

Appendix A

Code Listings

Listings

3.1	Getting labels	25
3.2	Constraint Solving for Isoceles Triangle	25
3.3	Constraint Solving for Isoceles Triangle	25
A.1	load.scm	74
A.2	main.scm	75
A.3	learning/load.scm	78
A.4	learning/core-knowledge.scm	79
A.5	learning/definitions.scm	80
A.6	learning/simplifier.scm	81
A.7	learning/student.scm	82
A.8	learning/walkthrough.scm	85
A.9	figure/core.scm	87
A.10	figure/line.scm	88
A.11	figure/direction.scm	92
A.12	figure/direction.scm	94
A.13	figure/vec.scm	96
A.14	figure/measurements.scm	98
A.15	figure/angle.scm	99
A.16	figure/bounds.scm	102
A.17	figure/circle.scm	104
A.18	figure/point.scm	105
A.19	figure/constructions.scm	106
A.20	figure/intersections.scm	109

A.21 figure/figure.scm	112
A.22 figure/math-utils.scm	113
A.23 figure/polygon.scm	114
A.24 figure/metadata.scm	117
A.25 figure/dependencies.scm	118
A.26 figure/randomness.scm	120
A.27 figure/transforms.scm	126
A.28 core/load.scm	128
A.29 core/animation.scm	129
A.30 core/macros.scm	131
A.31 core/print.scm	133
A.32 core/utils.scm	134

Listing A.1: load.scm

```

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

```

Listing A.2: main.scm

```

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

```

```

67     (figure (polygon-from-points a c b d))))
68
69 (define (line-test)
70   (let-geo* ((a (random-point))
71              (b (random-point))
72              (c (random-point))
73              (d (random-point))
74              (l1 (line-from-points a b))
75              (l2 (line-from-points c d))
76              (e (intersect-lines l1 l2))
77              (f (random-point-on-line l1))
78              (cir (circle-from-points e f)))
79     (figure a b c d l1 l2 e f cir)))
80
81 (define (angle-test)
82   (let-geo* (((t (a b c)) (random-triangle))
83              (a-1 (smallest-angle (angle-from-points a b c)))
84              (a-2 (smallest-angle (angle-from-points b c a)))
85              (a-3 (smallest-angle (angle-from-points c a b)))
86              (l1 (angle-bisector a-1))
87              (l2 (angle-bisector a-2))
88              (l3 (angle-bisector a-3))
89              (center-point
90                (intersect-lines (ray->line l1)
91                                 (ray->line l2)))
92              (radius-line
93                (perpendicular (line-from-points b c)
94                               center-point))
95              (radius-point
96                (intersect-lines radius-line
97                                 (line-from-points b c)))
98              (cir (circle-from-points
99                    center-point
100                    radius-point))
101              (pb1 (perpendicular-bisector
102                    (make-segment a b)))
103              (pb2 (perpendicular-bisector
104                    (make-segment b c)))
105              (pb-center (intersect-lines pb1 pb2))
106              (circum-cir (circle-from-points
107                           pb-center
108                           a)))
109     (figure t cir a-1 a-2 a-3
110            pb-center
111            circum-cir
112            center-point)))
113
114 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
115 ;; Run commands
116
117 (define current-figure demo-figure)
118
119 (define c
120   (if (environment-bound? (the-environment) 'c)
121       c
122       (canvas)))
123
124 (define (close)
125   (ignore-errors (lambda () (graphics-close (canvas-g c)))))
126
127 (define *num-inner-loop* 5)
128 (define *num-outer-loop* 5)
129
130
131 (define (run-figure current-figure-proc)
132   (let ((analysis-data (make-analysis-collector)))
133     (run-animation
134      (lambda ()

```



```

135     (let ((current-figure (current-figure-proc)))
136       (draw-figure current-figure c)
137       (let ((analysis-results (analyze-figure current-figure)))
138         (save-results (print analysis-results) analysis-data))
139       )))
140   (display "--- Results ---\n")
141   (analyze-figure current-figure)
142   (print-analysis-results analysis-data))
143
144 (define interesting-figures
145   (list
146     debug-figure
147     parallel-lines-converse
148     perpendicular-bisector-equidistant
149     perpendicular-bisector-converse
150     demo-figure
151     linear-pair
152     vertical-angles
153     corresponding-angles
154     cyclic-quadrilateral))
155
156 (define (r)
157   (for-each (lambda (figure)
158             (run-figure figure))
159     interesting-figures)
160   'done)
161
162 ;(r)

```

Listing A.3: learning/load.scm

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

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

```

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

```

Listing A.5: learning/definitions.scm

```

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

```

Listing A.6: learning/simplifier.scm

```

1  ;;; simplifier.scm --- simplifies definitions
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - interfaces to manipulator
7
8  ;; Future:
9  ;; - Support more complex topologies.
10
11  ;;; Code:
12
13  ;;; Main Interface ;;;
14
15  (define (simplify-definition
16           n-sides
17           relationships)
18    #f)
19
20  (define (relationships->constraints relationships)
21    '())
22
23  (define (relationship->constraint rel)
24    '())
25
26  (define (establish-polygon-topology-for-n-gon n-sides)
27    (cond ((= n-sides 3)
28           (m:establish-polygon-topology 'a 'b 'c))
29          ((= n-sides 4)
30           (m:establish-polygon-topology 'a 'b 'c 'd))))
31
32  (define (relationships->figure n-sides relationships)
33    (initialize-scheduler)
34    (let ((m (apply
35              m:mechanism
36              (cons (establish-polygon-topology-for-n-gon n-sides)
37                    (relationships->constraints relationships)))))
38      (m:build-mechanism m)
39      (m:solve-mechanism m)
40      (let ((f (m:mechanism->figure m)))
41        f)))

```

Listing A.7: learning/student.scm

```

1  ;; student.scm -- base model of a student's knowlege
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Definitions, constructions, theorems
7  ;; - "What is"
8
9  ;; Future:
10 ;; - Simplifiers of redudant / uninteresting info
11 ;; - Propose own investigations?
12
13 ;; Code:
14
15 ;;;;;;;;;;;;;; Student Structure ;;;;;;;;;;;;;;
16
17 (define-record-type <student>
18   (%make-student definitions)
19   student?
20   (definitions student-definitions))
21
22 (define (make-student)
23   (%make-student (make-key-weak-eq-hash-table)))
24
25
26 ;;;;;;;;;;;;;; Building Predicates ;;;;;;;;;;;;;;
27
28 (define (build-predicate-for-definition s def)
29   (let ((classifications (definition-classifications def))
30         (observations (definition-observations def)))
31     (let ((classification-predicate
32            (lambda (obj)
33              (every
34                (lambda (classification)
35                  (or ((definition-predicate (student-lookup s classification))
36                      obj)
37                      (begin (if *explain*
38                              (pprint '(failed-classification
39                                      ,classification)))
39                          #f)))
34                classifications))))
42     (lambda args
43       (and (apply classification-predicate args)
44            (every (lambda (o) (satisfies-observation o args))
45                  observations))))))
46
47 ;;;;;;;;;;;;;; Definitions ;;;;;;;;;;;;;;
48
49 (define (add-definition! s def)
50   (if (not (definition-predicate def))
51       (set-definition-predicate!
52         def
53         (build-predicate-for-definition s def)))
54   (hash-table/put! (student-definitions s)
55                    (definition-name def)
56                    def))
57
58 (define (lookup-definition s name)
59   (hash-table/get (student-definitions s)
60                   name
61                   #f))
62
63 ;;;;;;;;;;;;;; Current Student ;;;;;;;;;;;;;;
64
65 (define *current-student* #f)
66

```

```

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

```

```

135 (define (learn-term term object-generator)
136   (let ((v (lookup term)))
137     (if (not (eq? v 'unknown))
138       (pprint '(already-known ,term))
139       (let ((example (name-polygon (object-generator))))
140         (let* ((base-terms (examine example))
141                (simple-base-terms (simplify-base-terms base-terms))
142                (base-definitions (map lookup base-terms))
143                (base-observations (flatten (map definition-observations
144                                              base-definitions)))
145                (fig (figure (with-dependency '<premise>' example)))
146                (observations (analyze-figure fig))
147                (simplified-observations
148                 (simplify-observations observations base-observations)))
149           (run-figure (lambda () (figure (object-generator))))
150           (pprint observations)
151           (let ((new-def
152                  (make-restrictions-definition
153                   term
154                   simple-base-terms
155                   simplified-observations
156                   object-generator)))
157             (add-definition! *current-student* new-def)
158             'done))))))

