# SICP section 1.3.1

July 11, 2007 at 05:45 Tags SICP

I like that higher-order functions are presented so early in the book. In some sense, it makes the inherent simplicity and power of Lisp apparent. Imagine pointers to functions explained in the first chapter of a book about C ! Since the syntax of Lisp is so uniform, the distinctions between code and data blend, which makes higher-order functions much more natural.

Anyway, this section is a good place to recall the major difference between Scheme and Common Lisp in regards to treating symbols as functions. I referred to this topic in the introduction post of this series, so I'll get right to the code. This is the implementation of `sum`, in CL:

```
(defun cube (x)
  (* x x x))

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

(defun sum-integers (a b)
  (sum #'identity a #'1+ b))

(defun pi-sum (a b)
  (defun pi-term (x)
    (/ 1.0 (* x (+ x 2))))
  (defun pi-next (x)
    (+ x 4))
  (sum #'pi-term a #'pi-next b))

(defun integral (f a b dx)
  (defun add-dx (x)
    (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) #'add-dx b)
    dx))
```

In short – functions live in a separate namespace from variables in CL. Therefore, we can't just call a function by placing it as the leftmost symbol in a form (unlike Scheme). Rather, we must specify explicitly that it is a function call by using the `funcall` function. In addition, we can't just pass a name of a function around like a variable, we must use the special #' tag (which is really syntactic sugar for the `function` form), as the code above shows[1].

### Exercise 1.29

Below is the implementation of Simpson's Rule in terms of `sum`.

```
(defun simpson-integral (f a b n)
  (let ((h (float (/ (- b a) n))))
    (defun simpson-term (k)
      (*  (funcall f (+ a (* k h)))
          (cond ((or (= k 0) (= k n)) 1)
                ((oddp k) 4)
                (t 2))))
    (*  (/ h 3)
        (sum #'simpson-term 0 #'1+ n))))
```

Note that it is different from `integral` in the sense that the `sum` is invoked on the range 0..n and not a..b, because the computation of `simpson-term` must be aware of the k.

Also note the explicit call to `float`. This is because CLISP did the computation in rational numbers instead of floating point numbers, and to compare the results to `integral` I wanted floating point. Here is the comparison:

```
(print 'integral)
(print (integral #'cube 0 1 0.01))
(print (integral #'cube 0 1 0.001))
(print 'simpson-integral)
(print (simpson-integral #'cube 0 1 100))
(print (simpson-integral #'cube 0 1 1000))

==>

INTEGRAL
0.24998708
0.24999388
SIMPSON-INTEGRAL
0.24999997
0.2500002
```

Obviouly, the approximation provided by Simpson's Rule is better – the results converges better with the same amount of iterations.

### Exercise 1.30

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

### Exercise 1.31

It is very simple to design `product` since it's very similar to `sum` (this similarity is the central topic of this and the following two exercises).

```
(defun product (term a next b)
  (if (> a b)
    1
    (*  (funcall term a)
        (product term (funcall next a) next b))))
```

And here we use `product`:

```
(defun factorial (n)
  (product #'identity 1 #'1+ n))

(defun wallis-pi (n)
  (defun wallis-term (k)
    (let ((nom
            (if (evenp k)
              (+ k 2)
              (+ k 1)))
          (denom
            (if (evenp k)
              (+ k 1)
              (+ k 2))))
      (float (/ nom denom))))
  (* 4 (product #'wallis-term 1 #'1+ n)))
```

The second part of the exercise asks to design an iterative process for `product`, which is also very similar to `sum-iter`:

```
(defun product-iter (term a next b)
  (defun iter (a result)
    (if (> a b)
      result
      (iter (funcall next a)
            (* (funcall term a) result))))
  (iter a 1))
```

### Exercise 1.32

Here is the recursive definition of `accumulator` and `sum` defined in terms of it:

```
(defun accumulator (combiner null-value term a next b)
  (if (> a b)
    null-value
    (funcall combiner
      (funcall term a)
      (accumulator combiner null-value term (funcall next a) next b))))

(defun sum (term a next b)
  (accumulator #'+ 0 term a next b))
```

And here is the iterative version and `product` defined in terms of it:

```
(defun accumulator-iter (combiner null-value term a next b)
  (defun iter (a result)
    (if (> a b)
      result
      (iter (funcall next a)
            (funcall combiner (funcall term a) result))))
  (iter a null-value))

(defun product (term a next b)
  (accumulator-iter #'* 1 term a next b))
```

Exercise 1.33

```
(defun filtered-accumulator (combiner null-value term a next b filter)
  (cond ((> a b) null-value)
        ((funcall filter a)
          (funcall combiner
              (funcall term a)
              (filtered-accumulator combiner null-value term (funcall next a) next
        ; call without combining this term
        (t (filtered-accumulator combiner null-value term (funcall next a) next b

(defun sum-squares-of-primes (a b)
  (filtered-accumulator #'+ 0 #'square a #'1+ b #'prime?))

(defun product-of-relatively-prime (n)
  (defun relatively-prime-to-n? (k)
    (= (gcd k n) 1))
  (filtered-accumulator #'* 1 #'identity 1 #'1+ (1- n) #'relatively-prime-to-n?))
```

## Conclusion

I firmly believe that abstraction is one of the (if not*the*) most important concepts to grasp about programming. If you understand what abstractions are and how to use them, you are a better programmer. Curiously, many programmers are unaware of a whole level of abstraction – the higher-order functional abstraction. This is probably because the more popular languages (of until a few years ago) don't allow such abstractions in any simple manner. Imagine writing `filtered-accumulator` in C or Pascal! Lisp, however, supports these abstractions at its very base. Moreover, the uniform syntax of Lisp makes these abstractions look very natural, which allows tutorials on Lisp to introduce them very early in the teaching. Seeing and using such powerful abstractions right from the beginning is definitely very helpful for developing good programming skills. This is probably why people say that merely learning and understanding Lisp can make you a better programmer for life.

## Footnotes

[1] There are several ways to do these things in Common Lisp. For example, the`apply` function can be used in a similar manner to `funcall`. Also, function names can be passed with the simple ' quote instead of the special #' quote. I don't want to delve too much into the details of CL, however, unless there is a very specific need.

For comments, please send me an email.



Back to top