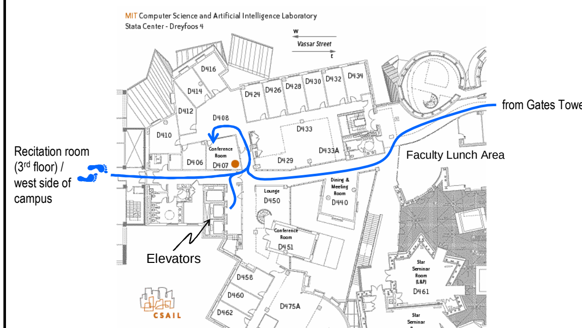
6.001 Recitation 7: Data Abstraction II

28 Feb 2007
RI: Gerald Dalley (dalleyg@mit.edu)

Announcements

- Solutions, handouts, *etc.*:
  - http://people.csail.mit.edu/dalleyg/6.001/SP2007/
  - `primes-in-range` discussion & orders of growth
- Office Hours
  - Thursdays, 2-3PM, 32-D407

Office Hours Location…



Overview

- Today: prime factorization, an extended example
- This is a nice example for several reasons:
  - Interesting design decisions
  - Practice with writing types
    - (using `prime` and `pf` as new types)
  - Related to primality testing from yesterday's lecture
  - primes are also important to Project 1…which is due next Friday.

Designing a data abstraction

- *Prime factorization:* representing an integer as the product of its prime factors

$$2 = 2 = 2^1$$
$$4 = 2 \bullet 2 = 2^2$$
$$6 = 2 \bullet 3 = 2^1 \bullet 3^1$$
$$40 = 2 \bullet 2 \bullet 2 \bullet 5 = 2^3 \bullet 5^1$$
$$187 = 11 \bullet 17 = 11^1 \bullet 17^1$$

What about 1?
What about 0?
What about negative integers?

Designing a data abstraction: constructors

New types
prime = subset of integers that are prime
pf = prime factorization data type

```
(make-prime-factors n):                                   integer → pf
    (make-prime-factors 40) → 2*2*2*5
```

```
(make-prime-factors factors):                       list<prime> → pf
    (make-prime-factors (list 2 2 2 5)) → 2*2*2*5
```

```
(make-prime-factors p):                                   prime → pf
(add-prime-factor p pf):                              prime, pf → pf
    (make-prime-factors 2) => 2
    (add-prime-factor 5 (make-prime-factors 2)) → 2*5
```

```
prime-factors-of-1:                                               pf
(add-prime-factor p pf):                              prime, pf → pf
    (add-prime-factor
      5
      (add-prime-factor 2 prime-factors-of-1)) → 2*5
```


Designing a data abstraction: accessors

For now, assume our constructor is `(make-prime-factors n)`

```
(get-number pf):                                           pf → int
  contract: (get-number (make-prime-factors n)) = n
```

```
(get-all-factors pf):                              pf → list<prime>
  contract:
   product of (get-all-factors (make-prime-factors n)) = n
```

```
(get-unique-factors pf):                           pf → list<prime>
(get-multiplicity pf p):                      pf, prime → integer
  contract:
   let
      (p0 ... pi) = (get-unique-factors (make-prime-factors n))
   and
      mi = (get-multiplicity (make-prime-factors n) pi)
   then
      n = product of pi^mi
```

Designing a data abstraction: operators

`(=pf pf1 pf2)`: tests whether two factorizations are the same	*pf,pf → boolean*
`(divides-pf? pf1 pf2)`: tests whether `pf1` divides evenly into `pf2`	*pf,pf → boolean*
`(has-factor? pf p)`: tests whether `p` is a prime factor of `pf`	*pf,prime → boolean*
`(*pf pf1 pf2)`: returns factorization of `n1*n2`	*pf,pf → pf*
`(/pf pf1 pf2)`: returns factorization of `n1/n2` **if n2 divides n1**	*pf,pf → pf*
`(gcd-pf pf1 pf2)`: returns factorization of greatest common divisor of `pf1` and `pf2`	*pf,pf → pf*

+pf, -pf
Not really appropriate for this data type. The only way to do it is converting to integer and then factorizing again.

