Appendix A

Code Listings

Listings

Listing A.1: load.scm

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

;;; Code:

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

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

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

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

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

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

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

'done-loading
```

Listing A.2: main.scm

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


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

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

;;; Other Examples:

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

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

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

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

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

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

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

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

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

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

(define (angle-test)
  (let-geo* (((t (a b c)) (random-triangle))
             (a-1 (smallest-angle (angle-from-points a b c)))
             (a-2 (smallest-angle (angle-from-points b c a)))
             (a-3 (smallest-angle (angle-from-points c a b)))
             (l1 (angle-bisector a-1))
             (l2 (angle-bisector a-2))
             (l3 (angle-bisector a-3))
             (center-point
              (intersect-lines (ray->line l1)
                               (ray->line l2)))
             (radius-line
              (perpendicular (line-from-points b c)
                             center-point))
             (radius-point
              (intersect-lines radius-line
                               (line-from-points b c)))
             (cir (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 cir a-1 a-2 a-3
            pb-center
            circum-cir
            center-point)))

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

(define current-figure demo-figure)

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

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

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


(define (run-figure current-figure-proc)
  (let ((analysis-data (make-analysis-collector)))
    (run-animation
     (lambda ()
```

```
      (let ((current-figure (current-figure-proc)))
        (draw-figure current-figure c)
        (let ((analysis-results (analyze-figure current-figure)))
          (save-results (print analysis-results) analysis-data))
        )))
   (display "--- Results ---\n")
   (analyze-figure current-figure)
   (print-analysis-results analysis-data)))

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

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

;(r)
```

Listing A.3: learning/load.scm

```
;;; load.scm -- Load learning module
(for-each (lambda (f) (load f))
          '("definitions"
            "student"
            "core-knowledge"
            "simplifier"))
```

Listing A.4: learning/core-knowledge.scm

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

;;; Commentary:

;;; Code:

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

(define (provide-core-knowledge student)
  (for-each (lambda (def)
              (add-definition! student def))
            primitive-definitions)
  (for-each (lambda (def)
              (add-definition! student def))
            built-in-definitions))

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

(define primitive-definitions
  (list
   (make-primitive-definition 'point point? random-point)
   (make-primitive-definition 'line line? random-line)
   (make-primitive-definition 'segment segment? random-segment)
   (make-primitive-definition 'polygon polygon? random-polygon)
   (make-primitive-definition 'circle circle? random-circle)
   (make-primitive-definition 'angle angle? random-angle)))

;;;;;;;;;;;;;;;;;;;;;;;; Built-in Definitions ;;;;;;;;;;;;;;;;;;;;;;;;

(define (polygon-n-sides-observation n)
  (make-observation
   '()
   (make-polygon-n-sides-relationship n)
   (list (with-source car '<premise>))))

(define built-in-definitions
  (list
   ;; Triangle
   (make-restrictions-definition
    'triangle '(polygon)
    (list (polygon-n-sides-observation 3))
    random-triangle)
   ;; Quadrilateral
   (make-restrictions-definition
    'quadrilateral '(polygon)
    (list (polygon-n-sides-observation 4))
    random-quadrilateral)

   ;; Isoceles Triangle!
   #|
   (make-restrictions-definition
    'isoceles-triangle 'triangle
    (list (lambda (t)
            (let* ((a (polygon-point-ref t 0))
                   (b (polygon-point-ref t 1))
                   (c (polygon-point-ref t 2)))
              (segment-equal-length? (make-segment a b)
                                     (make-segment a c)))))

   random-isoceles-triangle))
|#
   ))
```

Listing A.5: learning/definitions.scm

```
;;; definitions.scm --- representation and interaction with definitions

;;; Commentary:

;; Ideas:
;; - primitive definitions

;; Future:
;; - relationship-based definitions

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;; Basic Structure ;;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <definition>
  (%make-definition name classifications observations predicate generator)
  definition?
  (name definition-name)
  (classifications definition-classifications)
  (observations definition-observations)
  (predicate definition-predicate set-definition-predicate!)
  (generator definition-generator))

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

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

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

(define (make-restrictions-definition
         name classifications observations generator)
  (%make-definition name classifications observations #f generator))

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

(define (print-definition def)
  (list (definition-name def)
        (definition-classifications def)
        (map print (definition-observations def))))

(defhandler print print-definition
  definition?)

(define (print-primitive-definition def)
  'primitive-definition)

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

Listing A.6: learning/simplifier.scm

```
;;; simplifier.scm --- simplifies definitions

;;; Commentary:

;; Ideas:
;; - interfaces to manipulator

;; Future:
;; - Support more complex topologies.

;;; Code:

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

(define (simplify-definition
         n-sides
         relationships)
  #f)

(define (relationships->constraints relationships)
  '())

(define (relationship->constraint rel)
  '())

(define (establish-polygon-topology-for-n-gon n-sides)
  (cond ((= n-sides 3)
         (m:establish-polygon-topology 'a 'b 'c))
        ((= n-sides 4)
         (m:establish-polygon-topology 'a 'b 'c 'd))))

(define (relationships->figure n-sides relationships)
  (initialize-scheduler)
  (let ((m (apply
            m:mechanism
            (cons (establish-polygon-topology-for-n-gon n-sides)
                  (relationships->constraints relationships)))))
    (m:build-mechanism m)
    (m:solve-mechanism m)
    (let ((f (m:mechanism->figure m)))
      f)))
```

Listing A.7: learning/student.scm

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

;;; Commentary:

;; Ideas:
;; - Definitions, constructions, theorems
;; - "What is"

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

;;; Code:

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

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

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


;;;;;;;;;;;;;;;;;;;;;;;; Building Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;

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

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

(define (add-definition! s def)
  (if (not (definition-predicate def))
      (set-definition-predicate!
       def
       (build-predicate-for-definition s def)))
  (hash-table/put! (student-definitions s)
                   (definition-name def)
                   def))

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

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

(define *current-student* #f)

```

```
(define (student-lookup s term)
  (lookup-definition s term))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Query ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (lookup term)
  (let ((result (student-lookup *current-student* term)))
    (if (not result)
        'unknown
        result)))

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

(define *explain* #f)

(define (is-a? term obj)
  (show-element obj)
  (let ((def (lookup term)))
    (if (eq? def 'unknown)
        `(,term unknown)
        (fluid-let ((*explain* #t))
          ((definition-predicate def) obj)))))

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

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

(define (examine object)
  (show-element object)
  (let ((base-terms (filter (lambda (term)
                              (internal-is-a? term object))
                            (hash-table/key-list
                             (student-definitions *current-student*)))))
    base-terms))

;;;;;;;;;;;;;;;;;;;;;;; Simplifying base terms ;;;;;;;;;;;;;;;;;;;;;;;

(define (simplify-base-terms terms)
  (let ((parent-terms (append-map
                       (lambda (t) (definition-classifications (lookup t)))
                           terms)))
    (filter (lambda (t) (not (memq t parent-terms)))
            terms)))

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

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

;;;;;;;;;;;;;;;;;;;;;;;; Initializing Student ;;;;;;;;;;;;;;;;;;;;;;;;

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


```

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

Listing A.8: learning/walkthrough.scm

```
;;; Sample:

;;;;;;;;;;;;;;;;;;;;;;;;; Looking up terms ;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Starts with limited knowledge

(what-is 'square)

(what-is 'rhombus)

;;; Knows primitive objects

(what-is 'line)

(what-is 'point)

(what-is 'polygon)

;;; And some built-in non-primitives

(what-is 'triangle)

(what-is 'quadrilateral)

;;;;;;;;;;;; Can idenitfy whether elements satisfy these ;;;;;;;;;;;;

(is-a? 'polygon (random-square))

(is-a? 'quadrilateral (random-square))

(is-a? 'triangle (random-square))

(is-a? 'segment (random-square))

(is-a? 'line (random-line))

;;;;;;;;;;;;;;;;; Can learn and explain new terms ;;;;;;;;;;;;;;;;;

(what-is 'isoc-t)

(learn-term 'isoc-t random-isoceles-triangle)

(what-is 'isoc-t)

(is-a? 'isoc-t (random-isoceles-triangle))

(is-a? 'isoc-t (random-equilateral-triangle))

(is-a? 'isoc-t (random-triangle))

(learn-term 'equi-t random-equilateral-triangle)

(what-is 'equi-t)

(is-a? 'equi-t (random-isoceles-triangle))

(is-a? 'equi-t (random-equilateral-triangle))

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

(learn-term 'pl random-parallelogram)

(what-is 'pl)

(learn-term 'kite random-kite)

```

```
(what-is 'kite)

(learn-term 'rh random-rhombus)

(what-is 'rh)

(learn-term 'rectangle random-rectangle)

(what-is 'rectangle)

(learn-term 'sq random-square)

(what-is 'sq)
```

Listing A.9: figure/load.scm

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

Listing A.10: figure/core.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

#|
Example Usage:

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

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

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

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

Listing A.11: figure/line.scm

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

;;; Commentary:

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

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

;;; Code:

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

(define-record-type &lt;segment&gt;
  (%segment p1 p2)
  segment?
  (p1 segment-endpoint-1)
  (p2 segment-endpoint-2))

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

;;; Alternate, helper constructors

(define (make-segment p1 p2)
  (let ((seg (%segment p1 p2)))
    (set-element-name!
     seg
     (symbol '*seg*: (element-name p1) '- (element-name p2)))
    (with-dependency
     `(segment ,p1 ,p2)
     seg)))

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

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

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

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

(define make-line %make-line)

```

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

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

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

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

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

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


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

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

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

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

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

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

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

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

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

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

```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; Transforms ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(define (segment->line segment)
```

```
  (ray->line (segment->ray segment)))

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

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

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

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

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

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

Listing A.12: figure/direction.scm

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

;;; Commentary:

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

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

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

;;; Code:

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

Listing A.13: figure/direction.scm

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

;;; Commentary:

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

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

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

;;; Code:

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

Listing A.14: figure/vec.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

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

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

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

;;; Returns new vecs

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

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

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

(define (reverse-vec v)
  (make-vec (- (vec-x v))
            (- (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.15: figure/measurements.scm

```
;;; measurements.scm

;;; Commentary:

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

;; Future:
;; - Arc Measure

;;; Code:

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

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

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

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

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

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

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

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

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

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

Listing A.16: figure/angle.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

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

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

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

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

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

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

```
    (make-angle d1 p d2)))
(defhandler angle-from angle-from-lines line? line?)

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

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

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

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

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

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

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

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

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

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

```
      (linear-pair-internal? a2 a1)))

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

Listing A.17: figure/bounds.scm

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

;;; Commentary:

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

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

;;; Code:

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

;;; Max bounds of the graphics window

(define *g-min-x* -1)
(define *g-max-x*  1)
(define *g-min-y* -1)
(define *g-max-y*  1)

