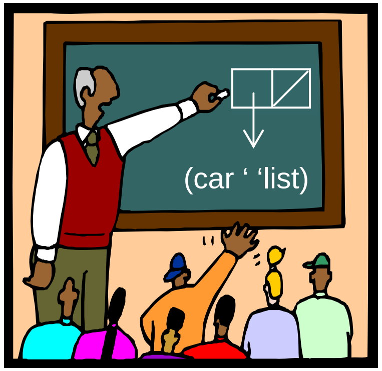
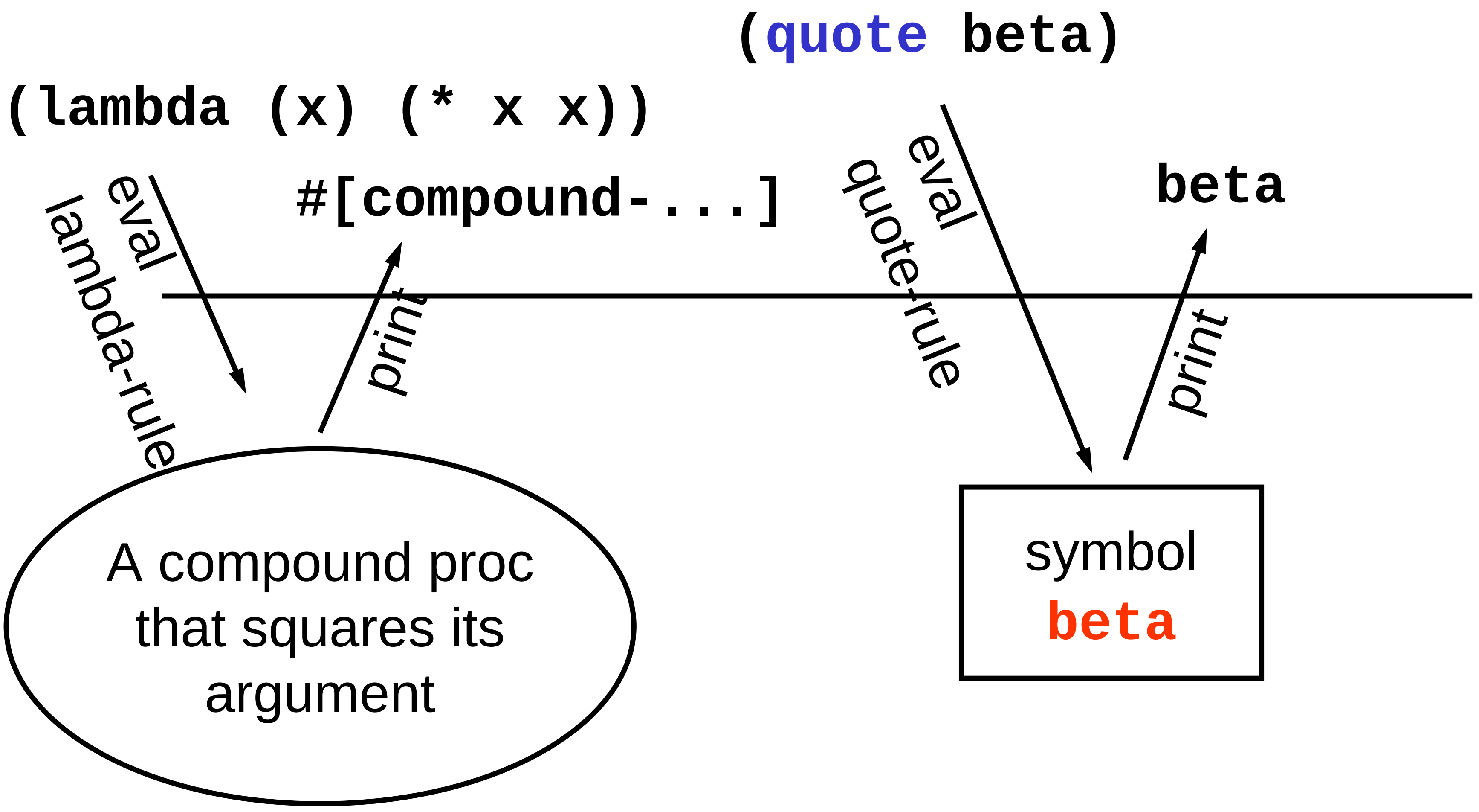
6.001 recitation 3/14/06

