```


Listing A.23: perception/observation.scm

```

1  ;; observation.scm -- observed relationships
2
3  ;; Commentary:
4
5  ;; Future:
6  ;; - Observation equality is more complicated!
7
8  ;; Code:
9
10 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Observation ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
11
12 (define-record-type <observation>
13   (make-observation relationship args)
14   observation?
15   (relationship observation-relationship)
16   (args observation-args))
17
18 (define (observation-equal? obs1 obs2)
19   (equal? (print-observation obs1)
20           (print-observation obs2)))
21
22 (define (print-observation obs)
23   (cons
24     (print (observation-relationship obs))
25     (map print (observation-args obs))))
26
27 (defhandler print print-observation observation?)
28
29 (define (print-observations obs-list)
30   (map print-observation obs-list))
31
32 (define (observation-with-premises obs)
33   (cons (observation-relationship obs)
34         (map element-dependencies->list (observation-args obs))))

```

Listing A.24: perception/analyzer.scm

```

1  ;; analyzer.scm --- Tools for analyzing Diagram
2
3  ;; Commentary
4
5  ;; Ideas:
6  ;; - Analyze figure to dermine properties "beyond coincidence"
7  ;; - Use dependency structure to eliminate some obvious examples.
8
9  ;; Future:
10 ;; - Add More "interesting properties"
11 ;; - Create storage for learned properties.
12 ;; - Output format, add names
13 ;; - Separate "discovered" from old properties.
14
15 ;; Code:
16
17 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Main Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
18
19 (define (all-observations figure)
20   (analyze figure))
21
22 (define (analyze-figure figure)
23   (all-observations figure))
24
25 ;; Given a figure, report what's interesting
26 (define (all-observations figure)
27   (number-figure-random-dependencies! figure)
28   (let* ((points (figure-points figure))
29          (angles (figure-angles figure))
30          (implied-segments '() ; (point-pairs->segments (all-pairs points))
31          )
32          (linear-elements (append
33                            (figure-linear-elements figure)
34                            implied-segments))
35          (segments (append (figure-segments figure)
36                             implied-segments)))
37   (append
38     (extract-relationships points
39                           (list concurrent-points-relationship
40                                concentric-relationship
41                                concentric-with-center-relationship))
42     (extract-relationships segments
43                           (list equal-length-relationship))
44     (extract-relationships angles
45                           (list equal-angle-relationship
46                                supplementary-angles-relationship
47                                complementary-angles-relationship))
48     (extract-relationships linear-elements
49                           (list parallel-relationship
50                                concurrent-relationship
51                                perpendicular-relationship
52                                ))))
53
54 (define (extract-relationships elements relationships)
55   (append-map (lambda (r)
56                 (extract-relationship elements r))
57               relationships))
58
59 (define (extract-relationship elements relationship)
60   (map (lambda (tuple)
61         (make-observation relationship tuple))
62        (report-n-wise
63          (relationship-arity relationship)
64          (relationship-predicate relationship)
65          elements)))
66

```

```

67 ;;;;;;;;;;;;;;;;;;;;;;;;;; Interesting Observations ;;;;;;;;;;;;;;;;;;;;;;;;;;
68
69 (define (interesting-observations figure-proc)
70   (set! *obvious-observations* '())
71   (let ((all-obs (all-observations (figure-proc))))
72     (pprint *obvious-observations*)
73     (pprint all-obs)
74     (set-difference all-obs *obvious-observations*
75                     observation-equal?)))
76
77 (define *obvious-observations* #f)
78
79 (define (save-obvious-observation! obs)
80   (if *obvious-observations*
81       (begin
82         (pprint obs)
83         (set! *obvious-observations*
84               (cons obs
85                     *obvious-observations*))))))
86
87 ;;;;;;;;;;;;;;;;;;;;;;;;;; Cross products, pairs ;;;;;;;;;;;;;;;;;;;;;;;;;;
88
89 ;;; General procedres for generating pairs
90 (define (all-pairs elements)
91   (all-n-tuples 2 elements))
92
93 (define (all-n-tuples n elements)
94   (cond ((zero? n) '())
95         ((< (length elements) n) '())
96         (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))))))))))
97
98 ;;;;;;;;;;;;;;;;;;;;;;;;;; Obvious Segments ;;;;;;;;;;;;;;;;;;;;;;;;;;
99
100 (define (segment-for-endpoint p1)
101   (let ((dep (element-dependency p1)))
102     (and dep
103          (or (and (eq? (car dep) 'segment-endpoint-1)
104                  (cadr dep))
105              (and (eq? (car dep) 'segment-endpoint-2)
106                  (cadr dep))))))
107
108 (define (derived-from-same-segment? p1 p2)
109   (and
110    (segment-for-endpoint p1)
111    (segment-for-endpoint p2)
112    (eq? (segment-for-endpoint p1)
113          (segment-for-endpoint p2))))
114
115 (define (polygon-for-point p1)
116   (let ((dep (element-dependency p1)))
117     (and dep
118          (and (eq? (car dep) 'polygon-point)
119               (cons (caddr dep)
120                     (cadr dep))))))
121
122 (define (adjacent-in-same-polygon? p1 p2)

```

```

135 (let ((poly1 (polygon-for-point p1))
136       (poly2 (polygon-for-point p2)))
137   (and poly1 poly2
138     (eq? (car poly1) (car poly2))
139     (or (= (abs (- (cdr poly1)
140                   (cdr poly2)))
141         1)
142         (and (= (cdr poly1) 0)
143               (= (cdr poly2) 3))
144         (and (= (cdr poly1) 3)
145               (= (cdr poly2) 0))))))
146
147 (define (point-pairs->segments ppairs)
148   (filter (lambda (segment) segment)
149     (map (lambda (point-pair)
150         (let ((p1 (car point-pair))
151               (p2 (cadr point-pair)))
152           (and (not (point-equal? p1 p2))
153               (not (derived-from-same-segment? p1 p2))
154               (not (adjacent-in-same-polygon? p1 p2))
155               (make-auxiliary-segment
156                 (car point-pair)
157                 (cadr point-pair)))))) ; TODO: Name segment
158     ppairs)))
159
160 ;;;;;;;;;;;;;; Dealing with pairs ;;;;;;;;;;;;;;
161
162 ;; Check for pairwise equality
163 (define ((nary-predicate n predicate) tuple)
164   (apply predicate tuple))
165
166 ;; Merges "connected-components" of pairs
167 (define (merge-pair-groups elements pairs)
168   (let ((i 0)
169         (group-ids (make-key-weak-eq-hash-table))
170         (group-elements (make-key-weak-eq-hash-table))) ; Map from pair
171     (for-each (lambda (pair)
172         (let ((first (car pair))
173               (second (cadr pair)))
174           (let ((group-id-1 (hash-table/get group-ids first i))
175                 (group-id-2 (hash-table/get group-ids second i)))
176             (cond ((and (= group-id-1 i)
177                          (= group-id-2 i))
178                   ;; Both new, new groups:
179                   (hash-table/put! group-ids first group-id-1)
180                   (hash-table/put! group-ids second group-id-1))
181                 ((= group-id-1 i)
182                  (hash-table/put! group-ids first group-id-2))
183                 ((= group-id-2 i)
184                  (hash-table/put! group-ids second group-id-1)))
185             (set! i (+ i 1))))
186       pairs)
187     (for-each (lambda (elt)
188         (hash-table/append group-elements
189                           (hash-table/get group-ids elt 'invalid)
190                           elt))
191       elements)
192     (hash-table/remove! group-elements 'invalid)
193     (hash-table/datum-list group-elements)))
194
195 (define (report-n-wise n predicate elements)
196   (let ((tuples (all-n-tuples n elements)))
197     (filter (nary-predicate n predicate) tuples)))
198
199 ;;;;;;;;;;;;;; Results: ;;;;;;;;;;;;;;
200
201 (define (make-analysis-collector)
202   (make-equal-hash-table))

```

```
203
204 (define (save-results results data-table)
205   (hash-table/put! data-table results
206     (+ 1 (hash-table/get data-table results 0))))
207
208 (define (print-analysis-results data-table)
209   (hash-table/for-each
210     data-table
211     (lambda (k v)
212       (pprint (list v (cons 'discovered k))))))
```

Listing A.25: graphics/load.scm

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

Listing A.26: 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.27: 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*))))
76     (m (segment-midpoint segment)))
77   (let ((label-point (add-to-point m v)))
78     (canvas-draw-text canvas
79         (point-x label-point)
80         (point-y label-point)
81         (symbol->string (element-name segment))))))
82
83 (define (draw-line canvas line)
84   (let ((p1 (line-p1 line)))
85     (let ((p2 (add-to-point
86       p1
87       (unit-vec-from-direction (line-direction line))))))
88       (draw-segment canvas (extend-to-max-segment p1 p2))))
89
90 (define (draw-ray canvas ray)
91   (let ((p1 (ray-endpoint ray)))
92     (let ((p2 (add-to-point
93       p1
94       (unit-vec-from-direction (ray-direction ray))))))
95       (draw-segment canvas (ray-extend-to-max-segment p1 p2))))
96
97 (define (draw-circle canvas c)
98   (let ((center (circle-center c))
99     (radius (circle-radius c)))
100     (canvas-draw-circle canvas
101         (point-x center)
102         (point-y center)
103         radius)))
104
105 (define *angle-mark-radius* 0.05)
106 (define (draw-angle canvas a)
107   (let* ((vertex (angle-vertex a))
108     (d1 (angle-arm-1 a))
109     (d2 (angle-arm-2 a))
110     (angle-start (direction-theta d2))
111     (angle-end (direction-theta d1)))
112     (canvas-draw-arc canvas
113         (point-x vertex)
114         (point-y vertex)
115         *angle-mark-radius*
116         angle-start
117         angle-end)))
118
119 ;;; Add to generic operations
120
121 (add-to-draw-element! point? draw-point)
122 (add-to-draw-element! segment? draw-segment)
123 (add-to-draw-element! circle? draw-circle)
124 (add-to-draw-element! angle? draw-angle)
125 (add-to-draw-element! line? draw-line)
126 (add-to-draw-element! ray? draw-ray)
127
128 (add-to-draw-label! point? draw-point-label)
129
130 ;;; Canvas for x-graphics
131
132 (define (x-graphics) (make-graphics-device 'x))
133
134 (define (canvas)

```

```

135 (let ((g (x-graphics)))
136   (graphics-enable-buffering g)
137   (list 'canvas g)))
138
139 (define (canvas-g canvas)
140   (cadr canvas))
141
142 (define (canvas? x)
143   (and (pair? x)
144         (eq? (car x 'canvas))))
145
146 (define (clear-canvas canvas)
147   (graphics-clear (canvas-g canvas)))
148
149 (define (canvas-draw-circle canvas x y radius)
150   (graphics-operation (canvas-g canvas)
151                       'draw-circle
152                       x y radius))
153
154 (define (canvas-draw-text canvas x y text)
155   (graphics-draw-text (canvas-g canvas) x y text))
156
157 (define (canvas-draw-arc canvas x y radius
158                       angle-start angle-end)
159   (let ((angle-sweep
160         (fix-angle-0-2pi (- angle-end
161                               angle-start))))
162     (graphics-operation (canvas-g canvas)
163                         'draw-arc
164                         x y radius radius
165                         (rad->deg angle-start)
166                         (rad->deg angle-sweep)
167                         #f)))
168
169 (define (canvas-fill-circle canvas x y radius)
170   (graphics-operation (canvas-g canvas)
171                       'fill-circle
172                       x y radius))
173
174 (define (canvas-draw-line canvas x1 y1 x2 y2)
175   (graphics-draw-line (canvas-g canvas)
176                       x1 y1
177                       x2 y2))
178
179 (define (canvas-set-color canvas color)
180   (graphics-operation (canvas-g canvas) 'set-foreground-color color)
181 )

```

Listing A.28: manipulate/load.scm

```
1 ;;; load.scm -- Load manipulate
2 (for-each (lambda (f) (load f))
3           '("linkages"
4             "region"
5             "constraints"
6             "topology"
7             "mechanism"
8             "main"))
```

Listing A.29: manipulate/linkages.scm

```

1  ;; linkages.scm --- Bar/Joint propagators between directions and coordinates
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Join "Identify" bars and joints to build mechanism
7  ;;   versions of diagrams
8  ;; - Use propagator system to deal with partial information
9  ;; - Used Regions for partial info about points,
10 ;; - Direction Intervals for partial info about joint directions.
11
12 ;; Future:
13 ;; - Other Linkages?
14 ;; - Draw partially assembled linkages
15
16 ;;; Example:
17
18 #|
19 (let* ((s1 (m:make-bar))
20        (s2 (m:make-bar))
21        (j (m:make-joint)))
22   (m:instantiate (m:joint-theta j) (/ pi 2) 'theta)
23   (c:id (m:bar-length s1)
24         (m:bar-length s2))
25   (m:instantiate-point (m:bar-p2 s1) 4 0 'bar-2-endpoint)
26   (m:instantiate-point (m:bar-p1 s1) 2 -2 'bar-2-endpoint)
27   (m:identify-out-of-arm-1 j s1)
28   (m:identify-out-of-arm-2 j s2)
29   (run)
30   (m:examine-point (m:bar-p2 s2)))
31 |#
32
33 ;;; Code:
34
35 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; TMS Interfaces ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
36
37 (define (m:instantiate cell value premise)
38   (add-content cell
39     (make-tms (contingent value (list premise)))))
40
41 (define (m:examine-cell cell)
42   (let ((v (content cell)))
43     (cond ((nothing? v) v)
44           ((tms? v)
45            (contingent-info (tms-query v)))
46           (else v))))
47
48 (defhandler print
49   (lambda (cell) (print (m:examine-cell cell))))
50 cell?)
51
52 (define (m:contradictory? cell)
53   (contradictory? (m:examine-cell cell)))
54
55 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Reversing directions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
56
57 (define m:reverse-direction
58   (make-generic-operation 1 'm:reverse-direction))
59 (defhandler m:reverse-direction
60   reverse-direction direction?)
61 (defhandler m:reverse-direction
62   reverse-direction-interval direction-interval?)
63
64 (propagatify m:reverse-direction)
65
66 (define (ce:reverse-direction input-cell)

```

```

67 (let-cells (output-cell)
68   (name! output-cell (symbol 'reverse- (name input-cell)))
69   (p:m:reverse-direction input-cell output-cell)
70   (p:m:reverse-direction output-cell input-cell)
71   output-cell))
72
73 ;;;;;;;;;;;;;; Adding to directions ;;;;;;;;;;;;;;
74
75 (define (m:add-interval-to-direction d i)
76   (if (empty-interval? i)
77       (error "Cannot add empty interval to direction"))
78   (make-direction-interval-from-start-dir-and-size
79     (add-to-direction d (interval-low i))
80     (- (interval-high i)
81        (interval-low i))))
82
83 (define (m:add-interval-to-standard-direction-interval di i)
84   (if (empty-interval? i)
85       (error "Cannot add empty interval to direction"))
86   (let ((di-size (direction-interval-size di))
87         (i-size (- (interval-high i)
88                     (interval-low i)))
89         (di-start (direction-interval-start di)))
90     (make-direction-interval-from-start-dir-and-size
91       (add-to-direction di-start (interval-low i))
92       (+ di-size i-size))))
93
94 (define (m:add-interval-to-full-circle-direction-interval fcdi i)
95   (if (empty-interval? i)
96       (error "Cannot add empty interval to direction"))
97   fcdi)
98
99 (define (m:add-interval-to-invalid-direction-interval fcdi i)
100   (if (empty-interval? i)
101       (error "Cannot add empty interval to direction"))
102   (error "Cannot add to invalid direction in"))
103
104 (define m:add-to-direction
105   (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 ;;; TODO: Support Intervals for thetas?
168 (defhandler m:subtract-directions
169   (lambda (di1 di2)
170     nothing)
171   direction-interval? direction-interval?)
172
173 (defhandler m:subtract-directions
174   m:standard-direction-interval-minus-direction
175   standard-direction-interval? direction?)
176
177 (defhandler m:subtract-directions
178   m:full-circle-direction-interval-minus-direction
179   full-circle-direction-interval? direction?)
180
181 (defhandler m:subtract-directions
182   m:direction-minus-standard-direction-interval
183   direction? standard-direction-interval?)
184
185 (defhandler m:subtract-directions
186   m:direction-minus-full-circle-direction-interval
187   direction? full-circle-direction-interval?)
188
189 (propagatify m:subtract-directions)
190
191 ;;;;;;;;;;;;;; Vec ;;;;;;;;;;;;;;
192 (define-record-type <m:vec>
193   (%m:make-vec dx dy length direction)
194   m:vec?
195   (dx m:vec-dx)
196   (dy m:vec-dy)
197   (length m:vec-length)
198   (direction m:vec-direction))
199
200
201 ;;; Allocate and wire up the cells in a vec
202 (define (m:make-vec vec-id)

