Symbols
Robots & A-lists

Dr. Kimberle Koile
Symbols in Scheme

A compound proc that squares its argument

(lambda (x) (* x x))

(quote beta)

# [compound-...]

symbol beta

eval

lambda-rule

print

eval

quote-rule

print
Symbols are ordinary values

\[(\text{list } 1 \ 2) \quad \Rightarrow \quad (1 \ 2)\]

\[(\text{list (quote delta) (quote gamma)}) \quad \Rightarrow \quad (\text{delta gamma})\]
practice

(define a 1)
(define b 2)

What does the Scheme interpreter print for each of the expressions:

- (list a b) = (1 2)
- (list 'a 'b) = (a b)
- (list 'a b) = (a 2)
- (car 'a) = error

the symbol a is not a pair
(define a 1)
(define b 2)

What does the Scheme interpreter print for each of the expressions:

(car "a")
(cadr "list")
((cadr "list) a b)

- (car "a") prints quote
- (cadr "list") prints list
- ((cadr "list) a b) prints error

quote - this is the symbol quote, not the procedure:
(car (quote (quote a)))
(car (quote a)) => quote

list is a symbol, not a procedure
Wallace and Gromit
Wallace and Gromit have just finished their vacation on the moon and are about to head back to Earth in their rocket ship (located at position G below). The local robot desperately wants to go back with them, but must hurry to get to the rocket ship in time. (He’s at S below.) He has to navigate around two obstacles (shown as triangles AEF and BCD.) He uses his nifty search engine to find the best path. In recitation 14 (October 27) we’ll figure out which way he goes. Today let’s figure out the representations needed for his search engine. Below is a graph representing possible paths from the robot’s starting location (S) to the rocket ship’s location (G). The graph consists of nodes (labeled S, and A to G) which are connected by links (aka arcs or edges). Nodes have such properties as id, e.g., S; links in which they are endpoints; and estimated distances to the goal node. Links have properties such as the nodes that are endpoints and length; e.g., the link between S and B has endpoints node S and node B, and a length of 5 (units not specified). Paths through the graph can be represented as ordered sets of nodes and/or links.
Data representation: symbols and alists

<table>
<thead>
<tr>
<th>Link lengths:</th>
<th>Estimates of distance to G from:</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-A 6</td>
<td>A 7</td>
</tr>
<tr>
<td>S-B 5</td>
<td>B 9</td>
</tr>
<tr>
<td>S-C 4</td>
<td>C 13</td>
</tr>
<tr>
<td>A-B 1</td>
<td>D 7</td>
</tr>
<tr>
<td>A-D 7</td>
<td>E 4</td>
</tr>
<tr>
<td>A-E 3</td>
<td>F 4</td>
</tr>
<tr>
<td>A-F 7</td>
<td>G 0</td>
</tr>
<tr>
<td>B-C 6</td>
<td>S 1</td>
</tr>
</tbody>
</table>

Represent it as a symbol: \((B \ D \ 6)\)
Data abstraction: symbols and alists

(define *node-data* '((A 7) (B 9) (C 13) (D 7) (E 4) (F 4) (G 0) (S 1)))

(define *link-data* '(((S A) 6) ((S B) 5) ((S C) 4) ((A B) 1) ((A D) 7) ((A E) 3) ((A F) 7) ((B C) 6) ((B D) 6) ((B E) 5) ((C D) 6) ((D E) 5) ((D G) 8) ((E F) 6) ((E G) 4) ((F G) 4)))
What does the Scheme interpreter print for each of the expressions:

(assoc 'A *node-data*) => (A 7)
(assoc A *node-data*) => error, undefined var A
(assoc 9 *node-data*) => #f
(assoc '7 *node-data*) => #f
(assoc '(C 13) *node-data*) => #f