```


Listing A.8: learning/walkthrough.scm

```

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

```

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

Listing A.9: figure/load.scm

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

Listing A.10: figure/core.scm

```

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

```

Listing A.11: figure/line.scm

```

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

```

```

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

```

```

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

```

```

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

```


Listing A.12: figure/direction.scm

```

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

```

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

Listing A.13: figure/direction.scm

```

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

```

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

Listing A.14: figure/vec.scm

```

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

```

```

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

```

Listing A.15: figure/measurements.scm

```

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

```

Listing A.16: figure/angle.scm

```

1  ;;; angle.scm --- Angles
2
3  ;;; Commentary:
4
5  ;; Ideas:
6  ;; - Initially three points, now vertex + two directions
7  ;; - Counter-clockwise orientation
8  ;; - Uniquely determining from elements forces directions
9  ;; - naming of "arms" vs. "directions"
10
11 ;; Future Ideas:
12 ;; - Automatically discover angles from diagrams (e.g. from a pile of
13 ;;   points and segments)
14 ;; - Angle intersections
15
16 ;;; Code:
17
18 ;;;;;;;;;;;;;;;;;;;;;;;;; Angles ;;;;;;;;;;;;;;;;;;;;;;;;;
19
20 ;;; dir1 and dir2 are directions of the angle arms
21 ;;; The angle sweeps from dir2 *counter clockwise* to dir1
22 (define-record-type <angle>
23   (make-angle dir1 vertex dir2)
24   angle?
25   (dir1 angle-arm-1)
26   (vertex angle-vertex)
27   (dir2 angle-arm-2))
28
29 (declare-element-component-handler
30   (component-procedure-from-getters
31     ray-from-arm-1
32     angle-vertex
33     ray-from-arm-2)
34   angle?)
35
36 ;;;;;;;;;;;;;;;;;;;;;;;;; Transformations on Angles ;;;;;;;;;;;;;;;;;;;;;;;;;
37
38 (define (reverse-angle a)
39   (let ((d1 (angle-arm-1 a))
40         (v (angle-vertex a))
41         (d2 (angle-arm-2 a)))
42     (make-angle d2 v d1)))
43
44 (define (smallest-angle a)
45   (if (> (angle-measure a) pi)
46       (reverse-angle a)
47       a))
48
49 ;;;;;;;;;;;;;;;;;;;;;;;;; Alternate Constructors ;;;;;;;;;;;;;;;;;;;;;;;;;
50
51 (define (angle-from-points p1 vertex p2)
52   (let ((arm1 (direction-from-points vertex p1))
53         (arm2 (direction-from-points vertex p2)))
54     (make-angle arm1 vertex arm2)))
55
56 (define (smallest-angle-from-points p1 vertex p2)
57   (smallest-angle (angle-from-points p1 vertex p2)))
58
59 ;;;;;;;;;;;;;;;;;;;;;;;;; Angle from pairs of elements ;;;;;;;;;;;;;;;;;;;;;;;;;
60
61 (define angle-from (make-generic-operation 2 'angle-from))
62
63 (define (angle-from-lines l1 l2)
64   (let ((d1 (line->direction l1))
65         (d2 (line->direction l2))
66         (p (intersect-lines l1 l2)))

```



```

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

```

```
135      (linear-pair-internal? a2 a1)))
136
137 (define (vertical-angles? a1 a2)
138   (and (point-equal? (angle-vertex a1)
139                      (angle-vertex a2))
140        (direction-opposite? (angle-arm-1 a1)
141                              (angle-arm-1 a2))
142        (direction-opposite? (angle-arm-2 a1)
143                              (angle-arm-2 a2))))
```

Listing A.17: figure/bounds.scm

```

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

```

```

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

```

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

```

Listing A.19: figure/point.scm

```

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

```

Listing A.20: figure/constructions.scm

```

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

```

```

67 (let ((ray-endpt (ray-endpoint r))
68       (ray-dir (ray-direction r)))
69   (or (point-equal? ray-endpt p)
70       (let ((dir-to-p (direction-from-points ray-endpt p)))
71         (direction-equal? dir-to-p ray-dir))))
72
73 ;;;;;;;;;;;;;;;;;;;;;;;;; Construction of lines ;;;;;;;;;;;;;;;;;;;;;;;;;
74
75 (define (perpendicular linear-element point)
76   (let* ((direction (->direction linear-element))
77         (rotated-direction (rotate-direction-90 direction)))
78     (make-line point rotated-direction)))
79
80 ;; endpoint-1 is point, endpoint-2 is on linear-element
81 (define (perpendicular-to linear-element point)
82   (let ((pl (perpendicular linear-element point)))
83     (let ((i (intersect-linear-elements pl (->line linear-element))))
84       (make-segment point i))))
85
86 (define (perpendicular-line-to linear-element point)
87   (let ((pl (perpendicular linear-element point)))
88     pl))
89
90 (define (perpendicular-bisector segment)
91   (let ((midpt (segment-midpoint segment)))
92     (perpendicular (segment->line segment)
93                     midpt)))
94
95 (define (angle-bisector a)
96   (let* ((d1 (angle-arm-1 a))
97         (d2 (angle-arm-2 a))
98         (vertex (angle-vertex a))
99         (radians (angle-measure a))
100        (half-angle (/ radians 2))
101        (new-direction (add-to-direction d2 half-angle)))
102     (make-ray vertex new-direction)))
103
104 (define (polygon-angle-bisector polygon vertex-angle)
105   (angle-bisector (polygon-angle polygon vertex-angle)))
106
107 ;;;;;;;;;;;;;;;;;;;;;;;;; Higher-order constructions ;;;;;;;;;;;;;;;;;;;;;;;;;
108
109 (define (circumcenter t)
110   (let ((p1 (polygon-point-ref t 0))
111         (p2 (polygon-point-ref t 1))
112         (p3 (polygon-point-ref t 2)))
113     (let ((l1 (perpendicular-bisector (make-segment p1 p2)))
114          (l2 (perpendicular-bisector (make-segment p1 p3))))
115       (intersect-linear-elements l1 l2)))
116
117 ;;;;;;;;;;;;;;;;;;;;;;;;; Concurrent Linear Elements ;;;;;;;;;;;;;;;;;;;;;;;;;
118
119 (define (concurrent? l1 l2 l3)
120   (let ((i-point (intersect-linear-elements l1 l2)))
121     (and i-point
122          (on-element? i-point l3))))
123
124 (define (concentric? p1 p2 p3 p4)
125   (and (not (point-equal? p1 p2))
126        (not (point-equal? p1 p3))
127        (not (point-equal? p1 p4))
128        (not (point-equal? p2 p3))
129        (not (point-equal? p2 p4))
130        (not (point-equal? p3 p4))
131        (let ((pb-1 (perpendicular-bisector
132                     (make-segment p1 p2)))
133              (pb-2 (perpendicular-bisector
134                     (make-segment p2 p3))))
135          (intersect-linear-elements pb-1 pb-2))))

```



```

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

```

Listing A.21: figure/intersections.scm

```

1  ;; intersections.scm --- Intersections
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Unified intersections
7  ;; - Separation of core computations
8
9  ;; Future:
10 ;; - Amb-like selection of multiple intersections, or list?
11 ;; - Deal with elements that are exactly the same
12
13 ;; Code:
14
15 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Computations ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
16
17 ;; http://en.wikipedia.org/wiki/Line%E2%80%93line\_intersection
18 ;; line 1 through p1, p2 with line 2 through p3, p4
19 (define (intersect-lines-by-points p1 p2 p3 p4)
20   (let ((x1 (point-x p1))
21         (y1 (point-y p1))
22         (x2 (point-x p2))
23         (y2 (point-y p2))
24         (x3 (point-x p3))
25         (y3 (point-y p3))
26         (x4 (point-x p4))
27         (y4 (point-y p4)))
28     (let* ((denom
29            (det (det x1 1 x2 1)
30                  (det y1 1 y2 1)
31                  (det x3 1 x4 1)
32                  (det y3 1 y4 1)))
33            (num-x
34             (det (det x1 y1 x2 y2)
35                   (det x1 1 x2 1)
36                   (det x3 y3 x4 y4)
37                   (det x3 1 x4 1)))
38            (num-y
39             (det (det x1 y1 x2 y2)
40                   (det y1 1 y2 1)
41                   (det x3 y3 x4 y4)
42                   (det y3 1 y4 1))))
43       (if (= denom 0)
44           '()
45           (let
46             ((px (/ num-x denom))
47              (py (/ num-y denom)))
48              (list (make-point px py)))))))
49
50 ;; http://mathforum.org/library/drmath/view/51836.html
51 (define (intersect-circles-by-centers-radii c1 r1 c2 r2)
52   (let* ((a (point-x c1))
53          (b (point-y c1))
54          (c (point-x c2))
55          (d (point-y c2))
56          (e (- c a))
57          (f (- d b))
58          (p (+ (square e)
59                 (square f)))
60          (k (/ (- (+ (square p) (square r1))
61                    (square r2))
62                 (* 2 p))))
63     (if (> k r1)
64         (error "Circle's don't intersect")
65         (let* ((t (sqrt (- (square r1)
66                             (square k))))
67                (x1 (+ a (- e t)))
68                (y1 (+ b (- f t)))
69                (x2 (+ a (+ e t)))
70                (y2 (+ b (+ f t)))
71                (p1 (make-point x1 y1))
72                (p2 (make-point x2 y2))
73                (l1 (list p1))
74                (l2 (list p2)))
75             (list l1 l2))))))

```

