

CS51 Assignment 1: A Library by Its Cover

Due: Friday, 15 February 2008 at 5:00 PM

Total Points: 60 (including 10 style points)

In this assignment you will get practice with Scheme and recursion, and write two different implementations of the same abstract interface: a dictionary.

1 Setup

As will be the case for each subsequent assignment, copy over the requisite files from the `~lib51/pub/assigns/asst1` directory into your own `~/cs51/asst1` directory. If you are having trouble doing this, see assignment 0 for an example.

2 Style

For this and all subsequent assignments, a substantial portion of your grade is based on style. Remember to consult the CS 51 style guide on the course website at <http://www.courses.fas.harvard.edu/~lib51/resources> if you have any doubts about what constitutes clear, well-written code. Even if you don't have specific questions, be sure to review the style guide to refresh your memory!

3 Testing your code

How do we know when a program is correct? That is, how can we ensure that the code we write will do what we intend it to do? We can read it carefully,¹ but that's usually not enough. We can try to prove it correct, but for most programs that's too difficult to do. We probably have to write some tests.

¹Or in some cases, ask someone else to read it carefully—but not in CS51.

In CS51, we provide you scaffolding named, appropriately, `do-tests`. `Do-tests` makes writing tests and reading their output easier. Suppose we want to test `right-triangle?` from the previous problem set. Using `do-tests`, we might write:

```
(do-tests (right-triangle? 3 4 5)           #t
          (right-triangle? 4 5 3)           #t
          (right-triangle? 5 4 3)           #t
          (right-triangle? 0 0 0)           #f
          (right-triangle? 'three 'four 'five) #f
          (right-triangle? -3 -4 -5)         #f)
```

This runs six tests, each of which evaluates an expression on the left (such as `(right-triangle? 3 4 5)`) and compares it to the value of the expression on the right (such as `#t`). If they are `equal?`, the test passes and no output is produced, but if they aren't, an informative error message is displayed.

Good tests exercise every branch of your code. In the example above, note that it tests a number of cases that the function might encounter: three with “normal” input (`3 4 5`, `4 5 3`, and `5 4 3`), a boundary condition (`0 0 0`), and two bad inputs (`'three 'four 'five`, `-3 -4 -5`). In general, your tests should cover every path through your program. This means that you will have to write tests with your specific code in mind. For example, suppose your Scheme function contains this piece of code:

```
(if (qux? obj)
    (handle-qux obj)
    (handle-nonqux obj))
```

A good set of tests for that code will provide cases where `obj` is not a `qux` and cases where it is, to ensure that both branches of the `if` work properly.

Try `do-tests` now. Start Scheme in Emacs and type something like `(do-tests (- 2 1) 4)`. You'll get back an error message:

```
Test failed.
expression:      (- 2 1)
evaluated to:    1
should equal:    4
```

In this and future assignments, we may provide you with a number of tests for the code you write, and often we will require you to add tests of your own.

3.1 New make target

Now that we have `do-tests`, running `make tests` loads your file silently, showing output only if tests fail. If you want to see a trace of your code being run, use `make trace` instead.

4 Exercise those Cons Cells (12 points total)

The answers to these questions, all all other questions on this assignment, should go in the file `asst1.scm`.

Exercise 1. 6 points

Each expression typed into the Scheme *REPL* contains a blank (____). What should go in the blanks to produce the results shown?

(a) [2 points]

```
> (cons 'foo ____)  
(foo)
```

(b) [2 points]

```
> (append ____ '())  
(bar)
```

(c) [2 points]

```
> (car (____ (cdr '(a (b c) d))))  
b
```

Exercise 2. 6 points

(a) [2 points] What will this function return? If it generates an error, explain why.

```
> (cons 7 (list 5 6 7 8))
```

(b) [2 points] What will this function return? If it generates an error, explain why.

```
> (append 7 (list 5 6 7 8))
```

(c) [2 points] Rewrite the second function, still using `append`, so that it returns the same value as the first function.

