lib/basic/graph-draw.ath

```
(load-file "rsarray.ath")

(load-file "maps.ath")

module Graph-Draw {

private define vs := val->string

define dot-executable := (cell "")

define set-dot-executable := lambda (path) (set! dot-executable path)

(set-dot-executable "\"c:\\Program Files (x86)\\GraphViz 2.28\\bin\\dot.exe\"")

private define string? :=
 lambda (v) (&& (list? v) (for-each v char?))

private define val->string :=
  lambda (v) check {(string? v) => v | else => (vs v)}

private define val->string-nl :=
 lambda (v)
   let {res := (val->string v)}
     (flatten (map lambda (c)
                    check {(equal? c '\n) => ['\\ 'n]
                         | else => [c]}
                  res))

# A graph is represented as a triple [ht node-vector counter]
# where ht is a hash table mapping values to indices in node-vector,
# and counter is a cell used to generate fresh successive indices.
# Node-vector is the resizable array that holds information about
# the vertices.

(define (make-graph N)
  (check ((less? N 1) [(make-hash-table 100) (make-rs-array 5000 () 1000) (cell 1)])
         (else [(make-hash-table N) (make-rs-array N () 4096) (cell 1)])))

private define [graph-ht graph-array graph-counter] := [first second third]

private define inc-counter :=
  lambda (counter)
    let {x := (ref counter);
         _ := (set! counter (x plus 1))}
      x

# A node (i.e. an element of the node-vector array) is a quadruple
# consisting of: (1) a value; (2) that value's string representation;
# (3) a mapping from attribute labels (represented as meta-identifiers) to
# attribute values (represented as strings); and (4) a list of edges, where
# an edge is a triple.

# Specifically, an edge is a triple [unique-id index attribute-mapping]
# where unique-id is some value that can uniquely identify
# the edge (distinguish it from all other edges in the graph);
# the index number indicates the (index of the) target node;
# and attribute-mapping is a map from attribute labels
# (represented as meta-identifiers) to attribute
# values (represented as strings).

private define unique-edge-counter := (cell 0)

private define make-unique-edge-id :=
  lambda ()
    let {res := (ref unique-edge-counter);
         _ := (set! unique-edge-counter (plus res 1))}
      res
```

```
68
69 (define (make-node v str attr-map edge-list)
70    [v str attr-map edge-list])
71
72 (define (make-default-node v str)
73   [v str empty-map []])
74
75 private define [get-node-value get-node-value-string get-node-attribute-map get-node-edge-list] :=
76                [first second third fourth]
77
78 # (define get-node-value first)
79 # (define get-node-value-string second)
80 # (define get-node-attribute-map third)
81 # (define get-node-edge-list fourth)
82
83 (define (make-default-edge i)
84   [(make-unique-edge-id) i empty-map])
85
86 (define (make-edge i unique-id)
87   [unique-id i empty-map])
88
89 (define (make-full-edge unique-id i attr-map)
90   [unique-id i attr-map])
91
92 (define get-edge-unique-id first)
93 (define get-edge-target second)
94 (define get-edge-attribute-map third)
95
96 (define (add-entry ht A counter v str)
97   (let ((index (inc-counter counter))
98         (_ (enter ht str index))
99         (new-node (make-default-node v str))
100        (_ (rs-array-set A index new-node)))
101    index))
102
103 (define (add-node g v)
104   (let ((ht (graph-ht g))
105         (A  (graph-array g))
106         (counter (graph-counter g))
107         (str (val->string-nl v)))
108     (match (look-up ht str)
109       (() (seq (add-entry ht A counter v str) ()))
110       (_ ()))))
111
112 # Note: pass the unit () as the value of the unique-edge-id parameter
113 # if you do not care to give the edge a unique id:
114
115 (define (add-edge g v1 v2 unique-edge-id)
116   (let ((ht (graph-ht g))
117         (A  (graph-array g))
118         (counter (graph-counter g))
119         (str1 (val->string-nl v1))
120         (str2 (val->string-nl v2))
121         (index1 (match (look-up ht str1)
122                   (() (add-entry ht A counter v1 str1))
123                   (i i)))
124         (index2 (match (look-up ht str2)
125                   (() (add-entry ht A counter v2 str2))
126                   (i i)))
127         (node (rs-array-sub A index1))
128         (new-edge (check ((equal? unique-edge-id ()) (make-default-edge index2))
129                          (else (make-edge index2 unique-edge-id)))))
130    (rs-array-set A index1
131       (make-node (get-node-value node)
132                  (get-node-value-string node)
133                  (get-node-attribute-map node)
134                  (add new-edge (get-node-edge-list node))))))
135
136 (define (make-node-id index)
137   (join "v" (val->string index)))
```

```
138
139 (define (set-node-attribute G v attr attr-val)
140   (let ((ht (graph-ht G))
141         (A  (graph-array G))