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

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

(define (ray-extend-to-max-segment p1 p2)
  (let ((x1 (point-x p1))
        (y1 (point-y p1))
        (x2 (point-x p2))
        (y2 (point-y p2)))
    (let ((dx (- x2 x1))
          (dy (- y2 y1)))
      (cond
       ((= 0 dx) (make-segment
                  (make-point x1 *g-min-y*)
                  (make-point x1 *g-max-y*)))
```

```
67        ((= 0 dy) (make-segment
68                   (make-point *g-min-x* y1)
69                   (make-point *g-min-y* y1)))
70        (else
71         (let ((t-xmin (/ (- *g-min-x* x1) dx))
72               (t-xmax (/ (- *g-max-x* x1) dx))
73               (t-ymin (/ (- *g-min-y* y1) dy))
74               (t-ymax (/ (- *g-max-y* y1) dy)))
75           (let* ((sorted (sort (list t-xmin t-xmax t-ymin t-ymax) <))
76                  (min-t (cadr sorted))
77                  (max-t (caddr sorted))
78                  (min-x (+ x1 (* min-t dx)))
79                  (min-y (+ y1 (* min-t dy)))
80                  (max-x (+ x1 (* max-t dx)))
81                  (max-y (+ y1 (* max-t dy))))
82             (make-segment p1
83                           (make-point max-x max-y)))))))))
```

Listing A.18: figure/circle.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

Listing A.19: figure/point.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

(defhandler print
  print-point point?)

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

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

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

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

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

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

Listing A.20: figure/constructions.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

(define (on-ray? p r)
```

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

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

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

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

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

(define (perpendicular-bisector segment)
  (let ((midpt (segment-midpoint segment)))
    (perpendicular (segment->line segment)
                   midpt)))

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

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

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

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

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

(define (concurrent? l1 l2 l3)
  (let ((i-point (intersect-linear-elements l1 l2)))
    (and i-point
         (on-element? i-point l3))))

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

```
            (pb-3 (perpendicular-bisector
                   (make-segment p3 p4))))
        (concurrent? pb-1 pb-2 pb-3))))

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

Listing A.21: figure/intersections.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

```
               (x1 (+ a (/ (* e k) p)))
               (y1 (+ b (/ (* f k) p)))
               (dx (/ (* f t) p))
               (dy (- (/ (* e t) p))))
          (list (make-point (+ x1 dx)
                            (+ y1 dy))
                (make-point (- x1 dx)
                            (- y1 dy)))))))

;;; Intersect circle centered at c with radius r and line through
;;; points p1, p2
;;; http://mathworld.wolfram.com/Circle-LineIntersection.html
(define (intersect-circle-line-by-points c r p1 p2)
  (let ((offset (sub-points (make-point 0 0) c)))
    (let ((p1-shifted (add-to-point p1 offset))
          (p2-shifted (add-to-point p2 offset)))
      (let ((x1 (point-x p1-shifted))
            (y1 (point-y p1-shifted))
            (x2 (point-x p2-shifted))
            (y2 (point-y p2-shifted)))
        (let* ((dx (- x2 x1))
               (dy (- y2 y1))
               (dr (sqrt (+ (square dx) (square dy))))
               (d (det x1 x2 y1 y2))
               (disc (- (* (square r) (square dr)) (square d))))
          (if (< disc 0)
              (list)
              (let ((x-a (* d dy))
                    (x-b (* (sgn dy) dx (sqrt disc)))
                    (y-a (- (* d dx)))
                    (y-b (* (abs dy) (sqrt disc))))
                (let ((ip1 (make-point
                            (/ (+ x-a x-b) (square dr))
                            (/ (+ y-a y-b) (square dr))))
                      (ip2 (make-point
                            (/ (- x-a x-b) (square dr))
                            (/ (- y-a y-b) (square dr)))))
                  (if (close-enuf? 0 disc) ;; Tangent
                      (list (add-to-point ip1 (reverse-vec offset)))
                      (list (add-to-point ip1 (reverse-vec offset))
                            (add-to-point ip2 (reverse-vec offset))))))))))))

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

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

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

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

(define (intersect-circle-line cir line)
  (let ((center (circle-center cir))
        (radius (circle-radius cir))
        (p1 (line-p1 line))
```

```
        (p2 (line-p2 line)))
    (intersect-circle-line-by-points center radius p1 p2)))

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

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

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

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

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

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

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

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

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

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

Listing A.22: figure/figure.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

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

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

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

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

Listing A.23: figure/math-utils.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Basic Operators ;;;;;;;;;;;;;;;;;;;;;;;;;;

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

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

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

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

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

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

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

Listing A.24: figure/polygon.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

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

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

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

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

(define (name-polygon polygon)
  (for-each (lambda (i)
              (set-element-name! (polygon-point-ref polygon i)
```

```
                                (nth-letter-symbol (+ i 1))))
            (iota (polygon-n-points polygon)))
  polygon)

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

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

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

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

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

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

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

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

```

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

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

Listing A.25: figure/metadata.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

Listing A.26: figure/dependencies.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

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


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

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

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

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

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

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

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

```
  (num %random-dependency-num set-random-dependency-num!))

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

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

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

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

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

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

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

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

Listing A.27: figure/randomness.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

(define *wiggle-ratio* 0.15)

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

;;; Random Values - distances, angles

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

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

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

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

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

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

(define (random-point-around p)
  (let ((x (point-x p))
```

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

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

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

(define (random-point-on-segment seg)
  (let* ((p1 (segment-endpoint-1 seg))
         (p2 (segment-endpoint-2 seg))
         (t (rand-range 0.0 1.0))
         (v (sub-points p2 p1)))
    (add-to-point p1 (scale-vec v t))))

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

(define (random-circle)
  (let ((pr1 (random-point))
```

```
        (pr2 (random-point)))
    (circle-from-points (midpoint pr1 pr2) pr1)))

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

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

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

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

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

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

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

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

```

```
;;;;;;;;;;;;;;;;;;;;;; Random Quadrilaterals ;;;;;;;;;;;;;;;;;;;;;;

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

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

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

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

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

(define (random-rhombus)
```

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

Listing A.28: figure/transforms.scm

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

;;; Commentary:

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

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

;;; Code:

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

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

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

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

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

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

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

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

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

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

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

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

(define (translate-angle-by vec a)
  (define (translate-point p) (translate-point-by vec p))
```

```
  (make-angle (angle-arm-1 a)
              (translate-point (angle-vertex a))
              (angle-arm-2 a)))

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

;;; Reflections

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

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

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

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

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

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

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

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

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

Listing A.29: perception/load.scm

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

Listing A.30: perception/observation.scm

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

;;; Commentary:

;;; Code:

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

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

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

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

(defhandler print print-observation observation?)


;;;;;;;;;;;;;;;;;;;;;;; Checking observation ;;;;;;;;;;;;;;;;;;;;;;;

(define (satisfies-observation obs new-premise)
  (let ((new-args
         (map (lambda (arg)
                ((element-source arg) new-premise))
              (observation-args obs)))
        (rel (observation-relationship obs)))
    (or (relationship-holds rel new-args)
        (begin (if *explain*
                   (pprint `(failed-observation ,obs)))
               #f))))

;;;;;;;;;;;;;;;;;;;;; Simplifying observations ;;;;;;;;;;;;;;;;;;;;;

(define (simplify-observations observations base-observations)
  (define memp (member-procedure observation-equal?))
  (filter
   (lambda (o) (not (memp o base-observations)))
   observations))
```

Listing A.31: perception/analyzer.scm

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

;;; Commentary

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

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

;;; Code:

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

(define (analyze-figure figure)
  (analyze figure))

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

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

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

;;;;;;;;;;;;;;;;;;;;;; Cross products, pairs ;;;;;;;;;;;;;;;;;;;;;;

;;; General proceudres for generating pairs
(define (all-pairs elements)
```

```
  (all-n-tuples 2 elements))

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

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

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

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

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

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

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

```
135
136 ;;;;;;;;;;;;;;;;;;;;;;;; Dealing with pairs ;;;;;;;;;;;;;;;;;;;;;;;;
137
138 ;;; Check for pairwise equality
139 (define ((nary-predicate n predicate) tuple)
140   (apply predicate tuple))
141
142 ;;; Merges "connected-components" of pairs
143 (define (merge-pair-groups elements pairs)
144   (let ((i 0)
145         (group-ids (make-key-weak-eq-hash-table))
146         (group-elements (make-key-weak-eq-hash-table))) ; Map from pair
147     (for-each (lambda (pair)
148                 (let ((first (car pair))
149                       (second (cadr pair)))
150                   (let ((group-id-1 (hash-table/get group-ids first i))
151                         (group-id-2 (hash-table/get group-ids second i)))
152                     (cond ((and (= group-id-1 i)
153                                 (= group-id-2 i))
154                            ;; Both new, new groups:
155                            (hash-table/put! group-ids first group-id-1)
156                            (hash-table/put! group-ids second group-id-1))
157                           ((= group-id-1 i)
158                            (hash-table/put! group-ids first group-id-2))
159                           ((= group-id-2 i)
160                            (hash-table/put! group-ids second group-id-1)))
161                     (set! i (+ i 1)))))
162               pairs)
163     (for-each (lambda (elt)
164                 (hash-table/append group-elements
165                                    (hash-table/get group-ids elt 'invalid)
166                                    elt))
167               elements)
168     (hash-table/remove! group-elements 'invalid)
169     (hash-table/datum-list group-elements)))
170
171 (define (report-n-wise n predicate elements)
172   (let ((tuples (all-n-tuples n elements)))
173     (filter (nary-predicate n predicate) tuples)))
174
175 ;;;;;;;;;;;;;;;;;;;;;;;;;;;; Results: ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
176
177 (define (make-analysis-collector)
178   (make-equal-hash-table))
179
180 (define (save-results results data-table)
181   (hash-table/put! data-table results
182                    (+ 1 (hash-table/get data-table results 0))))
183
184 (define (print-analysis-results data-table)
185   (hash-table/for-each
186    data-table
187    (lambda (k v)
188      (pprint (list v (cons 'discovered k))))))
```

Listing A.32: graphics/load.scm

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

Listing A.33: graphics/appearance.scm

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

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

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

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

Listing A.34: graphics/graphics.scm

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

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

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

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

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


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

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

```
                       (symbol->string (element-name segment))))))

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

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

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

(define *angle-mark-radius* 0.1)
(define (draw-angle canvas a)
  (let* ((vertex (angle-vertex a))
         (d1 (angle-arm-1 a))
         (d2 (angle-arm-2 a))
         (angle-start (direction-theta d2))
         (angle-end (direction-theta d1)))
    (canvas-draw-arc canvas
                     (point-x vertex)
                     (point-y vertex)
                     *angle-mark-radius*
                     angle-start
                     angle-end)))

;;; Add to generic operations

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

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

;;; Canvas for x-graphics

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

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

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

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

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

```

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

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

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

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

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

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

Listing A.35: manipulate/load.scm

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

Listing A.36: manipulate/linkages.scm

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

;;; Commentary:

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

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

;;; Example:

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

;;; Code:

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

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

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

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

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

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

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

(propagatify m:reverse-direction)

(define (ce:reverse-direction input-cell)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

(propagatify m:add-to-direction)

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

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

