SICP section 2.5.1

September 14, 2007 at 09:38 Tags SICP

The code for this section is in Scheme.

Exercise 2.77

When represented in Scheme notation, the expression in figure 2.24 is:

```
(cons 'complex (cons 'rectangular (cons 3 4)))
=>
(complex rectangular 3 . 4)
```

When Louis tries to evaluate `(magnitude z)`, what is actually called is: `(apply-generic 'magnitude z)`. `apply-generic` looks in the table of operations trying to find the operation `magnitude` for `z`.

Here's `apply-generic` again:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types -- APPLY-GENERIC"
            (list op type-tags))))))
```

It examines the tag of `z` and finds that it's `(complex)` (the `type-tag` function just takes the `car` as the type tag). However, no `magnitude` operation is registered for this type, hence the error. The operations Alyssa proposes:

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

Place `magnitude` and other selectors for type `(complex)` into the table – which makes it work as expected.

`apply-generic` is invoked twice in the process – first dispatching to the`magnitude` of `(complex)` (which is what calls `apply-generic` the second time) and second to the `magnitude` of `(rectangular)`.

Exercise 2.78

```
(define (attach-tag type-tag contents)
  (if (number? contents)
    contents
    (cons type-tag contents)))

(define (type-tag datum)
  (cond ((number? datum) 'scheme-number)
        ((pair? datum) (car datum))
        (else
          (error "Bad tagged datum -- TYPE-TAG" datum))))

(define (contents datum)
  (cond ((number? datum) datum)
        ((pair? datum) (cdr datum))
        (else
          (error "Bad tagged datum -- TYPE-TAG" datum))))
```

This code demonstrates one of the sad facts of life in programming –*you often have to break your abstraction barriers for the sake of efficiency*. All languages do it, and Lisp / Scheme is no different. It's a pity that having built such a pretty, orthogonal arithmetic package, we now have to brutally slice into the generic functions that make it possible and add special cases for special values.

Now we can write simpler code for working with numbers:

```
(print (add 5 19))
=>
24
```

Exercise 2.79

```
(define (equ? x y) (apply-generic 'equ? x y))

;; Add into install-scheme-number-package
  (put 'equ? '(scheme-number scheme-number)
    (lambda (x y) (= x y)))

;; Add into install-rational-package
  (define (equal-rat? x y)
    (and
      (= (numer x) (numer y))
      (= (denom x) (denom y))))

  (put 'equ? '(rational rational)
    (lambda (x y) (equal-rat? x y)))

;; Add into install-complex-package
  (define (equal-complex? z1 z2)
    (and
      (= (real-part z1) (real-part z2))
      (= (imag-part z1) (imag-part z2))))

  (put 'equ? '(complex complex)
      (lambda (z1 z2) (equal-complex? z1 z2)))
```

Note that the results of the functions placed into the table for `equ?` are not tagged. This is because the result of the equality operator is boolean.

Exercise 2.80

```
(define (=zero? x) (apply-generic '=zero? x))

;; Add into install-scheme-number-package
  (put '=zero? '(scheme-number)
    (lambda (x) (= x 0)))

;; Add into install-rational-package
  (put '=zero? '(rational)
    (lambda (x) (= (numer x) 0)))

;; Add into install-complex-package
  (put '=zero? '(complex)
    (lambda (z)
      (and  (= (real-part z) 0)
            (= (imag-part z) 0))))
```

For comments, please send me an email.

top