SICP section 1.1

June 21, 2007 at 05:50 **Tags** SICP

This is the first real post in the "SICP reading" series. It refers to section 1.1 of the book, discussing interesting points and solving selected exercises (I'm trying to fully solve all the non-trivial ones).

Sections 1.1.1 – 1.1.6

This section presents the very basics of Lisp (the Scheme dialect, to be exact). I think that the choice of Lisp for a teaching language can be clearly justified by seeing how short and simple the section is. The basic rules and forms presented here are enough to write very complex Lisp programs.

Substitution model for procedure application

In order to understand what's going under the hood when a Lisp procedure is called, the authors present a simplified model:

> To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.

This model will work for the first few chapters of the book but later will break done when closures and environments are presented. For now, however, it is a useful tool for mechanical understanding of how Lisp evaluates expressions.

Exercise 1.4

This is an interesting example of how Common Lisp is different from Scheme. I want to dwell on it here for a moment. The function presented is (Scheme)

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

The `if` returns a function, either + or -, according to its argument. The function is then applied to `a` and `b`.

To rewrite it in Common Lisp, we must remember that unlike Scheme, Common Lisp has a separate namespace for functions. Therefore the `if` must return something CL will recognize as a function, and which will then be *explicitly* applied to the arguments

```
(defun a-plus-abs-b (a b)
  (funcall (if (> b 0) #'+ #'-) a b))
```

The different name and syntax of `defun` as opposed to `define` aside, see the usage of `#'` as function specifiers (meaning "the following symbol is a function") and `funcall` as an explicit application of the functions.

By the way, `#'` is a shorthand for the `function` special form, so the above can be rewritten in CL as

```
(defun a-plus-abs-b (a b)
  (funcall (if (> b 0) (function +) (function -)) a b))
```

Exercise 1.5

The authors presented normal-order evaluation vs. applicative-order evaluation in section 1.1.5. Applicative-order is what Scheme / CL actually use, while normal-order evaluation is a useful tool which is sometimes coded explicitly (and the authors promise to show how in chapters 3 and 4).

Anyway, the code presented for this exercise is (Scheme):

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))

(test 0 (p))
```

Obviously, the definition of `p` is invalid – calling it calls `p` again, and so on recursively ad infinitum (a good interpreter will probably catch it as a stack overflow, or infinite recursion).

So, when the call to `test` is evaluated in applicative order, its arguments are evaluated first. Hence, `(p)` is evaluated, and the program fails with an error. However, with normal order evaluation, `(p)` is not evaluated until it's actually needed, which never happens, because the comparison of x with 0 suceeds and the call just returns 0.

Section 1.1.7

A very nice quote about the difference between mathematical functions and programming procedures:

> The contrast between function and procedure is a reflection of the general distinction between describing properties of things and describing how to do things, or, as it is sometimes referred to, the distinction between declarative knowledge and imperative knowledge. In mathematics we are usually concerned with declarative (what is) descriptions, whereas in computer science we are usually concerned with imperative (how to) descriptions.

Exercise 1.6

The normal `if` form is special – it evaluates its `then-clause` part only if the predicate is true, and its `else-clause` part only if the predicate is false. The `new-if` function proposed here evaluates both parts, always (applicative-order, remember?). This is easily demonstrated:

```
(defun new-if (predicate then-clause else-clause)
  (cond (predicate then-clause)
        (t else-clause)))

(print "new-if:")
(new-if (= 2 3) (print "yes") (print "no"))

(print "if:")
(if (= 2 3) (print "yes") (print "no"))
```

So when used in `sqrt-iter`, `new-if` will cause an infinite loop by repeatedly calling itself, even when the guess is good enough.

Exercise 1.7

There indeed is a problem with very large and very small numbers in the current implementation. For example, `(mysqrt 92400000000000000)` fails with a "program stack overflow" message, while `(mysqrt 0.0005)` gives a bad result – 0.036, instead of the expected 0.022 (Note that I changed the function name to `mysqrt` in order to avoid a clash with the `sqrt` built into CL.

The problem with very small numbers is easy to understand. Since we have an explicit epsilon value in `guess` – 0.001, the function will give incorrect results for inputs that approach the epsilon. Consider the 0.005 I used in my example – the square of 0.022 is 0.00484, which is within 0.001 of 0.005, while 0.022 is 50% smaller than the correct answer 0.036;

The problem with large numbers is a little more tricky. The square root computations must be performed in floating point arithmetic, which has limited precision in CL systems (on the other hand, integer arithmetic is arbitrarily large). Floating point numbers are usually represented with an exponent and a mantissa, and when the exponent grows large, the "quantas" the number can advance in become large. In our example, after a few iterations the number `3.039737E8` was reached and from there the process entered infinite recursion because `(improve 3.039737E8)` returns `3.039737E8` itself.

To solve the problem, we'll do as suggested by the authors, and implement `good-enough?` as a test of how `guess` changes from one iteration to the next and stop when the change is a very small fraction of the guess. Here is the improved code:

```
(defun  sqrt-iter (guess x)
  (let ((improved-guess (improve guess x)))
    (if (close-enough? guess improved-guess)
      improved-guess
      (sqrt-iter improved-guess x))))

(defun improve (guess x)
  (average guess (/ x guess)))

(defun average (x y)
  (/ (+ x y) 2))

(defun close-enough? (a b)
  (let ((ratio (/ a b)))
    (and (< ratio 1.001) (> ratio 0.999))))

(defun square (x)
  (* x x))

(defun mysqrt (x)
  (sqrt-iter 1.0 x))
```

Note: I implemented `square` explicitly because it's not built into CL. Also, I've used the `let` form in this code although it wasn't taught in the book yet.

The `good-enough?` function was replaced by `close-enough?` that is used to judge whether two consecutive guesses are close enough to each other. The real improvement here is that this judgement does not rely on some fixed number like 0.001 but rather on a quantity relative to the guess itself. As expected, this new method provides good results for the whole range of numbers, including very small and very large.

Exercise 1.8

Naturally, the code is similar to exercise 1.7, except for the `improve` function which follows the formula given in the book:

```
(defun improve (guess x)
  (/ (+ (/ x (square guess))
        (* 2 guess))
     3)))
```

For comments, please send me an email.

Back to top