Announcements
• Solutions, handouts, etc.: http://people.csail.mit.edu/dalleyg/6.001/SP2007/primesinvrange discussion & orders of growth
• Office Hours
  • Thursdays, 2-3PM, 32-D407

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: constructors

<table>
<thead>
<tr>
<th>New types</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>prime</td>
<td>= subset of integers that are prime</td>
<td></td>
</tr>
<tr>
<td>pf</td>
<td>= prime factorization data type</td>
<td></td>
</tr>
</tbody>
</table>

### Prime Factors Constructors
- (make-prime-factors n): \( \text{Integer} \rightarrow \text{pf} \)
  \( \text{(make-prime-factors 40)} \rightarrow 2^3 \cdot 5 \)
  \( \text{(make-prime-factors \text{\{11 2 2 2 5\)}})} \rightarrow 2^2 \cdot 5^2 \)
- (add-prime-factor p pf): \( \text{prime}, \text{pf} \rightarrow \text{pf} \)
- (add-prime-factors pf): \( \text{pf} \rightarrow \text{pf} \)
  \( \text{(add-prime-factors \text{(add-prime-factor 2 pf)}} \rightarrow 2^5 \)

Designing a data abstraction: accessors

For now, assume our constructor is \( (\text{make-prime-factors } n) \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>get-number pf</td>
<td>( \text{pf} \rightarrow \text{int} )</td>
<td></td>
</tr>
<tr>
<td>contract</td>
<td>( (\text{get-number (make-prime-factors } n)) \rightarrow n )</td>
<td></td>
</tr>
<tr>
<td>get-all-factors pf</td>
<td>( \text{pf} \rightarrow \text{list&lt;prime&gt;} )</td>
<td></td>
</tr>
<tr>
<td>contract</td>
<td>( \text{product of (get-all-factors (make-prime-factors } n)} \rightarrow n )</td>
<td></td>
</tr>
<tr>
<td>get-unique-factors pf</td>
<td>( \text{pf} \rightarrow \text{list&lt;prime&gt;} )</td>
<td></td>
</tr>
<tr>
<td>contract</td>
<td>( (\text{get-unique-factors (make-prime-factors } n)) \rightarrow n )</td>
<td></td>
</tr>
<tr>
<td>get-multiplicity pf p</td>
<td>( \text{pf}, \text{prime} \rightarrow \text{int} )</td>
<td></td>
</tr>
<tr>
<td>contract</td>
<td>( (\text{get-multiplicity (make-prime-factors } n) \rightarrow \text{product of } p^m )</td>
<td></td>
</tr>
</tbody>
</table>

Prime factorization:

- 2 = \( 2^1 \)
- 4 = \( 2 \cdot 2 = 2^2 \)
- 6 = \( 2 \cdot 3 = 2^1 \cdot 3^1 \)
- 40 = \( 2 \cdot 2 \cdot 2 \). \( 5 = 2^3 \cdot 5^1 \)
- 187 = \( 11 \cdot 17 = 11^1 \cdot 17^1 \)

What about 1?
What about 0?
What about negative integers?
Representations also have implicit assumptions

\[ \text{(define (make-prime-factors lst) lst)} \]
\[ \text{(define (*pf pf1 pf2) (append pf1 pf2))} \]
\[ \text{... assumes order doesn't matter} \]
\[ \text{(define (get-multiplicity pf p) (cond ((null? pf) 0) ((= (car pf) p) (+ 1 (get-multiplicity (cdr pf) p))) ((> (car pf) p) 0) (else (get-multiplicity (cdr pf) p))))} \]
\[ \text{... assumes sorted order} \]
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))

Procedures inside the abstraction boundary "know" that the real representation is \(2 \times 5 \times 2\), and depend on it.

Procedures outside don't care about the representation.

Summary of data abstraction design

1. Choose constructors and accessors that are useful to clients and that make it possible to write the operators you need.
   - Constructors and accessors should be complete; 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 contract between the constructors and accessors.

2. Choose representation that is appropriate to the operators you need (that makes the operators readable and efficient).
   - Write down the assumptions implicit in your representation.

3. Respect abstraction boundaries 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.