```

```

203 (let-cells (dx dy length direction)
204   (name! dx (symbol vec-id '-dx))
205   (name! dy (symbol vec-id '-dy))
206   (name! length (symbol vec-id '-len))
207   (name! direction (symbol vec-id '-dir))
208
209   (p:make-direction
210     (e:atan2 dy dx) direction)
211   (p:sqrt (e:+ (e:square dx)
212               (e:square dy))
213     length)
214   (p:* length (e:direction-cos direction) dx)
215   (p:* length (e:direction-sin direction) dy)
216   (%m:make-vec dx dy length direction)))
217
218 (define (m:print-vec v)
219   `(m:vec (,(print (m:vec-dx v))
220                ,(print (m:vec-dy v)))
221            ,(print (m:vec-length v))
222            ,(print (m:vec-direction v))))
223
224 (defhandler print m:print-vec m:vec?)
225
226 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Point ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
227 (define-record-type <m:point>
228   (%m:make-point x y region)
229   m:point?
230   (x m:point-x)
231   (y m:point-y)
232   (region m:point-region))
233
234 ;;; Allocate cells for a point
235 (define (m:make-point id)
236   (let-cells (x y region)
237     (name! x (symbol id '-x))
238     (name! y (symbol id '-y))
239     (name! region (symbol id '-region))
240     (p:m:x-y->region x y region)
241     (p:m:region->x region x)
242     (p:m:region->y region y)
243     (%m:make-point x y region)))
244
245 (define (m:x-y->region x y)
246   (m:make-singular-point-set (make-point x y)))
247
248 (propagatify m:x-y->region)
249
250 (define (m:region->x region)
251   (if (m:singular-point-set? region)
252       (point-x (m:singular-point-set-point region))
253       nothing))
254
255 (define (m:region->y region)
256   (if (m:singular-point-set? region)
257       (point-y (m:singular-point-set-point region))
258       nothing))
259
260 (propagatify m:region->x)
261 (propagatify m:region->y)
262
263 (define (m:instantiate-point p x y premise)
264   (m:instantiate (m:point-x p)
265                   x premise)
266   (m:instantiate (m:point-y p)
267                   y premise)
268   (m:instantiate (m:point-region p)
269                   (m:make-singular-point-set (make-point x y))
270                   premise))

```

```

271
272 (define (m:examine-point p)
273   (list 'm:point
274         (m:examine-cell (m:point-x p))
275         (m:examine-cell (m:point-y p))))
276
277 (define (m:print-point p)
278   `(m:point ,(print (m:point-x p))
279             ,(print (m:point-y p))
280             ,(print (m:point-region p))))
281
282 (defhandler print m:print-point m:point?)
283
284 ;;; Set p1 and p2 to be equal
285 (define (m:identify-points p1 p2)
286   (for-each (lambda (getter)
287               (c:id (getter p1)
288                     (getter p2)))
289             (list m:point-x m:point-y m:point-region)))
290
291 ;;; Bar ;;;
292
293 (define-record-type <m:bar>
294   (%m:make-bar p1 p2 vec)
295   m:bar?
296   (p1 m:bar-p1)
297   (p2 m:bar-p2)
298   (vec m:bar-vec))
299
300 (define (m:bar-direction bar)
301   (m:vec-direction (m:bar-vec bar)))
302
303 (define (m:bar-length bar)
304   (m:vec-length (m:bar-vec bar)))
305
306 (define (m:print-bar b)
307   `(m:bar
308     ,(print (m:bar-name b))
309     ,(print (m:bar-p1 b))
310     ,(print (m:bar-p2 b))
311     ,(print (m:bar-vec b))))
312
313 (defhandler print m:print-bar m:bar?)
314
315 ;;; Allocate cells and wire up a bar
316 (define (m:make-bar bar-id)
317   (let ((bar-key (m:make-bar-name-key bar-id)))
318     (let ((p1 (m:make-point (symbol bar-key '-p1))
319                             (p2 (m:make-point (symbol bar-key '-p2))))
320           (name! p1 (symbol bar-key '-p1))
321           (name! p2 (symbol bar-key '-p2))
322           (let ((v (m:make-vec bar-key)))
323             (c:+ (m:point-x p1)
324                   (m:vec-dx v)
325                   (m:point-x p2))
326             (c:+ (m:point-y p1)
327                   (m:vec-dy v)
328                   (m:point-y p2))
329             (let ((bar (%m:make-bar p1 p2 v)))
330               (m:p1->p2-bar-propagator p1 p2 bar)
331               (m:p2->p1-bar-propagator p2 p1 bar)
332               bar))))))
333
334 ;;; TODO: Combine p1->p2 / p2->p1
335 (define (m:x-y-direction->region px py direction)
336   (if (direction? direction)
337       (let ((vertex (make-point px py)))
338         (m:make-ray vertex direction))

```



```

339     nothing))
340
341 (propagatify m:x-y-direction->region)
342
343 (define (m:x-y-length-di->region px py length dir-interval)
344   (if (direction-interval? dir-interval)
345     (let ((vertex (make-point px py)))
346       (m:make-arc vertex length dir-interval))
347     nothing))
348 (propagatify m:x-y-length-di->region)
349
350 (define (m:region-length-direction->region pr length dir)
351   (if (direction-interval? dir)
352     nothing
353     (m:translate-region
354      pr
355      (vec-from-direction-distance dir length))))
356 (propagatify m:region-length-direction->region)
357
358
359 (define (m:p1->p2-bar-propagator p1 p2 bar)
360   (let ((p1x (m:point-x p1))
361         (p1y (m:point-y p1))
362         (p1r (m:point-region p1))
363         (p2r (m:point-region p2))
364         (length (m:bar-length bar))
365         (dir (m:bar-direction bar)))
366     (p:m:x-y-direction->region p1x p1y dir p2r)
367     (p:m:x-y-length-di->region p1x p1y length dir p2r)
368     (p:m:region-length-direction->region p1r length dir p2r)))
369
370 (define (m:p2->p1-bar-propagator p2 p1 bar)
371   (let ((p2x (m:point-x p2))
372         (p2y (m:point-y p2))
373         (p1r (m:point-region p1))
374         (p2r (m:point-region p2))
375         (length (m:bar-length bar))
376         (dir (m:bar-direction bar)))
377     (p:m:x-y-direction->region p2x p2y (ce:reverse-direction dir) p1r)
378     (p:m:x-y-length-di->region p2x p2y length (ce:reverse-direction dir) p1r)
379     (p:m:region-length-direction->region
380      p2r length (ce:reverse-direction dir) p1r)))
381
382 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Joint ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
383 ;;; Direction-2 is counter-clockwise from direction-1 by theta
384 (define-record-type <m:joint>
385   (%m:make-joint vertex dir-1 dir-2 theta)
386   m:joint?
387   (vertex m:joint-vertex)
388   (dir-1 m:joint-dir-1)
389   (dir-2 m:joint-dir-2)
390   (theta m:joint-theta))
391
392 (define *max-joint-swing* pi)
393
394 (define (m:make-joint joint-id)
395   (let ((joint-key (m:make-joint-name-key joint-id)))
396     (let ((vertex (m:make-point (symbol joint-key '-vertex))))
397       (let-cells (dir-1 dir-2 theta)
398         (name! dir-1 (symbol joint-key '-dir-1))
399         (name! dir-2 (symbol joint-key '-dir-2))
400         (name! theta (symbol joint-key '-theta))
401         (name! vertex (symbol joint-key '-vertex))
402         (p:m:add-to-direction
403          dir-1 theta dir-2)
404         (p:m:add-to-direction
405          dir-2 (e:negate theta) dir-1)
406         (p:m:subtract-directions

```

```

407         dir-2 dir-1
408         theta)
409         (m:instantiate theta (make-interval 0 *max-joint-swing*) 'theta)
410         (%m:make-joint vertex dir-1 dir-2 theta))))))
411
412 (define (m:print-joint j)
413   `(m:joint
414     ,(print (m:joint-name j))
415     ,(print (m:joint-dir-1 j))
416     ,(print (m:joint-vertex j))
417     ,(print (m:joint-dir-2 j))
418     ,(print (m:joint-theta j))))
419
420 (defhandler print m:print-joint m:joint?)
421
422 ;;; T00D: Abstract?
423 (define (m:identify-out-of-arm-1 joint bar)
424   (m:set-endpoint-1 bar joint)
425   (m:set-joint-arm-1 joint bar)
426   (m:identify-points (m:joint-vertex joint)
427                       (m:bar-p1 bar))
428   (c:id (m:joint-dir-1 joint)
429         (m:bar-direction bar)))
430
431 (define (m:identify-out-of-arm-2 joint bar)
432   (m:set-endpoint-1 bar joint)
433   (m:set-joint-arm-2 joint bar)
434   (m:identify-points (m:joint-vertex joint)
435                       (m:bar-p1 bar))
436   (c:id (m:joint-dir-2 joint)
437         (m:bar-direction bar)))
438
439 (define (m:identify-into-arm-1 joint bar)
440   (m:set-endpoint-2 bar joint)
441   (m:set-joint-arm-1 joint bar)
442   (m:identify-points (m:joint-vertex joint)
443                       (m:bar-p2 bar))
444   (c:id (ce:reverse-direction (m:joint-dir-1 joint))
445         (m:bar-direction bar)))
446
447 (define (m:identify-into-arm-2 joint bar)
448   (m:set-endpoint-2 bar joint)
449   (m:set-joint-arm-2 joint bar)
450   (m:identify-points (m:joint-vertex joint)
451                       (m:bar-p2 bar))
452   (c:id (ce:reverse-direction (m:joint-dir-2 joint))
453         (m:bar-direction bar)))
454
455 ;;;;;;;;;;;;;; Storing Adjacencies ;;;;;;;;;;;;;;
456
457 (define (m:set-endpoint-1 bar joint)
458   (eq-append! bar 'm:bar-endpoints-1 joint))
459
460 (define (m:bar-endpoints-1 bar)
461   (or (eq-get bar 'm:bar-endpoints-1)
462       '()))
463
464 (define (m:set-endpoint-2 bar joint)
465   (eq-append! bar 'm:bar-endpoints-2 joint))
466
467 (define (m:bar-endpoints-2 bar)
468   (or (eq-get bar 'm:bar-endpoints-2)
469       '()))
470
471 (define (m:set-joint-arm-1 joint bar)
472   (eq-put! joint 'm:joint-arm-1 bar))
473
474 (define (m:joint-arm-1 joint)

```

```

475 (eq-get joint 'm:joint-arm-1))
476
477 (define (m:set-joint-arm-2 joint bar)
478   (eq-put! joint 'm:joint-arm-2 bar))
479
480 (define (m:joint-arm-2 joint)
481   (eq-get joint 'm:joint-arm-2))
482
483 ;;;;;;;;;;;;;;;;;;;;;;;;;; Named Linkages ;;;;;;;;;;;;;;;;;;;;;;;;;;
484
485 (define (m:make-bar-name-key bar-id)
486   (symbol 'm:bar:
487     (m:bar-id-p1-name bar-id) ':
488     (m:bar-id-p2-name bar-id)))
489
490 (define (m:make-joint-name-key joint-id)
491   (symbol 'm:joint:
492     (m:joint-id-dir-1-name joint-id) ':
493     (m:joint-id-vertex-name joint-id) ':
494     (m:joint-id-dir-2-name joint-id)))
495
496 (define (m:name-element! element name)
497   (eq-put! element 'm:name name))
498
499 (define (m:element-name element)
500   (or (eq-get element 'm:name)
501       '*unnamed*))
502
503 (define (m:make-named-bar p1-name p2-name)
504   (let ((bar (m:make-bar (m:bar p1-name p2-name))))
505     (m:name-element! (m:bar-p1 bar) p1-name)
506     (m:name-element! (m:bar-p2 bar) p2-name)
507     bar))
508
509 (define (m:bar-name bar)
510   (m:bar
511     (m:element-name (m:bar-p1 bar))
512     (m:element-name (m:bar-p2 bar))))
513
514 (define (m:bars-name-equivalent? bar-1 bar-2)
515   (or (m:bar-id-equal?
516       (m:bar-name bar-1)
517       (m:bar-name bar-2))
518       (m:bar-id-equal?
519       (m:bar-name bar-1)
520       (m:reverse-bar-id (m:bar-name bar-2)))))
521
522 (define (m:bar-p1-name bar)
523   (m:element-name (m:bar-p1 bar)))
524
525 (define (m:bar-p2-name bar)
526   (m:element-name (m:bar-p2 bar)))
527
528 (define (m:make-named-joint arm-1-name vertex-name arm-2-name)
529   (let ((joint-id (m:joint arm-1-name
530                           vertex-name
531                           arm-2-name)))
532     (let ((joint (m:make-joint joint-id)))
533       (m:name-element! (m:joint-dir-1 joint) arm-1-name)
534       (m:name-element! (m:joint-vertex joint) vertex-name)
535       (m:name-element! (m:joint-dir-2 joint) arm-2-name)
536       joint)))
537
538 (define (m:joint-name joint)
539   (m:joint
540     (m:joint-dir-1-name joint)
541     (m:joint-vertex-name joint)
542     (m:joint-dir-2-name joint)))

```

```

543
544 (define (m:joint-vertex-name joint)
545   (m:element-name (m:joint-vertex joint)))
546
547 (define (m:joint-dir-1-name joint)
548   (m:element-name (m:joint-dir-1 joint)))
549
550 (define (m:joint-dir-2-name joint)
551   (m:element-name (m:joint-dir-2 joint)))
552
553 ;;;;;;;;;;;;;;;;;; Symbolic Bar / Joint Identifiers ;;;;;;;;;;;;;;;;;;
554
555 ;; Maybe Move?
556
557 (define-record-type <m:bar-id>
558   (%m:make-bar-id p1-name p2-name)
559   m:bar-id?
560   (p1-name m:bar-id-p1-name)
561   (p2-name m:bar-id-p2-name))
562
563 (define (m:bar-id-equal? bar-id-1 bar-id-2)
564   (and (eq? (m:bar-id-p1-name bar-id-1)
565             (m:bar-id-p1-name bar-id-2))
566        (eq? (m:bar-id-p2-name bar-id-1)
567             (m:bar-id-p2-name bar-id-2))))
568
569 (define (m:bar p1-name p2-name)
570   (%m:make-bar-id p1-name p2-name))
571
572 (defhandler print m:make-bar-name-key m:bar-id?)
573
574 (define (m:reverse-bar-id bar-id)
575   (%m:make-bar-id (m:bar-id-p2-name bar-id)
576                   (m:bar-id-p1-name bar-id)))
577
578 ;; Joints:
579
580 (define-record-type <m:joint-vertex-id>
581   (%m:make-joint-verex-id vertex-name)
582   m:joint-vertex-id?
583   (vertex-name m:joint-vertex-id-name))
584
585 (define-record-type <m:joint-id>
586   (%m:make-joint-id dir-1-name vertex-name dir-2-name)
587   m:joint-id?
588   (dir-1-name m:joint-id-dir-1-name)
589   (vertex-name m:joint-id-vertex-name)
590   (dir-2-name m:joint-id-dir-2-name))
591
592 (defhandler print m:make-joint-name-key m:joint-id?)
593
594 (define (m:joint arg1 . rest)
595   (cond ((null? rest)
596         (%m:make-joint-verex-id arg1))
597         ((= 2 (length rest))
598          (%m:make-joint-id arg1 (car rest) (cadr rest)))
599         (else
600          (error "m:joint was called with the wrong number of arguments."))))
601
602 ;;;;;;;;;;;;;;;;;; Tables and Accessors for named linkages ;;;;;;;;;;;;;;;;;;
603 (define (m:make-bars-by-name-table bars)
604   (let ((table (make-key-weak-eqv-hash-table)))
605     (for-each (lambda (bar)
606                 (let ((key (m:make-bar-name-key (m:bar-name bar))))
607                   (if (hash-table/get table key #f)
608                       (error "Bar key already in bar name table" key))
609                   (hash-table/put! table key bar)))
610               bars)
611     table)

