SICP section 3.3.3

October 04, 2007 at 12:58 Tags SICP

The code for this section is in Common Lisp.

The `assoc` function the authors define exists in Common Lisp, and the one-dimensional table presented here is called an *association list*, or *alist*.

Here's a sample session:

```
[11]> (defvar tbl '((a . 1) (b . 2) (c . 3)))
TBL
[12]> (assoc 'c tbl)
(C . 3)
[13]> (assoc 'd tbl)
NIL
[14]> (assoc 'c tbl :test #'equalp)
(C . 3)
```

As the last line shows, `assoc` accepts a test function as an argument (it uses `eql` by default).

Exercise 3.24

I'll use the idiomatic CL way of keyword arguments to set `same-key?`. Keyword arguments are a very useful feature CL supports out-of-the-box.

```
(defun make-table (&key (same-key? #'eql))
  (let ((local-table (list '*table*)))
    (labels (
      (tassoc (key table)
        (assoc key table :test same-key?))

      (lookup (key-1 key-2)
        (let ((subtable (tassoc key-1 (cdr local-table))))
          (if subtable
            (let ((record (tassoc key-2 (cdr subtable))))
              (if record
                (cdr record)
                nil))
            nil)))

      (insert! (key-1 key-2 val)
        (let ((subtable (tassoc key-1 (cdr local-table))))
          (if subtable
            (let ((record (tassoc key-2 (cdr subtable))))
              (if record
                (setf (cdr record) val)
                (setf (cdr subtable)
                      (cons (cons key-2 val)
                            (cdr subtable)))))
            (setf (cdr local-table)
                  (cons (list key-1
                              (cons key-2 val))
                        (cdr local-table)))))
        'ok)

      (dispatch (m)
        (case m
          ('lookup-proc #'lookup)
          ('insert-proc! #'insert!)
          (otherwise (error "Bad dispatch ~a" m)))))

      #'dispatch)))

(defvar operation-table (make-table :same-key? #'equal))

(defvar get (funcall operation-table 'lookup-proc))
(defvar put (funcall operation-table 'insert-proc!))
```

Here’s sample usage:

```
(funcall put 'x 'y 12)
=> OK
(funcall get 'x 'y)
=> 12
```

Exercise 3.25

One simple and effective way to achieve this is to use a list as the key. You'll also need to pass in an equality predicate that knows how to compare lists, for example `equal`.

Exercise 3.26

I'll use the binary tree representation that was presented for sets in section 2.3.3 of the book. In the solution of exercise 2.66 I've already implemented a lookup, so what's left is to package it all in a "local table" and add the insertion function.

```
; Generic binary tree
;
(defun make-tree (entry left right)
  (list entry left right))

(defun make-leaf (entry)
  (list entry nil nil))

(defun entry (tree)
  (car tree))

(defun set-entry! (tree ent)
  (setf (car tree) ent))

(defun left-branch (tree)
  (cadr tree))

(defun set-left-branch! (tree lb)
  (setf (cadr tree) lb))

(defun right-branch (tree)
  (caddr tree))

(defun set-right-branch! (tree lb)
  (setf (caddr tree) lb))

; Records
;
(defun make-record (key data)
  (list key data))

(defun key (record)
  (car record))

(defun data (record)
  (cadr record))

; Table implemented as a binary tree
;
(defun make-table (&key (<? #'<))
  (let ((local-table (cons '*head* nil)))
    (labels (
      (tree-root ()
        (cdr local-table))
```

```
(set-tree-root! (node)
  (setf (cdr local-table) node))

(node-lookup (key node)
  (if (null node)
    nil
    (let*  ((cur-entry (entry node))
             (cur-key (key cur-entry)))
      (cond ((funcall <? key cur-key)
              (node-lookup
                key
                (left-branch node)))
             ((funcall <? cur-key key)
              (node-lookup
                key
                (right-branch node)))
             (t ; equal
              cur-entry)))))

(lookup (key)
  (node-lookup key (cdr local-table)))

(node-insert (key data node)
  (let*  ((cur-entry (entry node))
           (cur-key (key cur-entry)))
    (cond ((funcall <? key cur-key)
             (if (null (left-branch node))
              (set-left-branch!
                node
                (make-leaf
                  (make-record key data)))
              (node-insert
                key data (left-branch node))))
           ((funcall <? cur-key key)
             (if (null (right-branch node))
              (set-right-branch!
                node
                (make-leaf
                  (make-record key data)))
              (node-insert
                key data (right-branch node))))
           (t ; equal
             (set-entry!
              node (make-record key data))))))

(insert! (key data)
  (if (null (tree-root))
    (set-tree-root!
      (make-leaf (make-record key data)))
    (node-insert key data (tree-root))))
```

```
      (dispatch (m)
        (case m
          ('lookup-proc #'lookup)
          ('insert-proc! #'insert!)
          (otherwise (error "Bad dispatch ~a" m)))))

      #'dispatch)))
```

Note the usage of a generic *less than* operator. This is an accepted technique which is used in, for example, the Standard Template Library of C++. By providing a *less than* operator, we can compare keys and infer their ordering (and even know whether they're equal).

Here's some sample code that demonstrates how this works:

```
(defvar my-t (make-table))
(defvar get (funcall my-t 'lookup-proc))
(defvar put (funcall my-t 'insert-proc!))

(funcall put 5 55)
(funcall put 6 66)
(funcall put 3 33)
(funcall put 9 99)
(funcall put 1 11)
(funcall put 2 22)

(funcall get 9)
=> (9 99)

(funcall get 7)
=> NIL

(funcall put 7 77)

(funcall get 7)
=> (7 77)
```

Exercise 3.27

I won't draw the env diagram, but I will explain what's going on.

When `memoize` is called, and environment with the variable `table` is created, and the function returned points to this environment. Each time the function is executed (each call to `memo-fib`), it is executed within the body of the lambda that `memoize` returns – which checks for the value in the table.

`(memo-fib n)` takes `n` steps to compute its result because it never computes the same result twice. The call tree is flattened into a linear list.

The scheme would not work if we had defined `memo-fib` to be `(memoize fib)`. This is because `fib` still calls itself (`fib`) instead of `memo-fib` in its recursive calls.
