SICP sections 2.1.1 - 2.1.3

July 25, 2007 at 05:48 Tags SICP

In chapter 1, computational processes and procedures were in the spotlight. Now we'll plunge right into chapter 2 which deals with another fundamental topic of programming – data. How data is represented, processed and abstracted is the name of the game now. So, join me for a discussion of the text and the solution of the exercises of chapter 2. Let's go.

Exercise 2.1

```
(defun make-rat (n d)
  (labels (
    (make-rat-reduce (n d)
      (let ((g (gcd n d)))
        (cons (/ n g) (/ d g)))))

    (cond ((and (< n 0) (< d 0))
            (make-rat-reduce (- n) (- d)))
          ((and (< d 0) (> n 0))
            (make-rat-reduce (- n) (- d)))
          (t
            (make-rat-reduce n d)))))

(defun numer (x)
  (car x))

(defun denom (x)
  (cdr x))
```

Note that the first two cases in the `cond` return the same result, so they can be unified (via an `or` form). However, for the sake of simplicity I prefer it this way – it makes the different cases clearer.

Section 2.1.2

Data abstraction is presented in this section – "Abstraction Barriers". As I've mentioned before in my blog, I firmly believe that abstraction is the most important thing a programmer must understand in order to be any good. Perhaps this applies to other engineering disciplines as well. The problems we face are either not interesting or very complex, and very complex problems must be somehow divided into layers of abstraction, or otherwise we wouldn't be able to create anything. You can't write code for a web application constantly thinking of the voltage levels on transistors' gates inside the CPU. This is taken way too far, of course, but it makes the point. Naturally, you can't even write web applications worrying about the underlying implementation of

the C runtime, atop which most of your OS, web-server and scripting language run.

OK, enough philosophy for now, let's see the solutions of the exercises for this section.

Exercise 2.2

```
(defun make-segment (start end)
  (cons start end))

(defun start-segment (segment)
  (car segment))

(defun end-segment (segment)
  (cdr segment))

(defun make-point (x y)
  (cons x y))

(defun x-point (point)
  (car point))

(defun y-point (point)
  (cdr point))

(defun print-point (point)
  (format t "(~F,~F)~%" (x-point point) (y-point point)))

(defun midpoint-segment (segment)
  (let ((segstart (start-segment segment))
        (segend (end-segment segment)))
    (make-point (average  (x-point segstart)
                          (x-point segend))
                (average  (y-point segstart)
                          (y-point segend)))))

(defvar aa (make-point 4 6))
(defvar bb (make-point 9 15))

(print-point
  (midpoint-segment (make-segment aa bb)))
=> (6.5,10.5)
```

Exercise 2.3

If we define the interface to the rectangle abstraction, we can go on defining the computation of perimeters and areas without actually seeing the code for the rectangle:

```
; Rectangle abstraction:
;
; * make-rect: constructs a rectangle given its two opposing
;   points
; * rect-width: returns the rectangle's width
; * rect-height: returns the rectangle's height
;

(defun rect-perimeter (rect)
  (+  (* 2 (rect-width rect))
      (* 2 (rect-height rect))))

(defun rect-area (rect)
  (*  (rect-width rect)
      (rect-height rect)))
```

And here are two different representations:

```
; Representation 1: stores the two opposing points
;
(defun make-rect (p1 p2)
  (cons p1 p2))

(defun rect-width (rect)
  (abs (- (x-point (car rect))
          (x-point (cdr rect)))))

(defun rect-height (rect)
  (abs (- (y-point (car rect))
          (y-point (cdr rect)))))

; Representation 2: stores the diagonal segment
;
(defun make-rect (p1 p2)
  (make-segment p1 p2))

(defun rect-width (rect)
  (abs (- (x-point (start-segment rect))
          (x-point (end-segment rect)))))

(defun rect-height (rect)
  (abs (- (y-point (start-segment rect))
          (y-point (end-segment rect)))))
```

These two representations are very similar, and we can easily envision others. For example: store a “width segment” and a “height segment”, or store all 4 points, or store one corner stone, width, height and the angle of one of them above the X plane.