```

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

```

```

135         (p2 (line-p2 line)))
136     (intersect-circle-line-by-points center radius p1 p2)))
137
138 (define standard-intersect
139   (make-generic-operation 2 'standard-intersect))
140
141 (defhandler standard-intersect
142   intersect-lines-to-list line? line?)
143
144 (defhandler standard-intersect
145   intersect-circles circle? circle?)
146
147 (defhandler standard-intersect
148   intersect-circle-line circle? line?)
149
150 (defhandler standard-intersect
151   (flip-args intersect-circle-line) line? circle?)
152
153 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Generic intersection ;;;;;;;;;;;;;;;;;
154
155 (define (intersect-linear-elements el-1 el-2)
156   (let ((i-list (standard-intersect (->line el-1)
157                                     (->line el-2))))
158     (if (null? i-list)
159         #f
160         (let ((i (car i-list)))
161           (if (or (not (on-element? i el-1))
162                 (not (on-element? i el-2)))
163               #f
164               i))))))
165
166 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; On Elements ;;;;;;;;;;;;;;;;;
167
168 (define on-element? (make-generic-operation 2 'on-element?))
169
170 (defhandler on-element? on-segment? point? segment?)
171 (defhandler on-element? on-line? point? line?)
172 (defhandler on-element? on-ray? point? ray?)

```

Listing A.22: figure/figure.scm

```

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

```

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

```

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

```

Listing A.24: figure/polygon.scm

```

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

```

```

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

```



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

Listing A.25: figure/metadata.scm

```
1  ;; metadata.scm - Element metadata
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Currently, names
7  ;; - Dependencies grew here, but are now separate
8
9  ;; Future:
10 ;; - Point/Linear/Circle adjacency - walk like graph
11
12 ;; Code:
13
14 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Names ;;;;;;;;;;;;;;;;;;
15
16 (define (set-element-name! element name)
17   (eq-put! element 'name name)
18   element)
19
20 (define (element-name element)
21   (or (eq-get element 'name)
22       '*unnamed*))
```

Listing A.26: figure/dependencies.scm

```

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

```

```

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

```

Listing A.27: figure/randomness.scm

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

339 (let* ((s1 (random-segment))
340        (p1 (segment-endpoint-1 s1))
341        (p2 (segment-endpoint-2 s1))
342        (p4 (rotate-about p1
343                          (rand-angle-measure)
344                          p2))
345        (p3 (add-to-point
346              p2
347              (sub-points p4 p1))))
348 (with-dependency
349   (make-random-dependency 'random-rhombus)
350   (polygon-from-points p1 p2 p3 p4)))

```

Listing A.28: figure/transforms.scm

```

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

```

```

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

```

Listing A.29: perception/load.scm

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

Listing A.30: perception/observation.scm

```

1  ;; observation.scm -- observed relationships
2
3  ;; Commentary:
4
5  ;; Code:
6
7  ;; Observation ;;;;;;;;;;;;;;;;;
8
9  (define-record-type <observation>
10    (make-observation premises relationship args)
11    observation?
12    (premises observation-premises)
13    (relationship observation-relationship)
14    (args observation-args))
15
16  (define (observation-equal? obs1 obs2)
17    (equal? (print-observation obs1)
18            (print-observation obs2)))
19
20  (define (print-observation obs)
21    (cons
22      (print (observation-relationship obs))
23      (map element-dependencies->list (observation-args obs))))
24
25  (defhandler print print-observation observation?)
26
27
28  ;; Checking observation ;;;;;;;;;;;;;;;;;
29
30  (define (satisfies-observation obs new-premise)
31    (let ((new-args
32          (map (lambda (arg)
33                ((element-source arg) new-premise))
34               (observation-args obs))))
35      (rel (observation-relationship obs)))
36      (or (relationship-holds rel new-args)
37          (begin (if *explain*
38                    (pprint '(failed-observation ,obs))
39                    #f)))))
40
41  ;; Simplifying observations ;;;;;;;;;;;;;;;;;
42
43  (define (simplify-observations observations base-observations)
44    (define memp (member-procedure observation-equal?))
45    (filter
46      (lambda (o) (not (memp o base-observations)))
47      observations))

```

Listing A.31: perception/analyzer.scm

```

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

```

```

67 (all-n-tuples 2 elements))
68
69 (define (all-n-tuples n elements)
70   (cond ((zero? n) '())
71         ((< (length elements) n) '())
72         (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))))))))))
85
86 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Obvious Segments ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
87
88 (define (segment-for-endpoint p1)
89   (let ((dep (element-dependency p1)))
90     (and dep
91          (or (and (eq? (car dep) 'segment-endpoint-1)
                    (cadr dep))
              (and (eq? (car dep) 'segment-endpoint-2)
                    (cadr dep))))))
95
96 (define (derived-from-same-segment? p1 p2)
97   (and
98    (segment-for-endpoint p1)
99    (segment-for-endpoint p2)
100    (eq? (segment-for-endpoint p1)
          (segment-for-endpoint p2))))
102
103 (define (polygon-for-point p1)
104   (let ((dep (element-dependency p1)))
105     (and dep
106          (and (eq? (car dep) 'polygon-point)
                (cons (caddr dep)
                      (cadr dep))))))
109
110 (define (adjacent-in-same-polygon? p1 p2)
111   (let ((poly1 (polygon-for-point p1))
         (poly2 (polygon-for-point p2)))
112     (and poly1 poly2
113          (eq? (car poly1) (car poly2))
114          (or (= (abs (- (cdr poly1)
                        (cdr poly2)))
                1)
              (and (= (cdr poly1) 0)
                    (= (cdr poly2) 3))
              (and (= (cdr poly1) 3)
                    (= (cdr poly2) 0))))))
122
123 (define (point-pairs->segments ppairs)
124   (filter (lambda (segment) segment)
125           (map (lambda (point-pair)
126                 (let ((p1 (car point-pair))
                      (p2 (cadr point-pair)))
127                   (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
133   ppairs)))
134

```



```

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)
         (group-ids (make-key-weak-eq-hash-table))
         (group-elements (make-key-weak-eq-hash-table))) ; Map from pair
     (for-each (lambda (pair)
                 (let ((first (car pair))
                       (second (cadr pair)))
                   (let ((group-id-1 (hash-table/get group-ids first i))
                         (group-id-2 (hash-table/get group-ids second i)))
                     (cond ((and (= group-id-1 i)
                                  (= group-id-2 i))
                            ; Both new, new groups:
                            (hash-table/put! group-ids first group-id-1)
                            (hash-table/put! group-ids second group-id-1))
                           ((= group-id-1 i)
                            (hash-table/put! group-ids first group-id-2))
                           ((= group-id-2 i)
                            (hash-table/put! group-ids second group-id-1)))
                       (set! i (+ i 1))))
                 pairs)
     (for-each (lambda (elt)
                 (hash-table/append group-elements
                                     (hash-table/get group-ids elt 'invalid)
                                     elt))
                 elements)
     (hash-table/remove! group-elements 'invalid)
     (hash-table/datum-list group-elements)))
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
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
                    (+ 1 (hash-table/get data-table results 0))))
182
183
184 (define (print-analysis-results data-table)
185   (hash-table/for-each
    data-table
    (lambda (k v)
      (pprint (list v (cons 'discovered k))))))
186
187
188