- Symbols
- Robots & A-lists



Dr. Kimberle Koile

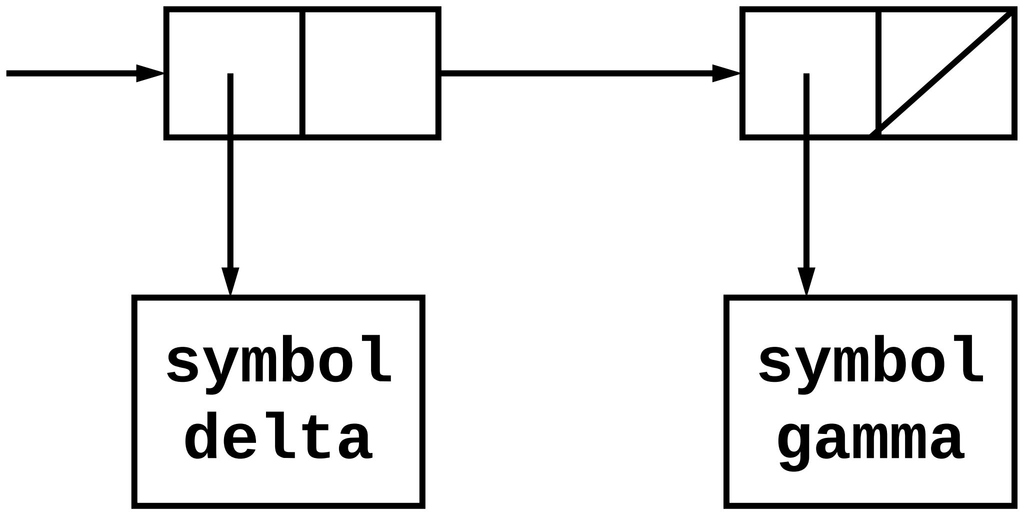
Symbols in Scheme



Symbols are ordinary values

(list 1 2) ==> (1 2)

(list (quote delta) (quote ~~gamma~~ delta))
==> (delta ~~gamma~~ delta)



compare (list 'a 'b)
'(a b)
'(list a b)

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

more practice

(define a 1)

(define b 2)

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

(car ''a)	quote ⊛
(cadr ''list)	list
((cadr ''list) a b)	error

(quote (quote list))
cdr : (list)
cadr : list

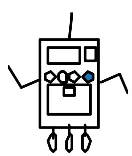
list is a symbol, not a procedure

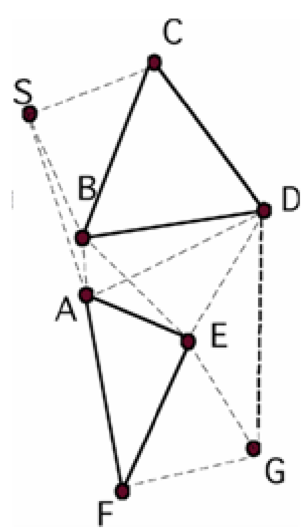
⊛ quote – this is the symbol quote, not the procedure:
(car (quote (quote a)))
(car (quote a)) ⇒ quote

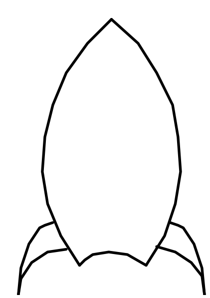
Wallace and Gromit

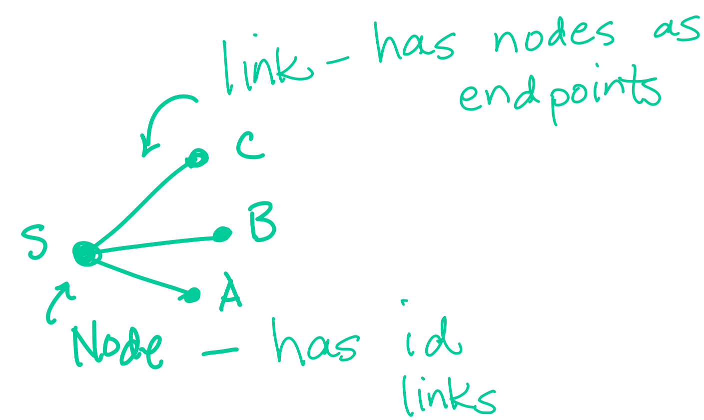
data abstraction, symbols, and search

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.