```

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

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

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

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

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

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

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

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

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

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

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

(propagatify m:subtract-directions)

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


;;; Allocate and wire up the cells in a vec
(define (m:make-vec vec-id)
```

```
  (let-cells (dx dy length direction)
    (name! dx (symbol vec-id '-dx))
    (name! dy (symbol vec-id '-dy))
    (name! length (symbol vec-id '-len))
    (name! direction (symbol vec-id '-dir))

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

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

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

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

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

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

(propagatify m:x-y->region)

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

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

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

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

```
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))
```

```
      nothing))

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

(define (m:x-y-length-di->region px py length dir-interval)
  (if (direction-interval? dir-interval)
      (let ((vertex (make-point px py)))
        (m:make-arc vertex length dir-interval))
      nothing))

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

(define (m:p1->p2-bar-propagator p1 p2 bar)
  (let ((p1x (m:point-x p1))
        (p1y (m:point-y p1))
        (p2r (m:point-region p2))
        (length (m:bar-length bar))
        (dir (m:bar-direction bar)))
    (p:m:x-y-direction->region p1x p1y dir p2r)
    (p:m:x-y-length-di->region p1x p1y length dir p2r)))

(define (m:p2->p1-bar-propagator p2 p1 bar)
  (let ((p2x (m:point-x p2))
        (p2y (m:point-y p2))
        (p1r (m:point-region p1))
        (length (m:bar-length bar))
        (dir (m:bar-direction bar)))
    (p:m:x-y-direction->region p2x p2y (ce:reverse-direction dir) p1r)
    (p:m:x-y-length-di->region p2x p2y length (ce:reverse-direction dir) p1r)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Joint  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Direction-2 is counter-clockwise from direction-1 by theta
(define-record-type <m:joint>
  (%m:make-joint vertex dir-1 dir-2 theta)
  m:joint?
  (vertex m:joint-vertex)
  (dir-1 m:joint-dir-1)
  (dir-2 m:joint-dir-2)
  (theta m:joint-theta))

(define *max-joint-swing* pi)

(define (m:make-joint joint-id)
  (let ((joint-key (m:make-joint-name-key joint-id)))
   (let ((vertex (m:make-point (symbol joint-key '-vertex))))
     (let-cells (dir-1 dir-2 theta)
       (name! dir-1 (symbol joint-key '-dir-1))
       (name! dir-2 (symbol joint-key '-dir-2))
       (name! theta (symbol joint-key '-theta))
       (name! vertex (symbol joint-key '-vertex))
       (p:m:add-to-direction
        dir-1 theta dir-2)
       (p:m:add-to-direction
        dir-2 (e:negate theta) dir-1)
       (p:m:subtract-directions
        dir-2 dir-1
        theta)
       (m:instantiate theta (make-interval 0 *max-joint-swing*) 'theta)
       (%m:make-joint vertex dir-1 dir-2 theta)))))

(define (m:print-joint j)
  `(m:joint
    ,(print (m:joint-name j))
    ,(print (m:joint-dir-1 j))
    ,(print (m:joint-vertex j))
    ,(print (m:joint-dir-2 j))
    ,(print (m:joint-theta j))))

```

```
(defhandler print m:print-joint m:joint?)

;;; TOOD: Abstract?
(define (m:identify-out-of-arm-1 joint bar)
  (m:set-endpoint-1 bar joint)
  (m:set-joint-arm-1 joint bar)
  (m:identify-points (m:joint-vertex joint)
                     (m:bar-p1 bar))
  (c:id (m:joint-dir-1 joint)
        (m:bar-direction bar)))

(define (m:identify-out-of-arm-2 joint bar)
  (m:set-endpoint-1 bar joint)
  (m:set-joint-arm-2 joint bar)
  (m:identify-points (m:joint-vertex joint)
                     (m:bar-p1 bar))
  (c:id (m:joint-dir-2 joint)
        (m:bar-direction bar)))

(define (m:identify-into-arm-1 joint bar)
  (m:set-endpoint-2 bar joint)
  (m:set-joint-arm-1 joint bar)
  (m:identify-points (m:joint-vertex joint)
                     (m:bar-p2 bar))
  (c:id (ce:reverse-direction (m:joint-dir-1 joint))
        (m:bar-direction bar)))

(define (m:identify-into-arm-2 joint bar)
  (m:set-endpoint-2 bar joint)
  (m:set-joint-arm-2 joint bar)
  (m:identify-points (m:joint-vertex joint)
                     (m:bar-p2 bar))
  (c:id (ce:reverse-direction (m:joint-dir-2 joint))
        (m:bar-direction bar)))

;;;;;;;;;;;;;;;;;;;;;;;; Storing Adjacencies ;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:set-endpoint-1 bar joint)
  (eq-append! bar 'm:bar-endpoints-1 joint))

(define (m:bar-endpoints-1 bar)
  (or (eq-get bar 'm:bar-endpoints-1)
      '()))

(define (m:set-endpoint-2 bar joint)
  (eq-append! bar 'm:bar-endpoints-2 joint))

(define (m:bar-endpoints-2 bar)
  (or (eq-get bar 'm:bar-endpoints-2)
      '()))

(define (m:set-joint-arm-1 joint bar)
  (eq-put! joint 'm:joint-arm-1 bar))

(define (m:joint-arm-1 joint)
  (eq-get joint 'm:joint-arm-1))

(define (m:set-joint-arm-2 joint bar)
  (eq-put! joint 'm:joint-arm-2 bar))

(define (m:joint-arm-2 joint)
  (eq-get joint 'm:joint-arm-2))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Named Linkages  ;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:make-bar-name-key bar-id)
  (symbol 'm:bar:
          (m:bar-id-p1-name bar-id) ':
```

