SICP sections 4.1.1 - 4.1.2

December 06, 2007 at 23:23 Tags SICP

The code for this section is in Common Lisp. However, the language I'm going to implement the evaluator for is Scheme. This is for two reasons. First, this is the language being implemented in the book, and I don't want to deviate from it too much. Second, the semantics of Scheme are somewhat simpler than those of CL, so for a first Lisp evaluator Scheme will do.

Note that I will first reimplement the authors' evaluator wholly in Common Lisp, in order to be able to test the solutions to the exercises. This includes rewriting code that is presented in later sections. Since the full code of the evaluator is quite big, I won't paste it wholly here but if you're interested, you can download it as a evaluator.lisp.

I also wrote a set of unit tests for the evaluator - evaluator_testing.lisp, using the lisp-unit unit testing package. Unit testing is an important part of developing correct software, and evaluators are one of the areas where unit testing is especially applicable, since their inputs and outputs can be easily tested, and a set of unit tests can exercise the code in a very good fashion. This unit testing suite will prove itself helpful when I'll want to test that the various changed to the evaluator made in the exercises haven't harmed its correctness.

Once I have this code loaded, I can play with the evaluator – and this is actually quite some fun, since I now know exactly what goes on under the hood. For example:

```
(interpret '(define xx 14))
(interpret
  '(define (average x y)
      (/ (+ x y) 2)))
(print (interpret '(average xx 66)))
=> 40
```

Another example, using a higher-order function:

```
(interpret
  '(define (make-adder-func x)
    (lambda (y) (+ x y))))

(interpret
  '(define add2 (make-adder-func 2)))

(print (interpret '(add2 6)))
=>
8
```

I'm using the helper function `interpret` which is implemented as follows:

```
(defun interpret (exp)
  (eval. exp the-global-environment))
```

Note the period in the name `eval.` – it is there to avoid a clash between the built-in `eval` and my own `eval`. A period was also, appended to `apply`, for the same reason.

Exercise 4.1

I can use the `let*` form to impose an order[1].

```
(defun list-of-values (exps env)
  (if (no-operands? exps)
    '()
    (let*  ((left (eval. (first-operand exps) env))
            (right (list-of-values (rest-operands exps) env)))
      (cons left right))))
```

The same trick can be used to evaluate from right to left.

Exercise 4.2

a. The evaluator accepts any list as a procedure application, so its clause must be last – after all the special cases were tested. Therefore Louis's plan is flawed – the evaluator will recognize definitions and assignments as procedure applications. For example, for the expression `(define x 3)` the evaluator will attempt to call the procedure `define` on arguments `x` and `3`, which is not what was intended.

You might wonder whether it's possible to just make `define` a procedure. It isn't, because a procedure application evaluates its arguments[2], and we wouldn't want to evaluate the `x` in the `define` shown above, because it is obviously undefined before the `define` is executed. Whenever we want to impose special evaluation rules on the arguments, we must introduce a special form.

b. We only need to change what the evaluator considers to be an applications. Since the evaluator was built in a very modular style that permits changing the underlying implementations of constructs easily, all we really have to do is rewrite:

```
(defun application? (exp)
  (tagged-list? exp 'call))
(defun operator (exp) (cadr exp))
(defun operands (exp) (cddr exp))
```

Exercise 4.3

Back in chapter 2, I implemented the data-directed dispatch code in Scheme. Now, as I'm coding the evaluator in Common Lisp, I will need the support tools for the operation table. Fortunately, using a hash table in CL is as simple as in Scheme:

```
(defvar *op-table* (make-hash-table :test #'equal))

(defun get-op (type)
  (gethash type *op-table*))

(defun put-op (type proc)
  (setf (gethash type *op-table*) proc))
```

If you carefully look at `eval` in the book (or at `eval.` in my CL code), you will notice that except a couple of special cases which can't be dispatched (like recognizing a variable, an application and a self-evaluating expression), all the other operations are handled similarly. First, the expression is recognized with a specialized test that checks if it's a tagged list of some sort, and then the appropriate evaluation procedure is called and handled the expression and environment as arguments.