5 Recursion is not mysterious (11 points total)

Exercise 3. 2 points

What does `mystery` do? Describe it in clear English. For example, suppose you were given this function:

```
(define (example x) (car (cdr x)))
```

We'd favor an answer such as, "this function takes a list as its argument and returns the second element of that list" over "this function gets an argument, x, and returns the car of the cdr of x." The key is to describe the function's behavior holistically.

Now give this mystery function a try.

```
(define (mystery x)
  (cond
    ((null? x) x)
    ((null? (car x)) (mystery (cdr x)))
    ((pair? (car x)) (append (mystery (car x)) (mystery (cdr x))))
    (else (cons (car x) (mystery (cdr x))))))
```

Exercise 4. Restriction enzyme.

9 points

A DNA molecule is composed of matched base pairs. In Scheme, we can represent the bases as symbols `a`, `c`, `g`, and `t`, and a strand of DNA as a list of symbols.

A *restriction enzyme* is a molecule that binds to a particular sequence of bases and then cuts the DNA at that point. Below is a Scheme function (`restrict enzyme dna`), which returns a list of numbered positions where a given `enzyme` would cut a given `dna` fragment. If the `enzyme` matches the front of `dna` that would be position 0, the `cdr` is position 1, and so on. Here's an example:

```
> (restrict '(t g a) '(g t g a g c t g t g a a))
(1 8)
```

In this example, `(t g a)` matches the DNA at positions 1 and 8.

Here is an incomplete implementation of `restrict`:

```
(define (restrict enzyme dna)
  (restrict-position-helper enzyme dna 0))

(define (restrict-position-helper enzyme dna position)
  (cond
    ((null? dna) '())
    ((restrict-mystery-helper? enzyme dna)
     (cons position
            (restrict-position-helper enzyme (cdr dna) (+ 1 position))))
    (else (restrict-position-helper enzyme (cdr dna) (+ 1 position))))
```

- (a) [2 points] What must `restrict-mystery-helper?` do? Describe it.
- (b) [4 points] Write `restrict-mystery-helper?`.
- (c) [3 points] Write tests for `restrict-mystery-helper?`. Your tests should cover each branch in the code.

6 Dictionaries

(26 points total)

A *dictionary*, also known as a finite map or associative array, is a collection of values randomly accessible by a *key*. A standard array (say, in C or Java) is indexed by integers, but a dictionary is usually indexed by other types of values. We'll use dictionaries indexed by symbols. We might, for example, create a dictionary that maps produce to colors:

$$\{\text{banana} \mapsto \text{yellow}, \text{spinach} \mapsto \text{green}, \text{pomegranate} \mapsto \text{red}\}$$

The dictionary above is a mathematical abstraction rather than a particular data structure. Just as we may use a linked list or an array to represent a sequence, there is more than one concrete representation for a dictionary. We define a dictionary in terms of an interface with four operations:

`(dict-new)` returns a new, empty dictionary.

`(dict-extend dict key value)` returns a dictionary like *dict*, but with *key* bound to *value*.

`(dict-lookup dict key)` returns the value that *key* is bound to in *dict*, or `()` if *key* is unbound.

`(dict-keys dict)` returns the keys in *dict* as a list.

We do not have a concrete representation yet, but we already know how the operations should work on an *abstract* representation of dictionaries:

```
(dict-keys {banana ↦ yellow, spinach ↦ green, pomegranate ↦ red})
  ⇒ (banana spinach pomegranate)
(dict-lookup {banana ↦ yellow, spinach ↦ green, pomegranate ↦ red} 'spinach)
  ⇒ green
(dict-extend {spinach ↦ green, pomegranate ↦ red} 'asparagus 'green)
  ⇒ {pomegranate ↦ red, spinach ↦ green, asparagus ↦ green}
(dict-extend {banana ↦ yellow, asparagus ↦ green} 'asparagus 'white)
  ⇒ {banana ↦ yellow, asparagus ↦ white}
```

Exercise 5.

