6.090, Building Programming Experience
Ben Vandiver
Lecture 8: Tags, Alists, Trees

From the Lecture 8 handout, directions are:

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

The following program may be written to satisfy these instructions:

```
(define (total-cost people-list)
  (if (null? people-list)
      0
      (+ (second (car people-list)) (total-cost (cdr people-list)))))
```

abstraction violation! (using second on a professor or a student)

Several predicate primitive procedures were named:

(number? x) returns a Boolean depending on whether or not x is a number
(string? x) does the same to check for a string
(pair? x) checks to see if x is a box; nil does not satisfy this procedure with true
(list? x) is it a string of boxes ending in nil? nil itself satisfies this predicate

The "theme" of the lecture was "You will see these topics again in 6.001"

Tagging a list is a standard way of making lists:

```
(define (make-professor name salary)
  (list "professor" name salary))
```

"professor" is the tag part of the list

Professor-name is now defined as the second of the list make-professor
Professor-salary is now defined as the third

(as opposed to first and second, respectively)

```
(define professor? x)
  (and (pair? x) (string? "professor" (car x))))
```

(We can't use list? above because nil is a list and that will cause the program to crash when it reaches the procedure car.)

Using the constructors and selectors that we created on the Lecture 8 handout, we can make contracts:

```
(define (total-cost people)
  (if (null? people)
      0
      (+ (if (professor? (car people))
             (professor-salary (car people))
             (gradstudent-salary (car people)))
         (total-cost (cdr people)))))
```

Association Lists

1. assoc – (assoc *key alist*) – returns association containing matching key or #f. alist is an association list. The following example is from the previous homework (homework 7 with the thesaurus entries):
key: word
value: synonyms

a→((“victory” (“triumph” “win”))
(“ball” (“rondure” “sphere”)))

a is the alist
(assoc “victory” a) => (“victory” (“triumph” “win”))

(assoc “defeat” a) => #f

b→((1 #t) (2 #f) (3 #t) (4 #f))

assoc returns the entire entry rather than simply the associated value for the sake of clarity. The following is a list of assoc commands and what is actually returned. After that is a list of the same assoc commands intended for counter example.

Actual:

```
(assoc 3 b)
;Value: (3 #t)
(assoc 2 b)
;Value: (2 #f)
(assoc 54 b)
;Value: #f
```

Counterexample (if only the associated value were returned):

```
(assoc 3 b)
;Value: #t
(assoc 2 b)
;Value: #f
(assoc 54 b)
;Value: #f
```

If only associated the value is returned, we cannot tell in a case such as the above whether the associated value is #f or whether the key itself is not a part of alist. (It would be good to note that when an key cannot be found in alist, the return value of assoc is #f.)

We can build an association and add it to a previous list as follows:

```
(cons (list 0 #t)
      b)
```

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

(del-assoc "victory" a) => (("ball" ("rondure" "sphere")))

Do Problems on Lecture 8 handout.

2. Rewrite lookup from homework 7 using assoc.

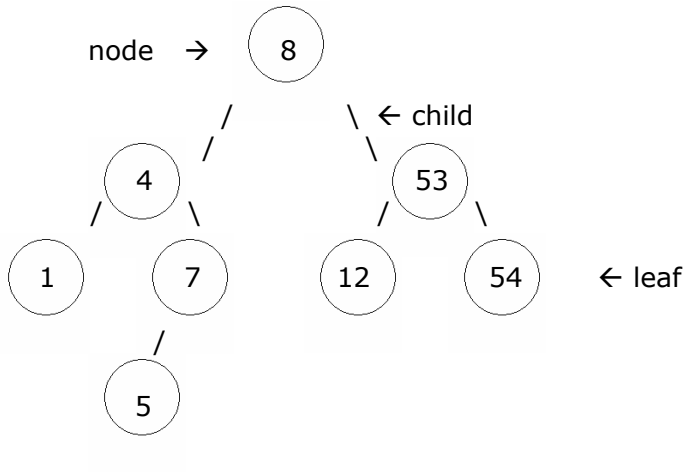
```
(define (lookup word thesaurus)
  (assoc word thesaurus))
```

The lookup written in the homework was a crippled version of assoc that only worked with strings as the key.

Trees

(1 2 3 … 100)
Looking through an unsorted list for some number (like 54) involves analyzing every element.
A sorted list reduces the effort, but there is still high effort.

This is where trees come in. We will be using binary trees (a max of two branches off each node). Every node has a value in it and has a left child and a right child as it's two branches. Each node in a binary tree may have 0 children, 1 child, or 2 children. A node with 0 children is a leaf.



(In class, leaf notation became a child not enclosed in a circle.)

Notice that everything to the left of a node is smaller than it and that everything to the right is larger. If we are looking for 1, we simply look to the left of 8. If 1 is present in the tree, this is where we can expect to find it. This has thrown half the problem size away. Compare it to testing the first element in the list – this only throws away 1 of the problem size.

If there are n nodes in a binary tree, it will take us $\log_2 n$ tries to find the number.
Leaves are at the bottom of the tree.
Node abstraction:
Tree is made up of nodes

List is made up of pairs

A tree is either a leaf or a node, by definition. 12 is a valid tree. The simplest tree is a leaf: no nodes, no decision.

Define the procedures listed under "Trees" on the Lecture 8 handout. Solutions are posted on the site under Lecture 8 Solutions.

For question 3, the idea is to preserve the already-existing leaves and nodes, but insert a new element in its proper place. This involves methodically breaking apart the entire tree and building it back up (with your procedure).

The last portion of the lecture involved the Animal Guessing Game. The instructions are listed under "Animal Guessing Game" on the Lecture 8 handout. (This involves downloading lec8.scm from the server on your computer)

A procedure defined in class:

```
(define (new-list-guesser knowledge)
  (if (null? knowledge)
      (begin (print-msg "I give up")
             nil)
      (if (ask-about-animal (car knowledge))
          (begin (print-msg "Yay!")
                 knowledge)
          (cons (car knowledge) (new-list-guesser (cdr knowledge))))))
```

Discussion suggested replacing nil above with a return of "improve-knowledge", where we would implement a procedure to add an animal to the computer's bank of knowledge. This would improve the game a little.