```

```

611     table))
612
613 ;;; Unordered
614 (define (m:find-bar-by-id table bar-id)
615   (or (hash-table/get table
616         (m:make-bar-name-key bar-id)
617         #f)
618       (hash-table/get table
619         (m:make-bar-name-key (m:reverse-bar-id bar-id))
620         #f)))
621
622 ;;; Joints:
623
624 (define (m:make-joints-by-vertex-name-table joints)
625   (let ((table (make-key-weak-eq-hash-table)))
626     (for-each
627       (lambda (joint)
628         (let ((key (m:joint-vertex-name joint)))
629           (hash-table/put!
630             table key
631             (cons
632               joint (hash-table/get table
633                                   key
634                                   '())))))
635       joints)
636     table))
637
638 (define (m:find-joint-by-vertex-name table vertex-name)
639   (let ((joints (hash-table/get table
640                                 vertex-name
641                                 #f)))
642     (cond ((null? joints) #f)
643           ((= (length joints) 1)
644            (car joints))
645           (else (error "Vertex name not unique among joints"
646                        (map m:joint-name joints))))))
647
648 (define (m:make-joints-by-name-table joints)
649   (let ((table (make-key-weak-eq-hash-table)))
650     (for-each (lambda (joint)
651                 (hash-table/put! table
652                                 (m:make-joint-name-key (m:joint-name joint))
653                                 joint))
654               joints)
655     table))
656
657 ;;; dir-2 is CCW from dir-1
658 (define (m:find-joint-by-id table joint-id)
659   (hash-table/get
660     table
661     (m:make-joint-name-key joint-id)
662     #f))
663
664 ;;;;;;;;;;;;;;;;;;;;;;;;;; Operations using Names ;;;;;;;;;;;;;;;;;;;;;;;;;;
665
666 (define (m:identify-joint-bar-by-name joint bar)
667   (let ((vertex-name (m:joint-vertex-name joint))
668         (dir-1-name (m:joint-dir-1-name joint))
669         (dir-2-name (m:joint-dir-2-name joint))
670         (bar-p1-name (m:bar-p1-name bar))
671         (bar-p2-name (m:bar-p2-name bar)))
672     (cond ((eq? vertex-name bar-p1-name)
673           (cond ((eq? dir-1-name bar-p2-name)
674                 (m:identify-out-of-arm-1 joint bar))
675             ((eq? dir-2-name bar-p2-name)
676                 (m:identify-out-of-arm-2 joint bar))
677             (else (error "Bar can't be identified with joint - no arm"
678                          bar-p2-name))))))

```

```

679      ((eq? vertex-name bar-p2-name)
680      (cond ((eq? dir-1-name bar-p1-name)
681              (m:identify-into-arm-1 joint bar))
682            ((eq? dir-2-name bar-p1-name)
683              (m:identify-into-arm-2 joint bar))
684            (else (error "Bar can't be identified with joint - no arm"
685                          bar-p1-name))))
686      (else (error "Bar can't be identified with joint - no vertex"
687                    vertex-name))))))
688
689 ;;;;;;;;;;;;;; Degrees of Freedom ;;;;;;;;;;;;;;
690
691 (define (m:specified? cell #!optional predicate)
692   (let ((v (m:examine-cell cell)))
693     (and
694      (not (nothing? v))
695      (or (default-object? predicate)
696          (predicate v)))))
697
698 (define (m:bar-length-specified? bar)
699   (m:specified? (m:bar-length bar) number?))
700
701 (define (m:bar-direction-specified? bar)
702   (m:specified? (m:bar-direction bar) direction?))
703
704 (define (m:joint-theta-specified? joint)
705   (m:specified? (m:joint-theta joint) number?))
706
707 ;;;;;;;;;;;;;; Point Predicates ;;;;;;;;;;;;;;
708
709 (define (m:point-specified? p)
710   (and (m:specified? (m:point-x p) number?)
711         (m:specified? (m:point-y p) number?)))
712
713 (define (m:point-contradictory? p)
714   (or (m:contradictory? (m:point-x p))
715       (m:contradictory? (m:point-y p))
716       (m:contradictory? (m:point-region p))))
717
718 ;;;;;;;;;;;;;; Bar Predicates ;;;;;;;;;;;;;;
719
720 (define (m:bar-p1-specified? bar)
721   (m:point-specified? (m:bar-p1 bar)))
722
723 (define (m:bar-p2-specified? bar)
724   (m:point-specified? (m:bar-p2 bar)))
725
726 (define (m:bar-p1-contradictory? bar)
727   (m:point-contradictory? (m:bar-p1 bar)))
728
729 (define (m:bar-p2-contradictory? bar)
730   (m:point-contradictory? (m:bar-p2 bar)))
731
732 (define (m:bar-anchored? bar)
733   (or (m:bar-p1-specified? bar)
734       (m:bar-p2-specified? bar)))
735
736 (define (m:bar-directioned? bar)
737   (and (m:bar-anchored? bar)
738        (m:specified? (m:bar-direction bar) direction?)))
739
740 (define (m:bar-direction-contradictory? bar)
741   (or (m:contradictory? (m:bar-direction bar))
742       (m:contradictory? (m:vec-dx (m:bar-vec bar))
743                           (m:vec-dy (m:bar-vec bar)))))
744
745 (define (m:bar-length-specified? bar)
746   (and (m:specified? (m:bar-length bar) number?)))

```

```

747
748 (define (m:bar-direction-specified? bar)
749   (and (m:specified? (m:bar-direction bar) number?)))
750
751 (define (m:bar-length-contradictory? bar)
752   (m:contradictory? (m:bar-length bar)))
753
754 (define (m:bar-length-dir-specified? bar)
755   (and (m:bar-length-specified? bar)
756         (m:bar-direction-specified? bar)))
757
758 (define (m:bar-fully-specified? bar)
759   (and (m:bar-p1-specified? bar)
760         (m:bar-p2-specified? bar)))
761
762 (define (m:bar-contradictory? bar)
763   (or (m:bar-p1-contradictory? bar)
764        (m:bar-p2-contradictory? bar)
765        (m:bar-direction-contradictory? bar)
766        (m:bar-length-contradictory? bar)))
767
768 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Joint Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
769
770 (define (m:joint-dir-1-specified? joint)
771   (m:specified? (m:joint-dir-1 joint) direction?))
772
773 (define (m:joint-dir-1-contradictory? joint)
774   (m:contradictory? (m:joint-dir-1 joint)))
775
776 (define (m:joint-dir-2-specified? joint)
777   (m:specified? (m:joint-dir-2 joint) direction?))
778
779 (define (m:joint-dir-2-contradictory? joint)
780   (m:contradictory? (m:joint-dir-2 joint)))
781
782 (define (m:joint-theta-contradictory? joint)
783   (m:contradictory? (m:joint-theta joint)))
784
785 (define (m:joint-anchored? joint)
786   (or (m:joint-dir-1-specified? joint)
787        (m:joint-dir-2-specified? joint)))
788
789 (define (m:joint-anchored-and-arm-lengths-specified? joint)
790   (and (m:joint-anchored? joint)
791         (m:bar-length-specified? (m:joint-arm-1 joint))
792         (m:bar-length-specified? (m:joint-arm-2 joint))))
793
794 (define (m:joint-specified? joint)
795   (m:specified? (m:joint-theta joint) number?))
796
797 (define (m:joint-dirs-specified? joint)
798   (and
799     (m:joint-dir-1-specified? joint)
800     (m:joint-dir-2-specified? joint)))
801
802 (define (m:joint-fully-specified? joint)
803   (and
804     (m:point-specified? (m:joint-vertex joint))
805     (m:joint-dir-1-specified? joint)
806     (m:joint-dir-2-specified? joint)))
807
808 (define (m:joint-contradictory? joint)
809   (or
810     (m:point-contradictory? (m:joint-vertex joint))
811     (m:joint-dir-1-contradictory? joint)
812     (m:joint-dir-2-contradictory? joint)
813     (m:joint-theta-contradictory? joint)))
814

```

```

815 ;;;;;;;;;; Specifying Values ;;;;;;;;;;
816
817 (define (m:joint-theta-if-specified joint)
818   (let ((theta-v (m:examine-cell
819                  (m:joint-theta joint))))
820     (if (number? theta-v) theta-v
821         0)))
822
823 (define (m:bar-max-inner-angle-sum bar)
824   (let ((e1 (m:bar-endpoints-1 bar))
825         (e2 (m:bar-endpoints-2 bar)))
826     (if (or (null? e1)
827            (null? e2))
828         0
829         (+ (apply max (map m:joint-theta-if-specified e1))
830            (apply max (map m:joint-theta-if-specified e2))))))
831
832 (define (m:joint-bar-sums joint)
833   (let ((b1 (m:joint-arm-1 joint))
834         (b2 (m:joint-arm-2 joint)))
835     (and (m:bar-length-specified? b1)
836          (m:bar-length-specified? b2)
837          (+ (m:examine-cell (m:bar-length b1))
838             (m:examine-cell (m:bar-length b2))))))
839
840 (define (m:random-theta-for-joint joint)
841   (let ((theta-range (m:examine-cell (m:joint-theta joint))))
842     (if (interval? theta-range)
843         (if (close-enuf? (interval-low theta-range)
844                           (interval-high theta-range))
845             (interval-low theta-range)
846             (begin
847                (safe-internal-rand-range
848                 (interval-low theta-range)
849                 (interval-high theta-range)))
850             (error "Attempting to specify theta for joint"))))
851
852 (define (m:random-bar-length)
853   (internal-rand-range 0.2 1.5))
854
855 (define (m:initialize-bar bar)
856   (if (not (m:bar-anchored? bar))
857       (m:instantiate-point (m:bar-p1 bar) 0 0 'initialize))
858   (let ((random-dir (random-direction)))
859     (m:instantiate (m:bar-direction bar)
860                    random-dir 'initialize)
861     (pp `(initializing-bar ,(print (m:bar-name bar))
862        ,(print random-dir))))
863
864 (define (m:initialize-joint joint)
865   (m:instantiate-point (m:joint-vertex joint) 0 0 'initialize)
866   (pp `(initializing-joint ,(print (m:joint-name joint)))))
867
868 ;;;;;;;;;; Assembling named joints into diagrams ;;;;;;;;;;
869
870 (define (m:assemble-linkages bars joints)
871   (let ((bar-table (m:make-bars-by-name-table bars)))
872     (for-each
873      (lambda (joint)
874        (let ((vertex-name (m:joint-vertex-name joint))
875              (dir-1-name (m:joint-dir-1-name joint))
876              (dir-2-name (m:joint-dir-2-name joint)))
877          (for-each
878           (lambda (dir-name)
879             (let ((bar (m:find-bar-by-id
880                        bar-table
881                        (m:bar vertex-name
882                          dir-name))))

```



```

883         (if (eq? bar #f)
884             (error "Could not find bar for" vertex-name dir-name))
885         (m:identify-joint-bar-by-name joint bar)))
886     (list dir-1-name dir-2-name)))
887 joints)))
888
889 #|
890 ;; Simple example of "solving for the third point"
891 (begin
892   (initialize-scheduler)
893   (let ((b1 (m:make-named-bar 'a 'c))
894         (b2 (m:make-named-bar 'b 'c))
895         (b3 (m:make-named-bar 'a 'b))
896         (j1 (m:make-named-joint 'b 'a 'c))
897         (j2 (m:make-named-joint 'c 'b 'a))
898         (j3 (m:make-named-joint 'a 'c 'b)))
899
900     (m:assemble-linkages
901       (list b1 b2 b3)
902       (list j2 j3 j1))
903
904     (m:initialize-joint j1)
905     (c:id (m:bar-length b1) (m:bar-length b2))
906
907     (m:instantiate (m:bar-length b3) 6 'b3-len)
908     (m:instantiate (m:bar-length b1) 5 'b1-len)
909     (run)
910     (m:examine-point (m:bar-p2 b1))))
911 ;Value: (m:point 3 4)
912
913 |#
914
915 ;;;;;;;;;;;;;; Converstion to Figure Elements ;;;;;;;;;;;;;;
916
917 ;;; TODO: Extract dependencies from TMS? or set names
918
919 (define (m:point->figure-point m-point)
920   (if (not (m:point-specified? m-point))
921       (let ((r (m:examine-cell (m:point-region m-point))))
922         (m:region->figure-elements r))
923       (let ((p (make-point (m:examine-cell (m:point-x m-point))
924                             (m:examine-cell (m:point-y m-point)))))
925         (set-element-name! p (m:element-name m-point))
926         p)))
927
928 (define (m:bar->figure-segment m-bar)
929   (if (not (m:bar-fully-specified? m-bar))
930       #f
931       (let ((p1 (m:point->figure-point (m:bar-p1 m-bar))
932             (p2 (m:point->figure-point (m:bar-p2 m-bar))))
933         (and (point? p1)
934              (point? p2)
935              (make-segment p1 p2)))))
936
937 (define (m:joint->figure-angle m-joint)
938   (if (not (m:joint-fully-specified? m-joint))
939       #f
940       (make-angle (m:examine-cell (m:joint-dir-2 m-joint))
941                   (m:point->figure-point (m:joint-vertex m-joint))
942                   (m:examine-cell (m:joint-dir-1 m-joint)))))

```

Listing A.30: manipulate/region.scm

```

1  ;;; regions.scm --- Region Information
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Points, Lines, Circles, Intersections
7  ;; - For now, semicircle (joints only go to 180deg to avoid
8  ;;   multiple solns.)
9
10 ;; Future:
11 ;; - Differentiate regions with 2 deg. of freedom
12 ;; - Improve contradiction objects
13
14 ;;; Code:
15
16 ;;; Point Sets ;;;
17
18 (define-record-type <m:point-set>
19   (%m:make-point-set points)
20   m:point-set?
21   (points m:point-set-points))
22
23 (define (m:make-point-set points)
24   (%m:make-point-set points))
25
26 (define (m:make-singular-point-set point)
27   (m:make-point-set (list point)))
28
29 (define (m:in-point-set? p point-set)
30   (pair? ((member-procedure point-equal?) p (m:point-set-points point-set))))
31
32 (define (m:singular-point-set? x)
33   (and (m:point-set? x)
34        (= 1 (length (m:point-set-points x)))))
35
36 (define (m:singular-point-set-point ps)
37   (if (not (m:singular-point-set? ps))
38       (error "Not a singular point set")
39       (car (m:point-set-points ps))))
40
41 (define (m:point-sets-equivalent? ps1 ps2)
42   (define delp (delete-member-procedure list-deletor point-equal?))
43   (define memp (member-procedure point-equal?))
44   (let lp ((points-1 (m:point-set-points ps1))
45            (points-2 (m:point-set-points ps2)))
46     (if (null? points-1)
47         (null? points-2)
48         (let ((p1 (car points-1)))
49           (if (memp p1 points-2)
50               (lp (cdr points-1)
51                   (delp p1 points-2))
52               #f))))))
53
54 (define (m:print-point-set ps)
55   (cons 'm:point-set
56         (map (lambda (p) (list 'point (point-x p) (point-y p)))
57              (m:point-set-points ps))))
58
59 (defhandler print
60   m:print-point-set m:point-set?)
61
62 ;;; Rays ;;;
63
64 (define-record-type <m:ray>
65   (%m:make-ray endpoint direction)
66   m:ray?

```

```

67 (endpoint m:ray-endpoint)
68 (direction m:ray-direction))
69
70 (define m:make-ray %m:make-ray)
71
72 (define (m:ray->figure-ray m:ray)
73   (with-color "red"
74     (make-ray (m:ray-endpoint m:ray)
75               (m:ray-direction m:ray))))
76
77 (define (m:on-ray? p ray)
78   (let ((endpoint (m:ray-endpoint ray)))
79     (or (point-equal? p endpoint)
80         (let ((dir (direction-from-points endpoint p)))
81           (direction-equal? dir (m:ray-direction ray))))))
82
83 (define (m:p2-on-ray ray)
84   (add-to-point (m:ray-endpoint ray)
85                 (unit-vec-from-direction (m:ray-direction ray))))
86
87 (define (m:rays-equivalent? ray1 ray2)
88   (and (point-equal? (m:ray-endpoint ray1)
89                     (m:ray-endpoint ray2))
90        (direction-equal? (m:ray-direction ray1)
91                          (m:ray-direction ray2))))
92
93 (define (m:print-ray ray)
94   (let ((endpoint (m:ray-endpoint ray)))
95     `(m:ray (,(point-x endpoint)
96              ,(point-y endpoint)
97              ,(direction-theta (m:ray-direction ray)))))
98
99 (defhandler print
100   m:print-ray m:ray?)
101
102 ;;;;;;;;;;;;;; Arcs ;;;;;;;;;;;;;;
103
104 (define-record-type <m:arc>
105   (m:make-arc center-point radius dir-interval)
106   m:arc?
107   (center-point m:arc-center)
108   (radius m:arc-radius)
109   (dir-interval m:arc-dir-interval))
110
111 ;;; Start direction + ccw pi radian
112 (define (m:make-semi-circle center radius start-direction)
113   (m:make-arc center radius
114               (make-direction-interval start-direction
115                                         (reverse-direction start-direction))))
116
117 (define (m:on-arc? p arc)
118   (let ((center-point (m:arc-center arc))
119         (radius (m:arc-radius arc)))
120     (let ((distance (distance p center-point))
121           (dir (direction-from-points center-point p)))
122       (and (close-enuf? distance radius)
123            (within-direction-interval?
124              dir
125              (m:arc-dir-interval arc))))))
126
127 (define (m:arcs-equivalent? arc1 arc2)
128   (and (point-equal? (m:arc-center arc1)
129                     (m:arc-center arc2))
130        (close-enuf? (m:arc-radius arc1)
131                      (m:arc-radius arc2))
132        (direction-interval-equal?
133          (m:arc-dir-interval arc1)
134          (m:arc-dir-interval arc2))))