To make all the operations data-dispatchable, we'll add a few more specialized evaluation functions:

```
(defun eval-lambda (exp env)
  (make-procedure
    (lambda-parameteres exp)
    (lambda-body exp)
    env))

(defun eval-begin (exp env)
  (eval-sequence (begin-actions exp) env))

(defun eval-cond (exp env)
  (eval. (cond->if exp) env))

(defun eval-quoted (exp env)
  (text-of-quotation exp))
```

Now we can insert all operations into the dispatch table:

```
(put-op 'quote #'eval-quoted)
(put-op 'set! #'eval-assignment)
(put-op 'define #'eval-definition)
(put-op 'if #'eval-if)
(put-op 'lambda #'eval-lambda)
(put-op 'begin #'eval-begin)
(put-op 'cond #'eval-cond)
```

And finally, implement `eval.` in a data-dispatch fashion:

```
(defun eval. (exp env)
  (cond ((self-evaluating? exp)
          exp)
        ((variable? exp)
          (lookup-variable-value exp env))
        ((get-op (car exp))
          (funcall (get-op (car exp)) exp env))
        ((application? exp)
          (apply.
            (eval. (operator exp) env)
            (list-of-values (operands exp) env)))
        (t
          (error "Unknown expression in EVAL: " exp))))
```

Having the unit test suite at hand proves very useful here, as I can quickly use it to verify that this code works correctly.

Note that had we used Louis's idea of identifying applications with `call`, as was implemented in exercise 4.2b, we could dispatch to an application as well, instead of keeping it as a "special" operation in `eval.`.

Exercise 4.4

Here are the functions to recognize and handle `and` and `or` as special forms:

```
; (or <exp1> ... <expN>)
(defun or? (exp) (tagged-list? exp 'or))

(defun eval-or (exp env)
  (dolist (e (cdr exp) nil)
    (let ((result (eval. e env)))
      (when (true? result)
        (return result)))))

; (and <exp1> ... <expN>)
(defun and? (exp) (tagged-list? exp 'and))

(defun eval-and (exp env)
  (dolist (e (cdr exp) (car (last exp)))
    (let ((result (eval. e env)))
      (when (false? result)
        (return nil)))))
```

And here is their part in `eval.` (using the original, not data-dispatch approach):

```
...
((or? exp)
  (eval-or exp env))
((and? exp)
  (eval-and exp env))
...
```

Alternatively, they can be implemented as derived expressions in terms of nested`if` expressions, similarly to the implementation of `cond`.

Exercise 4.5

The `expand-cond-clauses` function has to be modified thus:

```
(defun expand-cond-clauses (clauses)
  (if (null clauses)
    'false              ; no _else_ clause
    (let ((first (car clauses))
          (rest (cdr clauses)))
      (cond
        ((cond-else-clause? first)
          (if (null rest)
            (sequence->exp (cond-actions first))
            (error "ELSE clause isn't last " clauses)))
        ((extended-cond-syntax? first)
          (make-if
            (extended-cond-test first)
            (list
              (extended-cond-recipient first)
              (extended-cond-test first))
            (expand-cond-clauses rest)))
        (t
          (make-if
            (cond-predicate first)
            (sequence->exp (cond-actions first))
            (expand-cond-clauses rest)))))))
```

With some helper functions:

```
(defun extended-cond-syntax? (clause)
  (and
    (listp clause)
    (> (length clause) 2)
    (eq (cadr clause) '=>)))

(defun extended-cond-test (clause)
  (car clause))

(defun extended-cond-recipient (clause)
  (caddr clause))
```

Note the call to `make-if` in case of an extended syntax, it is an interesting case. What we want is to*create code* that will be run by `eval.`, so we simply create a list, which represents the procedure application. In some sense, writing such code is reminiscent of writing macros with CL's `defmacro` facility – because this is code that is being executed during the "compilation" phase.

