

# CS 320: Concepts of Programming Languages

Wayne Snyder  
Computer Science Department  
Boston University

---

## Lecture 16: Lazy Evaluation in Haskell

- Review: Lazy Evaluation and Simultaneous Let
- Infinite Lists
- Lazy Evaluation and Pattern Matching
- Lazy vs Strict Evaluation in Haskell
- Foldl and foldl'

# Lazy Evaluation and Simultaneous Definitions

When we think of bindings as simultaneous equations, we see how Haskell interprets equations in **let** and **where**:

```
x = 2 * z
y = 4
z = y + 1
```

equivalent to



```
x = 10
y = 4
z = 5
```

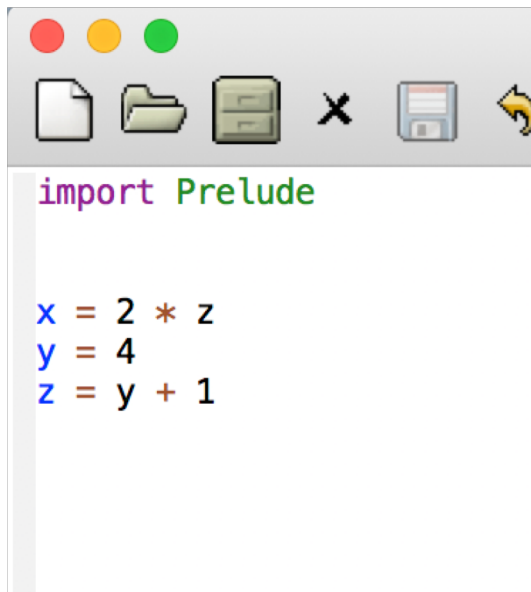
```
test = let x = 2 * z
        y = 4
        z = y + 1
      in (x, y, z)
```

```
test2 = (x, y, z) where
        x = 2 * z
        y = 4
        z = y + 1
```

```
Main> :r
[1 of 1] Compiling Main
( Main.hs, interpreted )
Ok, one module loaded.
Main> test
(10, 4, 5)
Main> test2
(10, 4, 5)
```

# Simultaneous Equations

The same thing is true of equations in your code:

A screenshot of a code editor window with a standard macOS-style title bar (red, yellow, green buttons) and a toolbar with icons for file operations. The code inside the editor is:

```
import Prelude

x = 2 * z
y = 4
z = y + 1
```

```
Main> :r
[1 of 1] Compiling Main
Main.hs, interpreted )
Ok, one module loaded.
Main> x
10
Main> y
4
Main> z
5
```

# Evaluation Order: Strict vs Lazy

Haskell uses lazy evaluation by default, although you can modify this to make functions strict.

```
Main> x = x + 1
Main> "NOOOO, DONT DO IT!!!!!"
"NOOOO, DONT DO IT!!!!!"
Main> x
```

-- Infinite digression, hit Control-c

If strict evaluation were being used, then  $x + 1$  would be evaluated first, and  $x$  is unbound (since the binding to  $x$  has not yet been made), as if

```
Main> x = x + 1
```

```
<interactive>:47:5: error: Variable not in scope: x
Main>
```

# Evaluation Order: Strict vs Lazy

But Haskell uses **lazy evaluation**, so

```
Main> x = x + 1
```

```
Main> x
```

Look up x, substitute the binding:

```
(x + 1)
```

Hm... look up x, substitute the binding:

```
((x + 1) + 1)
```

Hm... look up x, substitute the binding:

```
(( (x + 1) + 1) + 1)
```

etc. to infinite and beyond!

In this regard, as in many others, Haskell follows ordinary mathematical practice, rather than imperative programming language practice:

$$x = x + 1$$

has no solution in ordinary mathematics!