```

```

135
136 (define (m:print-arc arc)
137   (let ((center-point (m:arc-center arc))
138         (dir-interval (m:arc-dir-interval arc)))
139     `(m:arc (,(point-x center-point)
140              ,(point-y center-point))
141            ,(m:arc-radius arc)
142            (,(direction-theta (direction-interval-start dir-interval))
143              ,(direction-theta (direction-interval-end dir-interval)))))
144
145 (defhandler print
146   m:print-arc
147   m:arc?)
148
149 ;;;;;;;;;;;;;; Contradiction Objects ;;;;;;;;;;;;;;
150
151 (define-record-type <m:region-contradiction>
152   (m:make-region-contradiction error-regions)
153   m:region-contradiction?
154   (error-regions m:contradiction-error-regions))
155
156 ;; TODO: Maybe differeniate by error values
157 (define (m:region-contradictions-equivalent? rc1 rc2) #t)
158
159 (define (m:region-contradiction->figure-elements rc)
160   (map m:region->figure-elements (m:contradiction-error-regions rc)))
161
162 ;;;;;;;;;;;;;; Specific Intersections ;;;;;;;;;;;;;;
163
164 (define (m:intersect-rays ray1 ray2)
165   (let ((endpoint-1 (m:ray-endpoint ray1))
166         (endpoint-2 (m:ray-endpoint ray2))
167         (dir-1 (m:ray-direction ray1))
168         (dir-2 (m:ray-direction ray2)))
169     (if (direction-equal? dir-1 dir-2)
170         (cond ((m:on-ray? endpoint-1 ray2) ray1)
171               ((m:on-ray? endpoint-2 ray1) ray2)
172               ;; TODO: Determine error value
173               (else (m:make-region-contradiction (list ray1 ray2))))
174         (let ((ray1-p2 (m:p2-on-ray ray1))
175               (ray2-p2 (m:p2-on-ray ray2)))
176           (let ((intersections
177                  (intersect-lines-by-points endpoint-1 ray1-p2
178                                              endpoint-2 ray2-p2)))
179             (if (not (= 1 (length intersections)))
180                 (m:make-region-contradiction (list ray1 ray2))
181                 (let ((intersection (car intersections))
182                       (if (and (m:on-ray? intersection ray1)
183                                (m:on-ray? intersection ray2))
184                            (m:make-point-set (list intersection))
185                            ;; TODO: Determine error value
186                            (m:make-region-contradiction (list ray1 ray2))))))))))
187
188 (define (m:intersect-arcs arc1 arc2)
189   (let ((c1 (m:arc-center arc1))
190         (c2 (m:arc-center arc2))
191         (r1 (m:arc-radius arc1))
192         (r2 (m:arc-radius arc2)))
193     (if (point-equal? c1 c2)
194         (if (close-enuf? r1 r2)
195             (m:make-arc c1 r1
196                         (intersect-direction-intervals
197                          (m:arc-dir-interval arc1)
198                          (m:arc-dir-interval arc2)))
199             (m:make-region-contradiction (list arc1 arc2)))
200         (let ((intersections
201                (intersect-circles-by-centers-radii
202                 c1 r1 c2 r2)))
203

```

```

203     (let ((points
204           (filter (lambda (p)
205                     (and (m:on-arc? p arc1)
206                           (m:on-arc? p arc2)))
207                   intersections)))
208     (if (> (length points) 0)
209        (m:make-point-set points)
210        ;; TODO: Determine error value
211        (m:make-region-contradiction (list arc1 arc2))))))
212
213 (define (m:intersect-ray-arc ray arc)
214   (let ((center (m:arc-center arc))
215         (radius (m:arc-radius arc))
216         (endpoint (m:ray-endpoint ray))
217         (ray-p2 (m:p2-on-ray ray)))
218     (let ((intersections
219           (intersect-circle-line-by-points
220            center radius endpoint ray-p2)))
221       (let ((points
222             (filter (lambda (p)
223                       (and (m:on-ray? p ray)
224                             (m:on-arc? p arc)))
225                   intersections)))
226         (if (> (length points) 0)
227             (m:make-point-set points)
228             ;; TODO: Determine error value
229             (m:make-region-contradiction (list ray arc))))))
230
231 (define (m:intersect-arc-ray arc ray)
232   (m:intersect-ray-arc ray arc))
233
234 ;;;;;;;;;;;;;; Intersecting with Point Sets ;;;;;;;;;;;;;;
235
236 (define m:in-region? (make-generic-operation 2 'm:in-region?))
237
238 (defhandler m:in-region? m:in-point-set? point? m:point-set?)
239 (defhandler m:in-region? m:on-ray? point? m:ray?)
240 (defhandler m:in-region? m:on-arc? point? m:arc?)
241 (defhandler m:in-region? (lambda (p r) #f) point? m:region-contradiction?)
242
243 (define (m:intersect-point-set-with-region ps1 region)
244   (let ((results
245         (let lp ((points-1 (m:point-set-points ps1))
246                   (point-intersections '()))
247             (if (null? points-1)
248                 point-intersections
249                 (let ((p1 (car points-1)))
250                     (if (m:in-region? p1 region)
251                         (lp (cdr points-1)
252                             (cons p1 point-intersections))
253                         (lp (cdr points-1)
254                             point-intersections)))))))
255     (if (> (length results) 0)
256         (m:make-point-set results)
257         ;; TODO: Determine error value
258         (m:make-region-contradiction (list ps1 region))))
259
260 (define (m:intersect-region-with-point-set region ps)
261   (m:intersect-point-set-with-region ps region))
262
263 ;;;;;;;;;;;;;; Translating regions by Vec ;;;;;;;;;;;;;;
264
265 (define m:translate-region (make-generic-operation 2 'm:translate-region))
266
267 (define (m:translate-point-set ps vec)
268   (m:make-point-set
269    (map (lambda (p) (add-to-point p vec))
270         (m:point-set-points ps))))

```

```

271 (defhandler m:translate-region m:translate-point-set m:point-set? vec?)
272
273 (define (m:translate-ray ray vec)
274   (m:make-ray
275     (add-to-point (m:ray-endpoint ray) vec)
276     (m:ray-direction ray)))
277 (defhandler m:translate-region m:translate-ray m:ray? vec?)
278
279 (define (m:translate-arc arc vec)
280   (m:make-arc
281     (add-to-point (m:arc-center arc) vec)
282     (m:arc-radius arc)
283     (m:arc-dir-interval arc)))
284 (defhandler m:translate-region m:translate-arc m:arc? vec?)
285
286 ;;;;;;;;;;;;;; Generic Intersect Regions "Merge" ;;;;;;;;;;;;;;
287
288 (define m:intersect-regions (make-generic-operation 2 'm:intersect-regions))
289
290 ;; Same Type
291 (defhandler m:intersect-regions
292   m:intersect-rays m:ray? m:ray?)
293 (defhandler m:intersect-regions
294   m:intersect-arcs m:arc? m:arc?)
295
296 ;; Arc + Ray
297 (defhandler m:intersect-regions
298   m:intersect-ray-arc m:ray? m:arc?)
299 (defhandler m:intersect-regions
300   m:intersect-arc-ray m:arc? m:ray?)
301
302 ;; Point Sets
303 (defhandler m:intersect-regions
304   m:intersect-region-with-point-set any? m:point-set?)
305 (defhandler m:intersect-regions
306   m:intersect-point-set-with-region m:point-set? any?)
307
308 ;; Contradictions
309 (defhandler m:intersect-regions (lambda (a b) a) m:region-contradiction? any?)
310 (defhandler m:intersect-regions (lambda (a b) b) any? m:region-contradiction?)
311
312 ;;;;;;;;;;;;;; Generic Equivalency ;;;;;;;;;;;;;;
313
314 (define m:region-equivalent?
315   (make-generic-operation 2 'm:region-equivalent? (lambda (a b) #f)))
316
317 (defhandler m:region-equivalent?
318   m:point-sets-equivalent? m:point-set? m:point-set?)
319
320 (defhandler m:region-equivalent?
321   m:rays-equivalent? m:ray? m:ray?)
322
323 (defhandler m:region-equivalent?
324   m:arcs-equivalent? m:arc? m:arc?)
325
326 (defhandler m:region-equivalent?
327   m:region-contradictions-equivalent?
328   m:region-contradiction?
329   m:region-contradiction?)
330
331 ;;;;;;;;;;;;;; Interface to Propagator System ;;;;;;;;;;;;;;
332
333 (define (m:region? x)
334   (or (m:point-set? x)
335       (m:ray? x)
336       (m:arc? x)
337       (m:region-contradiction? x)))
338

```

```

339
340 (defhandler equivalent? m:region-equivalent? m:region? m:region?)
341
342 (defhandler merge m:intersect-regions m:region? m:region?)
343
344 (defhandler contradictory? m:region-contradiction? m:region?)
345
346 #|
347 Simple Examples
348 (pp (let-cells (c)
349   (add-content c (m:make-arc (make-point 1 0) (sqrt 2)
350     (make-direction-interval
351       (make-direction (/ pi 8))
352       (make-direction (* 7 (/ pi 8))))))
353
354   (add-content c (m:make-ray (make-point -3 1) (make-direction 0)))
355   (add-content c (m:make-ray (make-point 1 2)
356     (make-direction (* 7 (/ pi 4))))))
357   (content c)))
358
359 (let ((a (make-point 0 0))
360       (b (make-point 1 0))
361       (c (make-point 0 1))
362       (d (make-point 1 1)))
363   (let-cells (cell)
364     (add-content cell
365       (make-tms
366         (contingent (m:make-point-set (list a b c))
367           '(a))))
368     (add-content cell
369       (make-tms
370         (contingent (m:make-point-set (list a d))
371           '(a))))
372     (pp (tms-query (content cell)))))
373 |#
374 ;;;;;;;;;;;;;;;;;;;;;;;;; To Figure elements ;;;;;;;;;;;;;;;;;;;;;;;;;
375
376 (define m:region->figure-elements
377   (make-generic-operation 1 'm:region->figure-elements (lambda (r) #f)))
378
379 (defhandler m:region->figure-elements
380   m:ray->figure-ray
381   m:ray?)
382
383 (defhandler m:region->figure-elements
384   m:region-contradiction->figure-elements
385   m:region-contradiction?)

```

Listing A.31: manipulate/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
34           (m:bar-length bar-1)
35           (m:bar-length bar-2))))))
36
37 (define (m:c-angle-equal joint-id-1 joint-id-2)
38   (m:make-constraint
39     'm:c-angle-equal
40     (list joint-id-1 joint-id-2)
41     (lambda (m)
42       (let ((joint-1 (m:lookup m joint-id-1))
43             (joint-2 (m:lookup m joint-id-2)))
44         (c:id (m:joint-theta joint-1)
45               (m:joint-theta joint-2))))))
46
47 (define (m:c-right-angle joint-id)
48   (m:make-constraint
49     'm:right-angle
50     (list joint-id)
51     (lambda (m)
52       (let ((joint (m:lookup m joint-id)))
53         (c:id
54           (m:joint-theta joint)
55           (/ pi 2))))))
56
57 ;; p2 between p1 p3 in a line
58 (define (m:c-line-order p1-id p2-id p3-id)
59   (list
60     (m:make-named-bar p1-id p2-id)
61     (m:make-named-bar p2-id p3-id)
62     (m:make-named-joint p1-id p2-id p3-id)
63     (m:c-full-angle (m:joint p1-id p2-id p3-id))))
64
65 (define (m:c-full-angle joint-id)
66   (m:make-constraint

```



```

67 'm:full-angle
68 (list joint-id)
69 (lambda (m)
70   (let ((joint (m:lookup m joint-id)))
71     (c:id
72      (m:joint-theta joint)
73      pi))))))
74
75 (define (m:equal-joints-in-sum equal-joint-ids
76                                   all-joint-ids
77                                   total-sum)
78   (m:make-constraint
79    'm:equal-joints-in-sum
80    all-joint-ids
81    (lambda (m)
82      (let ((all-joints (m:multi-lookup m all-joint-ids))
83            (equal-joints (m:multi-lookup m equal-joint-ids)))
84        (let ((other-joints
85              (set-difference all-joints equal-joints eq?)))
86          (c:id (m:joint-theta (car equal-joints))
87               (ce:/
88                (ce:- total-sum
89                     (ce:multi+ (map m:joint-theta other-joints)))
89                (length equal-joints))))))))))
91
92 (define (n-gon-angle-sum n)
93   (* n (- pi (/ (* 2 pi) n))))
94
95 (define (m:polygon-sum-slice all-joint-ids)
96   (m:make-slice
97    (m:make-constraint
98     'm:joint-sum
99     all-joint-ids
100    (lambda (m)
101      (let ((all-joints (m:multi-lookup m all-joint-ids))
102            (total-sum (n-gon-angle-sum (length all-joint-ids))))
103        (m:joints-constrained-in-sum all-joints total-sum))))))
104
105 ;;;;;;;;;;;;;; Applying and Marking Constrained Elements ;;;;;;;;;;;;;;
106
107 (define (m:constrained? element)
108   (not (null? (m:element-constraints element))))
109
110 (define (m:element-constraints element)
111   (or (eq-get element 'm:constraints)
112       '()))
113
114 (define (m:set-element-constraints! element constraints)
115   (eq-put! element 'm:constraints constraints))
116
117 (define (m:mark-constraint element constraint)
118   (m:set-element-constraints!
119    element
120    (cons constraint
121          (m:element-constraints element))))
122
123 (define (m:apply-constraint m constraint)
124   (for-each (lambda (element-id)
125               (m:mark-constraint
126                (m:lookup m element-id)
127                constraint))
128             (m:constraint-args constraint))
129   ((m:constraint-procedure constraint) m))
130
131 ;;;;;;;;;;;;;; Slices ;;;;;;;;;;;;;;
132
133 ;;; Slices are constraints that are processed once the normal
134 ;;; constraints have been applied.

```

```

135
136 (define-record-type <m:slice>
137   (m:make-slice constraint)
138   m:slice?
139   (constraint m:slice-constraint))
140
141 (define (m:apply-slice m slice)
142   (m:apply-constraint m (m:slice-constraint slice)))
143
144 ;;;;;;;;;;;;;;;;;;;;;;;;; Propagator Utils ;;;;;;;;;;;;;;;;;;;;;;;;;
145
146 (define (ce:multi+ cells)
147   (cond ((null? cells) 0)
148         ((null? (cdr cells)) (car cells))
149         (else
          (ce:+ (car cells)
                (ce:multi+ (cdr cells))))))
150
151 (define (ce:+ (car cells)
152               (ce:multi+ (cdr cells))))
153
154 ;;;;;;;;;;;;;;;;;;;;;;;;; Slices (for sums) ;;;;;;;;;;;;;;;;;;;;;;;;;
155
156 (define (m:equal-values-in-sum equal-cells all-cells total-sum)
157   (let ((other-values (set-difference all-cells equal-cells eq?)))
158     (c:id (car equal-cells)
            (ce:/
              (ce:- total-sum
                    (ce:multi+ other-values))
              (length equal-cells)))))
159
160 (define (m:sum-slice elements cell-transformer equality-predicate total-sum)
161   (let ((equivalence-classes
          (partition-into-equivalence-classes elements equality-predicate))
         (all-cells (map cell-transformer elements)))
162     (cons
      (c:id total-sum
            (ce:multi+ all-cells))
      (filter identity
               (map (lambda (equiv-class)
                      (and (> (length equiv-class) 1)
                           (begin
                             (m:equal-values-in-sum
                               (map cell-transformer equiv-class)
                               all-cells
                               total-sum))))
                    equivalence-classes)))))
163
164 (define (angle-equal-constraint? c)
165   (eq? (m:constraint-type c) 'm:c-angle-equal))
166
167 (define (m:joints-constrained-equal-to-one-another? joint-1 joint-2)
168   (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))))
169     (not (null? (set-intersection joint-1-constraints
                                    joint-2-constraints
                                    (member-procedure eq?))))))
170
171 (define (m:joints-constrained-in-sum all-joints total-sum)
172   (m:sum-slice
    all-joints
    m:joint-theta
    m:joints-constrained-equal-to-one-another?
    total-sum))

```

Listing A.32: manipulate/topology.scm

```

1  ;; topology.scm --- Helpers for establishing topology for mechanism
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Simplify listing out all bar and joint orderings
7  ;; - Start with basic polygons, etc.
8
9  ;; Future:
10 ;; - Figure out making multi-in/out joints: (all pairs?)
11
12 ;; Code:
13
14 ;;;;;;;;;;;;;;;;;;;;;;;;;; Establish-topology ;;;;;;;;;;;;;;;;;;;;;;;;;;
15
16 ;; CCW point names
17 (define (m:establish-polygon-topology . point-names)
18   (if (< (length point-names) 3)
19       (error "Min polygon size: 3")
20       (let ((extended-point-names
21              (append point-names
22                       (list (car point-names) (cadr point-names)))))
23         (let ((bars
24                (map (lambda (p1-name p2-name)
25                      (m:make-named-bar p1-name p2-name))
26                     point-names
27                     (cdr extended-point-names)))
28               (joints
29                (map (lambda (p1-name vertex-name p2-name)
30                      (m:make-named-joint p1-name vertex-name p2-name))
31                     (cddr extended-point-names)
32                     (cdr extended-point-names)
33                     point-names)))
34           (append bars joints
35                   (list (m:polygon-sum-slice
36                         (map m:joint-name joints)))))))

```

Listing A.33: manipulate/mechanism.scm