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

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

```
(define (m:make-joints-by-vertex-name-table joints)
  (let ((table (make-key-weak-eq-hash-table)))
    (for-each
     (lambda (joint)
       (let ((key (m:joint-vertex-name joint)))
         (hash-table/put!
          table key
          (cons
           joint (hash-table/get table
                                 key
                                 '()))))))
     joints)
    table))

(define (m:find-joint-by-vertex-name table vertex-name)
  (let ((joints (hash-table/get table
                                vertex-name
                                #f)))
    (cond ((null? joints) #f)
          ((= (length joints) 1)
           (car joints))
          (else (error "Vertex name not unique among joints"
                       (map m:joint-name joints))))))

(define (m:make-joints-by-name-table joints)
  (let ((table (make-key-weak-eq-hash-table)))
    (for-each (lambda (joint)
                (hash-table/put! table
                                 (m:make-joint-name-key (m:joint-name joint))
                                 joint))
              joints)
    table))

;;; dir-2 is CCW from dir-1
(define (m:find-joint-by-id table joint-id)
  (hash-table/get
   table
   (m:make-joint-name-key joint-id)
   #f))

;;;;;;;;;;;;;;;;;;;;;; Operations using Names ;;;;;;;;;;;;;;;;;;;;;;

(define (m:identify-joint-bar-by-name joint bar)
  (let ((vertex-name (m:joint-vertex-name joint))
        (dir-1-name (m:joint-dir-1-name joint))
        (dir-2-name (m:joint-dir-2-name joint))
        (bar-p1-name (m:bar-p1-name bar))
        (bar-p2-name (m:bar-p2-name bar)))
    (cond ((eq? vertex-name bar-p1-name)
           (cond ((eq? dir-1-name bar-p2-name)
                  (m:identify-out-of-arm-1 joint bar))
                 ((eq? dir-2-name bar-p2-name)
                  (m:identify-out-of-arm-2 joint bar))
                 (else (error "Bar can't be identified with joint - no arm"
                              bar-p2-name))))
          ((eq? vertex-name bar-p2-name)
           (cond ((eq? dir-1-name bar-p1-name)
                  (m:identify-into-arm-1 joint bar))
                 ((eq? dir-2-name bar-p1-name)
                  (m:identify-into-arm-2 joint bar))
                 (else (error "Bar can't be identified with joint - no arm"
                              bar-p1-name))))
          (else (error "Bar can't be identified with joint - no vertex"
                       vertex-name)))))

;;;;;;;;;;;;;;;;;;;;;;; Degrees of Freedom  ;;;;;;;;;;;;;;;;;;;;;;

(define (m:specified? cell #!optional predicate)
```

```
  (let ((v (m:examine-cell cell)))
    (and
     (not (nothing? v))
     (or (default-object? predicate)
         (predicate v)))))

(define (m:bar-length-specified? bar)
  (m:specified? (m:bar-length bar)) number?)

(define (m:bar-direction-specified? bar)
  (m:specified? (m:bar-direction bar)) direction?)

(define (m:joint-theta-specified? joint)
  (m:specified? (m:joint-theta joint)) number?)

;;;;;;;;;;;;;;;;;;;;;;;;; Point Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:point-specified? p)
  (and (m:specified? (m:point-x p) number?)
       (m:specified? (m:point-y p) number?)))

(define (m:point-contradictory? p)
  (or (m:contradictory? (m:point-x p))
      (m:contradictory? (m:point-y p))
      (m:contradictory? (m:point-region p))))

;;;;;;;;;;;;;;;;;;;;;;;;;; Bar Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:bar-p1-specified? bar)
  (m:point-specified? (m:bar-p1 bar)))

(define (m:bar-p2-specified? bar)
  (m:point-specified? (m:bar-p2 bar)))

(define (m:bar-p1-contradictory? bar)
  (m:point-contradictory? (m:bar-p1 bar)))

(define (m:bar-p2-contradictory? bar)
  (m:point-contradictory? (m:bar-p2 bar)))

(define (m:bar-anchored? bar)
  (or (m:bar-p1-specified? bar)
      (m:bar-p2-specified? bar)))

(define (m:bar-directioned? bar)
  (and (m:bar-anchored? bar)
       (m:specified? (m:bar-direction bar) direction?)))

(define (m:bar-direction-contradictory? bar)
  (or (m:contradictory? (m:bar-direction bar))
      (m:contradictory? (m:vec-dx (m:bar-vec bar)))
      (m:contradictory? (m:vec-dy (m:bar-vec bar)))))

(define (m:bar-length-specified? bar)
  (and (m:specified? (m:bar-length bar) number?)))

(define (m:bar-direction-specified? bar)
  (and (m:specified? (m:bar-direction bar) number?)))

(define (m:bar-length-contradictory? bar)
  (m:contradictory? (m:bar-length bar)))

(define (m:bar-length-dir-specified? bar)
  (and (m:bar-length-specified? bar)
       (m:bar-direction-specified? bar)))

(define (m:bar-fully-specified? bar)
  (and (m:bar-p1-specified? bar)
```

```
       (m:bar-p2-specified? bar)))

(define (m:bar-contradictory? bar)
  (or (m:bar-p1-contradictory? bar)
      (m:bar-p2-contradictory? bar)
      (m:bar-direction-contradictory? bar)
      (m:bar-length-contradictory? bar)))

;;;;;;;;;;;;;;;;;;;;;;;;; Joint Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:joint-dir-1-specified? joint)
  (m:specified? (m:joint-dir-1 joint) direction?))

(define (m:joint-dir-1-contradictory? joint)
  (m:contradictory? (m:joint-dir-1 joint)))

(define (m:joint-dir-2-specified? joint)
  (m:specified? (m:joint-dir-2 joint) direction?))

(define (m:joint-dir-2-contradictory? joint)
  (m:contradictory? (m:joint-dir-2 joint)))

(define (m:joint-theta-contradictory? joint)
  (m:contradictory? (m:joint-theta joint)))

(define (m:joint-anchored? joint)
  (or (m:joint-dir-1-specified? joint)
      (m:joint-dir-2-specified? joint)))

(define (m:joint-anchored-and-arm-lengths-specified? joint)
  (and (m:joint-anchored? joint)
       (m:bar-length-specified? (m:joint-arm-1 joint))
       (m:bar-length-specified? (m:joint-arm-2 joint))))

(define (m:joint-specified? joint)
  (m:specified? (m:joint-theta joint) number?))

(define (m:joint-dirs-specified? joint)
  (and
   (m:joint-dir-1-specified? joint)
   (m:joint-dir-2-specified? joint)))

(define (m:joint-fully-specified? joint)
  (and
   (m:point-specified? (m:joint-vertex joint))
   (m:joint-dir-1-specified? joint)
   (m:joint-dir-2-specified? joint)))

(define (m:joint-contradictory? joint)
  (or
   (m:point-contradictory? (m:joint-vertex joint))
   (m:joint-dir-1-contradictory? joint)
   (m:joint-dir-2-contradictory? joint)
   (m:joint-theta-contradictory? joint)))

;;;;;;;;;;;;;;;;;;;;;;;;; Specifying Values ;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:joint-theta-if-specified joint)
  (let ((theta-v (m:examine-cell
                  (m:joint-theta joint))))
    (if (number? theta-v) theta-v
        0)))

(define (m:bar-max-inner-angle-sum bar)
  (let ((e1 (m:bar-endpoints-1 bar))
        (e2 (m:bar-endpoints-2 bar)))
    (if (or (null? e1)
            (null? e2))
```

```
        0
        (+ (apply max (map m:joint-theta-if-specified e1))
           (apply max (map m:joint-theta-if-specified e2))))))

(define (m:joint-bar-sums joint)
  (let ((b1 (m:joint-arm-1 joint))
        (b2 (m:joint-arm-2 joint)))
    (and (m:bar-length-specified? b1)
         (m:bar-length-specified? b2)
         (+ (m:examine-cell (m:bar-length b1))
            (m:examine-cell (m:bar-length b2))))))

(define (m:random-theta-for-joint joint)
  (let ((theta-range (m:examine-cell (m:joint-theta joint))))
    (if (interval? theta-range)
        (begin
          (safe-internal-rand-range
           (interval-low theta-range)
           (interval-high theta-range)))
        (error "Attempting to specify theta for joint"))))

(define (m:random-bar-length)
  (internal-rand-range 0.1 0.9))

(define (m:initialize-bar bar)
  (if (not (m:bar-anchored? bar))
      (m:instantiate-point (m:bar-p1 bar) 0 0 'initialize))
  (let ((random-dir (random-direction)))
    (m:instantiate (m:bar-direction bar)
                   random-dir 'initialize)
    (pp `(initializing-bar ,(print (m:bar-name bar))
                           ,(print random-dir)))))

(define (m:initialize-joint joint)
  (m:instantiate-point (m:joint-vertex joint) 0 0 'initialize)
  (pp `(initializing-joint ,(print (m:joint-name joint)))))

;;;;;;;;;; Assembling named joints into diagrams ;;;;;;;

(define (m:assemble-linkages bars joints)
  (let ((bar-table (m:make-bars-by-name-table bars)))
    (for-each
     (lambda (joint)
       (let ((vertex-name (m:joint-vertex-name joint))
             (dir-1-name (m:joint-dir-1-name joint))
             (dir-2-name (m:joint-dir-2-name joint)))
         (for-each
          (lambda (dir-name)
            (let ((bar (m:find-bar-by-id
                        bar-table
                        (m:bar vertex-name
                               dir-name))))
              (if (eq? bar #f)
                  (error "Could not find bar for" vertex-name dir-name))
              (m:identify-joint-bar-by-name joint bar)))
          (list dir-1-name dir-2-name))))
     joints)))

#|
 ;; Simple example of "solving for the third point"
 (begin
   (initialize-scheduler)
   (let ((b1 (m:make-named-bar 'a 'c))
         (b2 (m:make-named-bar 'b 'c))
         (b3 (m:make-named-bar 'a 'b))
         (j1 (m:make-named-joint 'b 'a 'c))
         (j2 (m:make-named-joint 'c 'b 'a))
         (j3 (m:make-named-joint 'a 'c 'b)))
```

```
883
884       (m:assemble-linkages
885        (list b1 b2 b3)
886        (list j2 j3 j1))
887
888       (m:initialize-joint j1)
889       (c:id (m:bar-length b1) (m:bar-length b2))
890
891       (m:instantiate (m:bar-length b3) 6 'b3-len)
892       (m:instantiate (m:bar-length b1) 5 'b1-len)
893       (run)
894       (m:examine-point (m:bar-p2 b1))))
895  ;Value: (m:point 3 4)
896
897 |#
898
899 ;;;;;;;;;;;;;;;;;; Converstion to Figure Elements ;;;;;;;;;;;;;;;;;;
900
901 ;;; TODO: Extract dependencies from TMS? or set names
902
903 (define (m:point->figure-point m-point)
904   (if (not (m:point-specified? m-point))
905       (let ((r (m:examine-cell (m:point-region m-point))))
906         (m:region->figure-elements r))
907       (let ((p (make-point (m:examine-cell (m:point-x m-point))
908                            (m:examine-cell (m:point-y m-point)))))
909         (set-element-name! p (m:element-name m-point))
910         p)))
911
912 (define (m:bar->figure-segment m-bar)
913   (if (not (m:bar-fully-specified? m-bar))
914       #f
915       (let ((p1 (m:point->figure-point (m:bar-p1 m-bar)))
916             (p2 (m:point->figure-point (m:bar-p2 m-bar))))
917         (and (point? p1)
918              (point? p2)
919              (make-segment p1 p2)))))
920
921 (define (m:joint->figure-angle m-joint)
922   (if (not (m:joint-fully-specified? m-joint))
923       #f
924       (make-angle (m:examine-cell (m:joint-dir-2 m-joint))
925                   (m:point->figure-point (m:joint-vertex m-joint))
926                   (m:examine-cell (m:joint-dir-1 m-joint)))))
```

Listing A.37: manipulate/region.scm

```
;;; regions.scm --- Region Information

;;; Commentary:

;; Ideas:
;; - Points, Lines, Circles, Intersections
;; - For now, semicircle (joints only go to 180deg to avoid
;;     multiple solns.)

;; Future:
;; - Differentiate regions with 2 deg. of freedom
;; - Improve contradiction objects

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Point Sets ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

(define (m:make-point-set points)
  (%m:make-point-set points))

(define (m:make-singular-point-set point)
  (m:make-point-set (list point)))

(define (m:in-point-set? p point-set)
  (pair? ((member-procedure point-equal?) p (m:point-set-points point-set))))

(define (m:singular-point-set? x)
  (and (m:point-set? x)
       (= 1 (length (m:point-set-points x)))))

(define (m:singular-point-set-point ps)
  (if (not (m:singular-point-set? ps))
      (error "Not a singular point set"))
  (car (m:point-set-points ps)))

(define (m:point-sets-equivalent? ps1 ps2)
  (define delp (delete-member-procedure list-deletor point-equal?))
  (define memp (member-procedure point-equal?))
  (let lp ((points-1 (m:point-set-points ps1))
           (points-2 (m:point-set-points ps2)))
    (if (null? points-1)
        (null? points-2)
        (let ((p1 (car points-1)))
          (if (memp p1 points-2)
              (lp (cdr points-1)
                  (delp p1 points-2))
              #f)))))

(define (m:print-point-set ps)
  (cons 'm:point-set
        (map (lambda (p) (list 'point (point-x p) (point-y p)))
             (m:point-set-points ps))))

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

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

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

```
  (endpoint m:ray-endpoint)
  (direction m:ray-direction))

(define m:make-ray %m:make-ray)

(define (m:ray->figure-ray m-ray)
  (with-color "red"
              (make-ray (m:ray-endpoint m-ray)
                        (m:ray-direction m-ray))))

(define (m:on-ray? p ray)
  (let ((endpoint (m:ray-endpoint ray)))
    (or (point-equal? p endpoint)
        (let ((dir (direction-from-points endpoint p)))
          (direction-equal? dir (m:ray-direction ray))))))

(define (m:p2-on-ray ray)
  (add-to-point (m:ray-endpoint ray)
                (unit-vec-from-direction (m:ray-direction ray))))

(define (m:rays-equivalent? ray1 ray2)
  (and (point-equal? (m:ray-endpoint ray1)
                     (m:ray-endpoint ray2))
       (direction-equal? (m:ray-direction ray1)
                         (m:ray-direction ray2))))

(define (m:print-ray ray)
  (let ((endpoint (m:ray-endpoint ray)))
    `(m:ray (,(point-x endpoint)
             ,(point-y endpoint))
            ,(direction-theta (m:ray-direction ray)))))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Arcs ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

;;; Start direction + ccw pi radian
(define (m:make-semi-circle center radius start-direction)
  (m:make-arc center radius
              (make-direction-interval start-direction
                                       (reverse-direction start-direction))))

(define (m:on-arc? p arc)
  (let ((center-point (m:arc-center arc))
        (radius (m:arc-radius arc)))
    (let ((distance (distance p center-point))
          (dir (direction-from-points center-point p)))
      (and (close-enuf? distance radius)
           (within-direction-interval?
            dir
            (m:arc-dir-interval arc))))))

(define (m:arcs-equivalent? arc1 arc2)
  (and (point-equal? (m:arc-center arc1)
                     (m:arc-center arc2))
       (close-enuf? (m:arc-radius arc1)
                    (m:arc-radius arc2))
       (direction-interval-equal?
        (m:arc-dir-interval arc1)
        (m:arc-dir-interval arc2))))
```

```
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)))
```

```
          (let ((points
                 (filter (lambda (p)
                           (and (m:on-arc? p arc1)
                                (m:on-arc? p arc2)))
                         intersections)))
            (if (> (length points) 0)
                (m:make-point-set points)
                ;; TODO: Determine error value
                (m:make-region-contradiction (list arc1 arc2))))))))

(define (m:intersect-ray-arc ray arc)
  (let ((center (m:arc-center arc))
        (radius (m:arc-radius arc))
        (endpoint (m:ray-endpoint ray))
        (ray-p2 (m:p2-on-ray ray)))
    (let ((intersections
           (intersect-circle-line-by-points
            center radius endpoint ray-p2)))
      (let ((points
             (filter (lambda (p)
                       (and (m:on-ray? p ray)
                            (m:on-arc? p arc)))
                     intersections)))
        (if (> (length points) 0)
            (m:make-point-set points)
            ;; TODO: Determine error value
            (m:make-region-contradiction (list ray arc)))))))

(define (m:intersect-arc-ray arc ray)
  (m:intersect-ray-arc ray arc))

;;;;;;;;;;;;;;;;;;; Intersecting with Point Sets ;;;;;;;;;;;;;;;;;;;

(define m:in-region? (make-generic-operation 2 'm:in-region?))

