CS 320: Concepts of Programming Languages

Wayne Snyder
Computer Science Department
Boston University

Lecture 15: Lambda Calculus II and Evaluation Order

- Properties of Beta-Reduction: Non-termination and confluence
- Evaluation Strategies and their meaning in programming languages
- Haskell Lazy Evaluation:
  - Simultaneous let definitions
  - Infinite Lists

Lambda Calculus: Properties of Beta-Reduction

Recall from last time:

Alpha-Conversion (change of bound variables):

$$\lambda x.\ E \quad \to_{\alpha} \quad \lambda x'.\ (\ E\ [x := x']\ )$$

where `x'` is a fresh variable (never seen before in this context).

Intuitively: change the bound variable x and every occurrence of an x corresponding to this binding to a new variable (your choice, but make sure it doesn't conflict with any other variables, either bound or free).

Examples:

$$\lambda x.\ (\ \lambda y.\ x\ y\ )\ \to_{\alpha}\ \lambda x'.\ (\ \lambda y.\ x'\ y\ )$$

$$\lambda x.\ (\ \lambda y.\ x\ (\lambda x\ .\ x\ y)\ )\ \to_{\alpha}\ \lambda x'.\ (\ \lambda y.\ x'\ (\lambda x\ .\ x\ y)\ )$$

Lambda Calculus: Properties of Beta-Reduction

Recall from last time:

Beta-Conversion (function application by parameter passing):

$$(\lambda x.\ E)\ F \quad \rightarrow_{\beta} \quad E[x := F]$$

where the term (λx. E) has undergone alpha-conversion as necessary to prevent free variable capture when making the substitution of F for x in E.

Examples:

$$(\lambda x.\ (\lambda y.\ x\ (\lambda x'\ .\ x'\ (\ y\ x\ )\ )\ )\ )\ z \rightarrow_{\beta} (\lambda y.\ z\ (\lambda x'\ .\ x'\ (\ y\ z\ )\ )\ )\ )$$

$$(\lambda x.\ (\lambda y.\ x\ (\lambda x\ .\ x\ (\ y\ x\ )\ )\ )\ )\ z \rightarrow_{\beta} (\lambda y.\ z\ (\lambda x\ .\ x\ (y\ x\ )\ )\ )\ )$$

$$(\lambda x.\ (\lambda y.\ x\ (\lambda x\ .\ x\ (\ y\ x\ )\ )\ )\ )\ y \rightarrow_{\alpha} (\lambda x.\ (\lambda y'.\ x\ (\lambda x\ .\ x\ (\ y'\ x\ )\ )\ )\ )\ y \rightarrow_{\beta} (\lambda y'.\ y\ (\lambda x\ .\ x\ (y'\ x\ ))))$$

$$\begin{aligned} (\lambda x.\ (\lambda y.\ y\ (\lambda y\ .\ y\ (\ y\ x\ )\ )\ )\ )\ y &\rightarrow_{\alpha} (\lambda x.\ (\lambda y'.\ y'\ (\lambda y\ .\ y\ (\ y\ x\ )\ )\ )\ )\ y \\ &\rightarrow_{\alpha} (\lambda x.\ (\lambda y'.\ y'\ (\lambda y''\ .\ y''\ (\ y''\ x\ )\ )\ )\ )\ y \\ &\rightarrow_{\beta} (\lambda y'.\ y'\ (\lambda y''\ .\ y''\ (\ y''\ y\ )\ )\ )\ ) \end{aligned}$$

Lambda Calculus: Properties of Beta-Reduction

Note that it does not matter in principle where the **beta-redex** is, and there could be more than one:

Examples:

$\underline{(\lambda z.\ (\lambda x.\ z\ x)\ )\ \ y}\ \ \rightarrow_{\beta}\ \ (\lambda x.\ \ y\ x\ )$ -- redex at top of expression

$z\ \ (\ z\ (\ \underline{(\lambda x.\ z\ x)\ \ y}\ )\ \ \rightarrow_{\beta}\ \ z\ \ (\ z\ (\ (\lambda x.\ z\ x)\ \ y\ )$ -- redex deep inside expression