```

Listing A.32: graphics/load.scm

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

Listing A.33: graphics/appearance.scm

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

Listing A.34: graphics/graphics.scm

```

1 (define (draw-figure figure canvas)
2   (clear-canvas canvas)
3   (for-each
4     (lambda (element)
5       (canvas-set-color canvas (element-color element))
6       ((draw-element element) canvas))
7     (all-figure-elements figure))
8   (for-each
9     (lambda (element)
10      (canvas-set-color canvas (element-color element))
11      ((draw-label element) canvas))
12     (all-figure-elements figure))
13   (graphics-flush (canvas-g canvas))
14   )
15
16 (define draw-element
17   (make-generic-operation 1 'draw-element
18     (lambda (e) (lambda (c) 'done))))
19
20 (define draw-label
21   (make-generic-operation 1 'draw-label (lambda (e) (lambda (c) 'done))))
22
23 (define (add-to-draw-element! predicate handler)
24   (defhandler draw-element
25     (lambda (element)
26       (lambda (canvas)
27         (handler canvas element)))
28     predicate))
29
30 (define (add-to-draw-label! predicate handler)
31   (defhandler draw-label
32     (lambda (element)
33       (lambda (canvas)
34         (handler canvas element)))
35     predicate))
36
37
38 (define *point-radius* 0.02)
39 (define (draw-point canvas point)
40   (canvas-fill-circle canvas
41     (point-x point)
42     (point-y point)
43     *point-radius*))
44 (define (draw-point-label canvas point)
45   (canvas-draw-text canvas
46     (+ (point-x point) *point-radius*)
47     (+ (point-y point) *point-radius*)
48     (symbol->string (element-name point))))
49
50 (define (draw-segment canvas segment)
51   (let ((p1 (segment-endpoint-1 segment))
52         (p2 (segment-endpoint-2 segment)))
53     (canvas-draw-line canvas
54       (point-x p1)
55       (point-y p1)
56       (point-x p2)
57       (point-y p2))))
58 (define (draw-segment-label canvas segment)
59   (let ((v (vec-from-direction-distance (rotate-direction-90
60     (segment->direction segment))
61     (* 2 *point-radius*)))
62     (m (segment-midpoint segment)))
63     (let ((label-point (add-to-point m v)))
64       (canvas-draw-text canvas
65         (point-x label-point)
66         (point-y label-point)

```

```

67         (symbol->string (element-name segment))))))
68
69 (define (draw-line canvas line)
70   (let ((p1 (line-p1 line))
71         (p2 (add-to-point
72              p1
73              (unit-vec-from-direction (line-direction line))))))
74     (draw-segment canvas (extend-to-max-segment p1 p2))))
75
76 (define (draw-ray canvas ray)
77   (let ((p1 (ray-endpoint ray))
78         (p2 (add-to-point
79              p1
80              (unit-vec-from-direction (ray-direction ray))))))
81     (draw-segment canvas (ray-extend-to-max-segment p1 p2))))
82
83 (define (draw-circle canvas c)
84   (let ((center (circle-center c))
85         (radius (circle-radius c)))
86     (canvas-draw-circle canvas
87                          (point-x center)
88                          (point-y center)
89                          radius)))
90
91 (define *angle-mark-radius* 0.1)
92 (define (draw-angle canvas a)
93   (let* ((vertex (angle-vertex a))
94          (d1 (angle-arm-1 a))
95          (d2 (angle-arm-2 a))
96          (angle-start (direction-theta d2))
97          (angle-end (direction-theta d1)))
98     (canvas-draw-arc canvas
99                      (point-x vertex)
100                     (point-y vertex)
101                     *angle-mark-radius*
102                     angle-start
103                     angle-end)))
104
105 ;;; Add to generic operations
106
107 (add-to-draw-element! point? draw-point)
108 (add-to-draw-element! segment? draw-segment)
109 (add-to-draw-element! circle? draw-circle)
110 (add-to-draw-element! angle? draw-angle)
111 (add-to-draw-element! line? draw-line)
112 (add-to-draw-element! ray? draw-ray)
113
114 (add-to-draw-label! point? draw-point-label)
115
116 ;;; Canvas for x-graphics
117
118 (define (x-graphics) (make-graphics-device 'x))
119
120 (define (canvas)
121   (let ((g (x-graphics)))
122     (graphics-enable-buffering g)
123     (list 'canvas g)))
124
125 (define (canvas-g canvas)
126   (cadr canvas))
127
128 (define (canvas? x)
129   (and (pair? x)
130        (eq? (car x 'canvas))))
131
132 (define (clear-canvas canvas)
133   (graphics-clear (canvas-g canvas)))
134

```

```

135 (define (canvas-draw-circle canvas x y radius)
136   (graphics-operation (canvas-g canvas)
137                       'draw-circle
138                       x y radius))
139
140 (define (canvas-draw-text canvas x y text)
141   (graphics-draw-text (canvas-g canvas) x y text))
142
143 (define (canvas-draw-arc canvas x y radius
144                       angle-start angle-end)
145   (let ((angle-sweep
146         (fix-angle-0-2pi (- angle-end
147                               angle-start))))
148     (graphics-operation (canvas-g canvas)
149                         'draw-arc
150                         x y radius radius
151                         (rad->deg angle-start)
152                         (rad->deg angle-sweep)
153                         #f)))
154
155 (define (canvas-fill-circle canvas x y radius)
156   (graphics-operation (canvas-g canvas)
157                       'fill-circle
158                       x y radius))
159
160 (define (canvas-draw-line canvas x1 y1 x2 y2)
161   (graphics-draw-line (canvas-g canvas)
162                       x1 y1
163                       x2 y2))
164
165 (define (canvas-set-color canvas color)
166   (graphics-operation (canvas-g canvas) 'set-foreground-color color)
167   )

```

Listing A.35: manipulate/load.scm

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

Listing A.36: manipulate/linkages.scm

```

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

```



```

67 (let-cells (output-cell)
68   (name! output-cell (symbol 'reverse- (name input-cell)))
69   (p:m:reverse-direction input-cell output-cell)
70   (p:m:reverse-direction output-cell input-cell)
71   output-cell))
72
73 ;;;;;;;;;;;;;; Adding to directions ;;;;;;;;;;;;;;
74
75 (define (m:add-interval-to-direction d i)
76   (if (empty-interval? i)
77       (error "Cannot add empty interval to direction"))
78   (make-direction-interval-from-start-dir-and-size
79     (add-to-direction d (interval-low i))
80     (- (interval-high i)
81        (interval-low i))))
82
83 (define (m:add-interval-to-standard-direction-interval di i)
84   (if (empty-interval? i)
85       (error "Cannot add empty interval to direction"))
86   (let ((di-size (direction-interval-size di))
87         (i-size (- (interval-high i)
88                     (interval-low i)))
89         (di-start (direction-interval-start di)))
90     (make-direction-interval-from-start-dir-and-size
91       (add-to-direction di-start (interval-low i))
92       (+ di-size i-size))))
93
94 (define (m:add-interval-to-full-circle-direction-interval fcdi i)
95   (if (empty-interval? i)
96       (error "Cannot add empty interval to direction"))
97   fcdi)
98
99 (define (m:add-interval-to-invalid-direction-interval fcdi i)
100   (if (empty-interval? i)
101       (error "Cannot add empty interval to direction"))
102   (error "Cannot add to invalid direction in"))
103
104 (define m:add-to-direction
105   (make-generic-operation 2 'm:add-to-direction))
106
107 (defhandler m:add-to-direction
108   m:add-interval-to-direction direction? interval?)
109
110 (defhandler m:add-to-direction
111   add-to-direction direction? number?)
112
113 (defhandler m:add-to-direction
114   m:add-interval-to-standard-direction-interval
115   standard-direction-interval? interval?)
116
117 (defhandler m:add-to-direction
118   m:add-interval-to-full-circle-direction-interval
119   full-circle-direction-interval? interval?)
120
121 (defhandler m:add-to-direction
122   m:add-interval-to-invalid-direction-interval
123   invalid-direction-interval? interval?)
124
125 (defhandler m:add-to-direction
126   shift-direction-interval direction-interval? number?)
127
128 (propagatify m:add-to-direction)
129
130 ;;;;;;;;;;;;;; Subtracting directions ;;;;;;;;;;;;;;
131
132 (defhandler generic-negate
133   (lambda (i) (mul-interval i -1)) %interval?)
134