(defhandler m:in-region? m:in-point-set? point? m:point-set?)
(defhandler m:in-region? m:on-ray? point? m:ray?)
(defhandler m:in-region? m:on-arc? point? m:arc?)
(defhandler m:in-region? (lambda (p r) #f) point? m:region-contradiction?)

(define (m:intersect-point-set-with-region ps1 region)
  (let ((results
         (let lp ((points-1 (m:point-set-points ps1))
                  (point-intersections '()))
           (if (null? points-1)
               point-intersections
               (let ((p1 (car points-1)))
                 (if (m:in-region? p1 region)
                     (lp (cdr points-1)
                         (cons p1 point-intersections))
                     (lp (cdr points-1)
                         point-intersections)))))))
    (if (> (length results) 0)
        (m:make-point-set results)
        ;;; TODO: Determine error value
        (m:make-region-contradiction (list ps1 region)))))

(define (m:intersect-region-with-point-set region ps)
  (m:intersect-point-set-with-region ps region))

;;;;;;;;;;;;;;;;; Generic Intersect Regions "Merge" ;;;;;;;;;;;;;;;;;

(define m:intersect-regions (make-generic-operation 2 'm:intersect-regions))

;;; Same Type
(defhandler m:intersect-regions
  m:intersect-rays m:ray? m:ray?)
(defhandler m:intersect-regions
```

```
  m:intersect-arcs m:arc? m:arc?)

;;; Arc + Ray
(defhandler m:intersect-regions
  m:intersect-ray-arc m:ray? m:arc?)
(defhandler m:intersect-regions
  m:intersect-arc-ray m:arc? m:ray?)

;;; Point Sets
(defhandler m:intersect-regions
  m:intersect-region-with-point-set any? m:point-set?)
(defhandler m:intersect-regions
  m:intersect-point-set-with-region m:point-set? any?)

;;; Contradictions
(defhandler m:intersect-regions (lambda (a b) a) m:region-contradiction? any?)
(defhandler m:intersect-regions (lambda (a b) b) any? m:region-contradiction?)

;;;;;;;;;;;;;;;;;;;;;;; Generic Equivalency ;;;;;;;;;;;;;;;;;;;;;;;

(define m:region-equivalent?
  (make-generic-operation 2 'm:region-equivalent? (lambda (a b) #f)))

(defhandler m:region-equivalent?
  m:point-sets-equivalent? m:point-set? m:point-set?)

(defhandler m:region-equivalent?
  m:rays-equivalent? m:ray? m:ray?)

(defhandler m:region-equivalent?
  m:arcs-equivalent? m:arc? m:arc?)

(defhandler m:region-equivalent?
  m:region-contradictions-equivalent?
  m:region-contradiction?
  m:region-contradiction?)

;;;;;;;;;;;;;;;;;; Interface to Propagator System ;;;;;;;;;;;;;;;;;;

(define (m:region? x)
  (or (m:point-set? x)
      (m:ray? x)
      (m:arc? x)
      (m:region-contradiction? x)))


(defhandler equivalent? m:region-equivalent? m:region? m:region?)

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

(defhandler contradictory? m:region-contradiction? m:region?)

#|
 Simple Examples
 (pp (let-cells (c)
    (add-content c (m:make-arc (make-point 1 0) (sqrt 2)
                               (make-direction-interval
                                (make-direction (/ pi 8))
                                (make-direction (* 7 (/ pi 8))))))

    (add-content c (m:make-ray (make-point -3 1) (make-direction 0)))
    (add-content c (m:make-ray (make-point 1 2)
                  (make-direction (* 7 (/ pi 4)))))
    (content c)))

 (let ((a (make-point 0 0))
       (b (make-point 1 0))
       (c (make-point 0 1))
```

```
       (d (make-point 1 1)))
     (let-cells (cell)
       (add-content cell
                    (make-tms
                     (contingent (m:make-point-set (list a b c))
                                 '(a))))
       (add-content cell
                    (make-tms
                     (contingent (m:make-point-set (list a d))
                                 '(a))))
       (pp (tms-query (content cell)))))
|#
;;;;;;;;;;;;;;;;;;;;;;;; To Figure elements ;;;;;;;;;;;;;;;;;;;;;;;;

(define m:region->figure-elements
  (make-generic-operation 1 'm:region->figure-elements (lambda (r) #f )))

(defhandler m:region->figure-elements
  m:ray->figure-ray
  m:ray?)

(defhandler m:region->figure-elements
  m:region-contradiction->figure-elements
  m:region-contradiction?)
```

Listing A.38: manipulate/constraints.scm

```
;;; constraints.scm --- Constraints for mechanisms

;;; Commentary:

;; Ideas:
;; - Abstraction for specifying constraints
;; - Length, angle equality
;; - Perpendicular / Parellel

;; Future:
;; - Constraints for other linkages?

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;; Constraint Structure ;;;;;;;;;;;;;;;;;;;;;;;

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

;;;;;;;;;;;;;;;;;;;;;;;; Constraint Types ;;;;;;;;;;;;;;;;;;;;;;;;

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

(define (m:c-angle-equal joint-id-1 joint-id-2)
  (m:make-constraint
   'm:c-angle-equal
   (list joint-id-1 joint-id-2)
   (lambda (m)
     (let ((joint-1 (m:lookup m joint-id-1))
           (joint-2 (m:lookup m joint-id-2)))
       (c:id (m:joint-theta joint-1)
             (m:joint-theta joint-2))))))

(define (m:c-right-angle joint-id)
  (m:make-constraint
   'm:right-angle
   (list joint-id)
   (lambda (m)
     (let ((joint (m:lookup m joint-id)))
       (c:id
        (m:joint-theta joint)
        (/ pi 2))))))

;;; p2 between p1 p3 in a line
(define (m:c-line-order p1-id p2-id p3-id)
  (list
   (m:make-named-bar p1-id p2-id)
   (m:make-named-bar p2-id p3-id)
   (m:make-named-joint p1-id p2-id p3-id)
   (m:c-full-angle (m:joint p1-id p2-id p3-id))))

(define (m:c-full-angle joint-id)
  (m:make-constraint
```

```
   'm:full-angle
   (list joint-id)
   (lambda (m)
     (let ((joint (m:lookup m joint-id)))
       (c:id
        (m:joint-theta joint)
        pi)))))

(define (m:equal-joints-in-sum equal-joint-ids
                               all-joint-ids
                               total-sum)
  (m:make-constraint
   'm:equal-joints-in-sum
   all-joint-ids
   (lambda (m)
     (let ((all-joints (m:multi-lookup m all-joint-ids))
           (equal-joints (m:multi-lookup m equal-joint-ids)))
       (let ((other-joints
              (set-difference all-joints equal-joints eq?)))
         (c:id (m:joint-theta (car equal-joints))
               (ce:/
                (ce:- total-sum
                      (ce:multi+ (map m:joint-theta other-joints)))
                (length equal-joints))))))))

;;;;;;;;;;;;; Applying and Marking Constrained Elements ;;;;;;;;;;;;;

(define (m:constrained? element)
  (not (null? (m:element-constraints element))))

(define (m:element-constraints element)
  (or (eq-get element 'm:constraints)
      '()))

(define (m:set-element-constraints! element constraints)
  (eq-put! element 'm:constraints constraints))

(define (m:mark-constraint element constraint)
  (m:set-element-constraints!
   element
   (cons constraint
         (m:element-constraints element))))

(define (m:apply-constraint m constraint)
  (for-each (lambda (element-id)
              (m:mark-constraint
               (m:lookup m element-id)
               constraint))
            (m:constraint-args constraint))
  ((m:constraint-procedure constraint) m))

;;;;;;;;;;;;;;;;;;;;;;;; Propagator Utils ;;;;;;;;;;;;;;;;;;;;;;;;

(define (ce:multi+ cells)
  (cond ((null? cells) 0)
        ((null? (cdr cells)) (car cells))
        (else
         (ce:+ (car cells)
               (ce:multi+ (cdr cells))))))
```

Listing A.39: manipulate/topology.scm

```
;;; topology.scm --- Helpers for establishing topology for mechanism

;;; Commentary:

;; Ideas:
;; - Simplify listing out all bar and joint orderings
;; - Start with basic polygons, etc.

;; Future:
;; - Figure out making multi-in/out joints: (all pairs?)

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;; Establish-topology ;;;;;;;;;;;;;;;;;;;;;;;;

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

Listing A.40: manipulate/mechanism.scm

```
;;; mechanism.scm --- Group of Bars / Joints

;;; Commentary:

;; Ideas:
;; - Grouping of bars and joints
;; - Integrate with establishing toplogy

;; Future:
;; - Also specify constraints with it
;; - Convert to Diagram

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;; Mechanism Structure ;;;;;;;;;;;;;;;;;;;;;;;;

(define-record-type <m:mechanism>
    (%m:make-mechanism bars joints constraints
                       bar-table joint-table joint-by-vertex-table)
    m:mechanism?
    (bars m:mechanism-bars)
    (joints m:mechanism-joints)
    (constraints m:mechanism-constraints)
    (bar-table m:mechanism-bar-table)
    (joint-table m:mechanism-joint-table)
    (joint-by-vertex-table m:mechanism-joint-by-vertex-table))

(define (m:make-mechanism bars joints constraints)
  (let ((bar-table (m:make-bars-by-name-table bars))
        (joint-table (m:make-joints-by-name-table joints))
        (joint-by-vertex-table (m:make-joints-by-vertex-name-table joints)))
    (%m:make-mechanism bars joints constraints
                       bar-table joint-table joint-by-vertex-table)))

(define (m:mechanism . args)
  (let ((elements (flatten args)))
    (let ((bars (m:dedupe-bars (filter m:bar? elements)))
          (joints (filter m:joint? elements))
          (constraints (filter m:constraint? elements)))
      (m:make-mechanism bars joints constraints))))

(define (m:print-mechanism m)
  `((bars ,(map print (m:mechanism-bars m)))
    (joints ,(map print (m:mechanism-joints m)))
    (constraints ,(map print (m:mechanism-constraints m)))))

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

;;;;;;;;;;;;;;;;;;;;;;;;;; Deduplication ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:dedupe-bars bars)
  (dedupe (member-procedure m:bars-name-equivalent?) bars))


;;;;;;;;;;;;;;;;;;;;;;;;;;;; Accessors ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:mechanism-joint-by-vertex-name m vertex-name)
  (m:find-joint-by-vertex-name
   (m:mechanism-joint-by-vertex-table m)
   vertex-name))

(define (m:mechanism-joint-by-names m dir-1-name vertex-name dir-2-name)
  (m:find-joint-by-names
   (m:mechanism-joint-table m)
   dir-1-name vertex-name dir-2-name))