How constructor choices affect operators

One constructor
Suppose our only constructor is `(make-prime-factors n)`
How do I write *pf: *pf,pf → pf* ?

```
(define (*pf pf1 pf2)
  (make-prime-factors
    (* (get-number pf1)
       (get-number pf2))))

(define (*pf pf1 pf2) :          pf,pf → pf
  (let ((combined-factors (append (get-all-factors pf1)
                                  (get-all-factors pf2))))
    ... how do I make a pf out of the resulting list of factors?
))
```

Let's provide two constructors:

```
(factorize n) :                  integer → pf
(make-prime-factors lst) :  list<prime> → pf

(define (*pf pf1 pf2)
  (make-prime-factors
    (append (get-all-factors pf1)
            (get-all-factors pf2)))
```


Many different representations are possible

```
(factorize 40);   40 = 2*2*2*5
⇒(2 2 2 5)        2*2*2*5    (sorted order)
⇒(2 5 2 2)        2*5*2*2    (order doesn't matter)
⇒((2 3) (5 1))    2^3 * 5^1
⇒(40 (2 5))                  stores n and its unique factors
```


Representation matters

```
; representation (2 2 2 5)
(define (get-multiplicity pf p)
  (cond ((null? pf) 0)
        ((= (car pf) p) (+ 1 (get-multiplicity (cdr pf) p)))
        ((> (car pf) p) 0)
        (else          (get-multiplicity (cdr pf) p))))

; representation ((2 3) (5 1)) (sorted order)
(define (get-multiplicity pf p)
  (cond ((null? pf) 0)
        ((= (caar pf) p) (get-multiplicity (cadar pf) p))
        ((> (car pf) p)  0)
        (else           (get-multiplicity (cdr pf) p))))

; representation (40 (2 5))
(define (get-multiplicity pf p)
  (define (multiplicity-of-p-in m)
    (if (divides? p m)
        (+ 1 (multiplicity-of-p-in (quotient m p)))
        0))
  (multiplicity-of-p-in (car pf)))
```


Representations also have implicit assumptions

```
(define (make-prime-factors lst) lst)

(define (*pf pf1 pf2)
  (append pf1 pf2))
    ... assumes order doesn't matter

(define (get-multiplicity pf p)
  (cond ((null? pf) 0)
        ((= (car pf) p)
         (+ 1 (get-multiplicity (cdr pf) p)))
        ((> (car pf) p) 0)
        (else (get-multiplicity (cdr pf) p))))
 ... assumes sorted order
```


~~Pain~~ Error Checking is Your Friend

```
(define (add-prime-factor p pf)
  (if (not (prime? p))
      (error "p is not prime")
      (cons p pf)))

(get-multiplicity
  (add-prime-factor
    24
    (make-prime-factor 1))
  2)
```

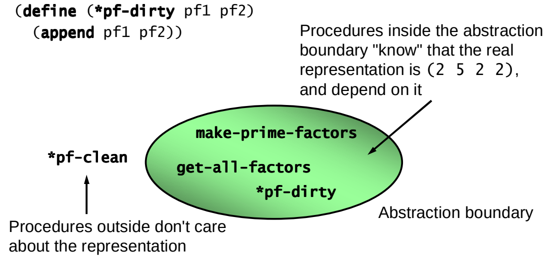
- Pain
  - Lets you know you're alive
  - Lets you know right away when something bad is happening
- Error checking
  - Lets you know *right away* when something bad is happening

Photo from:
http://wongkk.com/

Respect abstraction boundaries

```
(define (*pf-clean pf1 pf2)
  (make-prime-factors
    (append (get-all-factors pf1) (get-all-factors pf2))))

(define (*pf-dirty pf1 pf2)
  (append pf1 pf2))
```



Summary of data abstraction design

1. Choose **<u>constructors</u>** and **<u>accessors</u>** that are **useful** to clients and that make it possible to write the operators you need
   - Constructors and accessors should be **<u>complete</u>**: you need to be able to construct every possible object in the domain, and you need to be able to get out enough data to reconstruct the object
   - Write down the **<u>contract</u>** between the constructors and accessors
2. Choose **<u>representation</u>** that is appropriate to the operators you need (that makes the operators **<u>readable</u>** and **<u>efficient</u>**)
   - Write down the **<u>assumptions</u>** implicit in your representation
3. Respect **<u>abstraction boundaries</u>** as much as possible
   - Even within your abstraction's own code
   - Another way to say it: *Minimize the amount of code that "knows" what the real representation is.*