# Evaluation Order: Strict vs Lazy

This explains how simultaneous equations in **let** are evaluated, instead of storing values in the state/environment, we store unevaluated expressions and evaluate by need:

```
test = let x = 2 * z
        y = 4
        z = y + 1
      in x
```

```
Main> test
10
```

```
Bindings: [(x, (2 * z)), (y, 4), (z, (y+1))]
```

```
eval( test )
eval( x )
eval( 2 * z )
  eval( z )
  eval( y + 1 )
    eval( y )
    => 4
  => 5
=> 10
```

# Lazy Evaluation and Pattern Matching

Haskell only evaluates an expression when it has to, and ONLY as much as it has to. Pattern matching doesn't always force evaluation, though printing or adding does:

## What you see

## What is stored in Environment

Main> lst = [2*3,4+8,2^1000]	[(lst, [2*3,4+8,2^1000])]
Main> x = head lst	[(lst, [2*3,4+8,2^1000]), (x, )]
Main> y = head (tail lst)	[(lst, [2*3,4+8,2^1000]), (x, ), (y, )]
Main> add2 (x:y:_) = x + y	
Main> z = add2 lst	[(lst, [2*3,4+8,2^1000]), (x, ), (y, ), (z, +)]
Main> x 6	[(lst, [6,4+8,2^1000]), (x, ), (y, ), (z, +)]
Main> z 18	[(lst, [6,12,2^1000]), (x, ), (y, ), (z, 18)]

# Infinite Lists

One important consequence of lazy evaluation is that it allows the creation and manipulation of infinite data structures. The most common case is infinite lists, but infinite trees, and other infinite structures, are possible.

```
Main> a = [1..]    -- Create an infinite list
Main> a            -- NOOOOOOO!    Don't do this!
                    -- and if you do, hit control-c
```

```
Main> take 10 a
[1,2,3,4,5,6,7,8,9,10]
```

```
Main> b = [1,3..]
Main> take 10 b
[1,3,5,7,9,11,13,15,17,19]
```

```
Main> c = [ (x,x+1) | x <- [1..] ]
Main> take 5 c
[(1,2), (2,3), (3,4), (4,5), (5,6)]
```

Think of `[1..]` as an iterator which produces the values 1, 2, 3, etc. when you ask for them. Iterators are built into many languages. In Python, `range(...)` is an iterator.



# Infinite Lists

A function which needs to access an entire list can not be used with an infinite list:

```
Main> a = [1..]
```

```
Main> sum a           -- NOOOO! WAYNE, DON'T DO THESE!!!!
```

```
Main> length a
```

```
Main> last a
```

But those which only access a finite prefix of the list are fine:

```
Main> a !! 10
```

```
11
```

```
Main> takeWhile (<6) a
```

```
[1,2,3,4,5]
```

```
Main> (b,c) = splitAt 10 a
```

```
Main> b
```

```
[1,2,3,4,5,6,7,8,9,10]
```

```
Main> take 5 c
```

```
[11,12,13,14,15]
```

# Infinite Lists

And functions which create new infinite lists from other infinite lists will work, as long as they can (lazily) only access a prefix of the list.

```
Main> a = map (*10) [1,3..]
```

```
Main> take 10 a
```

# Infinite Lists

And functions which create new infinite lists from other infinite lists will work, as long as they can (lazily) only access a prefix of the list.

```
Main> a = map (*10) [1,3..]
```

```
Main> take 10 a
```

```
[10,30,50,70,90,110,130,150,170,190]
```

```
Main> b = filter (\x -> x `mod` 3 == 0) [1..]
```

```
Main> take 10 b
```

# Infinite Lists

And functions which create new infinite lists from other infinite lists will work, as long as they can (lazily) only access a prefix of the list.

```
Main> a = map (*10) [1,3..]
```

```
Main> take 10 a
```

```
[10,30,50,70,90,110,130,150,170,190]
```

```
Main> b = filter (\x -> x `mod` 3 == 0) [1..]
```

```
Main> take 10 b
```

```
[3,6,9,12,15,18,21,24,27,30]
```

```
Main> c = zip [1..] [10..]
```

```
Main> take 8 c
```

# Infinite Lists

And functions which create new infinite lists from other infinite lists will work, as long as they can (lazily) only access a prefix of the list.

```
Main> a = map (*10) [1,3..]
```

```
Main> take 10 a
```

```
[10,30,50,70,90,110,130,150,170,190]
```

```
Main> b = filter (\x -> x `mod` 3 == 0) [1..]
```

```
Main> take 10 b
```

```
[3,6,9,12,15,18,21,24,27,30]
```

```
Main> c = zip [1..] [10..]
```

```
Main> take 8 c
```

```
[(1,10), (2,11), (3,12), (4,13), (5,14), (6,15), (7,16), (8,17)]
```

```
Main> zip ['a'..'z'] [0..]
```