```

```
(define (m:multi-lookup m ids)
  (map (lambda (id) (m:lookup m id)) ids))

(define (m:lookup m id)
  (cond ((m:bar-id? id) (m:find-bar-by-id
                         (m:mechanism-bar-table m)
                         id))
        ((m:joint-id? id) (m:find-joint-by-id
                           (m:mechanism-joint-table m)
                           id))
        ((m:joint-vertex-id? id) (m:find-joint-by-vertex-name
                                  (m:mechanism-joint-by-vertex-table m)
                                  (m:joint-vertex-id-name id)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Specified ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (m:mechanism-fully-specified? mechanism)
  (and (every m:bar-fully-specified? (m:mechanism-bars mechanism))
       (every m:joint-fully-specified? (m:mechanism-joints mechanism))))

(define (m:mechanism-contradictory? mechanism)
  (or (any m:bar-contradictory? (m:mechanism-bars mechanism))
      (any m:joint-contradictory? (m:mechanism-joints mechanism))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Specify ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Should these be in Linkages?

(define *any-dir-specified* #f)
(define *any-point-specified* #f)

(define (any-one l)
  (let ((i (random (length l))))
    (list-ref l i)))

(define (m:pick-bar bars)
  (car (sort-by-key bars (negatep m:bar-max-inner-angle-sum))))

(define m:pick-joint-1 any-one)

(define (m:pick-joint joints)
  (car
   (append
    (sort-by-key
     (filter m:joint-bar-sums joints)
     m:joint-bar-sums)
    (filter (notp m:joint-bar-sums) joints))))

(define (m:specify-angle-if-first-time cell)
  (if (not *any-dir-specified*)
      (let ((dir (random-direction)))
        (set! *any-dir-specified* #t)
        (pp `(initializing-angle ,(name cell) ,(print dir)))
        (m:instantiate cell dir 'first-time-angle))))

(define (m:specify-point-if-first-time point)
  (if (not *any-point-specified*)
      (begin
        (set! *any-point-specified* #t)
        (pp `(initializing-point ,(name point) (0 0)))
        (m:instantiate-point point 0 0 'first-time-point))))

(define (m:specify-bar bar)
  (let ((v (m:random-bar-length)))
    (pp `(specifying-bar ,(print (m:bar-name bar)) ,v))
    (m:instantiate (m:bar-length bar) v 'specify-bar)
    (m:specify-angle-if-first-time (m:bar-direction bar))
    (m:specify-point-if-first-time (m:bar-p1 bar))))
```

```
135
136 (define (m:specify-joint joint)
137   (let ((v (m:random-theta-for-joint joint)))
138     (pp `(specifying-joint ,(print (m:joint-name joint)) ,v))
139     (m:instantiate (m:joint-theta joint) v 'specify-joint)
140     (m:specify-angle-if-first-time (m:joint-dir-1 joint))))
141
142 (define (m:initialize-joint-vertex joint)
143   (m:specify-point-if-first-time (m:joint-vertex joint)))
144
145 (define (m:initialize-joint-direction joint)
146   (m:specify-angle-if-first-time (m:joint-dir-1 joint)))
147
148 (define (m:initialize-bar-p1 bar)
149   (m:specify-point-if-first-time (m:bar-p1 bar)))
150
151 (define (m:specify-joint-if m predicate)
152   (let ((joints (filter (andp predicate (notp m:joint-specified?))
153                         (m:mechanism-joints m))))
154     (and (not (null? joints))
155          (m:specify-joint (m:pick-joint joints)))))
156
157 (define (m:initialize-joint-if m predicate)
158   (let ((joints (filter (andp predicate (notp m:joint-specified?))
159                         (m:mechanism-joints m))))
160     (and (not (null? joints))
161          (let ((j (m:pick-joint joints)))
162            (m:initialize-joint-direction j)))))
163
164 (define (m:specify-bar-if m predicate)
165   (let ((bars (filter (andp predicate (notp m:bar-length-specified?))
166                       (m:mechanism-bars m))))
167     (and (not (null? bars))
168          (m:specify-bar (m:pick-bar bars)))))
169
170 (define (m:initialize-bar-if m predicate)
171   (let ((bars (filter (andp predicate (notp m:bar-length-specified?))
172                       (m:mechanism-bars m))))
173     (and (not (null? bars))
174          (m:initialize-bar-p1 (m:pick-bar bars)))))
175
176 (define (m:specify-something m)
177   (or
178    (m:specify-bar-if m m:constrained?)
179    (m:specify-joint-if m m:constrained?)
180    (m:specify-joint-if m m:joint-anchored-and-arm-lengths-specified?)
181    (m:specify-joint-if m m:joint-anchored?)
182    (m:specify-bar-if m m:bar-directioned?)
183    (m:specify-bar-if m m:bar-anchored?)
184    (m:initialize-joint-if m m:joint-dirs-specified?)
185    (m:initialize-bar-if m m:bar-length-dir-specified?)
186    (m:initialize-bar-if m m:bar-direction-specified?)
187    (m:initialize-bar-if m m:bar-length-specified?)
188    (m:initialize-joint-if m m:joint-anchored?)
189    (m:initialize-joint-if m true-proc)
190    (m:initialize-bar-if m true-proc)))
191
192 ;;;;;;;;;;;;;;;;;;;;;;;; Applying constraints ;;;;;;;;;;;;;;;;;;;;;;;;
193
194 (define (m:apply-mechanism-constraints m)
195   (for-each (lambda (c)
196               (m:apply-constraint m c))
197             (m:mechanism-constraints m)))
198
199 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Build ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
200
201 (define (m:identify-vertices m)
202   (for-each (lambda (joints)
```

```
              (let ((first-vertex (m:joint-vertex (car joints))))
                (for-each (lambda (joint)
                            (m:identify-points first-vertex
                                               (m:joint-vertex joint)))
                          (cdr joints))))
            (hash-table/datum-list (m:mechanism-joint-by-vertex-table m))))

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

(define (m:initialize-solve)
  (set! *any-dir-specified* #f)
  (set! *any-point-specified* #f))

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

#|
 (begin
   (initialize-scheduler)
   (m:build-mechanism
    (m:mechanism
     (m:establish-polygon-topology 'a 'b 'c))))
|#

;;;;;;;;;;;;;;;;;;;;;;; Conversion to Figure ;;;;;;;;;;;;;;;;;;;;;;;

(define (m:mechanism->figure m)
  (let ((points
         (map (lambda (joint)
                (m:point->figure-point (m:joint-vertex joint)))
              (m:mechanism-joints m)))
        (segments (map m:bar->figure-segment (m:mechanism-bars m)))
        (angles (map m:joint->figure-angle (m:mechanism-joints m))))
    (apply figure (flatten (filter (lambda (x) (or x))
                           (append points segments angles))))))

(define (m:draw-mechanism m c)
  (draw-figure (m:mechanism->figure m) c))

#|
(let lp ()
  (initialize-scheduler)
  (let ((m (m:mechanism
            (m:establish-polygon-topology 'a 'b 'c 'd))))
    (pp (m:joint-anchored? (car (m:mechanism-joints m))))
    (m:build-mechanism m)
    (m:solve-mechanism m)
    (let ((f (m:mechanism->figure m)))
      (draw-figure f c)
      (pp (analyze-figure f)))))
|#
```

Listing A.41: manipulate/main.scm

```
;;; main.scm --- Main definitions and code for running the
;;; manipulation / mechanism-based code

;;; Examples

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

(define (arbitrary-right-triangle)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)
   (m:c-right-angle (m:joint 'a))))

(define (arbitrary-right-triangle-2)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)
   (m:c-right-angle (m:joint 'c))))

(define (quadrilateral-with-diagonals a b c d)
  (list
   (m:establish-polygon-topology a b c d)
   (m:establish-polygon-topology a b c)
   (m:establish-polygon-topology b c d)
   (m:establish-polygon-topology c d a)
   (m:establish-polygon-topology d a c)))

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

(define (quad-diagonals)
  (m:mechanism
   ;; Setup abcd with e in the middle:
   (quadrilateral-with-diagonals-intersection 'a 'b 'c 'd 'e)

   ;; Right Angle in Center:
   (m:c-right-angle (m:joint 'b 'e 'c))

   ;; Diagonals Equal
   ;;(m:c-length-equal (m:bar 'c 'a) (m:bar 'b 'd))
   (m:c-length-equal (m:bar 'c 'e) (m:bar 'a 'e))
   ;;(m:c-length-equal (m:bar 'b 'e) (m:bar 'd 'e))

   ;; Make it a square:
   ;;(m:c-length-equal (m:bar 'c 'e) (m:bar 'b 'e))
   ))

;;; Works:
(define (isoceles-triangle)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'b 'c))))

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

```
    (m:equal-joints-in-sum
     (list (m:joint 'a) (m:joint 'b))
     (list (m:joint 'a) (m:joint 'b) (m:joint 'c))
     pi)))

;;; Often works:
(define (arbitrary-quadrilateral)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)))

;;; Always works:
(define (parallelogram-by-sides)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'c 'd))
   (m:c-length-equal (m:bar 'b 'c)
                     (m:bar 'd 'a))))

(define (kite-by-sides)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'b 'c))
   (m:c-length-equal (m:bar 'c 'd)
                     (m:bar 'd 'a))))

(define (rhombus-by-sides)
  (m:mechanism
   (m:establish-polygon-topology 'a 'b 'c 'd)
   (m:c-length-equal (m:bar 'a 'b)
                     (m:bar 'b 'c))
   (m:c-length-equal (m:bar 'b 'c)
                     (m:bar 'c 'd))
   (m:c-length-equal (m:bar 'c 'd)
                     (m:bar 'a 'd))))

;;; Never works:
(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:equal-joints-in-sum
    (list (m:joint 'a) (m:joint 'c))
    (list (m:joint 'a) (m:joint 'b) (m:joint 'c) (m:joint 'd))
    (* 2 pi))
   (m:equal-joints-in-sum
    (list (m:joint 'b) (m:joint 'd))
    (list (m:joint 'a) (m:joint 'b) (m:joint 'c) (m:joint 'd))
    (* 2 pi))))

(define *m*)
(define (m:run-mechanism mechanism-proc)
  (initialize-scheduler)
  (let ((m (mechanism-proc)))
    (set! *m* m)
    (m:build-mechanism m)
    (m:solve-mechanism m)
    (let ((f (m:mechanism->figure m)))
      (draw-figure f c)
      ;;(pp (analyze-figure f))
      )))