To get the estimated distance to the goal for a node, we could use the *node-data* list and the procedure `assoc` or `find-assoc`:

```
(define *node-data* '((A 7) (B 9) (C 13) (D 7) (E 4) (F 4) (G 0) (S 1)))
```

```
(define *link-data* '(((S A) 6) ((S B) 5) ((S C) 4) ((A B) 1) ((A D) 7) ((A E) 3) ((A F) 7) ((B C) 6) ((B D) 6) ((B E) 5) ((C D) 6) ((D E) 5) ((D G) 8) ((E F) 6) ((E G) 4) ((F G) 4)))
```

To get the estimated distance to the goal for a node, we could use the *node-data* list and the procedure `assoc` or `find-assoc`:

```
(define (get-node-estimate node-id)
  (cadr (assoc node-id *node-data*)))
```

or
```
(define (get-node-estimate node-id)
  (find-assoc node-id *node-data*))
```
(define *node-data* '((A 7) (B 9) (C 13) (D 7) (E 4) (F 4) (G 0) (S 1)))

(define *link-data* '((((S A) 6) ((S B) 5) ((S C) 4) ((A B) 1) ((A D) 7) ((A E) 3) ((A F) 7) ((B C) 6) ((B D) 6) ((B E) 5) ((C D) 6) ((D E) 5) ((D G) 8) ((E F) 6) ((E G) 4) ((F G) 4)))

To get the length of a link, we could use *link-data* and assoc or find-assoc:

(define (get-link-length node-id1 node-id2)
  (let (cl-data (assoc (list node-id1 node-id2) *link-data*))
    (if cl-data (cdr cl-data) #f)))
Creating nodes using alists

Consider these representations for nodes and links:

```
(define (make-node id estimate-to-goal)
    (cons id estimate-to-goal))
(define (node-id node)
    (car node))
(define (node-estimate-to-goal node)
    (cdr node))
(define (make-link node1 node2 length)
    (list (list node1 node2) length))
```

```
(define *node-data* '((A 7) (B 9) (C 13) (D 7) (E 4) (F 4) (G 0) (S 1)))
```

1. Use map to create nodes using *node-data*.

```
(define *nodes*

(map (lambda (data)
    (make-node (car data) (cadr data))
    *node-data*)))
```

finding a node

2. Find a node in *nodes* given a symbol representing a node id.

Use this function to test for node-id equality:

```
(define equal-node-id? (id1 id2)
  (eq? id1 id2))
```

```
(define find-node (id)
  (define helper (nodes)
    (cond ((null? nodes) '())
      ((equal-node-id id (car nodes))
        (helper (cdr nodes)))
      (else helper *nodes*))
  (helper *nodes*))
```

Note: this one does more work but finds all.
3. Use map to create links using *link data*.

(define *link-data* '(((S A) 6) ((S B) 5) ((S C) 4) ((A B) 1) ((A D) 7) ((A E) 3)
   ((A F) 7) ((B C) 6) ((B D) 6) ((B E) 5) ((C D) 6) ((D E) 5)
   ((D G) 8) ((E F) 6) ((E G) 4) ((F G) 4)))

(define (make-link node1 node2 length)
   (list (list node1 node2) length))

(define *links*
   (map (λ (data)
          (let ((node-ids (car data))
                (make-link
                  (find-node (car node-ids))
                  (find-node (cdr node-ids))
                  (cadar data)))))
        *link-data*))
4. Assume our representation for nodes now includes links:

```
(define (make-node id estimate-to-goal links)
  (list id estimate-to-goal links))

(define (make-link node1 node2 length)
  (list (list node1 node2) length))

(define *node-data* '((A 7) (B 9) (C 13) (D 7) (E 4) (F 4) (G 0) (S 1)))

(define *link-data* '(((S A) 6) ((S B) 5) ((S C) 4) ((A B) 1) ((A D) 7) ((A E) 3)
  ((A F) 7) ((B C) 6) ((B D) 6) ((B E) 5) ((C D) 6) ((D E) 5)
  ((D G) 8) ((E F) 6) ((E G) 4) ((F G) 4)))
```

How do we create links when we need nodes they haven't been created yet?

1. Make all nodes with `(c)` for links
2. Walk through link data; for each data item, find corresponding node;
   make a link, add links to both nodes.