# Infinite Lists

Infinite data structures can also be created from **recursive equations**. Lazy evaluation means you only create as much of the list as you need. Just don't try to print it out or apply a function to the whole list!

$$a = 1:a$$

What list would satisfy this equation? Let's expand it out:

$$\begin{aligned} a &= 1:a \\ &= 1:(1:a) \\ &= 1:1:(1:a) \\ &= 1:1:1:1:1:1:1: \dots \\ &= [1, 1, 1, 1, 1, 1, 1, \dots ] \end{aligned}$$

Similarly,

$$\begin{aligned} b &= [1, 2, 3] ++ b \\ &= [1, 2, 3] ++ ([1, 2, 3] ++ b) \\ &= [1, 2, 3, 1, 2, 3] ++ ([1, 2, 3] ++ b) \\ &= [1, 2, 3, 1, 2, 3, 1, 2, 3, \dots ] \end{aligned}$$

# Infinite Lists

```
s = 1:t
```

```
t = 2:u
```

```
u = 3:s
```

```
-- can do this in a file, not in repl
```

```
Main> take 10 s
```

# Infinite Lists

```
s = 1:t  
t = 2:u  
u = 3:s
```

-- can do this in a file, not in repl

```
Main> take 10 s
```

```
[1,2,3,1,2,3,1,2,3,1]
```

```
Main> take 10 t
```

```
[2,3,1,2,3,1,2,3,1,2]
```

```
Main> take 10 u
```

```
[3,1,2,3,1,2,3,1,2,3]
```

```
Main> w = 1:(map (*2) w)
```

```
Main> take 10 w
```



# Infinite Lists

```
s = 1:t           -- can do this in a file, not in repl  
t = 2:u  
u = 3:s
```

```
Main> take 10 s  
[1,2,3,1,2,3,1,2,3,1]
```

```
Main> take 10 t  
[2,3,1,2,3,1,2,3,1,2]
```

```
Main> take 10 u  
[3,1,2,3,1,2,3,1,2,3]
```

```
Main> w = 1:(map (*2) w)
```

```
Main> take 10 w  
[1,2,4,8,16,32,64,128,256,512]
```

# Infinite Lists

```
s = 1:t           -- can do this in a file, not in repl
t = 2:u
u = 3:s
```

```
Main> take 10 s
[1,2,3,1,2,3,1,2,3,1]
```

```
Main> take 10 t
[2,3,1,2,3,1,2,3,1,2]
```

```
Main> take 10 u
[3,1,2,3,1,2,3,1,2,3]
```

```
Main> w = 1:(map (*2) w)
```

```
Main> take 10 w
[1,2,4,8,16,32,64,128,256,512]
```

# Infinite Lists

Combining recursion with list comprehensions is a typical Haskell idiom:

```
Main> a = [ x + 1 | x <- [1..] ]
```

```
Main> take 10 a
```

# Infinite Lists

Combining recursion with list comprehensions is a typical Haskell idiom:

```
Main> a = [ x + 1 | x <- [1..] ]
```

```
Main> take 10 a  
[2,3,4,5,6,7,8,9,10,11]
```

```
Main> b = [ x + y | (x,y) <- zip [1..] [5..] ]
```

```
Main> take 10 b
```

# Infinite Lists

Combining recursion with list comprehensions is a typical Haskell idiom:

```
Main> a = [ x + 1 | x <- [1..] ]
```

```
Main> take 10 a  
[2,3,4,5,6,7,8,9,10,11]
```

```
Main> b = [ x + y | (x,y) <- zip [1..] [5..] ]
```

```
Main> take 10 b  
[6,8,10,12,14,16,18,20,22,24]
```

```
Main> c = 1:[ x * 2 | x <- c ]
```

```
Main> take 10 c
```