#|
```

```
(let lp ()
  (initialize-scheduler)
  (pp 'start)
  (m:run-mechanism
   (lambda ()
     (m:mechanism
      ;;(m:establish-polygon-topology 'a 'b 'c)
      (m:make-named-bar 'a 'b)
      (m:make-named-bar 'b 'c)
      (m:make-named-bar 'c 'a)
      (m:make-named-joint 'c 'b 'a)
      (m:make-named-joint 'a 'c 'b)
      (m:make-named-joint 'b 'a 'c)

      (m:make-named-bar 'a 'd)
      (m:make-named-bar 'b 'd)
      (m:make-named-joint 'd 'a 'b)
      (m:make-named-joint 'a 'b 'd)
      (m:make-named-joint 'b 'd 'a)

      (m:make-named-bar 'c 'd)
      (m:make-named-joint 'a 'd 'c)
      (m:make-named-joint 'c 'a 'd)
      (m:make-named-joint 'd 'c 'a))))
  (lp))

(let lp ()
  (initialize-scheduler)
  (let ((m (m:mechanism
            (m:establish-polygon-topology 'a 'b 'c 'd))))
    (m:build-mechanism m)
    (m:solve-mechanism m)
    (let ((f (m:mechanism->figure m)))
      (draw-figure f c)
      (pp (analyze-figure f)))))
|#
```

Listing A.42: content/load.scm

```
;;; load.scm -- Load learning module
(for-each (lambda (f) (load f))
          '("investigations"))
```

Listing A.43: content/investigations.scm

```
1
2  ;;; [1] Linear Pair Conjecture
3  ;;; Givens: Angles a-1 and a-2 form a linear pair
4  ;;; Goal: m(a-1) + m(a-2) = 180 degrees
5  (define (linear-pair)
6    (let-geo* ((a (random-point))
7               (l1 (random-line-through-point a))
8               (r (random-ray-from-point a))
9               (a-1 (smallest-angle-from l1 r))
10              (a-2 (smallest-angle-from r (flip l1))))
11     (figure a l1 r a-1 a-2)))
12
13 ;;; [2] Vertical Angles Conjecture
14 ;;; Givens: Angles a-1 and a-2 are vertical angles
15 ;;; Goal: m(a-1) = m(a-2)
16 (define (vertical-angles)
17   (let-geo* ((l1 (random-line))
18              (c (random-point-on-line l1))
19              (l2 (rotate-randomly-about c l1))
20              (a-1 (smallest-angle-from l1 l2))
21              (a-2 (smallest-angle-from (flip l1) (flip l2))))
22     (figure l1 c l2 a-1 a-2)))
23
24 ;;; [3a] Corresponding Angles Conjecture
25 ;;; Givens: - Lines l1 and l2 are parallel
26 ;;;         - Line l3 is a transversal
27 ;;;         - a-1 and a-2 are resulting corresponding angles
28 ;;; Goal: m(a-1) = m(a-2)
29 (define (corresponding-angles)
30   (let-geo* ((l1 (random-line))
31              (l2 (translate-randomly l1))
32              (a (random-point-on-line l1))
33              (b (random-point-on-line l2))
34              (l3 (line-from-points a b))
35              (a-1 (smallest-angle-from l3 l2))
36              (a-2 (smallest-angle-from l3 l1)))
37             (figure l1 l2 a b l3 a-1 a-2)))
38 ;;; TODO: Translate randomly *multiple*
39 ;;; TODO: Multiple return values
40
41 ;;; [3b, 3c] Interior / alternate interior: ordering of angles and
42
43 ;;; [4] Converse of Parallel lines
44 ;;; Givens: - m(a-1) = m(a-2)
45 ;;;         - a-1, a-2, are either CA, AIA, AEA, etc. of Lines l1, l2
46 ;;; Goal: lines l1 and l2 are parallel
47 (define (parallel-lines-converse)
48   (let-geo* ((a-1 (random-angle))
49              (l3 (line-from-arm-1 a-1))
50              (a-2 (translate-randomly-along-line l3 a-1))
51              (l1 (line-from-arm-2 a-1))
52              (l2 (line-from-arm-2 a-2)))
53     (figure a-1 a-2 l1 l2 l3)))
54
55 ;;; [5] Perpendicular bisector conjecture
56 ;;; Givens: - p is a point on perpendicular bisector of segment (a, b)
57 ;;; Goal: p is equidistant from a and b
58 (define (perpendicular-bisector-equidistant)
59   (let-geo* (((s (a b)) (random-segment))
60              (l1 (perpendicular-bisector s))
61              (p (random-point-on-line l1)))
62             (figure s l1 p)))
63 ;;; TODO: Analyze equal segments not actually there...
64
65 ;;; [6] Converse of perpendicular bisector conjecture
66 ;;; Given: - a and b are equidistant from point p
```

```
;;; Goal: p is on the perpendicular bisector of a, b
(define (perpendicular-bisector-converse)
  (let-geo* ((p (random-point))
             (a (random-point))
             (b (rotate-randomly-about p a))
             (s (make-segment a b))
             (pb (perpendicular-bisector s)))
            (figure p a b s pb)))
;;; TODO: aux-segment

;;; [7] Shortest distance conjecture
;;; Givens: arbitrary point p, point a on line l
;;; Goal: Discover that shortest distance to line is along perpendicular
(define (shortest-distance)
  (let-geo* ((p (random-point))
             (l (random-line))
             (a (random-point-on-line l)))
    (figure p l a (make-auxiliary-segment p a))))
;;; TODO: Tricky, figure out how to minimize value, specify "minimize" property?

;;; [8] Angle bisector conjecture
;;; Given: angle a-1 of rays r-1, r-2, point a on angle-bisector l1
;;; Goal: Distnace from a to r-1 = distance a to r-2

(define (angle-bisector-distance)
  (let-geo* (((a (r-1 v r-2)) (random-angle))
             (ab (angle-bisector a))
             (p (random-point-on-ray ab))
             ((s-1 (p b)) (perpendicular-to r-1 p))
             ((s-2 (p c)) (perpendicular-to r-2 p)))
     (figure a r-1 r-2 ab p s-1 s-2)))
;;; Interesting, dependent on "shortest distance" from prior conjecture

;;; [9] Angle bisector concurrency
;;; Given: Triangle abc with angle-bisectors l1, l2, l3
;;; Goal: l1, l2, l3 are concurrent
(define (angle-bisector-concurrency)
  (let-geo* (((t1 (a b c)) (random-triangle))
             (((a-1 a-2 a-3)) (polygon-angles t1))
             (l1 (polygon-angle-bisector t1 a))
             (l2 (polygon-angle-bisector t1 b))
             (l3 (polygon-angle-bisector t1 c)))
    (figure t1 l1 l2 l3)))
;;; TODO: Concurrency of lines
;;; TODO: Draw markings for angle bisector

;;; [10] Perpendicular Bisector Concurrency
;;; Given: Triangle ABC with sides s1, s2, s3, perpendicular bisectors
;;; l1, l2, l3
;;; Goal: l1, l2, l3 are concurrent
(define (perpendicular-bisector-concurrency)
  (let-geo* (((t (a b c)) (random-triangle))
             (l1 (perpendicular-bisector (make-segment a b)))
             (l2 (perpendicular-bisector (make-segment b c)))
             (l3 (perpendicular-bisector (make-segment c a))))
    (figure t l1 l2 l3)))

;;; [11] Altitude Concurrency
;;; Given: Triangle ABC with altituds alt-1, alt2, alt-3
;;; Goal: alt-1, alt-2, alt-3 are concurrent
(define (altitude-concurrency)
  (let-geo* (((t (a b c)) (random-triangle))
             (alt-1 (perpendicular-line-to (make-segment b c) a))
             (alt-2 (perpendicular-line-to (make-segment a c) b))
             (alt-3 (perpendicular-line-to (make-segment a b) c)))
            (figure t alt-1 alt-2 alt-3)))
;;; TODO: Resist redundant concurrencies
;;; TODO: See if it can provide/learn a name for this point?
```

```
135
136 ;;;  [12] Circumcenter Conjecture
137 (define (circumcenter-figure)
138   (let-geo* (((t (a b c)) (random-triangle))
139              (c-center (circumcenter t)))
140     (figure t c-center (circle-from-points c-center a))))
141 ;;; TODO: Circumcenter macro?
142
143 ;;; [13] Incenter Conjecture
144 ;;; [14] Median Concurrency Conjecture
145 ;;; [15] Centroid Ratio Conjecture
146 ;;; [16] Center of Gravity Conjecture
147 ;;; [Exp.1] Euler Line Conjecture
148 ;;; [Exp.2] Euler Segment Conjecture
149 ;;; [17] Triangle Sum Conjecture
150 ;;; [18] Isoceles Triangle Conjecture
151 ;;; [19] Converse of Isoceles Triangle Conjecture
152 ;;; [20] Triangle Inequality Conjecture
153 ;;; [21] Side-Angle Inequaity Conjecture
154 ;;; [22] Triangle Exterior Angle Conjecture
155 ;;; [23] SSS Congruence Conjecture
156 ;;; [24] SAS Congruence Conjecture
157 ;;; [24b] SSA - Congruencey?
158 ;;; TODO: Provide some property to consider truth
159 ;;; [25] ASA Congruence Conjecture
160 ;;; [26] SAA Congruence Conjecture
161 ;;; [26b] AAA - Congruency?
162 ;;; [27] Vertex Angle Bisector Conjecture
163 ;;; [28] Equilateral/Eqiangular Triangle Conjecture
164 ;;; [29] Quadrilateral Sum Conjecture
165 ;;; [30] Pentagon Sum Conjecture
166 ;;; [31] Polygon Sum Conjecture
167 ;;; [32] Exterior Angle Sum Conjecture
168 ;;; [33] Equiangular Polygon Conjecture
169 ;;; [34] Kite Angles Conjecture
170 ;;; [35] Kite Diagonals Conjecture
171 ;;; [36] Kite Diagonal Biesctor Conjecture
172 ;;; [37] Kite Angle Bisector Conjecture
173 ;;; [38] Trapezoid Consecutive Angles Conjecture
174 ;;; [39] Isoceles Trapezoid Conjecture
175 ;;; [40] Isoceles Trapezoid Diagonals Conjecture
176 ;;; [41] Three Midsegments Conjecture
177 ;;; [42] Triangle Midsegment Conjecture
178 ;;; [43] Trapezoid Midsegment Conjecture
179 ;;; [44] Parallelogram Opposite Angles Conjecture
180
181 (define (parallelogram-opposite-angles)
182   (let-geo*
183       (((p (a b c d)) (random-parallelogram)))
184     (figure p)))
185 #|
186 ;;; [45] Parallelogram Consecutive Angles Conjecture
187 ;;; [46] Parallelogram Opposite Sides Conjecture
188 ;;; [47] Parallelogram Diagonals Conjecture
189 ;;; [48] Double-Edged Straitedge Conjecture
190 ;;; [49] Thombus Diagonals Conjecture
191 ;;; [50] Rhombus Angles Conjecture
192 ;;; [51] Rectangle Diagonals Conjecture
193 ;;; [52] Square Diagonals Conjecture
194 ;;; [53] Tangent Conjecture
195 ;;; [54] Tangent Segment Conjecture
196 ;;; [55] Chord Central Angles Conjecture
197 ;;; [56] Chord Arcs Conjecture
198 ;;; [57] Perpendicular to a Chord Conjecture
199 ;;; [58] Chord Distance to Center Conjecture
200 ;;; [59] Perpendicular Bisector of a Chord Conjecture
201 ;;; [60] Inscribed Angle Conjecture
202 ;;; [61] Inscribed Angles Intercepting Arcs Conjecture
```

