MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Department of Electrical Engineering and Computer Science
6.090—Building Programming Experience
IAP 2005

Lecture 8

Tags

```
; professor abstraction
(define (make-professor name salary)
  (list name salary))

(define (professor-name prof)
  (first prof))

(define (professor-salary prof)
  (second prof))

; graduate student abstraction
(define (make-gradstudent name salary)
  (list name salary))

(define (gradstudent-name grad)
  (first grad))

(define (gradstudent-salary grad)
  (second grad))
```

Given a list that contains both professors and graduate students, compute the total cost of their salaries.

```
(define (total-cost people-list)
```

Association Lists

Scheme

1. `assoc` - (`assoc` *key alist*) - returns association containing matching key or #f.

2. `del-assoc` - (`del-assoc` *key alist*) - returns a new alist with association with matching key removed.

Problems

1. Evaluate the following expressions, first guessing then checking with Scheme.

```
(define alst (list (list 1 2) (list 3 4) (list 5 6)))

(assoc 4 alst)

(assoc 3 alst)

(assoc 5 (cons (list 5 12) alst))

(del-assoc 5 alst)

(define alst2 (list (list "foo" 17) (list "bar" 42) (list "baz" 54)))

(assoc "foo" alst2)

(del-assoc "bar" alst2)

(assoc "yummy" alst2)

(assoc "yummy" alst)
```

2. Rewrite `lookup` from homework 7 using `assoc`.

```
(define (lookup word thesaurus)
```

Trees

```
(define (make-node val left right)
  (list "node" val left right))

(define (node? x)
  (and (pair? x) (string=? (car x) "node")))

(define (node-val node)
  (second node))
(define (node-left node)
  (third node))
(define (node-right node)
  (fourth node))

(define (leaf? x)
  (not (node? x)))
```

1. Write `tree-contains?`, which returns true if the tree contains the value as a leaf.

   ```
   (define (tree-contains? tree val)
   ```

2. Write `sum-tree`, which returns the sum of the leaves of the tree.

   ```
   (define (sum-tree tree)
   ```

```
(define (insert-list elem lst)
  (if (null? lst)
      (list elem)
      (if (< elem (car lst))
          (cons elem lst)
          (cons (car lst) (insert-list elem (cdr lst))))))

(define (avg v1 v2)
  (/ (+ v1 v2) 2))
```

3. Complete `insert-tree`, which returns a *new tree* with the value added to the correct place in the tree.

```
(define (insert-tree elem tree)
  (if (leaf? tree)
      (if (= elem tree)
          INSERT1
          (if (< elem tree)
              INSERT2
              INSERT3))
      (if (< elem (node-val tree))
          (make-node (node-val tree)
                     INSERT4
                     (node-right tree))
          (make-node (node-val tree)
                     (node-left tree)
                     INSERT5))))
```

Animal Guessing Game

Download `lec8.scm` from the website.

1. Write the `animal` abstraction
2. Write the `ask-about-animal` procedure, which should take an animal as input and ask the player if that is their animal

```
(ask-about-animal (make-animal "elephant"))

Is it a elephant (y or n)?        ;  ('n' key was struck)
;Value: #f
```

3. Look at the `play-game` procedure. This procedure uses a `guesser` procedure combined with some `knowledge` of animals in order to guess the player's animal. Let's start off by using a list of `animal`s as the knowledge. Implement `list-guesser`, which takes in a list of animals and asks the player about them until it guesses the animal or runs out of knowledge. If it succeeds, use `print-msg` to print out a victory message. If it runs out of knowledge without guessing the animal, print out `"I give up."`.
4. Look more closely at the `play-game` procedure. It uses the return value of the `guesser` as the new knowledge to use when playing the next game. Thus, we want to have the guesser return the knowledge. The reason `play-game` does this is it allows the guesser to ask a couple more questions when it fails to extend its knowledge to cover the situation where it lost:

```
(play-game new-list-guesser sample-list)

Is it a elephant (y or n)? n

Is it a hummingbird (y or n)? n
I give up.

What was your animal
(Please enter a string (surrounded by "s) and use C-x, C-e to submit it)
"thesaurus"

play again (y or n)? y

Is it a elephant (y or n)? n

Is it a hummingbird (y or n)? n

Is it a thesaurus (y or n)? y
Yay!

play again (y or n)? n
;Value: (("animal" "elephant") ("animal" "hummingbird") ("animal" "thesaurus"))
```

Write a `new-list-guesser` procedure which returns a new improve knowledge list each time it runs.

5. Most games of guess an animal are not played by repeated asking the player about every animal you know. By asking other yes-no questions, the scope of possible animals can be narrowed to a small range. The sounds like a job for trees!

   Implement the `question` abstraction: a question is a node in our knowledge tree.

6. Implement the `ask-question` procedure which asks the player the question.

7. The leaves of the tree are animals. Implement `tree-guesser` that takes in a tree as its knowledge and searchs the tree, asking questions to decide whether the left or right branch is the correct one.

```
(play-game new-tree-guesser sample-tree)

does it fly (y or n)? n

Is it a elephant (y or n)? y
Yay!

play again (y or n)? y

does it fly (y or n)? y

Is it a hummingbird (y or n)? y
Yay!

play again (y or n)? n
;Value: ("question" "does it fly" ("animal" "hummingbird") ("animal" "elephant"))
```

8. Once again, we should write our guesser such that it improves its knowledge each time. The `improve-tree` procedure has been given to you. Write `new-tree-guesser`.