```

1  ;; mechanism.scm --- Group of Bars / Joints
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Grouping of bars and joints
7  ;; - Integrate with establishing topology
8
9  ;; Future:
10 ;; - Also specify constraints with it
11 ;; - Convert to Diagram
12
13 ;; Code:
14
15 ;;;;;;;;;;;;;;;;; Mechanism Structure ;;;;;;;;;;;;;;;;;
16
17 (define-record-type <m:mechanism>
18   (%m:make-mechanism bars joints constraints slices
19     bar-table joint-table joint-by-vertex-table)
20   m:mechanism?
21   (bars m:mechanism-bars)
22   (joints m:mechanism-joints)
23   (constraints m:mechanism-constraints)
24   (slices m:mechanism-slices)
25   (bar-table m:mechanism-bar-table)
26   (joint-table m:mechanism-joint-table)
27   (joint-by-vertex-table m:mechanism-joint-by-vertex-table))
28
29 (define (m:make-mechanism bars joints constraints slices)
30   (let ((bar-table (m:make-bars-by-name-table bars))
31         (joint-table (m:make-joints-by-name-table joints))
32         (joint-by-vertex-table (m:make-joints-by-vertex-name-table joints)))
33     (%m:make-mechanism bars joints constraints slices
34       bar-table joint-table joint-by-vertex-table)))
35
36 (define (m:mechanism . args)
37   (let ((elements (flatten args)))
38     (let ((bars (m:dedupe-bars (filter m:bar? elements))
39           (joints (filter m:joint? elements))
40           (constraints (filter m:constraint? elements))
41           (slices (filter m:slice? elements)))
42       (m:make-mechanism bars joints constraints slices))))
43
44 (define (m:print-mechanism m)
45   `((bars ,(map print (m:mechanism-bars m)))
46     (joints ,(map print (m:mechanism-joints m)))
47     (constraints ,(map print (m:mechanism-constraints m)))))
48
49 (defhandler print m:print-mechanism m:mechanism?)
50
51 ;;;;;;;;;;;;;;;;; Deduplication ;;;;;;;;;;;;;;;;;
52
53 (define (m:dedupe-bars bars)
54   (dedupe (member-procedure m:bars-name-equivalent?) bars))
55
56
57 ;;;;;;;;;;;;;;;;; Accessors ;;;;;;;;;;;;;;;;;
58
59 (define (m:mechanism-joint-by-vertex-name m vertex-name)
60   (m:find-joint-by-vertex-name
61     (m:mechanism-joint-by-vertex-table m)
62     vertex-name))
63
64 (define (m:mechanism-joint-by-names m dir-1-name vertex-name dir-2-name)
65   (m:find-joint-by-names
66     (m:mechanism-joint-table m)

```

```

67   dir-1-name vertex-name dir-2-name))
68
69 (define (m:multi-lookup m ids)
70   (map (lambda (id) (m:lookup m id)) ids))
71
72 (define (m:lookup m id)
73   (cond ((m:bar-id? id) (m:find-bar-by-id
74                         (m:mechanism-bar-table m)
75                         id))
76         ((m:joint-id? id) (m:find-joint-by-id
77                             (m:mechanism-joint-table m)
78                             id))
79         ((m:joint-vertex-id? id) (m:find-joint-by-vertex-name
80                                     (m:mechanism-joint-by-vertex-table m)
81                                     (m:joint-vertex-id-name id))))))
82
83 ;;;;;;;;;;;;;; Specified ;;;;;;;;;;;;;;
84
85 (define (m:mechanism-fully-specified? mechanism)
86   (and (every m:bar-fully-specified? (m:mechanism-bars mechanism))
87        (every m:joint-fully-specified? (m:mechanism-joints mechanism))))
88
89 (define (m:mechanism-contradictory? mechanism)
90   (or (any m:bar-contradictory? (m:mechanism-bars mechanism))
91       (any m:joint-contradictory? (m:mechanism-joints mechanism))))
92
93 ;;;;;;;;;;;;;; Specify ;;;;;;;;;;;;;;
94
95 ;;; Should these be in Linkages?
96
97 (define *any-dir-specified* #f)
98 (define *any-point-specified* #f)
99
100 (define (any-one l)
101   (let ((i (random (length l))))
102     (list-ref l i)))
103
104 (define (m:pick-bar bars)
105   (car (sort-by-key bars (negatep m:bar-max-inner-angle-sum))))
106
107 (define m:pick-joint-1 any-one)
108
109 (define (m:pick-joint joints)
110   (car
111    (append
112     (sort-by-key
113      (filter m:joint-bar-sums joints)
114      m:joint-bar-sums)
115     (filter (notp m:joint-bar-sums) joints))))))
116
117 (define (m:specify-angle-if-first-time cell)
118   (if (not *any-dir-specified*)
119       (let ((dir (random-direction)))
120         (set! *any-dir-specified* #t)
121         (pp `(initializing-direction ,(name cell) ,(print dir)))
122         (m:instantiate cell dir 'first-time-angle))))
123
124 (define (m:specify-point-if-first-time point)
125   (if (not *any-point-specified*)
126       (begin
127         (set! *any-point-specified* #t)
128         (pp `(initializing-point ,(name point) (0 0)))
129         (m:instantiate point 0 0 'first-time-point))))
130
131 (define (m:specify-bar bar)
132   (let ((v (m:random-bar-length)))
133     (pp `(specifying-bar-length ,(print (m:bar-name bar)) ,v))
134     (m:instantiate (m:bar-length bar) v 'specify-bar)))

```

```

135     (m:specify-angle-if-first-time (m:bar-direction bar))
136     (m:specify-point-if-first-time (m:bar-pl bar))))
137
138 (define (m:specify-joint joint)
139   (let ((v (m:random-theta-for-joint joint)))
140     (pp `(specifying-joint-angle ,(print (m:joint-name joint)) ,v))
141     (m:instantiate (m:joint-theta joint) v 'specify-joint)
142     (m:specify-angle-if-first-time (m:joint-dir-1 joint))))
143
144 (define (m:initialize-joint-vertex joint)
145   (m:specify-point-if-first-time (m:joint-vertex joint)))
146
147 (define (m:initialize-joint-direction joint)
148   (m:specify-angle-if-first-time (m:joint-dir-1 joint)))
149
150 (define (m:initialize-bar-pl bar)
151   (m:specify-point-if-first-time (m:bar-pl bar)))
152
153 (define (m:specify-joint-if m predicate)
154   (let ((joints (filter (andp predicate (notp m:joint-specified?))
155                         (m:mechanism-joints m))))
156     (and (not (null? joints))
157          (m:specify-joint (m:pick-joint joints)))))
158
159 (define (m:initialize-joint-if m predicate)
160   (let ((joints (filter (andp predicate (notp m:joint-specified?))
161                         (m:mechanism-joints m))))
162     (and (not (null? joints))
163          (let ((j (m:pick-joint joints)))
164            (m:initialize-joint-direction j)))))
165
166 (define (m:specify-bar-if m predicate)
167   (let ((bars (filter (andp predicate (notp m:bar-length-specified?))
168                       (m:mechanism-bars m))))
169     (and (not (null? bars))
170          (m:specify-bar (m:pick-bar bars)))))
171
172 (define (m:initialize-bar-if m predicate)
173   (let ((bars (filter (andp predicate (notp m:bar-length-specified?))
174                       (m:mechanism-bars m))))
175     (and (not (null? bars))
176          (m:initialize-bar-pl (m:pick-bar bars)))))
177
178 (define (m:specify-something m)
179   (or
180    (m:specify-bar-if m m:constrained?)
181    (m:specify-joint-if m m:constrained?)
182    (m:specify-joint-if m m:joint-anchored-and-arm-lengths-specified?)
183    (m:specify-joint-if m m:joint-anchored?)
184    (m:specify-bar-if m m:bar-directioned?)
185    (m:specify-bar-if m m:bar-anchored?)
186    (m:initialize-joint-if m m:joint-dirs-specified?)
187    (m:initialize-bar-if m m:bar-length-dir-specified?)
188    (m:initialize-bar-if m m:bar-direction-specified?)
189    (m:initialize-bar-if m m:bar-length-specified?)
190    (m:initialize-joint-if m m:joint-anchored?)
191    (m:initialize-joint-if m true-proc)
192    (m:initialize-bar-if m true-proc)))
193
194 ;;;;;;;;;;;;;;;;;;;;;;;;;; Applying constraints ;;;;;;;;;;;;;;;;;;;;;;;;;;
195
196 (define (m:apply-mechanism-constraints m)
197   (for-each (lambda (c)
198               (m:apply-constraint m c))
199             (m:mechanism-constraints m)))
200
201 (define (m:apply-slices m)
202   (for-each (lambda (s)

```

```

203         (m:apply-slice m s))
204     (m:mechanism-slices m)))
205
206 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Build ;;;;;;;;;;;;;;;;;
207
208 (define (m:identify-vertices m)
209   (for-each (lambda (joints)
210     (let ((first-vertex (m:joint-vertex (car joints))))
211       (for-each (lambda (joint)
212         (m:identify-points first-vertex
213           (m:joint-vertex joint)))
214         (cdr joints))))
215     (hash-table/datum-list (m:mechanism-joint-by-vertex-table m))))
216
217 (define (m:build-mechanism m)
218   (m:identify-vertices m)
219   (m:assemble-linkages (m:mechanism-bars m)
220     (m:mechanism-joints m))
221   (m:apply-mechanism-constraints m)
222   (m:apply-slices m))
223
224 (define (m:initialize-solve)
225   (set! *any-dir-specified* #f)
226   (set! *any-point-specified* #f))
227
228 (define *m* #f)
229 (define (m:solve-mechanism m)
230   (set! *m* m)
231   (m:initialize-solve)
232   (let lp ()
233     (run)
234     (cond ((m:mechanism-contradictory? m)
235       (m:draw-mechanism m c)
236       #f)
237       ((not (m:mechanism-fully-specified? m))
238        (if (m:specify-something m)
239          (lp)
240          (error "Couldn't find anything to specify.")))
241       (else 'mechanism-built))))
242
243 (define (m:solve-mechanism-new m)
244   (set! *m* m)
245   (m:initialize-solve))
246
247 (define (m:specify-something-new m fail)
248   (let ((linkages (append (m:mechanism-bars m)
249     (m:mechanism-joints m))))
250     (let lp ((linkages (sort-linknages linkages)))
251       (if (null? linkages)
252         (fail)
253         (let ((first-linkage (car linkages))
254           (other-linkages (cdr linkages)))
255           (m:specify-linkage m first-linkage
256             (lambda ()
257               (lp (cdr linkages))))))))))
258
259 #|
260 (begin
261   (initialize-scheduler)
262   (m:build-mechanism
263     (m:mechanism
264       (m:establish-polygon-topology 'a 'b 'c))))
265 |#
266
267 ;;;;;;;;;;;;;;;;; Conversion to Figure ;;;;;;;;;;;;;;;;;
268
269 (define (m:mechanism->figure m)
270   (let ((points

```

```

271      (map (lambda (joint)
272            (m:point->figure-point (m:joint-vertex joint)))
273            (m:mechanism-joints m)))
274      (segments (map m:bar->figure-segment (m:mechanism-bars m)))
275      (angles (map m:joint->figure-angle (m:mechanism-joints m))))
276      (apply figure (flatten (filter (lambda (x) (or x))
277                                    (append points segments angles))))))
278
279 (define (m:draw-mechanism m c)
280   (draw-figure (m:mechanism->figure m) c))
281
282 #|
283 (let lp ()
284   (initialize-scheduler)
285   (let ((m (m:mechanism
286             (m:establish-polygon-topology 'a 'b 'c 'd))))
287     (pp (m:joint-anchored? (car (m:mechanism-joints m))))
288     (m:build-mechanism m)
289     (m:solve-mechanism m)
290     (let ((f (m:mechanism->figure m)))
291       (draw-figure f c)
292       (pp (analyze-figure f)))))
293 |#

```


Listing A.34: manipulate/main.scm

```

1  ;;; main.scm --- Main definitions and code for running the
2  ;;; manipulation / mechanism-based code
3
4  ;;; Examples
5
6  (define (arbitrary-triangle)
7    (m:mechanism
8      (m:establish-polygon-topology 'a 'b 'c)))
9
10 (define (arbitrary-right-triangle)
11   (m:mechanism
12     (m:establish-polygon-topology 'a 'b 'c)
13     (m:c-right-angle (m:joint 'a))))
14
15 (define (arbitrary-right-triangle-2)
16   (m:mechanism
17     (m:establish-polygon-topology 'a 'b 'c)
18     (m:c-right-angle (m:joint 'c))))
19
20 (define (quadrilateral-with-diagonals a b c d)
21   (list
22     (m:establish-polygon-topology a b c d)
23     (m:establish-polygon-topology a b c)
24     (m:establish-polygon-topology b c d)
25     (m:establish-polygon-topology c d a)
26     (m:establish-polygon-topology d a c)))
27
28 (define (quadrilateral-with-diagonals-intersection a b c d e)
29   (list
30     (quadrilateral-with-diagonals a b c d)
31     (m:establish-polygon-topology a b e)
32     (m:establish-polygon-topology b c e)
33     (m:establish-polygon-topology c d e)
34     (m:establish-polygon-topology d a e)
35     (m:c-line-order c e a)
36     (m:c-line-order b e d)))
37
38 (define (quad-diagonals)
39   (m:mechanism
40     ;; Setup abcd with e in the middle:
41     (quadrilateral-with-diagonals-intersection 'a 'b 'c 'd 'e)
42
43     (m:establish-polygon-topology 'a 'b 'e)
44     (m:establish-polygon-topology 'b 'c 'e)
45     (m:establish-polygon-topology 'c 'd 'e)
46     (m:establish-polygon-topology 'd 'a 'e)
47     (m:c-line-order 'c 'e 'a)
48     (m:c-line-order 'b 'e 'd))
49
50     ;; Right Angle in Center:
51     (m:c-right-angle (m:joint 'b 'e 'c))
52
53     ;; Diagonals Equal
54     (m:c-length-equal (m:bar 'c 'a) (m:bar 'b 'd))
55     (m:c-length-equal (m:bar 'c 'e) (m:bar 'a 'e))
56     (m:c-length-equal (m:bar 'b 'e) (m:bar 'd 'e))
57
58     ;; Make it a square:
59     (m:c-length-equal (m:bar 'c 'e) (m:bar 'b 'e))
60   ))
61
62 ;;; Works:
63 (define (isocles-triangle)
64   (m:mechanism
65     (m:establish-polygon-topology 'a 'b 'c)
66     (m:c-length-equal (m:bar 'a 'b)

```

```

67         (m:bar 'b 'c))))
68
69 (define (isocles-triangle-by-angles)
70   (m:mechanism
71     (m:establish-polygon-topology 'a 'b 'c)
72     (m:c-angle-equal (m:joint 'a)
73                       (m:joint 'b))
74     (m:equal-joints-in-sum
75       (list (m:joint 'a) (m:joint 'b))
76       (list (m:joint 'a) (m:joint 'b) (m:joint 'c))
77       pi)))
78
79 (define (isocles-triangle-by-angles)
80   (m:mechanism
81     (m:establish-polygon-topology 'a 'b 'c)
82     (m:c-angle-equal (m:joint 'a)
83                       (m:joint 'b))))
84
85 ;;; Often works:
86 (define (arbitrary-quadrilateral)
87   (m:mechanism
88     (m:establish-polygon-topology 'a 'b 'c 'd)))
89
90 ;;; Always works:
91 (define (parallelogram-by-sides)
92   (m:mechanism
93     (m:establish-polygon-topology 'a 'b 'c 'd)
94     (m:c-length-equal (m:bar 'a 'b)
95                       (m:bar 'c 'd))
96     (m:c-length-equal (m:bar 'b 'c)
97                       (m:bar 'd 'a))))
98
99 (define (kite-by-sides)
100   (m:mechanism
101     (m:establish-polygon-topology 'a 'b 'c 'd)
102     (m:c-length-equal (m:bar 'a 'b)
103                       (m:bar 'b 'c))
104     (m:c-length-equal (m:bar 'c 'd)
105                       (m:bar 'd 'a))))
106
107 (define (kite-by-angles-sides)
108   (m:mechanism
109     (m:establish-polygon-topology 'a 'b 'c 'd)
110     (m:c-length-equal (m:bar 'a 'b)
111                       (m:bar 'a 'd))
112     (m:c-angle-equal (m:joint 'b)
113                       (m:joint 'd))))
114
115 (define (rhombus-by-sides)
116   (m:mechanism
117     (m:establish-polygon-topology 'a 'b 'c 'd)
118     (m:c-length-equal (m:bar 'a 'b)
119                       (m:bar 'b 'c))
120     (m:c-length-equal (m:bar 'b 'c)
121                       (m:bar 'c 'd))
122     (m:c-length-equal (m:bar 'c 'd)
123                       (m:bar 'a 'd))))
124
125 (define (parallelogram-by-angles)
126   (m:mechanism
127     (m:establish-polygon-topology 'a 'b 'c 'd)
128     (m:c-angle-equal (m:joint 'a)
129                       (m:joint 'c))
130     (m:c-angle-equal (m:joint 'b)
131                       (m:joint 'd))))
132
133 (define ***)
134 (define (m:run-mechanism mechanism-proc)

```

