Lecture 13: Review; representing abstract data

CS 61A, Summer 2006

July 18, 2006

Instructor: Kevin Lin

1

Administrative stuff

- Midterm grades should be up
- It looks like the average was 30.4/40
- Midterm statistics: glookup -s mt1

- Project 1 grades should be up by the end of the week; I think most of you did pretty well on it.

2

Administrative stuff

- Get started on proj2 if you haven't already! Proj3 will be assigned at the end of this week.
- Start looking for project partners! If you don't have a partner, go to discussion/lab and/or talk to your TA.

3

Review: deep lists

Exercise: Write a procedure `(same-sublist? a b DL)` that checks whether **a** and **b** are contained in the same sublist of the deeplist **DL**. For example:

```
> (same-sublist? 'x 'y
        '(1 (2 3 (x y z) 4) 5))
#t
> (same-sublist? 'x 'y
        '(x ((1 2) y (3 4)) z))
#f
```

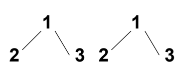
Use "car/cdr recursion". You may use the procedure member. Don't worry about efficiency.

4

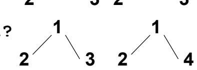
Review: Trees

Exercise: Write a procedure Tree-equal? that takes two Trees and checks whether they have the same data in the same locations.

```
> (Tree-equal?
```




```
)
#t
> (Tree-equal?
```




```
)
#f
```

5

Review: Scheme-1

Will the following work in Scheme-1? If it works, trace through what happens. If it doesn't work, explain why not.

```
Scheme-1: ((lambda (x) (x 2 2) +)
```

6

Scheme tricks

Remember that MAP can take a procedure of one argument and a LIST, and it applies the procedure to every argument of the list.

MAP can *also* take a procedure of N arguments, and N lists, and it will apply the procedure to the elements of the lists as follows:

```
> (map (lambda (x y) (* x y))
       '(2 2 2) '(1 2 3))
(2 4 6)
```


Scheme tricks

Also, as we saw earlier, APPLY may be used as follows:

```
> (apply + '(1 1 1 1 1))
5
> (apply square '(5))
25
```


Scheme tricks

Writing procedures that take arbitrarily many arguments:

```
(define (foo a b . rest)
  (cons b (map a rest)))

> (foo square 'baz 1 1 2 2)
(baz 1 1 4 4)
> (foo square 'baz 5)
(baz 25)
> (foo square 'baz)
(baz)
> (foo square)
ERROR: too few arguments
```


Back to representing abstract data …

Why do we call this segment "representing abstract data"?

Because our data is abstract, in the sense that it's not just data. It's not just words or numbers or lists or procedures or whatever.

It's data that *knows* stuff about itself; it's data that can *do* things by itself.

I've never seen a square that knows how to compute its own area. Have you?

Message passing

Yesterday we saw tagged data and data-directed programming.

Another method: message passing.

In message passing, the objects *themselves* know how to operate on themselves!

Message passing

Every object is a *dispatch procedure* that takes as its argument a *message* saying which operation to perform.

```
(define (make-square side)
  (lambda (msg)
    (cond ((equal? msg 'area)
           (* side side))
          ((equal? msg 'perimeter)
           (* 4 side))
          (else (error "unknown msg")))))

(define (area shape)
  (shape 'area))
```

Message passing

Message passing might seem overly complicated, and maybe for the example of shapes it is.

But we'll see that it is essential in object-oriented programming.

Message passing

Have we abandoned type-tagging? No:

```
(define (make-square side)
  (lambda (msg)
    (cond ((equal? msg 'area)
           (* side side))
          ((equal? msg 'perimeter)
           (* 4 side))
          ((equal? msg 'type-tag)
           'square)
          (else (error "unknown msg")))))
```


Data directed programming

We can *tag* data with information about what the data represents using `attach-tag`.

To select the contents and tag of a given tagged object, use `type-tag` and `contents`.

Data directed programming

For data-directed programming (or DDP), use PUT and GET.

Suppose we have written a package implementing rational numbers, real numbers, and complex numbers.

We want to be able to add different numbers of different types together.

Data directed programming

```
(put 'add '(rational rational) (lambda (x y) ...))
(put 'add '(rational complex) (lambda (x y) ...))
(put 'add '(real rational) (lambda (x y) ...))
(put 'add '(real complex) (lambda (x y) ...))
(put 'add '(complex complex) (lambda (x y) ...))
```

etc.

(Assume that rational numbers are pairs of integers, real numbers are in decimal form, and complex numbers are given by pairs of real numbers.)

Data directed programming

Now to get the procedure that adds a rational number and a complex number together, use
`(get 'add '(rational complex))`

Can we write a generic procedure that will do this for us?

Data directed programming

```
(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"
            (list op type-tags))))))
```


Data directed programming

How to use apply-generic?

```
> (apply-generic 'add (make-rational 1 3)
                      (make-complex 2 5))

; assuming that (make-rational 1 3) = 1/3
; and (make-complex 2 5) = 2+5i
```


Dyadic operations

How should you add a rational number to a complex number?

One possibility: *Raise* a rational number to the next type, real number, and then *raise* it again to the complex number type.

1/3 (rational) raises to 0.33333 (real) raises to 0.33333+0i (complex).

So 1/3+(2+5i) = 2.33333+5i.

Dyadic operations

But there is a problem with this:
1/3+(2+5i) = 2.33333+5i.

When we raise a rational number to a real number, we lose some information.

It would be better, perhaps, if we could get the answer 1/3+(2+5i) = (7/3)+5i.

But this would require us to change our implementation of complex numbers!

Dyadic operations

Moral: When dealing with numbers, things can get real hairy real fast. You will live longer if you only write programs that deal with integers.

Question: Suppose we want to write a program that will convert between different image formats (JPG, GIF, TIFF, PNG, etc). Suppose there are N different image formats. What is the minimum number of formatX-to-formatY converters that we need?

Theta(N^2)? Theta(N)? Something else?

Dyadic operations

In the case of numbers that we've shown we're lucky because we don't need to have N^2 raising algorithms, either.

This is because we have a "tower" of types: a rational number is a real number is a complex number.

So we don't need an explicit rational-to-complex raising algorithm. We can just compose rational-to-real and real-to-complex together.

We are not always so lucky!