# Infinite Lists

Combining recursion with list comprehensions is a typical Haskell idiom:

```
Main> a = [ x + 1 | x <- [1..] ]
```

```
Main> take 10 a  
[2,3,4,5,6,7,8,9,10,11]
```

```
Main> b = [ x + y | (x,y) <- zip [1..] [5..] ]
```

```
Main> take 10 b  
[6,8,10,12,14,16,18,20,22,24]
```

```
Main> c = 1:[ x * 2 | x <- c ]
```

```
Main> take 10 c  
[1,2,4,8,16,32,64,128,256,512]
```

# Infinite Lists

Combining finite lists with infinite lists works pretty well:

```
Main> h = [ (x,y) | x <- [1,2,3], y <- [2,4..] ]
```

```
Main> take 10 h
```

# Infinite Lists

Combining finite lists with infinite lists works pretty well:

```
Main> h = [ (x,y) | x <- [1,2,3], y <- [2,4..] ]
```

```
Main> take 10 h
```

```
[(1,2), (1,4), (1,6), (1,8), (1,10), (1,12), (1,14), (1,16), (1,18), (1,20)]
```

```
Main> i = [ (x,y) | x <- [1,3..], y <- [1,2,3] ]
```

```
Main> take 10 i
```



# Infinite Lists

Combining finite lists with infinite lists works pretty well:

```
Main> h = [ (x,y) | x <- [1,2,3], y <- [2,4..] ]
```

```
Main> take 10 h
```

```
[(1,2), (1,4), (1,6), (1,8), (1,10), (1,12), (1,14), (1,16), (1,18), (1,20)]
```

```
Main> i = [ (x,y) | x <- [1,3..], y <- [1,2,3] ]
```

```
Main> take 10 i
```

```
[(1,1), (1,2), (1,3), (3,1), (3,2), (3,3), (5,1), (5,2), (5,3), (7,1)]
```

But combining two infinite lists can be tricky!

```
Main> j = [ (x,y) | x <- [1..], y <- [10..] ]
```

```
Main> take 8 j
```

# Infinite Lists

Combining finite lists with infinite lists works pretty well:

```
Main> h = [ (x,y) | x <- [1,2,3], y <- [2,4..] ]
```

```
Main> take 10 h
```

```
[(1,2), (1,4), (1,6), (1,8), (1,10), (1,12), (1,14), (1,16), (1,18), (1,20)]
```

```
Main> i = [ (x,y) | x <- [1,3..], y <- [1,2,3] ]
```

```
Main> take 10 i
```

```
[(1,1), (1,2), (1,3), (3,1), (3,2), (3,3), (5,1), (5,2), (5,3), (7,1)]
```

But combining two infinite lists can be tricky!

```
Main> j = [ (x,y) | x <- [1..], y <- [10..] ]      -- BAD!
```

```
Main> take 8 j
```

```
[(1,10), (1,11), (1,12), (1,13), (1,14), (1,15), (1,16), (1,17)]
```

```
Main> k = [ x + y | (x,y) <- zip [1..] [5..] ]    -- GOOD!
```

```
Main> take 10 k
```

# Infinite Lists

Combining finite lists with infinite lists works pretty well:

```
Main> h = [ (x,y) | x <- [1,2,3], y <- [2,4..] ]
```

```
Main> take 10 h
```

```
[(1,2), (1,4), (1,6), (1,8), (1,10), (1,12), (1,14), (1,16), (1,18), (1,20)]
```

```
Main> i = [ (x,y) | x <- [1,3..], y <- [1,2,3] ]
```

```
Main> take 10 i
```

```
[(1,1), (1,2), (1,3), (3,1), (3,2), (3,3), (5,1), (5,2), (5,3), (7,1)]
```

But combining two infinite lists can be tricky!

```
Main> j = [ (x,y) | x <- [1..], y <- [10..] ]      -- BAD!
```

```
Main> take 8 j
```

```
[(1,10), (1,11), (1,12), (1,13), (1,14), (1,15), (1,16), (1,17)]
```

```
Main> k = [ x + y | (x,y) <- zip [1..] [5..] ]      -- GOOD!
```

```
Main> take 10 k
```

```
[6,8,10,12,14,16,18,20,22,24]
```

# Infinite Lists

But be careful that you are not somehow asking Haskell to look at a whole infinite list!

```
*Main> a = filter (<10) [1..]  
*Main> a  
[1,2,3,4,5,6,7,8,9]
```

```
*Main> a = filter (<10) [1..]  
*Main> a  
[1,2,3,4,5,6,7,8,9]^CInterrupted.
```

```
*Main> head (reverse (reverse [1..]))  
^C^C^CInterrupted.
```

# Programming with Infinite Lists

The power of infinite lists leads to some very interesting algorithms in Haskell, particularly when generating useful infinite series.