```

135 (initialize-scheduler)
136 (let ((m (mechanism-proc)))
137   (set! *m* m)
138   (m:build-mechanism m)
139   (if (not (m:solve-mechanism m))
140       (pp "Unsolvable!")
141       (let ((f (m:mechanism->figure m)))
142         (draw-figure f c)
143         ;;(pp (analyze-figure f))
144         ))))
145
146 #|
147 (let lp ()
148   (initialize-scheduler)
149   (pp 'start)
150   (m:run-mechanism
151     (lambda ()
152       (m:mechanism
153         ;;(m:establish-polygon-topology 'a 'b 'c)
154         (m:make-named-bar 'a 'b)
155         (m:make-named-bar 'b 'c)
156         (m:make-named-bar 'c 'a)
157         (m:make-named-joint 'c 'b 'a)
158         (m:make-named-joint 'a 'c 'b)
159         (m:make-named-joint 'b 'a 'c)
160
161         (m:make-named-bar 'a 'd)
162         (m:make-named-bar 'b 'd)
163         (m:make-named-joint 'd 'a 'b)
164         (m:make-named-joint 'a 'b 'd)
165         (m:make-named-joint 'b 'd 'a)
166
167         (m:make-named-bar 'c 'd)
168         (m:make-named-joint 'a 'd 'c)
169         (m:make-named-joint 'c 'a 'd)
170         (m:make-named-joint 'd 'c 'a))))
171   (lp))
172
173 (let lp ()
174   (initialize-scheduler)
175   (let ((m (m:mechanism
176     (m:establish-polygon-topology 'a 'b 'c 'd))))
177     (m:build-mechanism m)
178     (m:solve-mechanism m)
179     (let ((f (m:mechanism->figure m)))
180       (draw-figure f c)
181       (pp (analyze-figure f)))))
182 |#
183
184 (define (rect-demo-1)
185   (m:mechanism
186     (m:establish-polygon-topology 'a 'b 'c 'd)
187     (m:c-length-equal (m:bar 'a 'b)
188                       (m:bar 'b 'c))
189     (m:c-right-angle (m:joint 'd))))
190
191 (define (rect-demo-2)
192   (m:mechanism
193     (m:establish-polygon-topology 'a 'b 'c 'd)
194     (m:c-length-equal (m:bar 'a 'd)
195                       (m:bar 'b 'c))
196     (m:c-right-angle (m:joint 'd))
197     (m:c-angle-equal (m:joint 'a)
198                      (m:joint 'c))))
199
200 (define (rect-demo-3)
201   (m:mechanism
202     (m:establish-polygon-topology 'a 'b 'c 'd)

```

```
203 (m:c-length-equal (m:bar 'a 'd)
204                   (m:bar 'b 'c))
205 (m:c-right-angle (m:joint 'd))
206 (m:c-right-angle (m:joint 'b)))
```

Listing A.35: learning/load.scm

```
1 ;;; load.scm -- Load learning module
2 (for-each (lambda (f) (load f))
3           '("definitions"
4             "student"
5             "conjecture"
6             "simplifier"
7             "core-knowledge"))
```

Listing A.36: learning/core-knowledge.scm

```

1  ;; core-knowledge.scm -- Core knowledge of a student
2
3  ;; Commentary:
4
5  ;; Code:
6
7  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Adding to student ;;;;;;;;;;;;;;;;;;
8
9  (define (provide-core-knowledge student)
10   (for-each (lambda (def)
11               (add-definition! student def))
12             primitive-definitions)
13   (for-each (lambda (def)
14               (add-definition! student def))
15             built-in-definitions))
16
17  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Primitive definitions ;;;;;;;;;;;;;;;;;;
18
19  (define primitive-definitions
20   (list
21    (make-primitive-definition 'point point? random-point)
22    (make-primitive-definition 'line line? random-line)
23    (make-primitive-definition 'segment segment? random-segment)
24    (make-primitive-definition 'polygon polygon? random-polygon)
25    (make-primitive-definition 'circle circle? random-circle)
26    (make-primitive-definition 'angle angle? random-angle)))
27
28  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Built-in Definitions ;;;;;;;;;;;;;;;;;;
29
30  (define (polygon-n-sides-conjecture n)
31   (make-conjecture
32    '(polygon)
33    '<premise>)
34   (list car)
35   (make-polygon-n-sides-relationship n)))
36
37  (define built-in-definitions
38   (list
39    ;; Triangle
40    (make-restrictions-definition
41     'triangle '(polygon)
42     (list (polygon-n-sides-conjecture 3))
43     random-triangle)
44    ;; Quadrilateral
45    (make-restrictions-definition
46     'quadrilateral '(polygon)
47     (list (polygon-n-sides-conjecture 4))
48     random-quadrilateral)
49
50    ;; Isoceles Triangle!
51    #|
52    (make-restrictions-definition
53     'isoeles-triangle 'triangle
54     (list (lambda (t)
55              (let* ((a (polygon-point-ref t 0))
56                     (b (polygon-point-ref t 1))
57                     (c (polygon-point-ref t 2)))
58                (segment-equal-length? (make-segment a b)
59                                         (make-segment a c))))
60     random-isoeles-triangle))
61  |#
62  ))
63

```

Listing A.37: learning/definitions.scm

```

1  ;; definitions.scm --- representation and interaction with definitions
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - primitive definitions
7
8  ;; Future:
9  ;; - relationship-based definitions
10
11 ;; Code:
12
13 ;;;;;;;;;;;;;;;;;;;;;;;;;; Basic Structure ;;;;;;;;;;;;;;;;;;;;;;;;;;
14
15 (define-record-type <definition>
16   (%make-definition name classifications conjectures predicate generator)
17   definition?
18   (name definition-name)
19   (classifications definition-classifications)
20   (conjectures definition-conjectures)
21   (predicate definition-predicate set-definition-predicate!)
22   (generator definition-generator))
23
24 (define (make-primitive-definition name predicate generator)
25   (%make-definition name '() '() predicate generator))
26
27 (define (primitive-definition? def)
28   (and (definition? def)
29        (null? (definition-classifications def))))
30
31 ;;;;;;;;;;;;;;;;;;;;;;;;;; Higher-order Definitions ;;;;;;;;;;;;;;;;;;;;;;;;;;
32
33 (define (make-restrictions-definition
34         name classifications conjectures generator)
35   (%make-definition name classifications conjectures #f generator))
36
37 ;;;;;;;;;;;;;;;;;;;;;;;;;; Formatting ;;;;;;;;;;;;;;;;;;;;;;;;;;
38
39 (define (print-definition def)
40   (list (definition-name def)
41         (definition-classifications def)
42         (map print (definition-conjectures def))))
43
44 (defhandler print print-definition
45   definition?)
46
47 (define (print-primitive-definition def)
48   'primitive-definition)
49
50 (defhandler print print-primitive-definition
51   primitive-definition?)

```

Listing A.38: learning/conjecture.scm

```

1 ;; conjecture -- a proposed conjecture based on an observed relationship
2
3 ;;; Commentary
4
5 ;; Ideas:
6 ;; - Higher-level than raw observations reported by perception/analyzer
7
8 ;; Future:
9 ;; - More complicated premises
10 ;; - "Pattern-matching"
11
12 ;;; Code:
13
14 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Conjecture ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
15
16 (define-record-type <conjecture>
17   (make-conjecture premises constructions construction-procedures
18     relationship)
19   conjecture?
20   (premises conjecture-premises)
21   (constructions conjecture-constructions)
22   (construction-procedures conjecture-construction-procedures)
23   (relationship conjecture-relationship))
24
25 (define (print-conjecture conj)
26   (cons
27     (print (conjecture-relationship conj))
28     (conjecture-constructions conj)))
29
30 (defhandler print print-conjecture conjecture?)
31
32 (define (conjecture-equal? conj1 conj2)
33   (equal? (print conj1)
34     (print conj2)))
35
36 ;;; Whether
37 (define (satisfies-conjecture conj premise-instances)
38   (let ((new-args
39     (map
40       (lambda (construction-proc)
41         (construction-proc premise-instances))
42       (conjecture-construction-procedures conj))))
43     (rel (conjecture-relationship conj)))
44     (or (relationship-holds rel new-args)
45       #f
46       (begin (if *explain*
47         (pprint `(failed-conjecture ,conj)))
48         #f)))))
49
50
51 (define (conjecture-from-observation obs)
52   (make-conjecture
53     '()
54     (map element-dependencies->list (observation-args obs))
55     (map element-source (observation-args obs))
56     (observation-relationship obs)))
57
58
59 ;;; Removing redundant conjectures
60
61 (define (simplify-conjectures conjectures base-conjectures)
62   (define memp (member-procedure conjecture-equal?))
63   (filter
64     (lambda (o) (not (memp o base-conjectures)))
65     conjectures))

```


Listing A.39: learning/simplifier.scm

```

1  ;;; simplifier.scm --- simplifies definitions
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - interfaces to manipulator
7
8  ;; Future:
9  ;; - Support more complex topologies.
10
11 ;;; Code:
12
13 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Main Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
14
15 (define (observations->constraints observations)
16   (filter identity (map observation->constraint observations)))
17
18 (define (observation->constraint obs)
19   (let ((rel (observation-relationship obs))
20         (args (observation-args obs)))
21     (let ((constraint-proc (relationship->constraint rel))
22           (linkage-ids (args->linkage-ids args)))
23       (and constraint-proc
24             (every identity linkage-ids)
25             (apply constraint-proc
26                     (args->linkage-ids args))))))
27
28 (define (relationship->constraint rel)
29   (case (relationship-type rel)
30     ((equal-length) m:c-length-equal)
31     ((equal-angle) m:c-angle-equal)
32     (else #f)))
33
34 (define (args->linkage-ids args)
35   (map arg->linkage-id args))
36
37 (define arg->linkage-id (make-generic-operation 1 'arg->linkage-id
38                                                false-proc))
39
40 (define (segment->bar-id segment)
41   (m:bar (element-name (segment-endpoint-1 segment))
42          (element-name (segment-endpoint-2 segment))))
43 (defhandler arg->linkage-id segment->bar-id segment?)
44
45 (define (angle->joint-id angle)
46   (m:joint (element-name (angle-vertex angle))))
47 (defhandler arg->linkage-id angle->joint-id angle?)
48
49 (define (establish-polygon-topology-for-n-gon n-sides)
50   (cond ((= n-sides 3)
51          (m:establish-polygon-topology 'a 'b 'c))
52         ((= n-sides 4)
53          (m:establish-polygon-topology 'a 'b 'c 'd))))
54
55 (define (observations->figure topology observations)
56   (initialize-scheduler)
57   (pprint (observations->constraints observations))
58   (let ((m (apply
59             m:mechanism
60             (list
61              topology
62              (observations->constraints observations))))))
63     (m:build-mechanism m)
64     (if (not (m:solve-mechanism m))
65         (begin
66           (pp "Could not solve mechanism"))

```

```

67         #f)
68     (let ((f (m:mechanism->figure m)))
69         (pp "Solved!")
70         (show-figure f)
71         f)))
72
73 (define (topology-for-object obj)
74     (if (polygon? obj)
75         (establish-polygon-topology-for-n-gon
76             (polygon-n-points obj))
77         (error "Object isn't a polygon")))
78
79 ;; TODO: Make more general
80 (define (polygon-from-figure figure)
81     (let ((all-points (figure-points figure)))
82         (let lp ((i 1)
83                 (pts '()))
84             (let ((p (find-point all-points
85                                   (nth-letter-symbol i))))
86                 (if p
87                     (lp (+ i 1)
88                         (append pts (list p)))
89                     (apply polygon-from-points pts))))))
90
91 (define (find-point points name)
92     (let ((pts (filter
93                 (lambda (p)
94                     (eq? (element-name p) name))
95                 points)))
96         (and (not (null? pts))
97              (car pts))))

```

Listing A.40: learning/student.scm

```

1  ;; student.scm -- base model of a student's knowlege
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Definitions, constructions, theorems
7  ;; - "What is"
8
9  ;; Future:
10 ;; - Simplifiers of redudant / uninteresting info
11 ;; - Propose own investigations?
12
13 ;; Code:
14
15 ;;;;;;;;;;;;;; Student Structure ;;;;;;;;;;;;;;
16
17 (define-record-type <student>
18   (%make-student definitions)
19   student?
20   (definitions student-definitions))
21
22 (define (make-student)
23   (%make-student (make-key-weak-eq-hash-table)))
24
25 ;;;;;;;;;;;;;; Building Predicates ;;;;;;;;;;;;;;
26
27 (define (build-predicate-for-definition s def)
28   (let ((classifications (definition-classifications def))
29         (conjectures (definition-conjectures def)))
30     (let ((classification-predicate
31           (lambda (obj)
32             (every
33              (lambda (classification)
34                (or ((definition-predicate (student-lookup s classification))
35                    obj)
36                    (begin (if *explain*
37                              (pprint `(failed-classification
38                                      ,classification)))
39                          #f)))
34              classifications))))
35       (lambda args
36         (and (apply classification-predicate args)
37              (every (lambda (o) (satisfies-conjecture o args))
38                    conjectures))))))
39
40 ;;;;;;;;;;;;;; Definitions ;;;;;;;;;;;;;;
41
42 (define (add-definition! s def)
43   (if (not (definition-predicate def))
44       (set-definition-predicate!
45        def
46        (build-predicate-for-definition s def)))
47   (hash-table/put! (student-definitions s)
48                    (definition-name def)
49                    def))
50
51 (define (lookup-definition s name)
52   (hash-table/get (student-definitions s)
53                   name
54                   #f))
55
56 ;;;;;;;;;;;;;; Current Student ;;;;;;;;;;;;;;
57
58 (define *current-student* #f)
59
60 (define (student-lookup s term)

```

```

67 (or (lookup-definition s term)
68     *unknown*)
69
70 ;;;;;;;;;;;;;;;;;;;;;;;;;; Query ;;;;;;;;;;;;;;;;;;;;;;;;;;
71
72 (define (lookup term)
73   (student-lookup *current-student* term))
74
75 (define *unknown* 'unknown)
76 (define (unknown? x)
77   (eq? x *unknown*))
78
79 (define (what-is term)
80   (pprint (lookup term)))
81
82 (define *explain* #f)
83
84 (define (is-a? term obj)
85   (let ((def (lookup term)))
86     (if (unknown? def)
87         `(.term unknown)
88         (fluid-let ((*explain* #t)
89                     ((definition-predicate def) obj))))))
90
91 (define (internal-is-a? term obj)
92   (let ((def (lookup term)))
93     (if (unknown? def)
94         `(.term unknown)
95         ((definition-predicate def) obj))))
96
97 (define (show-me term)
98   (let ((def (lookup term)))
99     (if (unknown? def)
100        `(.term unknown)
101        (show-element ((definition-generator def))))))
102
103 (define (examine object)
104   (show-element object)
105   (let ((applicable-terms
106         (filter (lambda (term)
107                   (internal-is-a? term object))
108                 (all-known-terms))))
109     applicable-terms))
110
111 (define (all-known-terms)
112   (hash-table/key-list
113     (student-definitions *current-student*)))
114
115 ;;;;;;;;;;;;;;;;;;;;;;;;;; Simplifying base terms ;;;;;;;;;;;;;;;;;;;;;;;;;;
116
117 (define (simplify-base-terms terms)
118   (let ((parent-terms (append-map
119                       (lambda (t) (definition-classifications (lookup t)))
120                       terms)))
121     (filter (lambda (t) (not (memq t parent-terms)))
122             terms)))
123
124 ;;;;;;;;;;;;;;;;;;;;;;;;;; Graphics Interfaces ;;;;;;;;;;;;;;;;;;;;;;;;;;
125
126 (define (show-element element)
127   (if (polygon? element)
128       (name-polygon element)
129       (show-figure (figure element))))
130
131 (define (show-figure figure)
132   (draw-figure figure c))
133
134 ;;;;;;;;;;;;;;;;;;;;;;;;;; Analyzing Elements ;;;;;;;;;;;;;;;;;;;;;;;;;;

```