```

```

135 (define (m:standard-direction-interval-minus-direction di d)
136   (if (within-direction-interval? d di)
137     (make-interval
138       0
139       (subtract-directions (direction-interval-end di) d))
140     (make-interval
141       (subtract-directions (direction-interval-start di) d)
142       (subtract-directions (direction-interval-end di) d))))
143
144 (define (m:full-circle-direction-interval-minus-direction di d)
145   (make-interval
146     0 (* 2 pi)))
147
148 (define (m:direction-minus-standard-direction-interval d di)
149   (if (within-direction-interval? d di)
150     (make-interval
151       0
152       (subtract-directions d (direction-interval-start di)))
153     (make-interval
154       (subtract-directions d (direction-interval-end di))
155       (subtract-directions d (direction-interval-start di)))))
156
157 (define (m:direction-minus-full-circle-direction-interval d di)
158   (make-interval
159     0 (* 2 pi)))
160
161 (define m:subtract-directions
162   (make-generic-operation 2 'm:subtract-directions))
163
164 (defhandler m:subtract-directions
165   subtract-directions direction? direction?)
166
167 ;;; TODO: Support Intervals for thetas?
168 (defhandler m:subtract-directions
169   (lambda (di1 di2)
170     nothing)
171   direction-interval? direction-interval?)
172
173 (defhandler m:subtract-directions
174   m:standard-direction-interval-minus-direction
175   standard-direction-interval? direction?)
176
177 (defhandler m:subtract-directions
178   m:full-circle-direction-interval-minus-direction
179   full-circle-direction-interval? direction?)
180
181 (defhandler m:subtract-directions
182   m:direction-minus-standard-direction-interval
183   direction? standard-direction-interval?)
184
185 (defhandler m:subtract-directions
186   m:direction-minus-full-circle-direction-interval
187   direction? full-circle-direction-interval?)
188
189 (propagatify m:subtract-directions)
190
191 ;;;;;;;;;;;;;; Vec ;;;;;;;;;;;;;;
192 (define-record-type <m:vec>
193   (%m:make-vec dx dy length direction)
194   m:vec?
195   (dx m:vec-dx)
196   (dy m:vec-dy)
197   (length m:vec-length)
198   (direction m:vec-direction))
199
200
201 ;;; Allocate and wire up the cells in a vec
202 (define (m:make-vec vec-id)

```

```

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

```

```

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

```

```

339     nothing))
340
341 (propagatify m:x-y-direction->region)
342
343 (define (m:x-y-length-di->region px py length dir-interval)
344   (if (direction-interval? dir-interval)
345     (let ((vertex (make-point px py)))
346       (m:make-arc vertex length dir-interval))
347     nothing))
348
349 (propagatify m:x-y-length-di->region)
350
351 (define (m:p1->p2-bar-propagator p1 p2 bar)
352   (let ((plx (m:point-x p1))
353         (ply (m:point-y p1))
354         (p2r (m:point-region p2))
355         (length (m:bar-length bar))
356         (dir (m:bar-direction bar)))
357     (p:m:x-y-direction->region plx ply dir p2r)
358     (p:m:x-y-length-di->region plx ply length dir p2r)))
359
360 (define (m:p2->p1-bar-propagator p2 p1 bar)
361   (let ((p2x (m:point-x p2))
362         (p2y (m:point-y p2))
363         (p1r (m:point-region p1))
364         (length (m:bar-length bar))
365         (dir (m:bar-direction bar)))
366     (p:m:x-y-direction->region p2x p2y (ce:reverse-direction dir) p1r)
367     (p:m:x-y-length-di->region p2x p2y length (ce:reverse-direction dir) p1r)))
368
369 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Joint ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
370 ;;; Direction-2 is counter-clockwise from direction-1 by theta
371 (define-record-type <m:joint>
372   (%m:make-joint vertex dir-1 dir-2 theta)
373   m:joint?
374   (vertex m:joint-vertex)
375   (dir-1 m:joint-dir-1)
376   (dir-2 m:joint-dir-2)
377   (theta m:joint-theta))
378
379 (define *max-joint-swing* pi)
380
381 (define (m:make-joint joint-id)
382   (let ((joint-key (m:make-joint-name-key joint-id)))
383     (let ((vertex (m:make-point (symbol joint-key '-vertex))))
384       (let-cells (dir-1 dir-2 theta)
385         (name! dir-1 (symbol joint-key '-dir-1))
386         (name! dir-2 (symbol joint-key '-dir-2))
387         (name! theta (symbol joint-key '-theta))
388         (name! vertex (symbol joint-key '-vertex))
389         (p:m:add-to-direction
390          dir-1 theta dir-2)
391         (p:m:add-to-direction
392          dir-2 (e:negate theta) dir-1)
393         (p:m:subtract-directions
394          dir-2 dir-1
395          theta)
396         (m:instantiate theta (make-interval 0 *max-joint-swing*) 'theta)
397         (%m:make-joint vertex dir-1 dir-2 theta))))))
398
399 (define (m:print-joint j)
400   '(m:joint
401     ,(print (m:joint-name j))
402     ,(print (m:joint-dir-1 j))
403     ,(print (m:joint-vertex j))
404     ,(print (m:joint-dir-2 j))
405     ,(print (m:joint-theta j))))
406

```

```

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

```



```

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

```

```

679 (let ((v (m:examine-cell cell)))
680   (and
681     (not (nothing? v))
682     (or (default-object? predicate)
683         (predicate v))))))
684
685 (define (m:bar-length-specified? bar)
686   (m:specified? (m:bar-length bar) number?))
687
688 (define (m:bar-direction-specified? bar)
689   (m:specified? (m:bar-direction bar) direction?))
690
691 (define (m:joint-theta-specified? joint)
692   (m:specified? (m:joint-theta joint) number?))
693
694 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Point Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
695
696 (define (m:point-specified? p)
697   (and (m:specified? (m:point-x p) number?)
698        (m:specified? (m:point-y p) number?)))
699
700 (define (m:point-contradictory? p)
701   (or (m:contradictory? (m:point-x p))
702       (m:contradictory? (m:point-y p))
703       (m:contradictory? (m:point-region p))))
704
705 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Bar Predicates ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
706
707 (define (m:bar-p1-specified? bar)
708   (m:point-specified? (m:bar-p1 bar)))
709
710 (define (m:bar-p2-specified? bar)
711   (m:point-specified? (m:bar-p2 bar)))
712
713 (define (m:bar-p1-contradictory? bar)
714   (m:point-contradictory? (m:bar-p1 bar)))
715
716 (define (m:bar-p2-contradictory? bar)
717   (m:point-contradictory? (m:bar-p2 bar)))
718
719 (define (m:bar-anchored? bar)
720   (or (m:bar-p1-specified? bar)
721       (m:bar-p2-specified? bar)))
722
723 (define (m:bar-directioned? bar)
724   (and (m:bar-anchored? bar)
725        (m:specified? (m:bar-direction bar) direction?)))
726
727 (define (m:bar-direction-contradictory? bar)
728   (or (m:contradictory? (m:bar-direction bar))
729       (m:contradictory? (m:vec-dx (m:bar-vec bar))
730                          (m:vec-dy (m:bar-vec bar)))))
731
732 (define (m:bar-length-specified? bar)
733   (and (m:specified? (m:bar-length bar) number?)))
734
735 (define (m:bar-direction-specified? bar)
736   (and (m:specified? (m:bar-direction bar) number?)))
737
738 (define (m:bar-length-contradictory? bar)
739   (m:contradictory? (m:bar-length bar)))
740
741 (define (m:bar-length-dir-specified? bar)
742   (and (m:bar-length-specified? bar)
743        (m:bar-direction-specified? bar)))
744
745 (define (m:bar-fully-specified? bar)
746   (and (m:bar-p1-specified? bar)