$(\lambda z.\ \underline{(\lambda x.\ z\ x)\ (\lambda y.\ y)}\ )\ \rightarrow_{\beta}\ \ (\lambda z.\ \ z\ (\ \lambda y.\ \ y)\ )$ -- redex inside an abstraction

$\underline{(\ \lambda x.\ x\ y)\ (\underline{(\ \lambda x.\ x)\ (\ \lambda x.\ \ z\ x)})}\ \rightarrow_{\beta}\ \ ??$ **-- which one to reduce?**

Lambda Calculus: Properties of Beta-Reduction

There may be 0, 1, or more than 1 beta-redex. A lambda-expression with no beta-redexes is said to be in **normal form**. Such expressions maybe considered to be "values."

true $=_{def}$ $(\lambda x.\ \lambda y.\ x)$ two $=_{def}$ $(\lambda f.\ \lambda x.\ f\ (f\ x)\ )$

Evaluating a pure lambda calculus expression means to beta-reduce it to a normal form, if possible:

$$\underline{(\lambda x.\ x\ y)\ ((\lambda x.\ x)\ (\lambda x.\ z\ x))} \rightarrow_\beta (\underline{(\lambda x.\ x)\ (\lambda x.\ z\ x)})\ y \rightarrow_\beta (\lambda x.\ z\ x)\ y \rightarrow_\beta \underbrace{z\ y}_{\text{normal form}}$$

But this may not be possible! Beta-reductions may not terminate:

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow_\beta (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow_\beta (\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow_\beta \ \ldots\ldots$$

$$(\lambda x.\ (x\ x)\ x)\ (\lambda x.\ (x\ x)\ x) \rightarrow_\beta ((\lambda x.\ (x\ x)\ x)\ (\lambda x.\ (x\ x)\ x))\ (\lambda x.\ (x\ x)\ x) \rightarrow_\beta \ \ldots\ldots$$

Lambda Calculus: Properties of Beta-Reduction

When there is more than one redex, there are two important issues:

(1) Which one to reduce first? In general, what is our overall strategy for choosing redexes?

(2) Does it matter which strategy that we use? What are the consequences of choosing a strategy?

Issue (1) : There are two basic reduction strategies.

(A) Normal or Leftmost Order: "The leftmost, outermost redex is always reduced first. That is, ... the arguments are substituted into the body of an abstraction before the arguments are reduced." (Wikipedia)

$$\underline{(\lambda x.\, x\, y)\ ((\lambda x.\, x)\ (\lambda x.\ z\, x))} \rightarrow_\beta ((\lambda x.\, x)\ (\lambda x.\ z\, x))\ y$$

Lambda Calculus: Properties of Beta-Reduction

Reduction Strategies

(A) Normal or Leftmost Order: "The leftmost, outermost redex is always reduced first. That is, ... the arguments are substituted into the body of an abstraction before the arguments are reduced." (Wikipedia)

$$\underline{(\lambda x.\ x\ y)\ ((\lambda x.\ x)\ (\lambda x.\ z\ x))} \to_{\beta} ((\lambda x.\ x)\ (\lambda x.\ z\ x))\ y$$

(B) Applicative or Strict Order: "The rightmost, innermost redex is always reduced first. Intuitively this means a function's arguments are always reduced before the function itself. Applicative order always attempts to apply functions to normal forms, even when this is not possible." (Wikipedia)

$$(\lambda x.\ x\ y)\ (\underline{(\lambda x.\ x)\ (\lambda x.\ z\ x)}) \to_{\beta} (\lambda x.\ x\ y)\ (\lambda x.\ z\ x)$$

Lambda Calculus: Properties of Beta-Reduction

Issue (2): What are the consequences of choosing one strategy over the other?

Several important consequences:

(i) If there is any reduction sequence which terminates in a normal form, Normal Order will find one (which is why it is called "normal" order, since it finds normal forms).