142         (str (val->string-nl v)))
143     (match (look-up ht str)
144       (() (error (join "The graph contains no such node currently: " str)))
145       (index (let ((node (rs-array-sub A index))
146                    (v (get-node-value node))
147                    (str (get-node-value-string node))
148                    (attr-map (get-node-attribute-map node))
149                    (edge-list (get-node-edge-list node))
150                    (attr-map' (add-binding attr attr-val attr-map))
151                    (node' (make-node v str attr-map' edge-list)))
152                (rs-array-set A index node')))))
153
154
155 (define (find-edge-with-unique-id node-edge-list unique-edge-id)
156   (letrec ((loop (lambda (rest visited)
157                    (match rest
158                      ([] (error (join "find-edge-with-unique-id failed; no edge found with this unique-id: " (val->str
159                      ((list-of e more) (check ((equal? (get-edge-unique-id e) unique-edge-id) [e (join visited more)])
160                                               (else (loop more (add e visited)))))))))
161     (loop node-edge-list [])))
162
163 (define (set-edge-attribute G v unique-edge-id attr attr-val)
164   (let ((ht (graph-ht G))
165         (A  (graph-array G))
166         (counter (graph-counter G))
167         (str (val->string-nl v))
168         (index (match (look-up ht str)
169                    (() (error (join "Failed set-edge-attribute operation; there is no such node in the graph yet: " (v
170                    (i i)))
171         (node (rs-array-sub A index))
172         (node-val (get-node-value node))
173         (node-val-str (get-node-value-string node))
174         (node-attr-map (get-node-attribute-map node))
175         (node-edge-list (get-node-edge-list node))
176         (node-edge-list' (let (([edge rest-edges] (find-edge-with-unique-id node-edge-list unique-edge-id))
177                                (edge-target-node (get-edge-target edge))
178                                (edge-attr-map (get-edge-attribute-map edge))
179                                (edge-attr-map' (add-binding attr attr-val edge-attr-map))
180                                (edge' (make-full-edge unique-edge-id edge-target-node edge-attr-map')))
181                           (add edge' rest-edges))))
182     (rs-array-set A index
183        (make-node node-val node-val-str node-attr-map node-edge-list'))))
184
185
186 (define (make-attr-line attr-map)
187   (let ((make-av-pair (lambda (a-v-pair)
188                         (match a-v-pair
189                           ([a v] (let ((a-str (tail (val->string-nl a)))
190                                        (v-str (val->string-nl v))
191                                        (line (join a-str "=" v-str)))
192                                    line))))))
193         (pair-list (map make-av-pair (dom-range-list attr-map))))
194     (join "[" (separate pair-list ",") "];\n")))
195
196 (define (draw g file-name)
197   (let ((file (join file-name ".dot"))
198         (_ (delete-file file))
199         (ht (graph-ht g))
200         (A  (graph-array g))
201         (node-count (minus (ref (graph-counter g)) 1))
202         (draw-node (lambda (index)
203                      (let ((node (rs-array-sub A index))
204                            (node-id (make-node-id index))
205                            (v (get-node-value node))
206                            (str (get-node-value-string node))
207                            (attr-map (get-node-attribute-map node))
```

```
208                             (basic-node-line (join "\n" node-id " [label=\"" str "\"]" ";\n"))
209                             (_ (write-file file basic-node-line))
210                             (node-attribute-line (join "\n" node-id " " (make-attr-line attr-map)))
211                             (_ (write-file file node-attribute-line))
212                             (edge-list (get-node-edge-list node))
213                             (draw-edge (lambda (edge)
214                                          (let ((target-index (get-edge-target edge))
215                                                (edge-attr-map (get-edge-attribute-map edge))
216                                                (target-id (make-node-id target-index))
217                                                (edge-line (join "\n" node-id " -> " target-id " " (make-attr-line edge
218                                            (write-file file edge-line)))))
219                       (map-proc draw-edge (rev edge-list)))))
220         (L (from-to 1 node-count))
221         (_ (write-file file "digraph G {\n"))
222         (_ (map-proc draw-node L)))
223     (write-file file "\n}\n")))
224
225
226 (define viewer (cell ()))
227
228 (define (set-viewer v) (set! viewer v))
229