```

```

747     (m:bar-p2-specified? bar)))
748
749 (define (m:bar-contradictory? bar)
750   (or (m:bar-p1-contradictory? bar)
751       (m:bar-p2-contradictory? bar)
752       (m:bar-direction-contradictory? bar)
753       (m:bar-length-contradictory? bar)))
754
755 ;;;;;;;;;;;;;;; Joint Predicates ;;;;;;;;;;;;;;;
756
757 (define (m:joint-dir-1-specified? joint)
758   (m:specified? (m:joint-dir-1 joint) direction?))
759
760 (define (m:joint-dir-1-contradictory? joint)
761   (m:contradictory? (m:joint-dir-1 joint)))
762
763 (define (m:joint-dir-2-specified? joint)
764   (m:specified? (m:joint-dir-2 joint) direction?))
765
766 (define (m:joint-dir-2-contradictory? joint)
767   (m:contradictory? (m:joint-dir-2 joint)))
768
769 (define (m:joint-theta-contradictory? joint)
770   (m:contradictory? (m:joint-theta joint)))
771
772 (define (m:joint-anchored? joint)
773   (or (m:joint-dir-1-specified? joint)
774       (m:joint-dir-2-specified? joint)))
775
776 (define (m:joint-anchored-and-arm-lengths-specified? joint)
777   (and (m:joint-anchored? joint)
778        (m:bar-length-specified? (m:joint-arm-1 joint))
779        (m:bar-length-specified? (m:joint-arm-2 joint))))
780
781 (define (m:joint-specified? joint)
782   (m:specified? (m:joint-theta joint) number?))
783
784 (define (m:joint-dirs-specified? joint)
785   (and
786     (m:joint-dir-1-specified? joint)
787     (m:joint-dir-2-specified? joint)))
788
789 (define (m:joint-fully-specified? joint)
790   (and
791     (m:point-specified? (m:joint-vertex joint))
792     (m:joint-dir-1-specified? joint)
793     (m:joint-dir-2-specified? joint)))
794
795 (define (m:joint-contradictory? joint)
796   (or
797     (m:point-contradictory? (m:joint-vertex joint))
798     (m:joint-dir-1-contradictory? joint)
799     (m:joint-dir-2-contradictory? joint)
800     (m:joint-theta-contradictory? joint)))
801
802 ;;;;;;;;;;;;;;; Specifying Values ;;;;;;;;;;;;;;;
803
804 (define (m:joint-theta-if-specified joint)
805   (let ((theta-v (m:examine-cell
806                   (m:joint-theta joint))))
807     (if (number? theta-v) theta-v
808         0)))
809
810 (define (m:bar-max-inner-angle-sum bar)
811   (let ((e1 (m:bar-endpoints-1 bar))
812         (e2 (m:bar-endpoints-2 bar)))
813     (if (or (null? e1)
814             (null? e2))

```

```

815     0
816     (+ (apply max (map m:joint-theta-if-specified e1))
817        (apply max (map m:joint-theta-if-specified e2))))))
818
819 (define (m:joint-bar-sums joint)
820   (let ((b1 (m:joint-arm-1 joint))
821         (b2 (m:joint-arm-2 joint)))
822     (and (m:bar-length-specified? b1)
823          (m:bar-length-specified? b2)
824          (+ (m:examine-cell (m:bar-length b1))
825             (m:examine-cell (m:bar-length b2))))))
826
827 (define (m:random-theta-for-joint joint)
828   (let ((theta-range (m:examine-cell (m:joint-theta joint))))
829     (if (interval? theta-range)
830         (begin
831           (safe-internal-rand-range
832            (interval-low theta-range)
833            (interval-high theta-range)))
834         (error "Attempting to specify theta for joint"))))
835
836 (define (m:random-bar-length)
837   (internal-rand-range 0.1 0.9))
838
839 (define (m:initialize-bar bar)
840   (if (not (m:bar-anchored? bar))
841       (m:instantiate-point (m:bar-p1 bar) 0 0 'initialize))
842   (let ((random-dir (random-direction)))
843     (m:instantiate (m:bar-direction bar)
844                    random-dir 'initialize)
845     (pp '(initializing-bar ,(print (m:bar-name bar))
846        ,(print random-dir)))))
847
848 (define (m:initialize-joint joint)
849   (m:instantiate-point (m:joint-vertex joint) 0 0 'initialize)
850   (pp '(initializing-joint ,(print (m:joint-name joint)))))
851
852 ;;;;;;;;;; Assembling named joints into diagrams ;;;;;;;;;;
853
854 (define (m:assemble-linkages bars joints)
855   (let ((bar-table (m:make-bars-by-name-table bars)))
856     (for-each
857      (lambda (joint)
858        (let ((vertex-name (m:joint-vertex-name joint))
859              (dir-1-name (m:joint-dir-1-name joint))
860              (dir-2-name (m:joint-dir-2-name joint)))
861          (for-each
862           (lambda (dir-name)
863             (let ((bar (m:find-bar-by-id
864                        bar-table
865                        (m:bar vertex-name
866                          dir-name))))
867               (if (eq? bar #f)
868                   (error "Could not find bar for" vertex-name dir-name))
869               (m:identify-joint-bar-by-name joint bar)))
870            (list dir-1-name dir-2-name))))
871     joints)))
872
873 #|
874 ;; Simple example of "solving for the third point"
875 (begin
876   (initialize-scheduler)
877   (let ((b1 (m:make-named-bar 'a 'c))
878         (b2 (m:make-named-bar 'b 'c))
879         (b3 (m:make-named-bar 'a 'b))
880         (j1 (m:make-named-joint 'b 'a 'c))
881         (j2 (m:make-named-joint 'c 'b 'a))
882         (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 ;;;;;;;;;;;;;; Conversion 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

```

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

```

```

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

```

```

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

```



```

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

```

```

271 m:intersect-arcs m:arc? m:arc?)
272
273 ;; Arc + Ray
274 (defhandler m:intersect-regions
275   m:intersect-ray-arc m:ray? m:arc?)
276 (defhandler m:intersect-regions
277   m:intersect-arc-ray m:arc? m:ray?)
278
279 ;; Point Sets
280 (defhandler m:intersect-regions
281   m:intersect-region-with-point-set any? m:point-set?)
282 (defhandler m:intersect-regions
283   m:intersect-point-set-with-region m:point-set? any?)
284
285 ;; Contradictions
286 (defhandler m:intersect-regions (lambda (a b) a) m:region-contradiction? any?)
287 (defhandler m:intersect-regions (lambda (a b) b) any? m:region-contradiction?)
288
289 ;;;;;;;;; Generic Equivalency ;;;;;;;;;
290
291 (define m:region-equivalent?
292   (make-generic-operation 2 'm:region-equivalent? (lambda (a b) #f)))
293
294 (defhandler m:region-equivalent?
295   m:point-sets-equivalent? m:point-set? m:point-set?)
296
297 (defhandler m:region-equivalent?
298   m:rays-equivalent? m:ray? m:ray?)
299
300 (defhandler m:region-equivalent?
301   m:arcs-equivalent? m:arc? m:arc?)
302
303 (defhandler m:region-equivalent?
304   m:region-contradictions-equivalent?
305   m:region-contradiction?
306   m:region-contradiction?)
307
308 ;;;;;;;;; Interface to Propagator System ;;;;;;;;;
309
310 (define (m:region? x)
311   (or (m:point-set? x)
312       (m:ray? x)
313       (m:arc? x)
314       (m:region-contradiction? x)))
315
316
317 (defhandler equivalent? m:region-equivalent? m:region? m:region?)
318
319 (defhandler merge m:intersect-regions m:region? m:region?)
320
321 (defhandler contradictory? m:region-contradiction? m:region?)
322
323 #|
324 Simple Examples
325 (pp (let-cells (c)
326   (add-content c (m:make-arc (make-point 1 0) (sqrt 2)
327     (make-direction-interval
328       (make-direction (/ pi 8))
329       (make-direction (* 7 (/ pi 8))))))
330
331   (add-content c (m:make-ray (make-point -3 1) (make-direction 0)))
332   (add-content c (m:make-ray (make-point 1 2)
333     (make-direction (* 7 (/ pi 4))))))
334   (content c)))
335
336 (let ((a (make-point 0 0))
337       (b (make-point 1 0))
338       (c (make-point 0 1))

```

```

339     (d (make-point 1 1)))
340 (let-cells (cell)
341   (add-content cell
342     (make-tms
343       (contingent (m:make-point-set (list a b c))
344         '(a))))
345   (add-content cell
346     (make-tms
347       (contingent (m:make-point-set (list a d))
348         '(a))))
349   (pp (tms-query (content cell)))))
350 |#
351 ;;;;;;;;;;;;;; To Figure elements ;;;;;;;;;;;;;;
352
353 (define m:region->figure-elements
354   (make-generic-operation 1 'm:region->figure-elements (lambda (r) #f )))
355
356 (defhandler m:region->figure-elements
357   m:ray->figure-ray
358   m:ray?)
359
360 (defhandler m:region->figure-elements
361   m:region-contradiction->figure-elements
362   m:region-contradiction?)

```

Listing A.38: manipulate/constraints.scm