3 points

Based on this specification, we can write tests for dictionaries that are independent of our particular dictionary implementation. Below is a function that takes the four functions in the dictionary interface as arguments and runs several tests using them. ADD THREE TESTS TO THE `do-tests` BELOW.

```
(define (test-dictionary-functions new extend lookup keys)
  (do-tests
    (keys (new))                                     '()
    (keys (extend (new) 'a 'b))                     '(a)
    (lookup (extend (new) 'a 'b) 'a)                'b
```

```
(lookup (extend (extend (new) 'a 'b) 'c 'd) 'a) 'b)
YOUR-ANSWER-HERE)
```

6.1 Dictionaries as association lists

We will now implement dictionaries as association lists. An association list is a list of pairs, each with a key in the `car` and value in the `cdr`. We thus represent the abstract dictionary $\{\text{banana} \mapsto \text{yellow}, \text{spinach} \mapsto \text{green}, \text{pomegranate} \mapsto \text{red}\}$ as

```
((banana . yellow) (spinach . green) (pomegranate . red))
```

Here are implementations of `dict-new` and `dict-extend` for dictionaries-as-lists:

```
(define (dictlist-new) '())

(define (dictlist-extend dl key value)
  (cond
    ((null? dl) (list (cons key value)))
    ((equal? key (caar dl)) (cons (cons key value) (cdr dl)))
    (else
     (cons (car dl) (dictlist-extend (cdr dl) key value)))))
```

Exercise 6.

8 points

- (a) [3 points] Write an association list version of `dict-keys`; call it `dictlist-keys`.
 (b) [5 points] Write `dictlist-lookup`.

Now that we have a complete list-based dictionary implementation, we can test it with `test-dictionary-functions`:

```
(test-dictionary-functions dictlist-new dictlist-extend
                           dictlist-lookup dictlist-keys)
```

You may have noticed that `dict-keys` is difficult to test, because the specification doesn't tell us in what order the list should be. That is, it is correct for

```
(dict-keys {tangerine ↦ orange, eggplant ↦ purple})
```

to return either `(tangerine eggplant)` or `(eggplant tangerine)`. Once you have a particular implementation, however, you know which order to expect.

Exercise 7.

3 points

Write tests for `dictlist-keys`.

6.2 Dictionaries as binary trees

Our second representation for dictionaries is binary trees. You may recall that a binary tree is a linked data structure; we will define a tree to be either the empty tree, represented as `()`, or a node containing an object and left and right subtrees, represented as a list:

```
(define (tree-make-empty) '())
(define (tree-empty? tree) (null? tree))
(define (tree-make-node obj left right) (list obj left right))
(define (tree-obj tree) (car tree))
(define (tree-left tree) (cadr tree))
(define (tree-right tree) (caddr tree))
```

Moving on to dictionaries, then, we need a way to store both the key and the value in the object slot of a tree node. Since binary trees depend on having an ordering for their keys, we'll also need a way to order symbols.

```
(define (dicttree-make-node key value left right)
  (tree-make-node (cons key value) left right))
(define (dicttree-node-key dt) (car (tree-obj dt)))
(define (dicttree-node-value dt) (cdr (tree-obj dt)))
(define (symbol<? a b)
  (string<? (symbol->string a) (symbol->string b)))
```

Here are `dicttree-new` and `dicttree-lookup` for dictionary trees:

```
(define (dicttree-new) (tree-make-empty))

(define (dicttree-lookup dt key)
  (cond
    ((tree-empty? dt) '())
    ((equal? key (dicttree-node-key dt)) (dicttree-node-value dt))
    ((symbol<? key (dicttree-node-key dt))
     (dicttree-lookup (tree-left dt) key))
    (else
     (dicttree-lookup (tree-right dt) key))))
```

Exercise 8.

12 points

(a) [5 points] Write `dicttree-keys`. Your code for manipulating the dictionary tree should be written in terms of the `tree-` and `dicttree-node-` abstraction layer, not `cars` and `cdrs`.

(b) [7 points] Write `dicttree-extend`. (Don't worry about tree balancing.)

Now that we have a tree-based dictionary implementation, we can test it:

```
(test-dictionary-functions dicttree-new dicttree-extend
  dicttree-lookup dicttree-keys)
```

7 The Last Question

(1 point total)

Exercise 9.

1 point

About how long did this assignment take you to complete? How confident are you about your code? Answering the question is worth 1 point; what your answer actually is will not affect your grade provided that you answer it. However, it provides us with feedback that will help us improve this course for future years.

Submit your work with `make submit`.