Topic 14
Multiple Representations of Abstract Data – Data Directed Programming and Additivity

Section 2.4.3

Data-Directed Programming

Consider

```
; takes a tagged complex number and returns
; the real part
(define (real-part z)
  (cond ((rectangular? z)
          (real-part-rectangular (contents z)))
         ((polar? z)
          (real-part-polar (contents z)))
         (else
           (error
              "data type unknown to REAL-PART"
              z))))
```

Problems with that code

- Each interface procedure needs to know about all representations possible
- As more representation types are added, the procedure has to be rewritten
- Procedure gets longer
- Procedure gets more complicated
- Procedure is one big conditional statement

- Each representation alternative must be coded with no name conflicts with any other representation alternative.

The data-directed programming technique

- Replaces the conditional statement with a lookup table
- The chunks of code that were in the branches of the conditional are now indexed by the two dimensions of the lookup table (procedure X representation type)

Lookup table for complex numbers

Operations	Representation types	
	polar	rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

(named procedures or lambda expressions)

A prototype implementat of lookup table

- Basic operations are put and get
- Book's implementation isn't until the next chapter
- Our implementation is simple, suitable for rapid prototyping, but not the most efficient
- Can be replaced later by the book's better implementation without changing code built on top of it

Basic idea of our implementation

- Lookup table is just a list of triples of the form (operation type action)
- Simulates a sparse array of infinite size
- Put adds a triple to the list
- Get searches the list for the right triple and returns the action part
- The type is a list of the representation types of the operation's arguments

And now for the implementation

```
; operation-table will hold the triples of
; operations, their type, and the associated action
(define operation-table empty)

; takes an operation, the type of data it acts on, and an
; action that implements the operation on the type given.
; Adds the triple to the operation table
(define (put operation type action)
 (set! operation-table
      (cons (list operation type action)
           operation-table)))
```

(Forget you saw set!, the reassignment operator, until the next chapter.)

The get procedure

```
; takes an operator and a type, and returns the
; action that implements the operator for that type
; in the operation-table
(define (get operator type)
 (define (get-aux list)
  (cond ((null? list) #f)
        ((and (equal? operator (caar list))
         (equal? type (cadar list)))
         (caddar list))
        (else (get-aux (cdr list)))))
 (get-aux operation-table))
```

So? What do we do with it?

- Define a collection of procedures or a package for each representation
- These are installed in the table
- Complex-arithmetic selectors access the table by means of a general "operation" procedure called apply-generic

Each representation type in its own package

```
; define the selectors in the rectangular representation
;  each selector is defined as it was originally – and
;  must put installed into the operation table
(define (install-rectangular-package)
 (put 'real-part '(rectangular) car)
 (put 'imag-part '(rectangular) cdr)
 (put 'magnitude
      '(rectangular)
      (lambda (z)
        (sqrt (+ (square (car z))
                 (square (cdr z))))))
```

Rectangular package continued

```
 (put 'angle
      '(rectangular)
      (lambda (z) (atan (cdr z) (car z))))
 (put 'make-from-real-imag
      'rectangular
      (lambda (x y)
        (attach-tag 'rectangular
                    (cons x y))))
```

part 3

```
(put 'make-from-mag-ang
      'rectangular
      (lambda (r a)
        (attach-tag 'rectangular
                    (cons (* r (cos a))
                          (* r (sin a))))))
'done)
```

Polar package

```
; define the selectors in the polar representation
; each selector is defined as it was originally --
; and must put installed into the operation
(define (install-polar-package)
  (put 'magnitude '(polar) car)
  (put 'angle '(polar) cdr)
  (put 'real-part
       '(polar)
       (lambda (z)
         (* (car z) (cos (cdr z)))))
```

Polar package continued

```
(put 'imag-part
   '(polar)
   (lambda (z)
     (* (car z) (sin (cdr z)))))
(put 'make-from-mag-ang
   'polar
   (lambda (r a)
     (attach-tag 'polar (cons r a))))
```

Part 3

```
(put 'make-from-real-imag
     'polar
     (lambda (x y)
       (attach-tag
        'polar
        (cons (sqrt (+ (square x)
                       (square y)))
              (atan y x)))))
 'done)
```

The constructors

```
;; THE CONSTRUCTORS
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular)
   x
   y))

(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

Generic operation call

```
;;; general operation will implement selector
;; functions for complex numbers
(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
            "APPLY-GENERIC failed"
            (list op type-tags))))))
```

(note: operations can have more than one argument)

The selectors

```
; selectors are implemented in terms of the
; generic operation
(define (real-part z)
  (apply-generic 'real-part z))
(define (imag-part z)
  (apply-generic 'imag-part z))
(define (magnitude z)
  (apply-generic 'magnitude z))
(define (angle z)
  (apply-generic 'angle z))
```

Three programming styles

1) Data-directed programming, where code for each operator and representation type combination is stored in a 2-dimensional operation table

2) Conventional-style programming, where each operator decides what code to run by testing the representation types of its arguments

In effect, each operator has one row of operation table built into it.

Continued

3) Message passing, where each data object decides what code to run by testing the name of the operation it is being asked to perform

In effect, each data object has one column of the operation table built into it.

Instead of intelligent *operations* that know what kind of data they work on, each *data object* is intelligent and can dispatch its own operations.

A message passing version

```
; message passing version takes a real and imaginary
; part of a complex number and returns an intelligent data
; object that can dispatch its operations
(define (make-from-real-imag-mp x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part y)
          ((eq? op 'magnitude)(sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else
            (error "MAKE-FROM-REAL-IMAG failed"
                   op)))))
  dispatch)
```

Uses a different apply-generic

```
; the individual parts are defined as above e.g.
;(define (real-part z)
;  (apply-generic 'real-part z))
; etc.
; apply-generic for message passing version
; notice that this function simply calls the data
; object as an function applied to the operator
(define (apply-generic op arg)
  (arg op))

;or more directly,
(define (real-part-mp z)
  (z 'real-part))
```

Advantage

The main advantage of the message passing programming style is that it more effectively hides the internal structure of data objects so that programmers are not tempted to access the insides of the data objects with cars and cdrs.

The programing style for classes in C++ most closely resembles message passing.