Section 2.1.3

This section contains the most beautiful example of code in the book so far. First, the authors state that `cons, car, cdr` can be any three procedures as long as for any `x` and `y`, if `z` is

`(cons x y)`, then `(car z)` is `x` and `(cdr z)` is `y`. So any conceivable implementation of these procedures that satisfies these conditions will do. Consider then, this implementation:

```
(defun my-cons (x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (t (error "Argument not 0 or 1 -- CONS ~S~%" m)))))

(defun my-car (z)
  (funcall z 0))

(defun my-cdr (z)
  (funcall z 1))
```

These will comply to the definition stated above. Functionally, they're undistinguishable from the original `cons, car, cdr`. But where exactly is the data structure here ? Where is the data stored ? It is stored in a *closure* – the anonymous procedure returned by `my-cons`. This procedure is a closure because it captures some information from the external environment (in this case the values of `x` and `y`) at the moment of its definition. Closures and the progamming technique of dispatching (or message passing) will be discussed at length in the book, so we'll meet them a lot.

Exercise 2.4

Here it is even less clear where the data is stored. `my-cons` returns a function that takes a function argument and calls it on its two own arguments. `my-car` provides it with a function that selects the first of its two arguments. Thinking like this, `my-cdr` is very simple:

```
(defun my-cons (x y)
  (lambda (m) (funcall m x y)))

(defun my-car (z)
  (funcall z (lambda (p q) p)))

(defun my-cdr (z)
  (funcall z (lambda (p q) q)))
```

Exercise 2.5

```
(defun divides? (a b)
  (= (rem b a) 0))

(defun my-cons (a b)
  (* (expt 2 a) (expt 3 b)))

(defun my-car (z)
  (do ( (n 0 (1+ n))
        (aa z (/ aa 2)))
      ((not (divides? 2 aa)) n)))

(defun my-cdr (z)
  (do ( (n 0 (1+ n))
        (aa z (/ aa 3)))
      ((not (divides? 3 aa)) n)))
```

After writing code like this, it becomes immediately obvious that there's a repetition we can remove by abstracting it as a function:

```
(defun degree-of-factor (num f)
  "Finds the degree of factor f in number num"
  (do ( (deg 0 (1+ deg))
        (div num (/ div f)))
      ((not (divides? f div)) deg)))

(defun my-car (z)
  (degree-of-factor z 2))

(defun my-cdr (z)
  (degree-of-factor z 3))
```

Remember: **DRY** – **D*on't *R**epeat **Y**ourself – is one of the most fundamental rules in programming.

Exercise 2.6

Church Numerals are part of Lambda Calculus – a fascinating theoretic concept that is worth to learn about. Studying unusual topics like this can really stretch your mind in unexpected directions, and give you ideas on how to deal with practical problems in a better way.

> Church numerals are the representations of natural numbers under Church encoding. The higher-order function that represents natural number n is a function that maps any other function f to its n-fold composition

First let's rewrite in Lisp the definitions given in the book for `zero` and `add-1`:

```
(defvar zero
  (lambda (f)
    (lambda (x) x)))

(defun add-1 (n)
  (lambda (f)
    (lambda (x) (funcall f (funcall (funcall n f) x)))))
```

Now, let's use the substitution rule to see how `one` and `two` are created with applications of `add-1`:

```
; (add-1 zero)
; =>
; (lambda (f)
;   (lambda (x) (funcall f (funcall (funcall zero f) x))))
; =>
; (lambda (f)
;   (lambda (x) (funcall f (funcall (lambda (x) x) x))))
; =>
; (lambda (f)
;   (lambda (x) (funcall f x)))

(defvar one
  (lambda (f)
    (lambda (x) (funcall f x))))

; (add-1 one)
; =>
; (lambda (f)
;   (lambda (x) (funcall f (funcall (funcall one f) x))))
; =>
; (lambda (f)
;   (lambda (x) (funcall f (funcall (lambda (x) (funcall f x)) x))))
; =>
; (lambda (f)
;   (lambda (x) (funcall f (funcall f x))))

(defvar two
  (lambda (f)
    (lambda (x) (funcall f (funcall f x)))))
```

Indeed, this fits the definition quoted above (n-fold composition of `f` on `x`). We can easily think of the implementation of addition in these terms – adding Church numerals a and b is applying the composition a+b times:

```
(defun add (a b)
  (lambda (f)
    (lambda (x) (funcall (funcall a f) (funcall (funcall b f) x)))))
```

For comments, please send me an email.

Back to top