(ii) Applicative Order may not terminate, even when there does exist some terminating sequence. Example:

$\underline{(\lambda x.\ y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))} \rightarrow_{\beta} y$ -- normal order

$(\lambda x.\ y)\ (\underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)}) \rightarrow_{\beta} (\lambda x.\ y)\ (\underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)}) \rightarrow_{\beta}$ ..... -- applicative order

We explored this in hw 01! Preorder = normal order Postorder = applicative

Lambda Calculus: Properties of Beta-Reduction

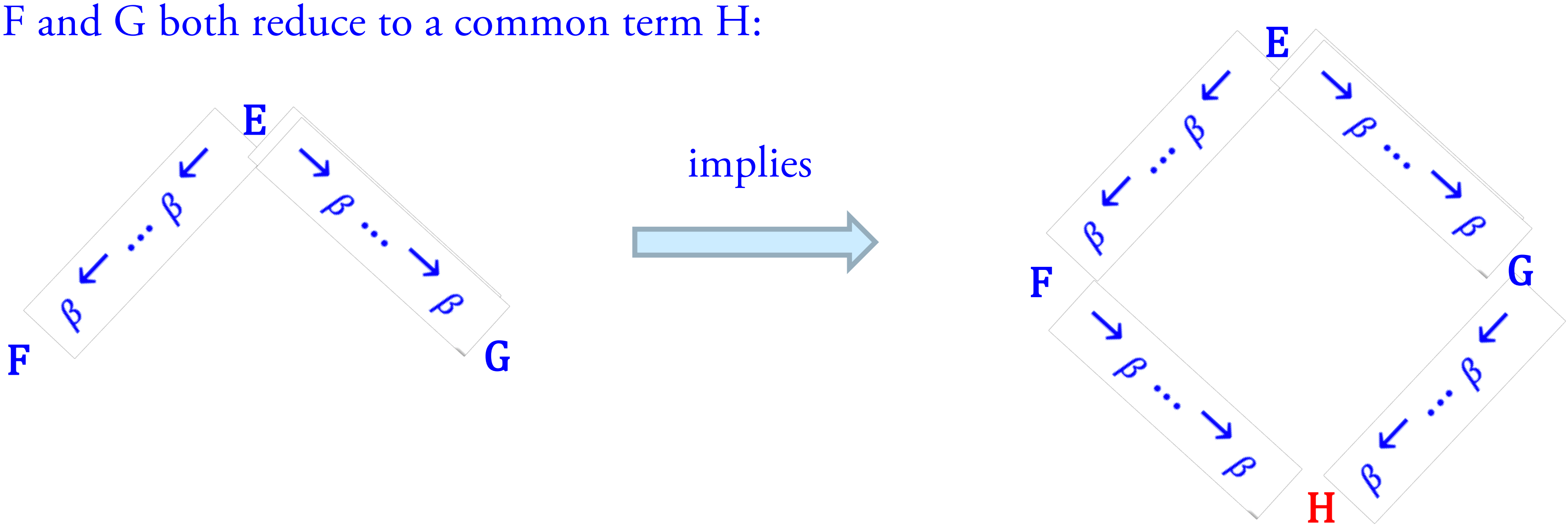
Issue (2): What are the consequences of choosing one strategy over the other?

(i) If there is any reduction sequence which terminates in a normal form, Normal Order will find one (which is why it is called "normal" order, since it finds normal forms).

(ii) Applicative Order may not terminate, even when there does exist some terminating sequence. ${}_{\beta}\!\leftarrow \dots {}_{\beta}\!\leftarrow$

(iii) Beta-reduction is confluent, so when normal forms exist, they are unique:

Confluence: If E reduces to two different expressions F $\neq G$,
then F and G both reduce to a common term H:



Lambda Calculus: Properties of Beta-Reduction

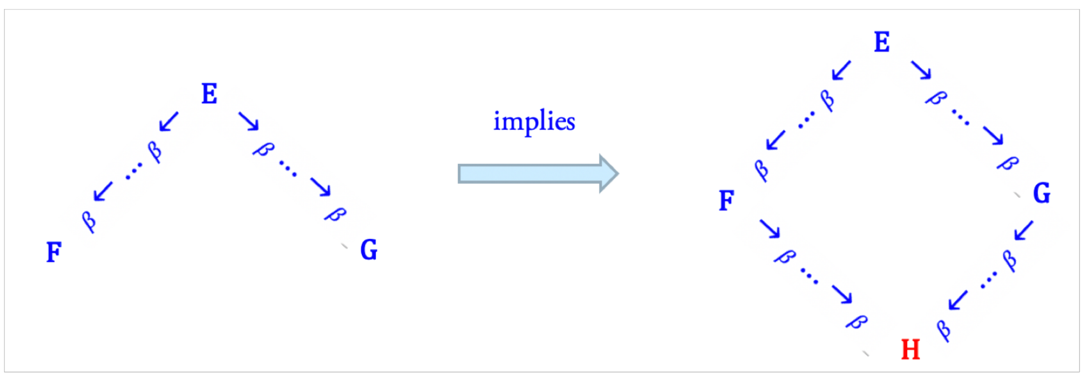
Issue (2): What are the consequences of choosing one strategy over the other?

(i) If there is any reduction sequence which terminates in a normal form, Normal Order will find one (which is why it is called "normal" order, since it finds normal forms).

(ii) Applicative Order may not terminate, even when there does exist some terminating sequence. $_\beta\!\leftarrow \ldots {}_\beta\!\leftarrow$

(iii) Beta-reduction is confluent, so when normal forms exist, they are unique:

Punchline: Normal Order will find a unique normal form when one exists; Applicative Order may not terminate, even when a normal form exists, but if it does, then that normal form is unique.



If F and G are normal forms, then F = H = G.

Evaluation Order in Programming Languages

So this means that except for the problem of non-termination, you will always get the same answer, no matter what strategy you use (even when we add arithmetic and other computational processes):

```
(\x ->  x * 3) ((\y -> y + 2) 5 )
```

normal order

applicative order

```
((\y -> y + 2) 5 ) *  3
```

```
(\x ->  x * 3)  (5 + 2)
```

```
(5 + 2) *  3
```

```
(\x ->  x * 3) 7
```

```
7 * 3
```

```
21
```

Evaluation Order in Programming Languages

Most languages use applicative/strict evaluation for function calls, so for example in Python we would have the following sequence of events:

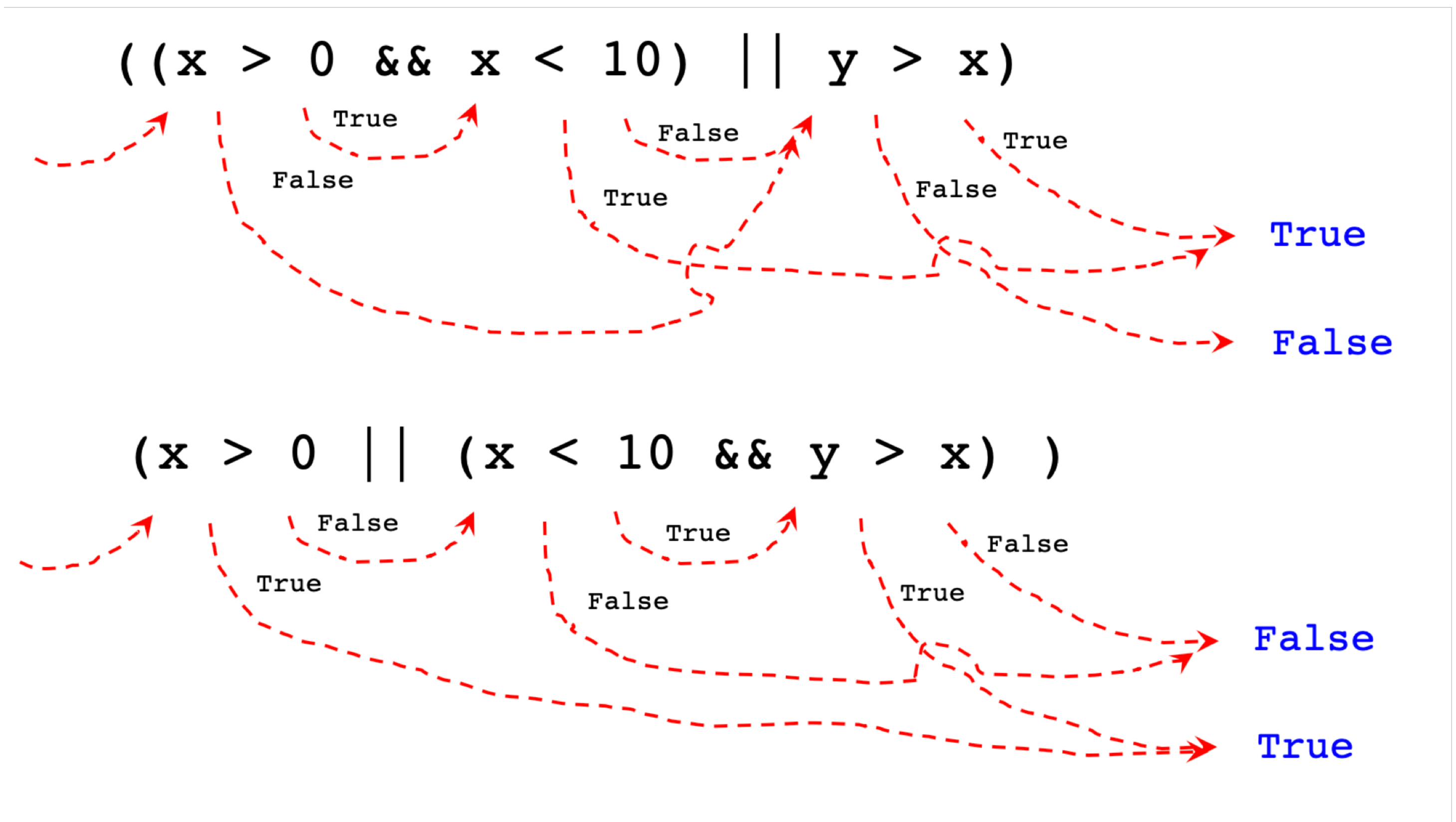
```
def times3(x):                          def plus2(y):
   return x * 3                            return y + 2
```

```
times3( plus2 ( 5 ) )          (\x ->  x * 3) ((\y -> y + 2) 5 )

evaluate 5

pass parameter to plus2:       (\x ->  x * 3) (5 + 2)
       y = 5

evaluate 5 + 2

return value 7                 (\x ->  x * 3) 7

pass parameter to times3:      7 * 3
       x = 7

evaluate 7 * 3

return 21                      21
```

Evaluation Order in Programming Languages

However, most languages also use some non-strict evaluation strategies, especially in two cases: Boolean operators and conditionals (if-then-else).

"Short-Circuit" Evaluation of Boolean Expressions:



Evaluation Order in Programming Languages

However, most languages also use some non-strict evaluation strategies, especially in two cases: Boolean operators and conditionals (if-then-else).

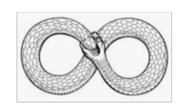
"Lazy" Evaluation of If-Then-Else:

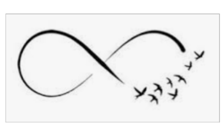
```
if A then B else C        --evaluate A, then evaluate
                          --only one of B or C
```

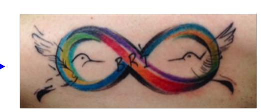
```
def ohNo(x):
     return ohNo(x)

def test(x):
   if x > 0:
      return "Positive!"
   else:
      return ohNo(x)
```

```
def cond(A,B,C):
   if A:
      return B
   else:
      return C
```

`test(10) => "Positive!"`

`cond( 10>0,10,ohNo(10)) =>` 

`test(-1) =>` 

`cond( -1>0,-1,ohNo(-1) )=>` 

Evaluation Order in Programming Languages

Haskell uses a version of Normal Order, called Lazy Evaluation, in which evaluation is ONLY done when absolutely necessary.

```
times3 x = x * 3                plus2 y = y + 2
```

```
(times3 (plus2 5))    =   (\x ->  x * 3) ((\y -> y + 2) 5 )

                      =   (((\y -> y + 2) 5 ) * 3)

                      =   ((5 + 2) * 3)

                      =   (7 * 3)

                      =   21
```

This is normal order evaluation.