230 (set-viewer "\"c:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe\"")
231
232 (define (draw-and-view g file-name viewer)
233   (let ((_ (draw g file-name))
234         (dot-program (ref dot-executable))
235         (dot-file (join file-name ".dot"))
236         (gif-file (join file-name ".gif"))
237         (command-string-1 (join dot-program " -Tgif " dot-file " -o " gif-file))
238 #         (_ (print "\nCommand-string-1: " command-string-1 "\n"))
239         (_ (exec-command command-string-1))
240         (command-string-2 (join viewer " " gif-file))
241 #         (_ (print "\nCommand-string-2: " command-string-2 "\n"))
242         (_ (exec-command command-string-2))
243 #         (_ (print "\nOK...\n"))
244       )
245    ()))
246
247 (define (draw-and-show g viewer)
248   (draw-and-view g "tmp.dot" (ref viewer)))
249
250 (define (draw-term-0 G counter t)
251   (let ((make-node (lambda (v)
252                      (let ((new-node (inc counter))
253                            (_ (add-node G new-node))
254                            (_ (set-node-attribute G new-node 'label (join "\"" (val->string-nl v) "\""))))
255                        new-node))))
256     (letrec ((add-info (lambda (t)
257                          (match t
258                            ((some-var v) (make-node v))
259                            (((some-symbol f) (some-list args))
260                                (let ((f-node (make-node f))
261                                      (_ (seq (map (lambda (root)
262                                                     (add-edge G f-node root ()))
263                                                   (map add-info args)))))
264                                  f-node))))))
265         (add-info t))))
266
267 (define (draw-term t viewer)
268   (let ((G (make-graph (term-size t)))
269         (counter (cell 0)))
270    (seq (draw-term-0 G counter t)
271         (draw-and-view G "term-graph" viewer))))
272
273
274 (define (draw-sentence p viewer)
275   (let ((G (make-graph (prop-size p)))
276         (counter (cell 0))
277         (make-node (lambda (v)
```

```
                      (let ((new-node (inc counter))
                            (_ (add-node G new-node))
                            (_ (set-node-attribute G new-node 'label (join "\"" (val->string-nl v) "\""))))
                        new-node))))
    (letrec ((add-info (lambda (p)
                         (match p
                           ((some-atom t) (draw-term-0 G counter t))
                           (((some-sent-con sc) (some-list args))
                               (let ((sc-node (make-node sc))
                                     (_ (seq (map (lambda (root)
                                                    (add-edge G sc-node root ()))
                                                  (map add-info args)))))
                                 sc-node))
                           (((some-quant Q) (some-list vars) (some-sentence body))
                              (let ((q-node (make-node Q))
                                    (_ (map (lambda (root)
                                              (add-edge G q-node root ()))
                                            (join (map (lambda (t) (draw-term-0 G counter t)) vars)
                                                  [(add-info body)]))))
                                q-node))))))
        (seq (add-info p)
             (draw-and-view G "term-graph" viewer)))))

(define (draw-sentence0 p)
  (let ((G (make-graph (prop-size p)))
        (counter (cell 0))
        (make-node (lambda (v)
                     (let ((new-node (inc counter))
                           (_ (add-node G new-node))
                           (_ (set-node-attribute G new-node 'label (join "\"" (val->string-nl v) "\""))))
                       new-node))))
    (letrec ((add-info (lambda (p)
                         (match p
                           ((some-atom t) (draw-term-0 G counter t))
                           (((some-sent-con sc) (some-list args))
                               (let ((sc-node (make-node sc))
                                     (_ (seq (map (lambda (root)
                                                    (add-edge G sc-node root ()))
                                                  (map add-info args)))))
                                 sc-node))
                           (((some-quant Q) (some-list vars) (some-sentence body))
                              (let ((q-node (make-node Q))
                                    (_ (map (lambda (root)
                                              (add-edge G q-node root ()))
                                            (join (map (lambda (t) (draw-term-0 G counter t)) vars)
                                                  [(add-info body)]))))
                                q-node))))))
        (seq (add-info p) G))))


}
```