Exercise 4.6

Here's the addition into `eval.`:

```
  ((let? exp)
    (eval. (let->combination exp) env))
```

And the corresponding utilities:

```
(defun let? (exp) (tagged-list? exp 'let))
(defun let-body (exp) (cddr exp))
(defun let-vars (exp)
  (mapcar #'car (cadr exp)))
(defun let-inits (exp)
  (mapcar #'cadr (cadr exp)))

(defun let->combination (exp)
  (cons
    (make-lambda (let-vars exp) (list (let-body exp)))
    (let-inits exp)))
```

Exercise 4.7

The example given in the book:

```
(let* ((x 3)
       (y (+ x 2))
       (z (+ x y 5)))
  (* x z))
```

Is equivalent to:

```
(let ((x 3))
  (let ((y (+ x 2)))
    (let ((z (+ x y 5)))
      (* x z))))
```

With this in mind, we can write the expansion code:

```
(defun let*->nested-lets (exp)
  (labels (
      (make-rec-let (initforms body)
        (if (null initforms)
          body
          (make-let
            (list (car initforms))
            (make-rec-let (cdr initforms) body)))))
    (make-rec-let (cadr exp) (caddr exp))))
```

I added `make-let` to encapsulate `let`'s creation:

```
(defun make-let (initforms body)
  (list 'let initforms body))
```

It is sufficient to add this to `eval.` to make `let*` work in the interpreter:

```
  ((let*? exp)
    (eval. (let*->nested-lets exp) env))
```

Exercise 4.8

Here's the modified `let->combination`:

```
(defun let->combination (exp)
  (if (named-let? exp)
    (sequence->exp
      (list
        (list
          'define
          (cons (named-let-name exp) (named-let-vars exp))
          (named-let-body exp))
        (cons
          (cadr exp)
          (named-let-inits exp))))
    (cons
      (make-lambda (let-vars exp) (list (let-body exp)))
      (let-inits exp))))
```

And these are the utility functions it uses:

```
(defun named-let? (exp)
  (atom (cadr exp)))
(defun named-let-name (exp) (cadr exp))
(defun named-let-vars (exp) (let-vars (cdr exp)))
(defun named-let-inits (exp) (let-inits (cdr exp)))
(defun named-let-body (exp) (let-body (cdr exp)))
```

Exercise 4.9

Here, for example, is the `while` iteration construct:

```
; (while <condition> <body>)
(defun while? (exp) (tagged-list? exp 'while))
(defun while-condition (exp) (cadr exp))
(defun while-body (exp) (caddr exp))

(defun while->combination (exp)
  (sequence->exp
    (list
      (list
        'define
        (list 'while-iter)
        (sequence->exp
          (list
            (while-body exp)
            (make-if
              (while-condition exp)
              (list 'while-iter)
              'true))))
      (list 'while-iter))))
```

With this addition to `eval.`:

```
((while? exp)
  (eval. (while->combination exp) env))
```

As I noted before, the similarity of this code to writing CL macros with `defmacro` is great. Using `defmacro` we can literally extend the CL compiler with our own constructs. Just as with macros there is the *expander* code that's evaluated at compile-time and *expanded* code that's evaluated at run-time, in our Scheme interpreter there's the code written in CL as part of the interpreter which is evaluated by CL, and the code in Scheme which is evaluated by `eval.`.

Note that untrivial pieces of code like `while->combination` are littered with calls to `list`, which is essential to build up expressions. When writing CL macros, the convenient back-quote ` and comma , provide facilities which allows one to avoid almost all calls to `list` when building code.

[1] Note that I could use a nested `let` for the same purpose. The outer `let` would be for the side to be evaluated first, and the inner `let` for the side to be evaluated second.

[2] In *applicative order* evaluation, which is what Lisp uses.

For comments, please send me an email.

Back to top