Evaluation Order in Programming Languages

But there is a serious efficiency problem with normal order: expressions may be duplicated and have to be evaluated multiple times:

```
square3 x = x * x * 3                    plus2 y = y + 2
```

```
(square3 (plus2 5))

        =  (\x -> x * x * 3) ((\y -> y + 2) 5 )

        =  (((\y -> y + 2) 5 ) * ((\y -> y + 2) 5 )  * 3)

        =  ((5 + 2) * ((\y -> y + 2) 5 )  * 3)

        =  (7 * ((\y -> y + 2) 5 ) * 3)

        =  (7 * (5 + 2) * 3)

        =  (7 * 7 * 3)

        =  (49 * 3) =  147
```

Evaluation Order in Programming Languages

Lazy evaluation fixes this by creating a temporary variable bound to the expression (called a "thunk") which is then only evaluated once:

```
square3 x = x * x * 3                plus2 y = y + 2

(square3 (plus2 5))

        = (\x -> x * x * 3) ((\y -> y + 2) 5 )

        =  (thunk * thunk * 3)
                       where thunk = ((\y -> y + 2) 5 )

        =  (thunk * thunk * 3)
                       where thunk = (5 + 2)

        =  (thunk * thunk * 3)
                       where thunk = 7

        =  (7 * thunk * 3) where thunk = 7

        =  (7 * 7 * 3) = (49 * 3) =  147
```

This is the programming language version of "memoizing."

Recall: Let and Where Expressions in Haskell

`let` and **`where`** expressions allow you to create local variables and avoid having to write lots of helper functions.

The parameters in a lambda expression are local variables which only have meaning inside the body of the lambda expression:

```
(\x -> x + 2*x - 1)
```

Scope of **x**

This is a familiar concept in programming languages:

```
def area(r):
      pi = 3.1415
      return pi * r * r
```

Scope of **r**

Scope of **pi**

In Haskell this is done using the **`let`** expression:

```
let <bindings> in <expression>
```

Scope of bindings

Let and Simultaneous Equations

Lazy evaluation in Haskell means that no expression is evaluated until it absolutely has to be. So in a let, nothing is evaluated until the variable has to be used; the net result is that equations in a let are "simultaneous" and order does not matter:

```
cylinder r h =
  let pi       = 3.1415
      sideArea = 2 * pi * r * h
      topArea  = pi * r^2
  in sideArea + 2 * topArea

cylinder r h =
  let sideArea = 2 * pi * r * h
      pi       = 3.1415
      topArea  = pi * r^2
in sideArea + 2 * topArea

cylinder r h =
  let sideArea = 2 * pi * r * h
      topArea  = pi * r^2
      pi       = 3.1415
  in sideArea + 2 * topArea
```

All these do exactly the same thing!

This is another example of how Haskell follows mathematical practice, not imperative programming.

Let and where in detail: how are bindings evaluated?

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)
```

Let and where in detail: how are bindings evaluated?

The same thing is true of equations in your code:

```haskell
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
```

Let and where in detail: how are bindings evaluated?

This leads to the following behavior with sets of bindings that have no solution as a set of simultaneous equations:

```haskell
import Prelude

x = 4
x = x + 1
```

```
*Main> :r
[1 of 1] Compiling Main

Main.hs:4:1: error:
    Multiple declarations of ‘x’
    Declared at: Main.hs:3:1
                 Main.hs:4:1
  |
4 | x = x + 1
  | ^
Failed, no modules loaded.
Prelude>
```

```haskell
import Prelude


x = x + 1
```

```
Prelude> :r
[1 of 1] Compiling Main                ( Main.
Ok, one module loaded.
*Main> "Wayne, noooooo, don\'t do it...."
"Wayne, noooooo, don't do it...."
*Main> x
^CInterrupted.
*Main>
*Main>
*Main>
```

Gets into infinite digression trying to figure out value of x! Had to hit **control-c** to stop it.

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 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. ad **infinitum**....

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; we only evaluate them when we have to:

```
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 )      -- look up x and substitute
eval( 2 * z )
   eval( z )
   eval( y + 1 )
      eval( y )
      => 4
   => 5
=> 10
```