CS410/510 Advanced Programming
Lecture 5:

Collections in Smalltalk
Hamming’s Problem Revisited
“List” Operations

• Last week you heard about list operations in Haskell

• For each there is a corresponding operation in Smalltalk; most work on any collection, not just lists.

• Advanced programmers use these operations; they almost never munge around with array indexes or pointers
Haskell ⇔ Smalltalk crib sheet

- `λ` map
- `λ` find
- `λ` filter
- `λ` all
- `λ` any
- `λ` foldl

- collect:
- detect:
- select:
- allSatisfy:
- anySatisfy:
- inject: into:
collect: captures a pattern

- If you ever find yourself writing a loop, or a recursive method, that builds a new collection based on an old one:

  - STOP!

    - Is this a collect:?
What about do:? 

- **do:** does some action on every element of a existing collection

- **collect:** builds a new collection based on applying a function to every element of an existing collection

- If you find yourself writing:
  
  ```smalltalk
  newCollection := <someclass> new.
  self do: [:each | newCollection add: (<an expression involving each>)].
  <proceed to use newCollection>
  ```

- Consider using **collect:** instead
Maybe types vs. Control

- Sometimes you don’t know if an element is in a collection

\[ \text{find} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Maybe a} \]

\[ \text{detect: } [ : \text{each} | \text{aBlock} ] \text{ ifNone: } [ \text{anotherBlock} ] \]

Examples:

\[ #(1 \ 3 \ 5) \text{ detect: } [: \text{each} | \text{each even} ] \rightarrow \text{error} \]

\[ #(1 \ 3 \ 5) \text{ detect: } [: \text{each} | \text{each even} ] \text{ ifNone: } [ 2 ] \rightarrow 2 \]

\[ #(1 \ 3 \ 4) \text{ detect: } [: \text{each} | \text{each even} ] \rightarrow 4 \]
Anonymous functions

- [: each | each even ] is an anonymous function
- What about named functions?
  - there aren’t any! Methods are not functions
- [ ] will turn a message-send into a function
  - [\( n \mid n + 1 \)] is the successor function
    - Haskell is briefer (+1)
- You could write a method that answers a function
folds

foldr substitutes from the right:

\( \text{foldr (:+) 0 [ 1, 2, 3 ]} \rightarrow 1 + 2 + 3 + 0 \)

or, more precisely: \( 1 + (2 + (3 + 0)) \)

foldl substitutes from the left:

\( \text{foldl (+) 0 [ 1, 2, 3 ]} \rightarrow 0 + 1 + 2 + 3 \)

or, more precisely: \( ((0 + 1) + 2) + 3 \)

\( \text{inject:into: is foldl} \)

\( (1 \text{ to: 3) inject: 0 into: [ :acc :each | acc + each ]} \)
inject:into: example

(1 to: 6)
  inject: Set new
  into: [:acc :each | each even
        ifTrue: [acc add: each]. acc]
inject:into: example

(1 to: 6)
  inject: Set new
  into: [:acc :each | each even
        ifTrue: [acc add: each]. acc]

⇒ a Set(6 2 4)
inject:into: example

(1 to: 6)
  inject: Set new
  into: [:acc :each | each even
        ifTrue: [acc add: each]. acc]
  ➞ a Set(6 2 4)

((1 to: 6) select: [:each | each even]) asSet
inject: into: example

(1 to: 6)
    inject: Set new
    into: [:acc :each | each even
         ifTrue: [acc add: each]. acc]

⇒ a Set(6 2 4)

((1 to: 6) select: [:each | each even]) asSet

what’s the difference?
common patterns captured by iterators

- **count: aPredicate**
  - answers the number of elements for which aPredicate is true

- **do: elementBlock separatedBy: separatorBlock**
  - execute the elementBlock for each element, and the separator block between the elements.

- **do: aBlock without: anItem**
  - execute aBlock for those elements that are not equal to anItem

- **detectMax: aBlock**
  - answer the element for which aBlock evaluates to the highest magnitude
...and on SequenceableCollections

- with: otherCollection collect: twoArgBlock
  - twoArgBlock calculates the elements of the result
- permutationsDo: aBlock
  - execute aBlock (self size factorial) times, with a single copy of self reordered in all possible ways.
- combinations: kk atATimeDo: aBlock
  - take my items kk at a time, and evaluate aBlock (self size take: kk) times, once for each combination. aBlock takes an array of elements; each combination only occurs once, and order of the elements does not matter.
“List Comprehensions”

• Generators

\[ [1..10] \]
\[ [1,5..25] \]

• Manipulators

\[ [ i * 2 | i <- [2..8] ] \]
\[ [ i * 2 | i <- [2..8]], even \ i \]
\[ [(i,j) | i <- [2..4], j <- [7..9]] \]
\[ zip [2..4] [7..9] \]
Programming is about finding patterns

- If the same pattern comes up in several places
  - abstract it into a programming language element (method, class, function)
  - replace all of the occurrences of the pattern with the abstraction
- once and only once
  - define the pattern once
Array printing example
Lessons from the Homework

- program with a partner
- fit into the framework
- don’t store all the values
- clean up after yourself
- avoid complexity (DTSTTCPW)
- write a printOn: method
- divide and conquer: separate complex cases
- when you inherit: override inappropriate methods
- write adequate tests
- don’t forget the class-side methods
Hamming’s problem

- h is an “Ordered Set”
- $1 \in h$
- $x \in h \Rightarrow 2x \in h, 3x \in h, 5x \in h.$
- generate all elements of $h < \text{limit}$
Let’s solve it

1. Write a test
2. make the test run green
3. clean up the code
   ➡ remove any duplication
4. repeat until done