SICP sections 1.3.2 - 1.3.3

July 13, 2007 at 13:30 **Tags** SICP

`lambda` is a very important tool in our quest for meaningful abstractions to make programming easier. Some procedures are inherently simple and one-time, and it's a shame to spend the extra time writing out their full definitions and making up names for them. So anonymous procedures, created with `lambda`, are the answer.

Here is some of the code from the previous sections rewritten with`lambda` instead of explicitly defined auxiliary functions:

```
(defun sum (term a next b)
  (if (> a b)
    0
    (+  (funcall term a)
        (sum term (funcall next a) next b))))

(defun pi-sum (a b)
  (sum  (lambda (x)
          (/ 1.0 (* x (+ x 2))))
        a
        (lambda (x) (+ x 4))
        b))

(defun integral (f a b dx)
  (*  (sum f
          (+ a (/ dx 2.0))
          (lambda (x) (+ x dx))
          b)
      dx))
```

The authors also presented `let` for defining temporary variables. I've been using `let` in some of the code already written for the previous sections, so there is no need to reintroduce it here.

Exercise 1.34

Here is the perverse invocation the authors refer to:

```
(defun f (g)
  (funcall g 2))

(f #'f)
```

Since `f` is passed as an argument to `f`, it will be called with 2 as the argument. But what is `f`, again ? A function that calls its argument as a function on 2. So, the second call will attempt to call 2 as a function (on 2), which is an error. CLISP complains:

```
*** - FUNCALL: 2 is not a function name; try using a symbol instead
```

Section 1.3.3

Here is the CL implementation of the fixed point search and finding the square root of 2 using it:

```
(defvar tolerance 0.00001)

(defun fixed-point (f first-guess)
  (labels (
    (close-enough? (v1 v2)
      (< (abs (- v1 v2)) tolerance))
    (try (guess)
      (let ((next (funcall f guess)))
        (if (close-enough? guess next)
          next
          (try next)))))

    (try first-guess)))

(defun average (a b)
  (/ (+ a b) 2))

(defun dampen-sqrt (x)
  (fixed-point
    (lambda (y)
      (average y (/ x y)))
    1.0))
```

I'm using here the `labels` form of CL instead of an internal `defun`, following a good advice given in the comments to the previous SICP post. In CL, unlike in Scheme, an internal `defun` creates a function with global scope, so it doesn't really do what we want it to do – which is define a localized function that is seen only inside the function definition it appears in. `labels` is the correct way to do this in CL. There are other forms that can achieve this goal, like `flet` – for differences between them turn to the Common Lisp Hyperspec

Note that the authors define `close-enough?` as an explicit function with a name, instead of using a `lambda`. This makes sense in this case, because the name `close-enough?` is meaningful and using the explicit function makes the code more readable. On the other hand, in the definition of `dampen-sort` the authors choose to represent the function computing the average of y and x/y as a `lambda`, because it doesn't harm readability and there is no point in making up a special function name for this singular case. There are no hard rules set in stone about when to use `lambda` and when to use an explicit function. In some cases one is better, in some the other. My rule of thumb in such situations is to go with the intuition – whatever feels more natural and readable in any specific case. This intuition improves with experience.

Exercise 1.35

First let's prove that phi is a fixed point of `x -> 1 + 1/x`. It's quite simple, since:

```
phi^2 = phi + 1   (1)
```

So substituting phi into the transformation:

```
1 + 1/phi =
(phi + 1)/phi =   / Applying (1)
(phi^2)/phi =
phi
```

So, indeed phi is a fixed point of `x -> 1 + 1/x`. Now, the code that computes it:

```
(fixed-point (lambda (x) (1+ (/ 1 x))) 1.0)
=>
1.618
```

Exercise 1.36

Here is the code:

```
(defvar tolerance 0.00001)

(defun fixed-point (f first-guess)
  (labels (
    (close-enough? (v1 v2)
      (< (abs (- v1 v2)) tolerance))
    (try (guess)
      (format t "Trying ~F~%" guess)
      (let ((next (funcall f guess)))
        (if (close-enough? guess next)
          next
          (try next)))))

    (try first-guess)))

(defun average (a b)
  (/ (+ a b) 2))

(defun xx (x)
  (/ (log 1000) (log x)))

(defun dampen-xx (x)
  (average x (xx x)))
```

Running without dampening:

```
(print (fixed-point #'xx 2.0))
=>
Trying 2.0
Trying 9.965784
Trying 3.0044723
Trying 6.279196
Trying 3.7598507
Trying 5.215844
Trying 4.182207
Trying 4.827765
Trying 4.3875937
Trying 4.67125
Trying 4.481404
Trying 4.6053658
Trying 4.523085
Trying 4.5771146
Trying 4.541383
Trying 4.5649033
Trying 4.5493727
Trying 4.5596066
Trying 4.552854
Trying 4.5573053
Trying 4.5543694
Trying 4.5563054
Trying 4.5550284
Trying 4.5558705
Trying 4.555315
Trying 4.555681
Trying 4.55544
Trying 4.5555987
Trying 4.5554943
Trying 4.555563
Trying 4.5555177
Trying 4.5555477
Trying 4.555528
Trying 4.555541

4.5555325
```

34 steps. Now, with dampening:

```
(print (fixed-point #'dampen-xx 2.0))
=>
Trying 2.0
Trying 5.982892
Trying 4.9221687
Trying 4.6282244
Trying 4.5683465
Trying 4.5577307
Trying 4.55591
Trying 4.555599
Trying 4.5555468

4.5555377
```

9 steps. Dampening certainly makes the computation converge much quicker in this case.

Exercise 1.37

Here is the implementation:

```
(defun cont-frac (n d k)
  (labels (
    (frac (i)
      (/  (funcall n i)
          (+  (funcall d i)
              (if (= i k)
                  0
                  (frac (1+ i)))))))

    (frac 1)))
```

This is a recursive process, of course. It takes k = 11 to reach 4-digit accuracy in the computation of 1/phi:

```
(print (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 11))
=>
0.6180556
```

And this is the iterative version:

```
(defun cont-frac-iter (n d k)
  (labels (
    (frac-iter (i result)
      (if (= i 0)
        result
        (frac-iter
          (1- i)
          (/  (funcall n i)
              (+ (funcall d i) result))))))

    (frac-iter k 0)))
```

As usual, it is a bit less elegant and obvious. I wonder if anyone first produces the iterative version for these exercises, and only then the recursive one. Somehow, resursive functions often greatly simplify matters. In the case of `cont-frac`, for instance, the recursive function is almost a transcription of the formula into a program. The iterative version, on the other hand, takes somewhat more thought to understand – because it has to generate the result from the bottom up (note that it counts down while the recursive function counts up).

Exercise 1.38

The only trick here is to realize the rule by which the elements of `Di` are generated, and this isn't too hard if you notice that the non-1s are all 3 elements away from each other and are successive multiples of 2.

```
(print
  (cont-frac
    (lambda (i) 1.0)
    (lambda (i)
      (let ((i+1 (1+ i)))
        (if (= (rem i+1 3) 0)
          (* 2.0 (/ i+1 3))
          1.0)))
    10))
=>
0.7182817
```

Which is `e - 2`!

Exercise 1.39

Again, transforming the fraction into a recursive process is very straightforward:

```
(defun tan-cf (x k)
  (labels ((tan-step (i)
    (/  (if (= i 1)
          x
          (square x))
        (-  (1- (* i 2))
            (if (= i k)
              0
              (tan-step (1+ i)))))))

    (tan-step 1)))
```

For comments, please send me an email.

Back to top