```

1  ;; constraints.scm --- Constraints for mechanisms
2
3  ;; Commentary:
4
5  ;; Ideas:
6  ;; - Abstraction for specifying constraints
7  ;; - Length, angle equality
8  ;; - Perpendicular / Parellel
9
10 ;; Future:
11 ;; - Constraints for other linkages?
12
13 ;; Code:
14
15 ;;;;;;;;;;;;;;;;;; Constraint Structure ;;;;;;;;;;;;;;;;;;
16
17 (define-record-type <m:constraint>
18   (m:make-constraint type args constraint-procedure)
19   m:constraint?
20   (type m:constraint-type)
21   (args m:constraint-args)
22   (constraint-procedure m:constraint-procedure))
23
24 ;;;;;;;;;;;;;;;;;; Constraint Types ;;;;;;;;;;;;;;;;;;
25
26 (define (m:c-length-equal bar-id-1 bar-id-2)
27   (m:make-constraint
28     'm:c-length-equal
29     (list bar-id-1 bar-id-2)
30     (lambda (m)
31       (let ((bar-1 (m:lookup m bar-id-1))
32             (bar-2 (m:lookup m bar-id-2)))
33         (c:id
34           (m:bar-length bar-1)
35           (m:bar-length bar-2))))))
36
37 (define (m:c-angle-equal joint-id-1 joint-id-2)
38   (m:make-constraint
39     'm:c-angle-equal
40     (list joint-id-1 joint-id-2)
41     (lambda (m)
42       (let ((joint-1 (m:lookup m joint-id-1))
43             (joint-2 (m:lookup m joint-id-2)))
44         (c:id (m:joint-theta joint-1)
45               (m:joint-theta joint-2))))))
46
47 (define (m:c-right-angle joint-id)
48   (m:make-constraint
49     'm:right-angle
50     (list joint-id)
51     (lambda (m)
52       (let ((joint (m:lookup m joint-id)))
53         (c:id
54           (m:joint-theta joint)
55           (/ pi 2))))))
56
57 ;; p2 between p1 p3 in a line
58 (define (m:c-line-order p1-id p2-id p3-id)
59   (list
60     (m:make-named-bar p1-id p2-id)
61     (m:make-named-bar p2-id p3-id)
62     (m:make-named-joint p1-id p2-id p3-id)
63     (m:c-full-angle (m:joint p1-id p2-id p3-id))))
64
65 (define (m:c-full-angle joint-id)
66   (m:make-constraint

```

```

67 'm:full-angle
68 (list joint-id)
69 (lambda (m)
70   (let ((joint (m:lookup m joint-id)))
71     (c:id
72      (m:joint-theta joint)
73      pi))))
74
75 (define (m:equal-joints-in-sum equal-joint-ids
76                                   all-joint-ids
77                                   total-sum)
78   (m:make-constraint
79    'm:equal-joints-in-sum
80    all-joint-ids
81    (lambda (m)
82      (let ((all-joints (m:multi-lookup m all-joint-ids))
83            (equal-joints (m:multi-lookup m equal-joint-ids)))
84        (let ((other-joints
85              (set-difference all-joints equal-joints eq?)))
86          (c:id (m:joint-theta (car equal-joints))
87               (ce:/
88                (ce:- total-sum
89                     (ce:multi+ (map m:joint-theta other-joints)))
89                (length equal-joints)))))))
91
92 ;;;;;;;;;;;;;; Applying and Marking Constrained Elements ;;;;;;;;;;;;;;
93
94 (define (m:constrained? element)
95   (not (null? (m:element-constraints element))))
96
97 (define (m:element-constraints element)
98   (or (eq-get element 'm:constraints)
99       '()))
100
101 (define (m:set-element-constraints! element constraints)
102   (eq-put! element 'm:constraints constraints))
103
104 (define (m:mark-constraint element constraint)
105   (m:set-element-constraints!
106    element
107    (cons constraint
108          (m:element-constraints element))))
109
110 (define (m:apply-constraint m constraint)
111   (for-each (lambda (element-id)
112               (m:mark-constraint
113                (m:lookup m element-id)
114                constraint))
115             (m:constraint-args constraint))
116   ((m:constraint-procedure constraint) m))
117
118 ;;;;;;;;;;;;;; Propagator Utils ;;;;;;;;;;;;;;
119
120 (define (ce:multi+ cells)
121   (cond ((null? cells) 0)
122         ((null? (cdr cells)) (car cells))
123         (else
124          (ce:+ (car cells)
125                (ce:multi+ (cdr cells))))))

```

Listing A.39: manipulate/topology.scm

```

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

```

Listing A.40: manipulate/mechanism.scm

```

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

```

```

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

```

```

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

```

Listing A.41: manipulate/main.scm

```

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

```

```

67 (m:equal-joints-in-sum
68 (list (m:joint 'a) (m:joint 'b))
69 (list (m:joint 'a) (m:joint 'b) (m:joint 'c))
70 pi)))
71
72 ;;; Often works:
73 (define (arbitrary-quadrilateral)
74 (m:mechanism
75 (m:establish-polygon-topology 'a 'b 'c 'd)))
76
77 ;;; Always works:
78 (define (parallelogram-by-sides)
79 (m:mechanism
80 (m:establish-polygon-topology 'a 'b 'c 'd)
81 (m:c-length-equal (m:bar 'a 'b)
82 (m:bar 'c 'd))
83 (m:c-length-equal (m:bar 'b 'c)
84 (m:bar 'd 'a))))
85
86 (define (kite-by-sides)
87 (m:mechanism
88 (m:establish-polygon-topology 'a 'b 'c 'd)
89 (m:c-length-equal (m:bar 'a 'b)
90 (m:bar 'b 'c))
91 (m:c-length-equal (m:bar 'c 'd)
92 (m:bar 'd 'a))))
93
94 (define (rhombus-by-sides)
95 (m:mechanism
96 (m:establish-polygon-topology 'a 'b 'c 'd)
97 (m:c-length-equal (m:bar 'a 'b)
98 (m:bar 'b 'c))
99 (m:c-length-equal (m:bar 'b 'c)
100 (m:bar 'c 'd))
101 (m:c-length-equal (m:bar 'c 'd)
102 (m:bar 'a 'd))))
103
104 ;;; Never works:
105 (define (parallelogram-by-angles)
106 (m:mechanism
107 (m:establish-polygon-topology 'a 'b 'c 'd)
108 (m:c-angle-equal (m:joint 'a)
109 (m:joint 'c))
110 (m:c-angle-equal (m:joint 'b)
111 (m:joint 'd)))
112
113 (m:equal-joints-in-sum
114 (list (m:joint 'a) (m:joint 'c))
115 (list (m:joint 'a) (m:joint 'b) (m:joint 'c) (m:joint 'd))
116 (* 2 pi))
117 (m:equal-joints-in-sum
118 (list (m:joint 'b) (m:joint 'd))
119 (list (m:joint 'a) (m:joint 'b) (m:joint 'c) (m:joint 'd))
120 (* 2 pi))))
121
122 (define *m*)
123 (define (m:run-mechanism mechanism-proc)
124 (initialize-scheduler)
125 (let ((m (mechanism-proc)))
126 (set! *m* m)
127 (m:build-mechanism m)
128 (m:solve-mechanism m)
129 (let ((f (m:mechanism->figure m)))
130 (draw-figure f c)
131 ;;(pp (analyze-figure f))
132 )))
133
134 #|

```

```

135 (let lp ()
136   (initialize-scheduler)
137   (pp 'start)
138   (m:run-mechanism
139     (lambda ()
140       (m:mechanism
141         ;;(m:establish-polygon-topology 'a 'b 'c)
142         (m:make-named-bar 'a 'b)
143         (m:make-named-bar 'b 'c)
144         (m:make-named-bar 'c 'a)
145         (m:make-named-joint 'c 'b 'a)
146         (m:make-named-joint 'a 'c 'b)
147         (m:make-named-joint 'b 'a 'c)
148
149         (m:make-named-bar 'a 'd)
150         (m:make-named-bar 'b 'd)
151         (m:make-named-joint 'd 'a 'b)
152         (m:make-named-joint 'a 'b 'd)
153         (m:make-named-joint 'b 'd 'a)
154
155         (m:make-named-bar 'c 'd)
156         (m:make-named-joint 'a 'd 'c)
157         (m:make-named-joint 'c 'a 'd)
158         (m:make-named-joint 'd 'c 'a))))
159   (lp))
160
161 (let lp ()
162   (initialize-scheduler)
163   (let ((m (m:mechanism
164     (m:establish-polygon-topology 'a 'b 'c 'd))))
165     (m:build-mechanism m)
166     (m:solve-mechanism m)
167     (let ((f (m:mechanism->figure m)))
168       (draw-figure f c)
169       (pp (analyze-figure f))))))
170 |#

```

Listing A.42: content/load.scm

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