```

135
136 (define (analyze-element element)
137   (if (polygon? element)
138       (name-polygon element))
139   (let ((fig (figure (with-dependency '<premise> element'))))
140     (show-figure fig)
141     (let ((obs-list (analyze-figure fig)))
142       (map observation-with-premises obs-list))))
143
144 ;;;;;;;;;;;;;; Initializing Student ;;;;;;;;;;;;;;
145
146 (define (initialize-student)
147   (let ((s (make-student)))
148     (provide-core-knowledge s)
149     (set! *current-student* s)))
150
151
152 (define (learn-term term object-generator)
153   (let ((v (lookup term)))
154     (if (not (eq? v 'unknown))
155         (pprint `(already-known ,term))
156         (let ((example (name-polygon (object-generator))))
157           (let* ((base-terms (examine example))
158                  (simple-base-terms (simplify-base-terms base-terms))
159                  (base-definitions (map lookup base-terms))
160                  (base-conjectures (flatten (map definition-conjectures
161                                                  base-definitions)))
162                  (fig (figure (with-dependency '<premise> example')))
163                  (observations (analyze-figure fig))
164                  (conjectures (map conjecture-from-observation observations))
165                  (simplified-conjectures
166                    (simplify-conjectures conjectures base-conjectures)))
167             (pprint conjectures)
168             (let ((new-def
169                    (make-restrictions-definition
170                     term
171                     simple-base-terms
172                     simplified-conjectures
173                     object-generator)))
174               (add-definition! *current-student* new-def)
175               'done))))))
176
177 (define (get-simple-definitions term)
178   (let ((def (lookup term)))
179     (if (unknown? def)
180         (error "Unknown term" term))
181     (let* ((object ((definition-generator def)))
182            (observations
183              (filter
184               observation->constraint
185               (all-observations
186                (figure (name-polygon object))))))
187      (map
188       (lambda (obs-subset)
189         (pprint obs-subset)
190         (let* ((topology (topology-for-object object))
191                (new-figure
192                  (observations->figure topology obs-subset)))
193           (if new-figure
194               (let ((new-polygon
195                      (polygon-from-figure new-figure)))
196                 (pprint new-polygon)
197                 (if (is-a? term new-polygon)
198                     (list 'valid-definition
199                           obs-subset)
200                     (list 'invalid-definition
201                           obs-subset)))
202               (list 'unknown-definition

```

```
203         obs-subset)))  
204 (all-subsets observations))))
```

Listing A.41: learning/walkthrough.scm

```

1  ;; Sample:
2
3  ;; Looking up terms
4
5  ;; Starts with limited knowledge
6
7  (what-is 'square)
8
9  (what-is 'rhombus)
10
11 ;; Knows primitive objects
12
13 (what-is 'line)
14
15 (what-is 'point)
16
17 (what-is 'polygon)
18
19 ;; And some built-in non-primitives
20
21 (what-is 'triangle)
22
23 (what-is 'quadrilateral)
24
25 ;; Can identify whether elements satisfy these
26
27 (show-element (random-parallelogram))
28
29 (is-a? 'polygon (random-square))
30
31 (is-a? 'quadrilateral (random-square))
32
33 (is-a? 'triangle (random-square))
34
35 (is-a? 'segment (random-square))
36
37 (is-a? 'line (random-line))
38
39 ;; Can learn and explain new terms
40
41 (what-is 'isoc-t)
42
43 (learn-term 'isoc-t random-isocles-triangle)
44
45 (what-is 'isoc-t)
46
47 (is-a? 'isoc-t (random-isocles-triangle))
48
49 (is-a? 'isoc-t (random-equilateral-triangle))
50
51 (is-a? 'isoc-t (random-triangle))
52
53 (learn-term 'equi-t random-equilateral-triangle)
54
55 (what-is 'equi-t)
56
57 (is-a? 'equi-t (random-isocles-triangle))
58
59 (is-a? 'equi-t (random-equilateral-triangle))
60
61 ;; Let's learn some basic quadrilaterals
62
63 (learn-term 'pl random-parallelogram)
64
65 (what-is 'pl)
66

```

```
67 (learn-term 'kite random-kite)
68
69 (what-is 'kite)
70
71 (learn-term 'rh random-rhombus)
72
73 (what-is 'rh)
74
75 (learn-term 'rectangle random-rectangle)
76
77 (what-is 'rectangle)
78
79 (learn-term 'sq random-square)
80
81 (what-is 'sq)
```


Listing A.42: content/load.scm

```
1 ;;; load.scm -- Load learning module
2 (for-each (lambda (f) (load f))
3         '("investigations"
4           "thesis-demos"))
```

Listing A.43: content/thesis-demos.scm

```

1  ;; thesis-demos.scm -- Examples for thesis demonstration chapter
2
3  ;; Code
4
5  ;; Basic Figure Example ;;
6
7  (define (triangle-with-perp-bisectors)
8    (let-geo* ((a (make-point 0 0))
9                (b (make-point 1.5 0))
10               (c (make-point 1 1))
11               (t (polygon-from-points a b c))
12               (pb1 (perpendicular-bisector (make-segment a b)))
13               (pb2 (perpendicular-bisector (make-segment b c)))
14               (pb3 (perpendicular-bisector (make-segment c a))))
15      (figure t pb1 pb2 pb3)))
16
17 (define (demo-figure-0)
18   (let-geo* (((s (a b)) (random-segment))
19             (pb (perpendicular-bisector s))
20             (p (random-point-on-line pb)))
21     (figure s pb
22            (make-segment a p)
23            (make-segment b p))))
24
25 (define (incircle-circumcircle)
26   (let-geo* (((t (a b c)) (random-triangle))
27             (((a-1 a-2 a-3)) (polygon-angles t))
28             (ab1 (angle-bisector a-1))
29             (ab2 (angle-bisector a-2))
30             ((radius-segment (center-point radius-point))
31              (perpendicular-to (make-segment a b)
32                                (intersect-linear-elements ab1 ab2)))
33             (incircle (circle-from-points
34                        center-point
35                        radius-point))
36             (pb1 (perpendicular-bisector
37                  (make-segment a b)))
38             (pb2 (perpendicular-bisector
39                  (make-segment b c)))
40             (pb-center (intersect-lines pb1 pb2))
41             (circum-cir (circle-from-points
42                          pb-center
43                          a)))
44     (figure t a-1 a-2 a-3
45            pb-center
46            radius-segment
47            incircle
48            circum-cir)))
49
50
51 (define (is-this-a-rectangle-2)
52   (m:mechanism
53    (m:establish-polygon-topology 'a 'b 'c 'd)
54    (m:c-length-equal (m:bar 'a 'd)
55                      (m:bar 'b 'c))
56    (m:c-right-angle (m:joint 'd))
57    (m:c-angle-equal (m:joint 'a)
58                    (m:joint 'c))))
59
60 (define (random-triangle-with-perp-bisectors)
61   (let-geo* ((t (random-triangle))
62             (a (polygon-point-ref t 0))
63             (b (polygon-point-ref t 1))
64             (c (polygon-point-ref t 2))
65             (pb1 (perpendicular-bisector (make-segment a b)))
66             (pb2 (perpendicular-bisector (make-segment b c)))

```

```

67         (pb3 (perpendicular-bisector (make-segment c a))))
68     (figure t pb1 pb2 pb3)))
69
70 (define (random-triangle-with-perp-bisectors)
71   (let-geo* (((t (a b c)) (random-triangle))
72             (pb1 (perpendicular-bisector (make-segment a b)))
73             (pb2 (perpendicular-bisector (make-segment b c)))
74             (pb3 (perpendicular-bisector (make-segment c a))))
75     (figure t pb1 pb2 pb3)))
76
77 (define (angle-bisector-distance)
78   (let-geo* (((a (r-1 v r-2)) (random-angle))
79             (ab (angle-bisector a))
80             (p (random-point-on-ray ab))
81             ((s-1 (p b)) (perpendicular-to r-1 p))
82             ((s-2 (p c)) (perpendicular-to r-2 p)))
83     (figure a r-1 r-2 ab p s-1 s-2)))
84
85 (define (simple-mechanism)
86   (m:mechanism
87     (m:make-named-bar 'a 'b)
88     (m:make-named-bar 'b 'c)
89     (m:make-named-joint 'a 'b 'c)
90     (m:c-right-angle (m:joint 'b))))
91
92 (define (parallelogram-figure)
93   (let-geo* (((p (a b c d)) (random-parallelogram)))
94     (figure p)))
95
96 (define (m:quadrilateral-with-intersecting-diagonals a b c d e)
97   (list (m:establish-polygon-topology a b e)
98         (m:establish-polygon-topology b c e)
99         (m:establish-polygon-topology c d e)
100        (m:establish-polygon-topology d a e)
101        (m:c-line-order c e a)
102        (m:c-line-order b e d)))
103
104 (define (kite-from-diagonals)
105   (m:mechanism
106     (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
107     (m:c-right-angle (m:joint 'b 'e 'c)) ;; Right Angle in Center
108     (m:c-length-equal (m:bar 'c 'e) (m:bar 'a 'e))))
109
110 (define (isocles-trapezoid-from-diagonals)
111   (m:mechanism
112     (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
113     (m:c-length-equal (m:bar 'a 'e) (m:bar 'b 'e))
114     (m:c-length-equal (m:bar 'c 'e) (m:bar 'd 'e))))
115
116 (define (parallelogram-from-diagonals)
117   (m:mechanism
118     (m:quadrilateral-with-intersecting-diagonals 'a 'b 'c 'd 'e)
119     (m:c-length-equal (m:bar 'a 'e) (m:bar 'c 'e))
120     (m:c-length-equal (m:bar 'b 'e) (m:bar 'd 'e))))

```

Listing A.44: core/load.scm

```
1 ;;; load.scm -- Load core
2 (for-each (lambda (f) (load f))
3           '("utils"
4             "macros"
5             "print"
6             "animation"))
```

Listing A.45: core/animation.scm

```

1  ;; animation.scm --- Animating and persisting values in figure constructions
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Animate a range
7  ;; - persist randomly chosen values across frames
8
9  ;; Future:
10 ;; - Backtracking, etc.
11 ;; - Save continuations?
12
13 ;; Code:
14
15 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Configurations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
16
17 (define *animation-steps* 15)
18
19 ;; ~30 Frames per second:
20 (define *animation-sleep* 30)
21
22 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Internal Constants ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
23 (define *is-animating?* #f)
24 (define *animation-value* 0)
25 (define *next-animation-index* 0)
26 (define *animating-index* 0)
27
28 (define (run-animation f-with-animations)
29   (fluid-let ((*is-animating?* #t)
30               (*persistent-values-table* (make-key-weak-eq-hash-table)))
31     (let lp ((animate-index 0))
32       (fluid-let
33         ((*animating-index* animate-index))
34         (let run-frame ((frame 0))
35           (fluid-let ((*next-animation-index* 0)
36                       (*next-value-index* 0)
37                       (*animation-value*
38                         (/ frame (* 1.0 *animation-steps*))))
39             (f-with-animations)
40             (sleep-current-thread *animation-sleep*)
41             (if (< frame *animation-steps*)
42                 (run-frame (+ frame 1))
43                 (if (< *animating-index* (- *next-animation-index* 1))
44                     (lp (+ animate-index 1))))))))))
45
46 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Animating Functions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
47
48 ;; f should be a function of one float argument in [0, 1]
49 (define (animate f)
50   (let ((my-index *next-animation-index*))
51     (set! *next-animation-index* (+ *next-animation-index* 1))
52     (f (cond ((< *animating-index* my-index) 0)
53            ((= *animating-index* my-index) *animation-value*)
54            ((> *animating-index* my-index) 1))))))
55
56 (define (animate-range min max)
57   (animate (lambda (v)
58              (+ min
59                 (* v (- max min))))))
60
61 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Persistence ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
62
63 (define *persistent-values-table* #f)
64 (define *next-value-index* 0)
65
66 (define (persist-value v)

```

```

67 (if (not *is-animating?*)
68     v
69     (let* ((my-index *next-value-index*)
70             (table-value (hash-table/get
71                             *persistent-values-table*
72                             my-index
73                             #f)))
74             (set! *next-value-index* (+ *next-value-index* 1))
75             (or table-value
76                 (begin
77                     (hash-table/put! *persistent-values-table*
78                                         my-index
79                                         v)
80                     v))))))

```

Listing A.46: core/macros.scm

```

1  ;; macros.scm --- Macros for let-geo* to assign names and variables
2  ;; to elements
3
4  ;; Commentary:
5
6  ;; Ideas:
7  ;; - Basic naming
8  ;; - Multiple assignment
9
10 ;; Future:
11 ;; - Warn about more errors
12 ;; - More efficient multiple-assignment for lists
13
14 ;; Code:
15
16 ;;;;;;;;;;;;;;;;;;;;;;;;;; Expanding Assignment ;;;;;;;;;;;;;;;;;;;;;;;;;;
17
18 (define *multiple-assignment-symbol* '*multiple-assignment-result*)
19
20 (define (expand-multiple-assignment lhs rhs)
21   (expand-compound-assignment
22     (list *multiple-assignment-symbol* lhs)
23     rhs))
24
25 (define (make-component-assignments key-name component-names)
26   (map (lambda (name i)
27         (list name `(element-component ,key-name ,i)))
28        component-names
29        (iota (length component-names))))
30
31 (define (expand-compound-assignment lhs rhs)
32   (if (not (= 2 (length lhs)))
33       (error "Malformed compound assignment LHS (needs 2 elements): " lhs))
34   (let ((key-name (car lhs))
35         (component-names (cadr lhs)))
36     (if (not (list? component-names))
37         (error "Component names must be a list:" component-names))
38     (let ((main-assignment (list key-name rhs))
39           (component-assignments (make-component-assignments
40                                   key-name
41                                   component-names)))
42       (cons main-assignment
43             component-assignments))))
44
45 (define (expand-assignment assignment)
46   (if (not (= 2 (length assignment)))
47       (error "Assignment in letgeo* must be of length 2, found:" assignment))
48   (let ((lhs (car assignment))
49         (rhs (cadr assignment)))
50     (if (list? lhs)
51         (if (= (length lhs) 1)
52             (expand-multiple-assignment (car lhs) rhs)
53             (expand-compound-assignment lhs rhs))
54         (list assignment))))
55
56 (define (expand-assignments assignments)
57   (append-map expand-assignment assignments))
58
59 ;;;;;;;;;;;;;;;;;;;;;;;;;; Extract Variable Names ;;;;;;;;;;;;;;;;;;;;;;;;;;
60
61 (define (variables-from-assignment assignment)
62   (flatten (list (car assignment))))
63
64 (define (variables-from-assignments assignments)
65   (append-map variables-from-assignment assignments))
66

```

```

67 (define (set-name-expressions symbols)
68   (map (lambda (s)
69         `(set-element-name! ,s (quote ,s)))
70        symbols))
71
72 ;;;;;;;;;;;;;; let-geo* ;;;;;;;;;;;;;;
73
74 ;;; Syntax for setting names for geometry objects declared via let-geo
75 (define-syntax let-geo*
76   (sc-macro-transformer
77     (lambda (exp env)
78       (let ((assignments (cadr exp))
79             (body (caddr exp)))
80         (let ((new-assignments (expand-assignments assignments))
81               (variable-names (variables-from-assignments assignments)))
82           (let ((result `(let*
83                           ,new-assignments
84                           ,@(set-name-expressions variable-names)
85                           ,body)))
86             result))))))

```


Listing A.47: core/print.scm

```

1
2 ;;; print.scm --- Print things nicely
3
4 ;;; Commentary:
5 ;;; - Default printing is not very nice for many of our record structure
6
7 ;;; Code:
8
9 ;;;;;;;;;;;;; Print ;;;;;;;;;;;;;;
10
11 (define print
12   (make-generic-operation 1 'print (lambda (x) x)))
13
14 (defhandler print
15   (lambda (p) (cons (print (car p))
16                     (print (cdr p))))
17   pair?)
18
19 (defhandler print
20   (lambda (l) (map print l))
21   list?)
22
23 (define (pprint x)
24   (pp (print x))
25   (display "\n"))

```

Listing A.48: core/utils.scm

