SICP section 2.4

September 09, 2007 at 05:44 **Tags** SICP

The code for this section is Scheme. Also, note that I skipped a couple of sub-sections in 2.3 - they're quite code-heavy and I hope to have them done soon.

Section 2.4 has a whole lot of text and not many exercises. It is an important and interesting read on the topic of multiple representations for data and generic operations. However, I feel that these topics are less in the domain of Lisp, since they are much more familiar from the world of C++, C#, Java and Ruby – object oriented languages.

In fact, Lisp has object orientation as well. In Common Lisp there's CLOS, and in Scheme there are a few OO packages flying around. While in languages like C++ object-orientation is a pre-designed feature, in Lisp it's just built in on top of the basic language as any other DSL. This is possible because the meta-programming Lisp allows with its closures, code-as-data and macros are a much more fundamental and powerful concept than OO, and so can be used to implement it[1].

Anyway, on to some code.

Exercise 2.73

a. Similarly to the examples of complex numbers, the code was turned into dispatching on the operator. `get` returns the appropriate functions for the operator it is given, and that function is called on the operands with the variable to do the actual derivation.

`number?` and `same-variable?` weren't assimilated into the data-directed dispatch because there's nothing to dispatch on for them. While the operator is part of the data (hence "data-directed" dispatch), these two are just predicates that examine properties of the data, and can't be dispatched on.

b. Although the authors don't explicitly ask to implement the `put` and `get` operations (since assignment isn't taught until the next chapter), I will do it since I want to test that my solution works:

```
;; Operation, type -> procedure
;; Dispatch table.
;;
(define *op-table* (make-hash-table 'equal))

(define (put op type proc)
  (hash-table-put! *op-table* (list op type) proc))

(define (get op type)
  (hash-table-get *op-table* (list op type) '()))
```

The table is implemented using Scheme's hash table facility. Note that I use lists as keys (*tuples* of operation and type) and so `equal` must be used as the comparison operator[2].

Now, the *deriv* package:

```
(define (install-deriv-package)
  ;; internal procedures
  (define (make-sum a1 a2) (list '+ a1 a2))
  (define (addend s) (car s))
  (define (augend s) (cadr s))

  (define (make-product m1 m2) (list '* m1 m2))
  (define (multiplier p) (car p))
  (define (multiplicand p) (cadr p))

  (define (deriv-sum exp var)
    (make-sum (deriv (addend exp) var)
              (deriv (augend exp) var)))

  (define (deriv-product exp var)
    (make-sum
      (make-product (multiplier exp)
                    (deriv (multiplicand  exp) var))
      (make-product (deriv (multiplier  exp) var)
                    (multiplicand exp))))

  ;; interface to the rest of the system
  (put 'deriv '+ deriv-sum)
  (put 'deriv '* deriv-product))
```

A small thing to note: the definitions of accessors `augend`, `multiplier`, etc.) are different from the originals – because now `deriv` only passes the expression without the operator into `deriv-sum` and `deriv-product`.

c. With the infrastructure already in place, this is easy. Just adding the following forms into `install-deriv-package` does the job, without the need to change `deriv` itself:

```
  (define (make-exponentation base exp)
    (list '** base exp))
  (define (base s) (car s))
  (define (exponent s) (cadr s))

  (define (deriv-exponentation exp var)
    (make-product
      (make-product
        (exponent exp)
        (make-exponentation
          (base exp)
          (- (exponent exp) 1)))
      (deriv (base exp) var)))

  (put 'deriv '** deriv-exponentation)
```

d. Since this works on the abstraction level of `get` and `put`, the only change required is switching the order of arguments given to `put` in `install-deriv-package`. The implementation of `put` and `get` is indifferent to the order of the arguments, as longs as it is consistent on the calling level.

Exercise 2.74

The requirements in this exercise are somewhat abstract and aren't easily testable without building the whole system, so the code I'm writing here wasn't really tested. The vast majority of exercises in SICP are very good. This exercise is one of the exceptions to that rule.

a. First, let's define a *generic file* as a tagged file type that contains the division it belongs to:

```
(define (make-generic-file division file)
  (list division file))

(define (division-of-generic-file gf)
  (car gf))

(define (file-of-generic-file gf)
  (cadr gf))
```

`make-generic-file` takes a division name and a file object and builds a generic file object. The other two functions are simple accessors.

With this in hand, we can write `get-record` as follows:

```
(define (get-record employee file)
  ((get 'get-record
        (division-of-generic-file file))
    employee
    (file-of-generic-file file)))
```

It applies the generic operation `get-record` to the employee and the file. It does this by first fetching the appropriate `get-record` for the division with which the file is tagged, and then calling it on the employee and the actual file.

b. Similarly to `get-record`, we can assume we have accessors for a generic employee object and

write:

```
(define (get-salary employee)
  ((get 'get-salary
        (division-of-generic-employee employee))
    (employee-of-generic-employee employee)))
```

c. In Common Lisp I'd use one of the plethora of `find` functions. In Scheme, they're not part of the standard but are commonly available as extensions. Here is the implementation, using `findf` from PLT Scheme's library of list utilities.

```
(require (lib "list.ss"))

(define (find-employee-record employee file-list)
  (findf
    (lambda (f)
      (let ((r (get-record employee f)))
        (if (null? r)
          #f
          r)))
    file-list))
```

Note the translation from `'()` to `#f`, since Scheme considers the empty list to be true in boolean expressions.

d. It needs to add the functions that know how to extract employee records from its own file format, and install them in the dispatch table with `put`.

Exercise 2.75

```
(define (make-from-mag-ang mag ang)
  (define (dispatch op)
    (cond
      ((eq? op 'real-part)
        (* mag (cos ang)))
      ((eq? op 'imag-part)
        (* mag (sin ang)))
      ((eq? op 'magnitude) mag)
      ((eq? op 'angle) ang)
      (else
        (error "Unknown op -- MAKE-FROM-MAG-ANG" op))))
  dispatch)
```

Exercise 2.76

- With explicit dispatch, every function will have to be changed when a new type is added. This isn't very good, of course.
- With data directed style, we'll have to install another package (like we did with `deriv` in Exercise 2.74), but from there the upper level calls don't have to change.
- With message passing, similarly to data directed style, we'll only have to add another type of "object" that accepts the same messages. The upper level calling code doesn't even have to

be aware of the fact a new type has been added.

[1] Contrast it with the opposite – OO in C++ can't be used to implement the special features of Lisp (without writing a full Lisp interpreter, of course).

[2] You can read about the Scheme equivalence operatorshere

For comments, please send me an email.

Back to top