## Prime Numbers:

```
Main> factors x = filter (\y -> x `mod` y == 0) [1..x]
Main> primes = [ x | x <- [1..], factors x == [1,x] ]
Main> take 20 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

## Factorials:

```
Main> fact = map (\n -> product [1..n]) [1..]
Main> take 10 fact
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

## Fibonacci Numbers:

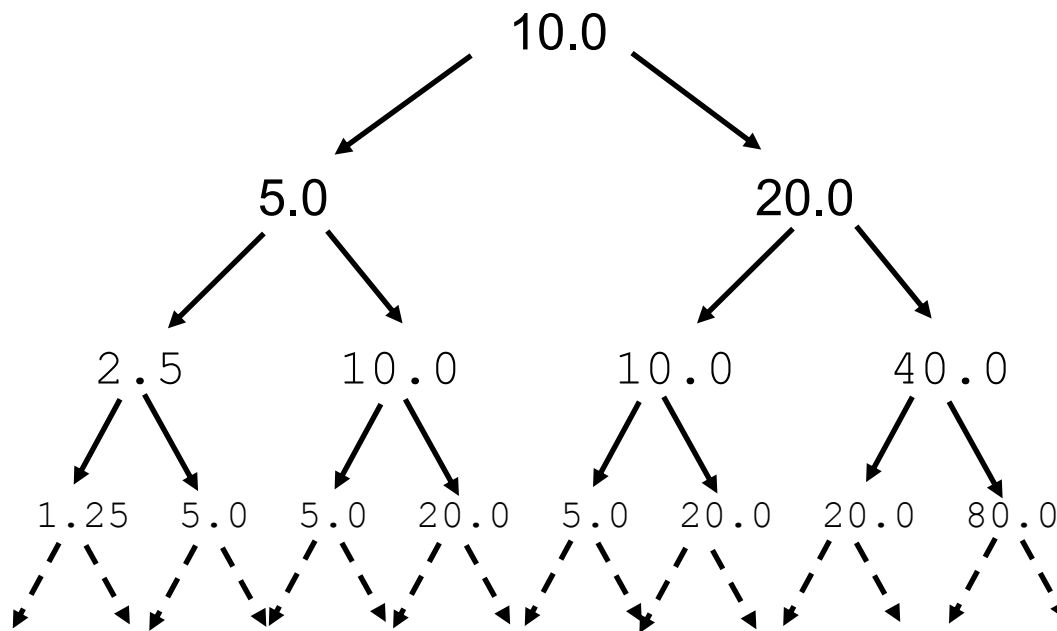
```
Main> fib = 1:1:[ x+y | (x,y) <- zip fib (tail fib) ]
Main> take 18 fib
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584]
```

# Infinite Trees

```
data Tree = Node Tree Double Tree deriving Show

tree :: Double -> Tree
tree x = Node (tree (x / 2)) x (tree (x * 2))

level :: Integer -> Tree -> [Double]
level 0 (Node _ x _) = [x]
level n (Node left x right) = (level (n-1) left) ++ (level (n-1) right)
```





# Infinite Trees

```
data Tree = Node Tree Double Tree deriving Show

tree :: Double -> Tree
tree x = Node (tree (x / 2)) x (tree (x * 2))

level :: Integer -> Tree -> [Double]
level 0 (Node _ x _) = [x]
level n (Node left x right) = (level (n-1) left) ++ (level (n-1) right)
```

```
Main> level 0 $ tree 10
[10.0]
```

```
Main> level 1 $ tree 10
[5.0,20.0]
```

```
Main> level 2 $ tree 10
[2.5,10.0,10.0,40.0]
```

```
Main> level 3 $ tree 10
[1.25,5.0,5.0,20.0,5.0,20.0,20.0,80.0]
```