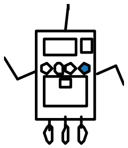


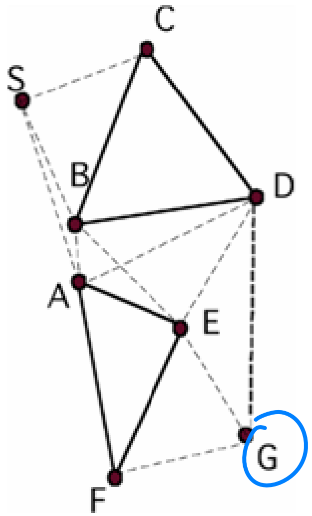




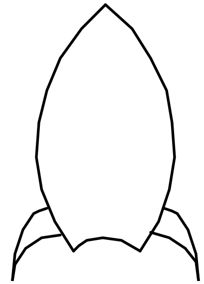


data representation: symbols and alists





represent it as a symbol



Link lengths:

S-A	6	B-D	6
S-B	5	B-E	5
S-C	4	C-D	6
A-B	1	D-E	5
A-D	7	D-G	8
A-E	3	E-F	6
A-F	7	E-G	4
B-C	6	F-G	4

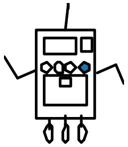
((B D) 6)

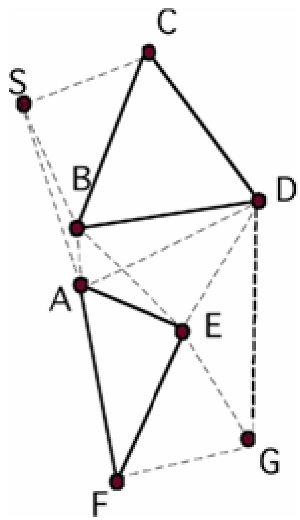
Estimates of distance to G from:

A	7
B	9
C	13
D	7
E	4
F	4
G	0
S	1

(A 7)

data abstraction: symbols and alists

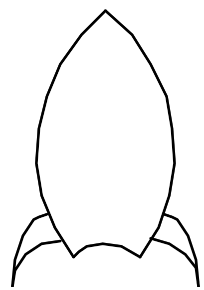




association list (aka alist : list of 2-element lists

assoc: given a key and an a-list, return the first item whose whose car is the key; element is a 2-element list

find-assoc: returns the value associated with a key



2 elt list
key, value

```
(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)))
```

alist practice: assoc

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

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

nodes and links

```
(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 (find-assoc node-id *node-data*)

nodes and links

(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 ((data (or
        (assoc (list node-id1 node-id2)
               *link-data*)
        (assoc (list node-id2 node-id1)
               *link-data*))))
  (if data (cadr data) #f))
```

define procedure find-link that takes 2 node ids as args (to avoid this duplicate code)

)

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 (λ (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))
          (else (helper (cdr nodes)))))
  (helper *nodes*)
)
```

or (let ((nodes (filter (λ (node) (equal-node-id (node-id node)
      (if nodes (car nodes) #f))

Note: this one does more work, like finds all.

creating a link

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 (cadr node-ids))
             (cadr data))))
   *link-data*)
)
```

creating a node containing links!

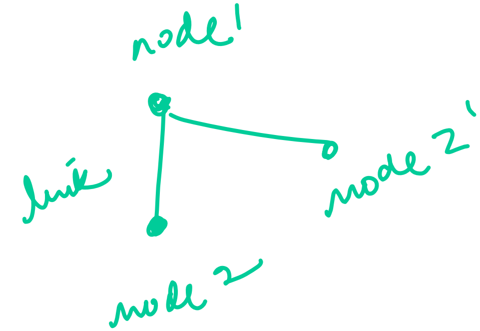
4. Assume our representation for nodes now includes links:

```
(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 '() for links
2. walk through link data; for each data item, find corresponding nodes, make a link, add link to both nodes.

Can you think of another algorithm?