```

```

67 ;; Goal: p is on the perpendicular bisector of a, b
68 (define (perpendicular-bisector-converse)
69   (let-geo* ((p (random-point))
70             (a (random-point))
71             (b (rotate-randomly-about p a))
72             (s (make-segment a b))
73             (pb (perpendicular-bisector s)))
74     (figure p a b s pb)))
75 ;; TODO: aux-segment
76
77 ;; [7] Shortest distance conjecture
78 ;; Givens: arbitrary point p, point a on line l
79 ;; Goal: Discover that shortest distance to line is along perpendicular
80 (define (shortest-distance)
81   (let-geo* ((p (random-point))
82             (l (random-line))
83             (a (random-point-on-line l)))
84     (figure p l a (make-auxiliary-segment p a))))
85 ;; TODO: Tricky, figure out how to minimize value, specify "minimize" property?
86
87 ;; [8] Angle bisector conjecture
88 ;; Given: angle a-1 of rays r-1, r-2, point a on angle-bisector l1
89 ;; Goal: Distance from a to r-1 = distance a to r-2
90
91 (define (angle-bisector-distance)
92   (let-geo* (((a (r-1 v r-2)) (random-angle))
93             (ab (angle-bisector a))
94             (p (random-point-on-ray ab))
95             ((s-1 (p b)) (perpendicular-to r-1 p))
96             ((s-2 (p c)) (perpendicular-to r-2 p)))
97     (figure a r-1 r-2 ab p s-1 s-2)))
98 ;; Interesting, dependent on "shortest distance" from prior conjecture
99
100 ;; [9] Angle bisector concurrency
101 ;; Given: Triangle abc with angle-bisectors l1, l2, l3
102 ;; Goal: l1, l2, l3 are concurrent
103 (define (angle-bisector-concurrency)
104   (let-geo* (((t1 (a b c)) (random-triangle))
105             (((a-1 a-2 a-3)) (polygon-angles t1))
106             (l1 (polygon-angle-bisector t1 a))
107             (l2 (polygon-angle-bisector t1 b))
108             (l3 (polygon-angle-bisector t1 c)))
109     (figure t1 l1 l2 l3)))
110 ;; TODO: Concurrency of lines
111 ;; TODO: Draw markings for angle bisector
112
113 ;; [10] Perpendicular Bisector Concurrency
114 ;; Given: Triangle ABC with sides s1, s2, s3, perpendicular bisectors
115 ;; l1, l2, l3
116 ;; Goal: l1, l2, l3 are concurrent
117 (define (perpendicular-bisector-concurrency)
118   (let-geo* (((t (a b c)) (random-triangle))
119             (l1 (perpendicular-bisector (make-segment a b)))
120             (l2 (perpendicular-bisector (make-segment b c)))
121             (l3 (perpendicular-bisector (make-segment c a))))
122     (figure t l1 l2 l3)))
123
124 ;; [11] Altitude Concurrency
125 ;; Given: Triangle ABC with altitudes alt-1, alt2, alt-3
126 ;; Goal: alt-1, alt-2, alt-3 are concurrent
127 (define (altitude-concurrency)
128   (let-geo* (((t (a b c)) (random-triangle))
129             (alt-1 (perpendicular-line-to (make-segment b c) a))
130             (alt-2 (perpendicular-line-to (make-segment a c) b))
131             (alt-3 (perpendicular-line-to (make-segment a b) c)))
132     (figure t alt-1 alt-2 alt-3)))
133 ;; TODO: Resist redundant concurrencies
134 ;; 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] Isocles Triangle Conjecture
151 ;;; [19] Converse of Isocles Triangle Conjecture
152 ;;; [20] Triangle Inequality Conjecture
153 ;;; [21] Side-Angle Inequality Conjecture
154 ;;; [22] Triangle Exterior Angle Conjecture
155 ;;; [23] SSS Congruence Conjecture
156 ;;; [24] SAS Congruence Conjecture
157 ;;; [24b] SSA - Congruency?
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/Equiangular 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] Isocles Trapezoid Conjecture
175 ;;; [40] Isocles 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

```

```

203 ;; [62] Angles Inscribed in a Semicircle Conjecture
204 ;; [63] Cyclic Quadrilateral Conjecture
205 |#
206 (define (cyclic-quadrilateral)
207   (let-geo*
208     ((cir (random-circle))
209      ((a b c d) (n-random-points-on-circle-ccw cir 4))
210      (q (polygon-from-points a b c d)))
211     (figure q)))
212 |#
213 ;; [64] Parallel Lines Intercepted Arcs Conjecture
214 ;; [65] Circumference Conjecture
215 ;; [66] Arc Length Conjecture
216 ;; [Exp.3] Intersecting Secants Conjecture
217 ;; [Exp.4] Intersecting Chords Conjecture
218 ;; [Exp.5] Tangent-Secant Conjecture
219 ;; [Exp.6] Intersecting Tangents Conjecture
220 ;; [Exp.7] Tangent-Chord Conjecture
221 ;; [67] Reflection Line Conjecture
222 ;; [68] Coordinate Transforms Conjecture
223 ;; [69] Minimal Path Conjecture
224 ;; [70] Reflections Across Parallel Lines Conjecture
225 ;; [71] Reflections Across Intersecting Lines Conjecture
226 ;; [72] Tessellating Triangles Conjecture
227 ;; [73] Tessellating Quadrilateral Conjecture
228 ;; [74] Rectangle Area Conjecture
229 ;; [75] Parallelogram Area Conjecture
230 ;; [76] Triangle Area Conjecture
231 ;; [77] Trapezoid Area Conjecture
232 ;; [78] Kite Area Conjecture
233 ;; [79] Regular Polygon Area Conjecture
234 ;; [80] Circle Area Conjecture
235 ;; [81] Pythagorean Theorem
236 ;; [82] Converse of Pythagorean Theorem
237 ;; [83] Isosceles Right Triangle Conjecture
238 ;; [84] 30-60-90 Triangle Conjecture
239 ;; [85] Distance Formula
240 ;; [86] Prism-Cylinder Volume Conjecture
241 ;; [87] Pyramid-Cone Volume Conjecture
242 ;; [Exp.8] Platonic Solids
243 ;; [88] Sphere Volume Conjecture
244 ;; [89] Sphere Surface Area Conjecture
245 ;; [91] AA Similarity Conjecture
246 ;; [92] SSS Similarity Conjecture
247 ;; [93] SAS Similarity Conjecture
248 ;; [94] Proportional Parts Conjecture
249 ;; [95] Angle Bisector / Opposite Side Conjecture
250 ;; [96] Proportional Area Conjecture
251 ;; [97] Proportional Volumes Conjecture
252 ;; [98] Parallel/Proportionality Conjecture
253 ;; [99] Extended Parallel/Proportionality Conjecture
254 ;; [100] SAS Triangle Area Conjecture
255 ;; [101] Law of Sines
256 ;; [102] Law of Cosines
257 ;; [Exp.9] Special Constructions
258 |#

```

Listing A.44: core/load.scm

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

Listing A.45: core/animation.scm

```

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

```

```

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

```

Listing A.46: core/macros.scm

```

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

```

```

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

```

Listing A.47: core/print.scm

```

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

```


Listing A.48: core/utils.scm

```

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

```

```

67         (or (eq-get element key) '()))))
68
69 (define (sort-by-key l key)
70   (sort l (lambda (v1 v2)
71             (< (key v1)
72                (key v2)))))
73
74 (define (sort-by-key-2 l key)
75   (let ((v (sort-by-key-2 l key)))
76     (pprint (map (lambda (x) (cons (name x) (key x))) v))
77     v))
78
79 (define ((negatep f) x)
80   (- (f x)))
81
82 (define ((flip-args f) x y)
83   (f y x))
84
85 (define (index-of el list equality-predicate)
86   (let lp ((i 0)
87            (l list))
88     (cond ((null? l) #f)
89           ((equality-predicate (car l) el)
90            i)
91           (else (lp (+ i 1) (cdr l))))))
92
93 ;;; (nth-letter-symbol 1) => 'a , 2 => 'b, etc.
94 (define (nth-letter-symbol i)
95   (symbol (make-char (+ 96 i) 0)))
96
97 (define (hash-table/append table key element)
98   (hash-table/put! table
99                    key
100                    (cons element
101                          (hash-table/get table key '()))))
102
103 (define (dedupe-by equality-predicate elements)
104   (dedupe (member-procedure equality-predicate) elements))
105
106 (define (dedupe member-predicate elements)
107   (cond ((null? elements) '())
108         (else
109          (let ((b1 (car elements)))
110            (if (member-predicate b1 (cdr elements))
111                (dedupe member-predicate (cdr elements))
112                (cons b1 (dedupe member-predicate (cdr elements))))))))

```