```
;;; [62] Angles Inscribed in a Semicircle Conjecture
;;; [63] Cyclic Quadrilateral Conjecture
|#
(define (cyclic-quadrilateral)
  (let-geo*
   ((cir (random-circle))
    (((a b c d)) (n-random-points-on-circle-ccw cir 4))
    (q (polygon-from-points a b c d)))
   (figure q)))
#|
;;; [64] Parallel Lines Intercepted Arcs Conjecture
;;; [65] Circumference Conjecture
;;; [66] Arc Length Conjecture
;;; [Exp.3] Intersecting Seacants Conjecture
;;; [Exp.4] Intersecting Chords Conjecture
;;; [Exp.5] Tangent-Secant Conjecture
;;; [Exp.6] Intersecting Tangents Conjecture
;;; [Exp.7] Tangent-Chord Conjecture
;;; [67] Reflection Line Conjecture
;;; [68] Coordinate Transforms Conjecture
;;; [69] Minimal Path Conjecture
;;; [70] Reflections Across Parallel Lines Conjecture
;;; [71] Reflections Across Intersecting Lines Conjecture
;;; [72] Tessellating Triangles Conjecture
;;; [73] Tesselating Quadrilateral Conjecture
;;; [74] Rectangle Area Conjecture
;;; [75] Parallelogram Area Conjecture
;;; [76] Triangle Area Conjecture
;;; [77] Trapezoid Area Conjecture
;;; [78] Kite Area Conjecture
;;; [79] Regular Polygon Area Conjecture
;;; [80] Circle Area Conjecture
;;; [81] Pythagorean Theorem
;;; [82] Converse of Pythagorean Theorem
;;; [83] Isoceles Right Triangle Conjecture
;;; [84] 30-60-90 Triangle Conjecture
;;; [85] Distance Formula
;;; [86] Prism-Cylinder Volume Conjecture
;;; [87] Pyramid-Cone Volume Conjecture
;;; [Exp.8] Platonic Solids
;;; [88] Sphere Volume Conjecture
;;; [89] Sphere Surface Area Conjecture
;;; [91] AA Similarity Conjecture
;;; [92] SSS Similarity Conjecture
;;; [93] SAS Similarity Conjecture
;;; [94] Proportional Parts Conjecture
;;; [95] Angle Bisector / Opposite Side Conjecture
;;; [96] Proportional Area Conjecture
;;; [97] Proportional Volumes Conjecture
;;; [98] Parallel/Porportionality Conjecture
;;; [99] Extended Parallel/Proportionality Conjecture
;;; [100] SAS Triangle Area Conjecture
;;; [101] Las of Sines
;;; [102] Law of Cosines
;;; [Exp.9] Special Constructions
|#
```

Listing A.44: core/load.scm

```
;;; load.scm -- Load core
(for-each (lambda (f) (load f))
          '("utils"
            "macros"
            "print"
            "animation"))
```

Listing A.45: core/animation.scm

```
;;; animation.scm --- Animating and persisting values in figure constructions

;;; Commentary:

;; Ideas:
;; - Animate a range
;; - persist randomly chosen values across frames

;; Future:
;; - Backtracking, etc.
;; - Save continuations?

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;;;;; Configurations ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *animation-steps* 15)

;; ~30 Frames per second:
(define *animation-sleep* 30)

;;;;;;;;;;;;;;;;;;;;;;;; Internal Constants ;;;;;;;;;;;;;;;;;;;;;;;;
(define *is-animating?* #f)
(define *animation-value* 0)
(define *next-animation-index* 0)
(define *animating-index* 0)

(define (run-animation f-with-animations)
  (fluid-let ((*is-animating?* #t)
              (*persistent-values-table* (make-key-weak-eq-hash-table)))
    (let lp ((animate-index 0))
      (fluid-let
          ((*animating-index* animate-index))
        (let run-frame ((frame 0))
          (fluid-let ((*next-animation-index* 0)
                      (*next-value-index* 0)
                      (*animation-value*
                       (/ frame (* 1.0 *animation-steps*))))
            (f-with-animations)
            (sleep-current-thread *animation-sleep*)
            (if (< frame *animation-steps*)
                (run-frame (+ frame 1))
                (if (< *animating-index* (- *next-animation-index* 1))
                    (lp (+ animate-index 1)))))))))))

;;;;;;;;;;;;;;;;;;;;;;;; Animating Functions ;;;;;;;;;;;;;;;;;;;;;;;;

;;; f should be a function of one float argument in [0, 1]
(define (animate f)
  (let ((my-index *next-animation-index*))
    (set! *next-animation-index* (+ *next-animation-index* 1))
    (f (cond ((< *animating-index* my-index) 0)
             ((= *animating-index* my-index) *animation-value*)
             ((> *animating-index* my-index) 1)))))

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Persistence ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *persistent-values-table* #f)
(define *next-value-index* 0)

(define (persist-value v)
```

```
(if (not *is-animating?*)
    v
    (let* ((my-index *next-value-index*)
           (table-value (hash-table/get
                         *persistent-values-table*
                         my-index
                         #f)))
      (set! *next-value-index* (+ *next-value-index* 1))
      (or table-value
          (begin
            (hash-table/put! *persistent-values-table*
                             my-index
                             v)
            v)))))
```

Listing A.46: core/macros.scm

```
;;; macros.scm --- Macros for let-geo* to assign names and variables
;;; to elements

;;; Commentary:

;; Ideas:
;; - Basic naming
;; - Multiple assignment

;; Future:
;; - Warn about more errors
;; - More efficient multiple-assignment for lists

;;; Code:

;;;;;;;;;;;;;;;;;;;;;;; Expanding Assignment ;;;;;;;;;;;;;;;;;;;;;;;

(define *multiple-assignment-symbol* '*multiple-assignment-result*)

(define (expand-multiple-assignment lhs rhs)
  (expand-compound-assignment
   (list *multiple-assignment-symbol* lhs)
   rhs))

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

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

(define (expand-assignment assignment)
  (if (not (= 2 (length assignment)))
      (error "Assignment in letgeo* must be of length 2, found:" assignment))
  (let ((lhs (car assignment))
        (rhs (cadr assignment)))
    (if (list? lhs)
        (if (= (length lhs) 1)
            (expand-multiple-assignment (car lhs) rhs)
            (expand-compound-assignment lhs rhs))
        (list assignment))))

(define (expand-assignments assignments)
  (append-map expand-assignment assignments))

;;;;;;;;;;;;;;;;;;;;;;; Extract Variable Names ;;;;;;;;;;;;;;;;;;;;;;;

(define (variables-from-assignment assignment)
  (flatten (list (car assignment))))

(define (variables-from-assignments assignments)
  (append-map variables-from-assignment assignments))

```

```
(define (set-name-expressions symbols)
  (map (lambda (s)
         `(set-element-name! ,s (quote ,s)))
       symbols))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; let-geo* ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Syntax for setting names for geometry objects declared via let-geo
(define-syntax let-geo*
  (sc-macro-transformer
   (lambda (exp env)
     (let ((assignments (cadr exp))
           (body (caddr exp)))
       (let ((new-assignments (expand-assignments assignments))
             (variable-names (variables-from-assignments assignments)))
         (let ((result`(let*
                           ,new-assignments
                         ,@(set-name-expressions variable-names)
                         ,body)))
           result))))))
```

Listing A.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

```
;;; close-enuf? floating point comparison from scmutils
;;; Origin: Gerald Jay Sussman

(define *machine-epsilon*
  (let loop ((e 1.0))
    (if (= 1.0 (+ e 1.0))
        (* 2 e)
        (loop (/ e 2)))))

(define *sqrt-machine-epsilon*
  (sqrt *machine-epsilon*))

#|
 (define (close-enuf? h1 h2 tolerance)
   (<= (magnitude (- h1 h2))
       (* .5 (max tolerance *machine-epsilon*)
          (+ (magnitude h1) (magnitude h2) 2.0))))
|#

(define (close-enuf? h1 h2 #!optional tolerance scale)
  (if (default-object? tolerance)
      (set! tolerance (* 10 *machine-epsilon*)))
  (if (default-object? scale)
      (set! scale 1.0))
  (<= (magnitude (- h1 h2))
      (* tolerance
         (+ (* 0.5
               (+ (magnitude h1) (magnitude h2)))
            scale))))

(define (assert boolean error-message)
  (if (not boolean) (error error-message)))

(define (flatten list)
  (cond ((null? list) '())
        ((list? (car list))
         (append (flatten (car list))
                 (flatten (cdr list))))
        (else (cons (car list) (flatten (cdr list))))))

(define ((notp predicate) x)
  (not (predicate x)))

(define ((andp p1 p2) x)
  (and (p1 x)
       (p2 x)))

(define (true-proc . args) #t)
(define (false-proc . args) #f)

(define (identity x) x)

;;; ps1 \ ps2
(define (set-difference set1 set2 member-predicate)
  (define delp (delete-member-procedure list-deletor member-predicate))
  (let lp ((set1 set1)
           (set2 set2))
    (if (null? set2)
        set1
        (let ((e (car set2)))
          (lp (delp e set1)
              (cdr set2))))))

(define (eq-append! element key val)
  (eq-put! element key
           (cons val
```

```
                (or (eq-get element key) '())))))

(define (sort-by-key l key)
  (sort l (lambda (v1 v2)
            (< (key v1)
               (key  v2)))))

(define (sort-by-key-2 l key)
  (let ((v (sort-by-key-2 l key)))
    (pprint (map (lambda (x) (cons (name x) (key x))) v))
    v))

(define ((negatep f) x)
  (- (f x)))

(define ((flip-args f) x y)
  (f y x))

(define (index-of el list equality-predicate)
  (let lp ((i 0)
           (l list))
    (cond ((null? l) #f)
          ((equality-predicate (car l) el)
           i)
          (else (lp (+ i 1) (cdr l))))))

;;; (nth-letter-symbol 1) => 'a , 2 => 'b, etc.
(define (nth-letter-symbol i)
  (symbol (make-char (+ 96 i) 0)))

(define (hash-table/append table key element)
  (hash-table/put! table
                   key
                   (cons element
                         (hash-table/get table key '()))))

(define (dedupe-by equality-predicate elements)
  (dedupe (member-procedure equality-predicate) elements))

(define (dedupe member-predicate elements)
  (cond ((null? elements) '())
        (else
         (let ((b1 (car elements)))
           (if (member-predicate b1 (cdr elements))
               (dedupe member-predicate (cdr elements))
               (cons b1 (dedupe member-predicate (cdr elements))))))))
```