```

1  ;; close-enuf? floating point comparison from scmutils
2  ;; Origin: Gerald Jay Sussman
3
4  (define *machine-epsilon*
5    (let loop ((e 1.0))
6      (if (= 1.0 (+ e 1.0))
7          (* 2 e)
8          (loop (/ e 2)))))
9
10 (define *sqrt-machine-epsilon*
11   (sqrt *machine-epsilon*))
12
13 #|
14 (define (close-enuf? h1 h2 tolerance)
15   (<= (magnitude (- h1 h2))
16       (* .5 (max tolerance *machine-epsilon*
17                  (+ (magnitude h1) (magnitude h2) 2.0)))))
18 |#
19
20 (define (close-enuf? h1 h2 #!optional tolerance scale)
21   (if (default-object? tolerance)
22       (set! tolerance (* 10 *machine-epsilon*))
23       (if (default-object? scale)
24           (set! scale 1.0))
25       (<= (magnitude (- h1 h2))
26           (* tolerance
27              (+ (* 0.5
28                  (+ (magnitude h1) (magnitude h2)))
29                  scale)))))
30
31 (define (assert boolean error-message)
32   (if (not boolean) (error error-message)))
33
34 (define (flatten list)
35   (cond ((null? list) '())
36         ((list? (car list))
37          (append (flatten (car list))
38                  (flatten (cdr list))))
39         (else (cons (car list) (flatten (cdr list))))))
40
41 (define ((notp predicate) x)
42   (not (predicate x)))
43
44 (define ((andp p1 p2) x)
45   (and (p1 x)
46        (p2 x)))
47
48 (define (true-proc . args) #t)
49 (define (false-proc . args) #f)
50
51 (define (identity x) x)
52
53 ;; ps1 \ ps2
54 (define (set-difference set1 set2 member-predicate)
55   (define delp (delete-member-procedure list-deletor member-predicate))
56   (let lp ((set1 set1)
57            (set2 set2))
58     (if (null? set2)
59         set1
60         (let ((e (car set2)))
61           (lp (delp e set1)
62               (cdr set2))))))
63
64 (define (set-intersection set1 set2 member-predicate)
65   (let lp ((set1 (dedupe member-predicate set1))
66            (intersection '()))
66     (if (null? set1)
67         intersection
68         (let ((e (car set1)))
9

```

```

67     (if (null? set1)
68         intersection
69         (let ((e (car set1)))
70             (lp (cdr set1)
71                 (if (member-predicate e set2)
72                     (cons e intersection)
73                     intersection))))))
74
75 (define (eq-append! element key val)
76     (eq-put! element key
77         (cons val
78             (or (eq-get element key) '()))))
79
80 (define (sort-by-key l key)
81     (sort l (lambda (v1 v2)
82         (< (key v1)
83            (key v2)))))
84
85 (define (sort-by-key-2 l key)
86     (let ((v (sort-by-key-2 l key)))
87         (pprint (map (lambda (x) (cons (name x) (key x))) v))
88         v))
89
90 (define ((negatep f) x)
91     (- (f x)))
92
93 (define ((flip-args f) x y)
94     (f y x))
95
96 (define (index-of el list equality-predicate)
97     (let lp ((i 0)
98         (l list))
99         (cond ((null? l) #f)
100             ((equality-predicate (car l) el)
101              i)
102             (else (lp (+ i 1) (cdr l))))))
103
104 ;;; (nth-letter-symbol 1) => 'a , 2 => 'b, etc.
105 (define (nth-letter-symbol i)
106     (symbol (make-char (+ 96 i) 0)))
107
108 (define (hash-table/append table key element)
109     (hash-table/put! table
110         key
111         (cons element
112             (hash-table/get table key '()))))
113
114 (define (dedupe-by equality-predicate elements)
115     (dedupe (member-procedure equality-predicate) elements))
116
117 (define (dedupe member-predicate elements)
118     (cond ((null? elements) '())
119           (else
120            (let ((b1 (car elements)))
121                (if (member-predicate b1 (cdr elements))
122                    (dedupe member-predicate (cdr elements))
123                    (cons b1 (dedupe member-predicate (cdr elements)))))))
124
125
126 (define (partition-into-equivalence-classes elements equivalence-predicate)
127     (let lp ((equivalence-classes '())
128         (remaining-elements elements))
129         (if (null? remaining-elements)
130             equivalence-classes
131             (lp
132                 (add-to-equivalence-classes
133                     equivalence-classes
134                     (car remaining-elements)

```

```

135         (member-procedure equivalence-predicate))
136         (cdr remaining-elements))))))
137
138 (define (add-to-equivalence-classes classes element memp)
139   (if (null? classes)
140       (list (list element))
141       (let ((first-class (car classes))
142             (remaining-classes (cdr classes)))
143         (if (memp element first-class)
144             (cons (cons element first-class)
145                   remaining-classes)
146             (cons first-class
147                   (add-to-equivalence-classes remaining-classes
148                                               element
149                                               memp))))))
150
151 (define (all-subsets elements)
152   (append-map
153     (lambda (n)
154       (all-n-tuples n elements))
155     (iota (+ (length elements) 1))))

```

Listing A.49: lib/eq-properties.scm

```

1  ;;; Traditional LISP property lists
2  ;;; extended to work on any kind of eq? data structure.
3
4  (declare (usual-integrations))
5
6  ;;; Property lists are a way of creating data that looks like a record
7  ;;; structure without committing to the fields that will be used until
8  ;;; run time. The use of such flexible structures is frowned upon by
9  ;;; most computer scientists, because it is hard to statically
10 ;;; determine the bounds of the behavior of a program written using
11 ;;; this stuff. But it makes it easy to write programs that confuse
12 ;;; such computer scientists. I personally find it difficult to write
13 ;;; without such crutches. -- GJS
14
15
16 (define eq-properties (make-eq-hash-table))
17
18 (define (eq-put! node property value)
19   (let ((plist (hash-table/get eq-properties node #f)))
20     (if plist
21         (let ((vcell (assq property (cdr plist))))
22           (if vcell
23               (set-cdr! vcell value)
24               (set-cdr! plist
25                           (cons (cons property value)
28                               (cdr plist))))))
29         (hash-table/put! eq-properties node
30                           (list node (cons property value)))))
31   'done)
32
33 (define (eq-adjoin! node property new)
34   (eq-put! node property
35             (eq-set/adjoin new
36                             (or (eq-get node property) '()))))
37   'done)
38
39 (define (eq-rem! node property)
40   (let ((plist (hash-table/get eq-properties node #f)))
41     (if plist
42         (let ((vcell (assq property (cdr plist))))
43           (if vcell
44               (hash-table/put! eq-properties node (delq! vcell plist))))))
45   'done)
46
47 (define (eq-get node property)
48   (let ((plist (hash-table/get eq-properties node #f)))
49     (if plist
50         (let ((vcell (assq property (cdr plist))))
51           (if vcell
52               (cdr vcell)
53               #f)))
54   #f))
55
56 (define (eq-plist node)
57   (hash-table/get eq-properties node #f))
58
59 (define (eq-path path)
60   (define (lp node)
61     (if node
62         (if (pair? path)
63             (eq-get ((eq-path (cdr path)) node)
64                       (car path))
65             node)
66         #f))

```

67 lp)

Listing A.50: lib/ghelper.scm

```

1
2 (define make-generic-operation make-generic-operator)
3
4 #|
5 ;;; Most General Generic-Operator Dispatch
6 (declare (usual-integrations)) ; for compiler
7
8 ;;; Generic-operator dispatch is implemented here by a
9 ;;; discrimination list (a "trie", invented by Ed Fredkin),
10 ;;; where the arguments passed to the operator are examined
11 ;;; by predicates that are supplied at the point of
12 ;;; attachment of a handler. (Handlers are attached by
13 ;;; ASSIGN-OPERATION alias DEFHANDLER).
14
15 ;;; The discrimination list has the following structure: it
16 ;;; is an improper alist whose "keys" are the predicates
17 ;;; that are applicable to the first argument. If a
18 ;;; predicate matches the first argument, the cdr of that
19 ;;; alist entry is a discrimination list for handling the
20 ;;; rest of the arguments. Each discrimination list is
21 ;;; improper: the cdr at the end of the backbone of the
22 ;;; alist is the default handler to apply (all remaining
23 ;;; arguments are implicitly accepted).
24
25 ;;; A successful match of an argument continues the search
26 ;;; on the next argument. To be the correct handler all
27 ;;; arguments must be accepted by the branch predicates, so
28 ;;; this makes it necessary to backtrack to find another
29 ;;; branch where the first argument is accepted if the
30 ;;; second argument is rejected. Here backtracking is
31 ;;; implemented using #f as a failure return, requiring
32 ;;; further search.
33
34 #| ;;; For example.
35 (define foo (make-generic-operator 2 'foo))
36
37 (defhandler foo + number? number?)
38
39 (define (symbolic? x)
40   (or (symbol? x)
41       (and (pair? x) (symbolic? (car x)) (list? (cdr x)))))
42
43 (define (+:symb x y) (list '+ x y))
44
45 (defhandler foo +:symb number? symbolic?)
46 (defhandler foo +:symb symbolic? number?)
47 (defhandler foo +:symb symbolic? symbolic?)
48
49 (foo 1 2)
50 ;Value: 3
51
52 (foo 1 'a)
53 ;Value: (+ 1 a)
54
55 (foo 'a 1)
56 ;Value: (+ a 1)
57
58 (foo ' (+ a 1) ' (+ 1 a))
59 ;Value: (+ (+ a 1) (+ 1 a))
60 |#
61
62 (define (make-generic-operator arity
63         #!optional name default-operation)
64   (let ((record (make-operator-record arity)))

```

```

65
66 (define (operator . arguments)
67   (if (not (acceptable-arglist? arguments arity))
68       (error:wrong-number-of-arguments
69         (if (default-object? name) operator name)
70         arity arguments))
71   (apply (find-handler (operator-record-tree record)
72                       arguments)
73          arguments))
74
75 (set-operator-record! operator record)
76
77 (set! default-operation
78   (if (default-object? default-operation)
79       (named-lambda (no-handler . arguments)
80         (error "Generic operator inapplicable:"
81               (if (default-object? name) operator name)
82               arguments))
83       default-operation))
84 (if (not (default-object? name)) ; Operation by name
85     (set-operator-record! name record))
86
87 (assign-operation operator default-operation)
88 operator))
89
90 #|
91 ;; To illustrate the structure we populate the
92 ;; operator table with quoted symbols rather
93 ;; than actual procedures.
94
95 (define blend
96   (make-generic-operator 2 'blend 'blend-default))
97
98 (pp (get-operator-record blend))
99 (2 . blend-default)
100
101 (defhandler blend 'b+b 'blue? 'blue?)
102 (defhandler blend 'g+b 'green? 'blue?)
103 (defhandler blend 'b+g 'blue? 'green?)
104 (defhandler blend 'g+g 'green? 'green?)
105
106 (pp (get-operator-record blend))
107 (2 (green? (green? . g+g) (blue? . g+b))
108   (blue? (green? . b+g) (blue? . b+b))
109   .
110   blend-default)
111 |#
112
113 #|
114 ;; Backtracking
115
116 ;; An operator satisfies bleen?
117 ;; if it satisfies either blue? or green?
118
119 (defhandler blend 'e+r 'bleen? 'red?)
120 (defhandler blend 'e+u 'bleen? 'grue?)
121
122 (pp (get-operator-record blend))
123 (2 (bleen? (grue? . e+u) (red? . e+r))
124   (green? (green? . g+g) (blue? . g+b))
125   (blue? (green? . b+g) (blue? . b+b))
126   .
127   blend-default)
128
129 ;; Consider what happens if we invoke
130 ;; (blend <bleen> <blue>)

```



```

131 |#

132
133 ;; This is the essence of the search.
134
135 (define (find-handler tree args)
136   (if (null? args)
137       tree
138       (find-branch tree
139                     (car args)
140                     (lambda (result)
141                       (find-handler result
142                                   (cdr args))))))
143
144 (define (find-branch tree arg next)
145   (let loop ((tree tree))
146     (cond ((pair? tree)
147            (or (and ((caar tree) arg)
148                    (next (cdar tree)))
149                (loop (cdr tree))))
150            ((null? tree) #f)
151            (else tree))))
152
153 (define (assign-operation operator handler
154               . argument-predicates)
155   (let ((record (get-operator-record operator))
156         (arity (length argument-predicates)))
157     (if record
158         (begin
159           (if (not (<= arity
160                       (procedure-arity-min
161                        (operator-record-arity record))))
162               (error "Incorrect operator arity:" operator))
163           (bind-in-tree argument-predicates
164                         handler
165                         (operator-record-tree record)
166                         (lambda (new)
167                           (set-operator-record-tree! record
168                                                         new))))
169         (error "Undefined generic operator" operator)))
170   operator)
171
172 (define defhandler assign-operation)
173
174 (define (bind-in-tree keys handler tree replace!)
175   (let loop ((keys keys) (tree tree) (replace! replace!))
176     (cond ((pair? keys) ; more argument-predicates
177           (let find-key ((tree* tree))
178             (if (pair? tree*)
179                 (if (eq? (caar tree*) (car keys))
180                     ;; There is already some discrimination
181                     ;; list keyed by this predicate: adjust it
182                     ;; according to the remaining keys
183                     (loop (cdr keys)
184                           (cdar tree*)
185                           (lambda (new)
186                             (set-cdr! (car tree*) new)))
187                     (find-key (cdr tree*)))
188                 (let ((better-tree
189                        (cons (cons (car keys) '()) tree)))
190                     ;; There was no entry for the key I was
191                     ;; looking for. Create it at the head of
192                     ;; the alist and try again.
193                     (replace! better-tree)
194                     (loop keys better-tree replace!))))))
195     ;; cond continues on next page.

```

```

196
197      ((pair? tree) ; no more argument predicates.
198      ;; There is more discrimination list here,
199      ;; because my predicate list is a proper prefix
200      ;; of the predicate list of some previous
201      ;; assign-operation. Insert the handler at the
202      ;; end, causing it to implicitly accept any
203      ;; arguments that fail all available tests.
204      (let ((p (last-pair tree)))
205        (if (not (null? (cdr p)))
206            (warn "Replacing a default handler:"
207                  (cdr p) handler))
208            (set-cdr! p handler)))
209      (else
210      ;; There is no discrimination list here. This
211      ;; handler becomes the discrimination list,
212      ;; accepting further arguments if any.
213      (if (not (null? tree))
214          (warn "Replacing a handler:" tree handler))
215      (replace! handler))))

216
217 (define *generic-operator-table* (make-eq-hash-table))
218
219 (define (get-operator-record operator)
220   (hash-table/get *generic-operator-table* operator #f))
221
222 (define (set-operator-record! operator record)
223   (hash-table/put! *generic-operator-table* operator
224                     record))
225
226 (define (make-operator-record arity) (cons arity '()))
227 (define (operator-record-arity record) (car record))
228 (define (operator-record-tree record) (cdr record))
229 (define (set-operator-record-tree! record tree)
230   (set-cdr! record tree))
231
232 (define (acceptable-arglist? lst arity)
233   (let ((len (length lst)))
234     (and (fix:<= (procedure-arity-min arity) len)
235          (or (not (procedure-arity-max arity))
236              (fix:>= (procedure-arity-max arity) len)))))

237
238 #|
239 ;;; Demonstration of handler tree structure.
240 ;;; Note: symbols were used instead of procedures
241
242 (define foo (make-generic-operator 3 'foo 'foo-default))
243
244 (pp (get-operator-record foo))
245 (3 . foo-default)
246
247 (defhandler foo 'two-arg-a-b 'a 'b)
248 (pp (get-operator-record foo))
249 (3 (a (b . two-arg-a-b)) . foo-default)
250
251 (defhandler foo 'two-arg-a-c 'a 'c)
252 (pp (get-operator-record foo))
253 (3 (a (c . two-arg-a-c) (b . two-arg-a-b)) . foo-default)
254
255 (defhandler foo 'two-arg-b-c 'b 'c)
256 (pp (get-operator-record foo))
257 (3 (b (c . two-arg-b-c))
258     (a (c . two-arg-a-c) (b . two-arg-a-b))
259     . foo-default)
260 |#

```

```

261
262 #|
263 (defhandler foo 'one-arg-b 'b)
264 (pp (get-operator-record foo))
265 (3 (b (c . two-arg-b-c) . one-arg-b)
266     (a (c . two-arg-a-c) (b . two-arg-a-b))
267     . foo-default)
268
269 (defhandler foo 'one-arg-a 'a)
270 (pp (get-operator-record foo))
271 (3 (b (c . two-arg-b-c) . one-arg-b)
272     (a (c . two-arg-a-c) (b . two-arg-a-b) . one-arg-a)
273     .
274     foo-default)
275
276 (defhandler foo 'one-arg-a-prime 'a)
277 ;Warning: Replacing a default handler:
278 ;         one-arg-a one-arg-a-prime
279
280 (defhandler foo 'two-arg-a-b-prime 'a 'b)
281 ;Warning: Replacing a handler:
282 ;         two-arg-a-b two-arg-a-b-prime

283
284 (defhandler foo 'three-arg-x-y-z 'x 'y 'z)
285 (pp (get-operator-record foo))
286 (3 (x (y (z . three-arg-x-y-z)))
287     (b (c . two-arg-b-c) . one-arg-b)
288     (a (c . two-arg-a-c)
289         (b . two-arg-a-b-prime)
290         .
291         one-arg-a-prime)
292     .
293     foo-default)
294 |#

295
296 ;;; Compatibility with previous extensible generics
297
298 (define make-generic-operation make-generic-operator)
299
300 (define (add-to-generic-operation! operator
301                                       applicability
302                                       handler)
303   ;; An applicability is a list representing a
304   ;; disjunction of lists, each representing a
305   ;; conjunction of predicates.
306
307   (for-each (lambda (conj)
308               (apply assign-operation
309                       operator
310                       handler
311                       conj)))
312   applicability))
313
314 |#

```


Bibliography