CS 410/510: Advanced Programming

Lecture 7: Hamming, Closures, Laziness

Mark P Jones
Portland State University

The Hamming Set:

hamming = { 1 }
∪ { 2 * x | x ∈ hamming }
∪ { 3 * x | x ∈ hamming }
∪ { 5 * x | x ∈ hamming }

hamming = { 1, 2, 3, 4, 5, 6, 8, 9, 10,
12, 15, 16, 18, 20, 24, … }

The Hamming Sequence:

```
hamming = 1 :
          (merge [ 2 * x | x <- hamming ]
          (merge [ 3 * x | x <- hamming ]
                 [ 5 * x | x <- hamming ]))

Main> hamming
[ 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18,
  20, 24, … ^C{Interrupted!}
Main>
```


The Hamming Sequence:

```
hamming = 1 :
          (merge (map (2*) hamming)
          (merge (map (3*) hamming)
                 (map (5*) hamming)))

Main> hamming
[ 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18,
  20, 24, … ^C{Interrupted!}
Main>
```

How does this work?

"Infinite" Lists in Haskell:

How do examples like the following work?

```
Main> [1..]
[1,2,3,4,5,6,7,8,9,10,11^C{Interrupted!}

Main> iterate (10*) 1
[1,10,100,1000,10000,100000,1000000^C{Interrupted!}

Main> fibs where fibs = 0 : 1 : [ x+y | (x,y) <- zip fibs (tail fibs) ]
[0,1,1,2,3,5,8,13,21,34,55,89,144,233, ^C{Interrupted!}

Main>
```


Closures, Delays, Thunks …

- Haskell Expressions are treated as:
  - Thunks
  - Closures
  - Delayed Computations
  - Suspensions
  - …
- Expressions are evaluated:
  - Lazily
  - On demand
  - By need
  - …

[1..]

The list [1..] is syntactic sugar for the expression enumFrom 1, where:

```
enumFrom n = n : enumFrom (n+1)
```

enumFrom	n
Code: instructions on how to produce the next element	**Data**: inputs that are needed to produce the next element

Closure/Thunk

[n..m]

The list [n..m] is syntactic sugar for the expression enumFromTo n m, where:

```
enumFromTo n m
    = if n<=m then n : enumFromTo (n+1) m
                else []
```

enumFromTo	n, m
Code: instructions on how to produce the next element	**Data**: inputs that are needed to produce the next element

Closure/Thunk

sum [1..10]

```
sum xs = sum’ 0 xs
   where sum’ n [] = n
           sum’ n (x:xs) = sum’ (n+x) xs
```

```
sum [1..10]
= sum’ 0 [1..10]
= sum’ 1 [2..10]
= sum’ 3 [3..10]
= sum’ 6 [4..10]
= …
= sum’ 55 [11..10]
= 55
```

```
t :=0; n:=1; m:=10;
while (n<=m) {
  t  := t + n;
  n := n+1;
}
```

sum’ t [n..m]

Closures in Smalltalk:

- Blocks provide a similar mechanism:
  - [i := i + 1] describes a computation, but doesn’t run it (yet)
  - aBlock value forces
- Essential to make control structures work:
  - aBool ifTrue: […] ifFalse: […]
- A bigger example:
  - BlockClosure>>doWhileFalse: conditionBlock
  - |result|
  - [result := self value. conditionBlock value] whileFalse.
  - ^ result

[1..]

In Smalltalk:

- A class EnumFrom, instance variable head
- A class method: EnumFrom with: head
- Accessor methods:

  EnumFrom>>> head
  ^ head

  EnumFrom>>> tail
  ^ EnumFrom with: (head+1)

map (mult*)

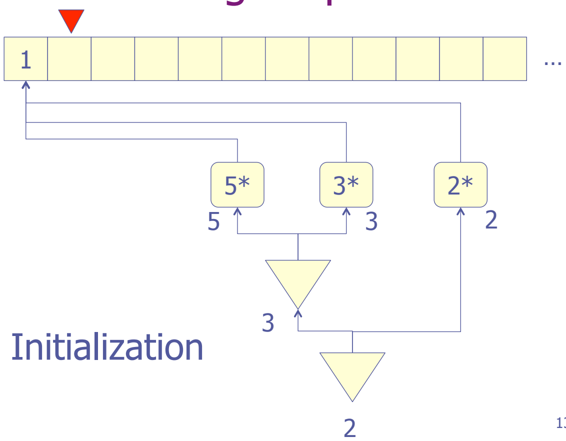
In Smalltalk:

- A class MultiplyBy, instance variables mult, aList
- A method: aList multiplyBy: mult
  (Which class should be home to this code?)
- Accessor methods:

  EnumFrom>>> head
  ^ aList head * mult

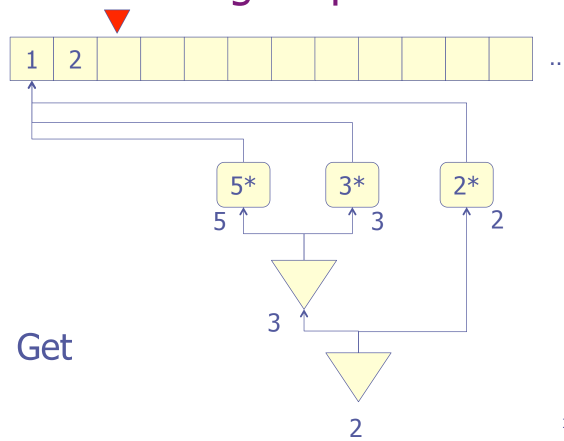
  EnumFrom>>> tail
  ^ aList tail multiplyBy: mult

The Hamming Sequence:



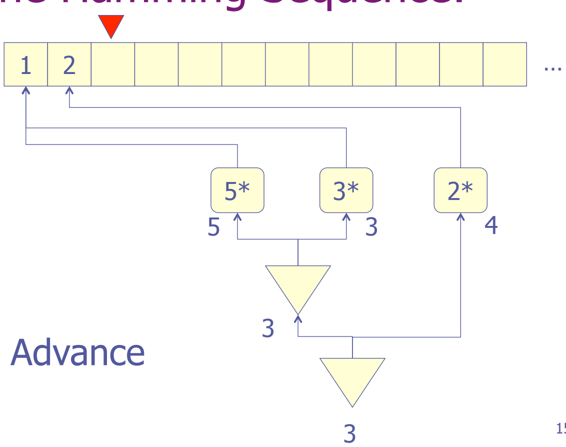
Initialization

The Hamming Sequence:



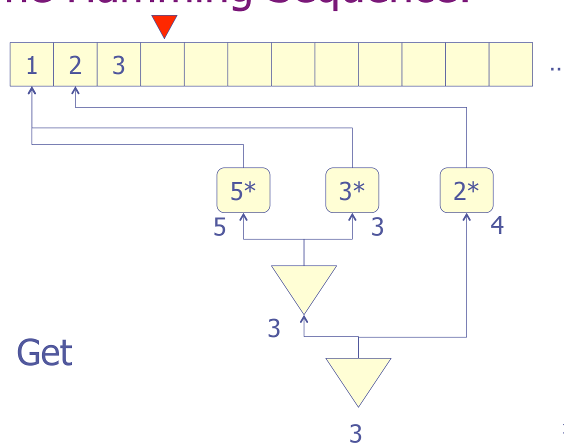
Get

The Hamming Sequence:



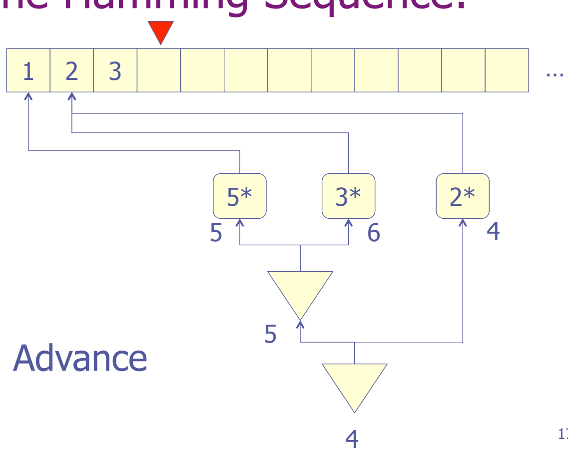
Advance

The Hamming Sequence:



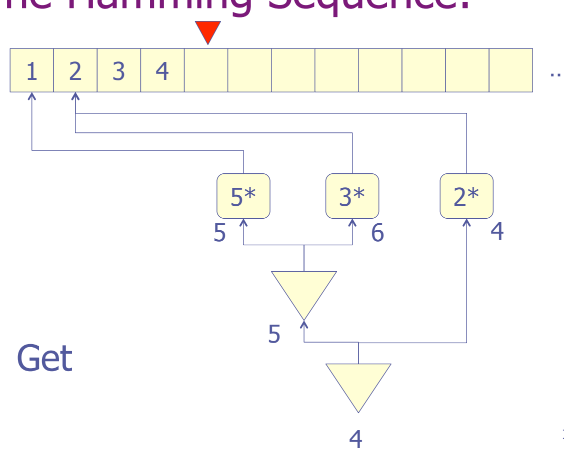
Get

The Hamming Sequence:



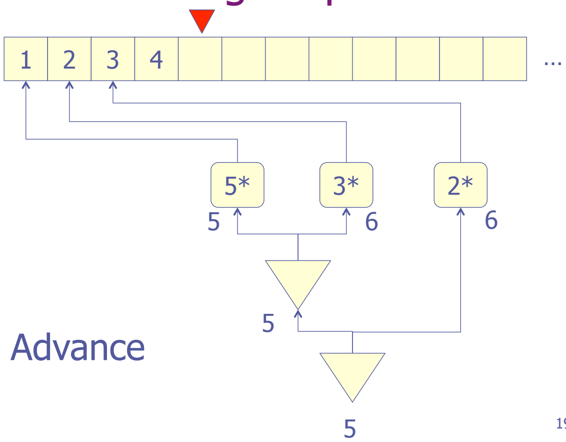
Advance

The Hamming Sequence:



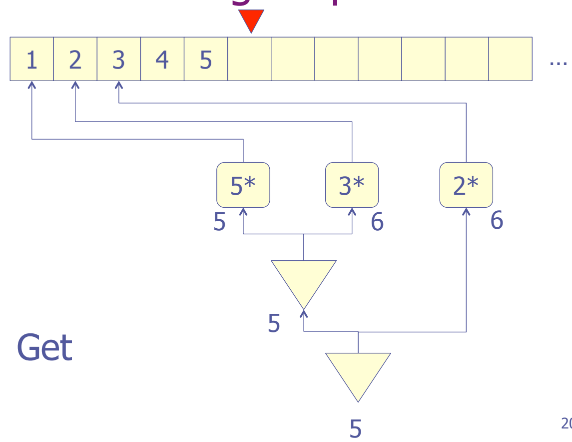
Get

The Hamming Sequence:



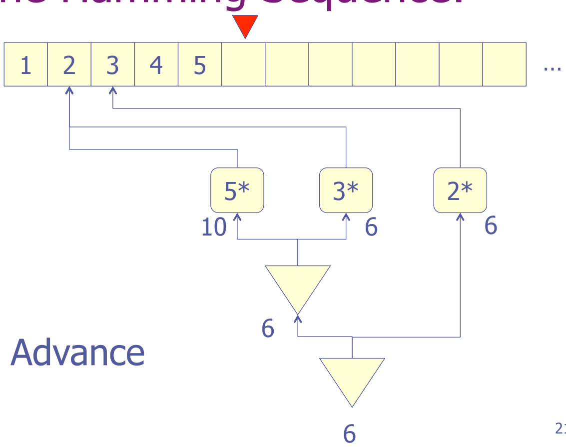
Advance

The Hamming Sequence:



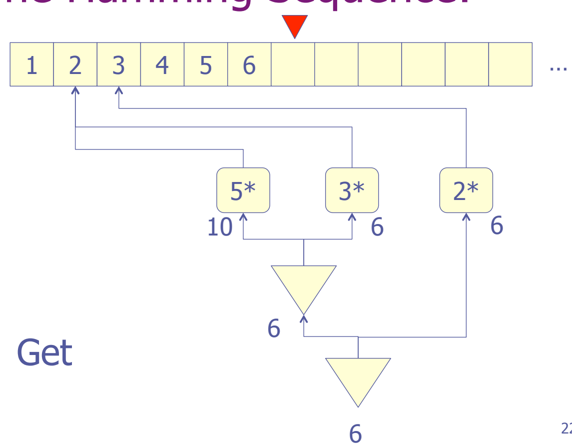
Get

The Hamming Sequence:



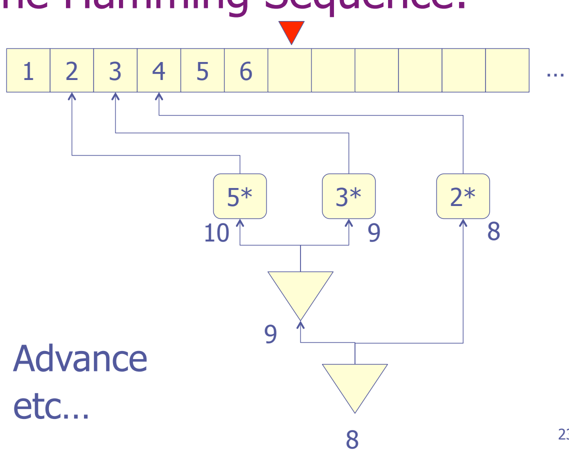
Advance

The Hamming Sequence:



Get

The Hamming Sequence:



Advance
etc…

Lists and Streams:

```
class List {
  int  head;
  List tail;
  List(int head) {
    this.head = head;
    this.tail = null;
  }
}
```

```
interface Stream {
  int  get();
  void advance();
}
```

Multiplier Streams:

```
class MultStream implements Stream {
  private int  mult;
  private List elems;
  MultStream(int mult, List elems) {
    this.mult  = mult;
    this.elems = elems;
  }

  public int get()         { return mult * elems.head; }
  public void advance() { elems = elems.tail; }
}
```


Merge Streams:

```
class MergeStream implements Stream {
  private Stream left, right;
  MergeStream(Stream left, Stream right) {
    this.left   = left;
    this.right = right;
  }

  public int get() {
    int l = left.get();
    int r = right.get();
    return (l<=r) ? l : r;
  }
```


Merge Streams (advance):

```
public void advance() {
  int l = left.get();
  int r = right.get();
  if (l == r) {
    left.advance();
    right.advance();
  } else if (l < r) {
    left.advance();
  } else {
    right.advance();
  }
```


Main Loop:

```
class Hamming {
  public static void main(String[] args) {
    List ham = new List(1);
    Stream s = new MergeStream(new MultStream(2, ham),
                  new MergeStream(new MultStream(3, ham),
                                          new MultStream(5, ham)));
    for (;;) {
      System.out.print(ham.head + ", ");
      int next = s.get();
      ham = ham.tail = new List(next);
      s.advance();
    }
  }
}
```


Observations:

- Hamming produces elements faster than the multiply/merge streams consume them
- We will never attempt to read uninitialized values
- The blue pointers are always behind the red pointer
- But the distance between the pointers will grow arbitrarily large … this can be considered a space leak

YAHS: (yet another Hamming solution)

```
factorOut        :: Int -> Int
factorOut n m
      | r == 0     = factorOut n q
      | otherwise = m
        where (q, r) = divMod m n

inHamming    :: Int -> Bool
inHamming    = (1==)
                    . factorOut 2
                    . factorOut 3
                    . factorOut 5
```

Summary:

- Programming with closures feels very natural in Haskell
  - Built-in support for lazy evaluation
  - Closure = function + arguments
  - Recursion
- But we can program with closures in other languages too!
  - One view of objects is as generalized closures:
    - Instance variables = Data
    - Methods = Multiple, parameterized Code entry points
- A powerful programming technique (not just for infinite lists)!

concat:

- concat :: [[a]] -> [a]
- concat [[1,2], [3,4,5], [6]]
  = [1,2,3,4,5,6]
- Laws:
  - filter p . concat = concat . map (filter p)
  - map f . concat = concat . map (map f)
  - concat . concat = concat . map concat

List Comprehensions:

General form:

- [expression | qualifiers]

where qualifiers are either:

- Generators: pat <- expr; or
- Guards: expr; or
- Local definitions: let defns

Works like a kind of generalized "for loop"

Examples:

[x*x | x <- [1..6]]
= [1, 4, 9, 16, 25, 36]

[x | x <- [1..27], 28 `mod` x == 0]
= [1, 2, 4, 7, 14]

[m | n <- [1..5], m <-[1..n]]
= [1, 1,2, 1,2,3, 1,2,3,4, 1,2,3,4,5]

Applications:

- Some "old friends":

  map f xs = [f x | x <- xs]
  filter p xs = [x | x <- xs, p x]
  concat xss = [x | xs <- xss, x <- xs]

- Can you define take, head, or (++) using a comprehension?

Laws of Comprehensions:

[x | x <- xs] = xs
[e | x <- xs] = map (\x -> e) xs

[e | True] = [e]
[e | False] = []

[e | gs_1, gs_2] = concat [[e | gs_2] | gs_1]

Example:

```
[ (x,y) | x <- [1,2], y <- [1,2] ]

= concat
    [ [ (x,y) | y <- [1,2]] | x <- [1,2] ]

= concat
    [ map (\y -> (x,y)) [1,2] | x <- [1,2] ]

= concat
    (map (\x ->
        map (\y -> (x,y)) [1,2]) [1,2])
```