Using `ghci`. Interactivity, directives.

Expressions for each of the basic data types: Integer, Float, Bool, Char, () — unit.

Tuples. Lists: arithmetic sequences, list comprehensions.

Bindings. The `let` expression.

Simple functions. Anonymous functions, functions as data, special declaration syntax, patterns, cases, guards.

Using `ghc`. Writing, compiling, and running a main program, interact.
Learning Haskell

- All things Haskell: haskell.org
- Tutorial: Learn You a Haskell for Great Good! by Miran Lipovača
- Searching for functions: Hoogle
Learning Haskell

- Higher-order functions
- Data structures
- Type inference
Using GHCi

```
$ ghci
```
Using GHCi

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```
Using GHCi

$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude> "Hello, " ++ "World!"
Using GHCi

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude> "Hello, " ++ "World!"
"Hello, World!"
Prelude> 
```
GHCI directives

$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCI configuration from /home/ryan/.ghci

λ> :help
Commands available from the prompt:

<statement> evaluate/run <statement>
: repeat last command
:{\n ..lines.. \n:}\n multiline command
:add [*]<module> ... add module(s) to the current target set
:browse[!] [[*]<mod>] display the names defined by module <mod>
(!!: more details; *: all top-level names)
:cd <dir> change directory to <dir>
:cmd <expr> run the commands returned by <expr>::IO String
:complete <dom> [<rng>] <s> list completions for partial input string
:ctags[!] [<file>] create tags file <file> for Vi (default: "tags")
GHCI directives

<statement>
:
:{\n ..lines.. \n:}\n:eval/\run <statement>
:repeat last command
:{\n ..lines.. \n:}
:multiline command
:add [*]<module> ...
:add module(s) to the current target set
:browse[!] [[*]<mod>]
;display the names defined by module <mod>
(!: more details; *: all top-level names)
:cd <dir>
:change directory to <dir>
:complete <dom> [rng] <s>
:list completions for partial input string
:help, ?
:display this list of commands
:info\[<name> ...]
:display information about the given names
:kind <type>
:show the kind of <type>
:load [*]<module> ...
:load module(s) and their dependents
:main [arguments] ...
:run the main function with the given arguments
:module [+/-] [*]<mod> ...
:set the context for expression evaluation
:quit
:exit GHCi
:reload
:reload the current module set
:type <expr>
:show the type of <expr>
:type +d <expr>
:show the type of <expr>, defaulting type variables
Interactive development

load — edit — reload — test; and repeat
Overview to skim quickly, or
Summary for later
Data

Integers

\[ \langle \text{integer} \rangle ::= \langle \text{digit} \rangle^+ \]

Tuples

\[ \langle \text{tuple} \rangle ::= (\langle \text{expr} \rangle, \langle \text{expr} \rangle) \]

Lists

\[ \langle \text{list} \rangle ::= [\ ] \]

\[ ::= \langle \text{expr} \rangle : \langle \text{list} \rangle \]

Functions

\[ \langle \text{function} \rangle ::= \backslash \langle \text{name} \rangle \rightarrow \langle \text{expr} \rangle \]

\[ ::= (\langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle) \]

\[ ::= (\langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle) \]
Integers

\[ \langle \text{integer} \rangle ::= \langle \text{digit} \rangle^+ \]
Some Integers

0

42

3

7

5429723947

567
The Type of Integers

Integer

Integer

Integer

Integer

Integer

Integer
Tuples

\[
\langle \text{tuple} \rangle ::= (\langle \text{expr} \rangle, \langle \text{expr} \rangle)
\]

 ::= (\langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle)

 ::= (\langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle, \langle \text{expr} \rangle)
\]
(5, 3+4)

(’z’, 1, 1, 1)

(True, 2, False, 3)

(’a’, True)

(4*7, (’a’, True), ’p’)

(2+3, 2*3, 2-3)
Lists

\[
\langle list \rangle ::= []
\]

::= \langle expr \rangle : \langle list \rangle
1:1: []

'a': 'b': 'c': []

():():[]

3:5:(3+6):7:[]

False:False:True:[]

True:False:True:[]
[Integer]

[Char] = String

[()]

[Integers]

[Bool]

[Bool]
Lists Have Special Syntax

[1,1]

['a','b','c'] = "abc"

[()(),()]

[3,5,3+6,7]

[False,False,True]

[True,False,True]
Functions

\[
\langle function \rangle ::= \langle name \rangle \rightarrow \langle expr \rangle
\]

- lambda
- formal parameter
- body
\[ xs \rightarrow 2 : xs \]

\[ i \rightarrow \max i 0 \]

\[ s \rightarrow s++" , "++s \]

\[ i \rightarrow i+2 \]

\[ x \rightarrow \sin (x+\pi) \]

\[ c \rightarrow \text{ord } c \]
[Integer] -> [Integer]

Integer -> Integer

String -> String

Integer -> Integer

Double -> Double

Char -> Integer
Data, all data exists apart from names. Data

meaningOfLive = 42

myVerySpecialPair = ('a', 0)

myShortList = [True, False, True]

add2 = \ i -> i+2

cons2 = \ xs -> 2:xs

double = \ s -> s ++ "", " " ++ s
Function Declarations Have Special Syntax

\[
\text{cons2 } xs = 2:xs \\
\text{cutOff } i = \text{max } i 0 \\
\text{doubleWord } s = s ++ ", " ++ s \\
\text{add2 } i = i + 2 \\
\text{f } x = \sin (x + \pi) \\
\text{toOrd } c = \text{ord } c
\]
Complex Canonical Values

(2, ('a', 5), "abc", True) :: (Integer, (Char, Integer), [Char], Bool)
([1], ('a', 5), "abc", True, "xyz") :: ([Integer], (Char, Integer), [Char], Bool, [Char])
(
  \i->i+2 , \x->sind(x+pi) 
) :: (Integer->Integer, Double->Double)
([1, True, "abc"), (2, True, "mno"), (3, True, "xyz")]) :: 
\ i -> (i,i) :: Integer -> (Integer,Integer)
\ p -> (fst p + snd p, fst p * snd p, fst p - snd p)
End of the quick overview
(proceed to basics), or
End of summary
(proceed to writing a program with GHC)
The Basics

1. Primitive: Integer, floating-point, character, boolean, unit
2. Structures: Tuple, lists, functions
3. Text: list of character
Integer

143
succ 34
3+4
5*7
9-4
negate 8

---- watch out: 3 * -8 NO!
3 * (-8)
Integer

\[
\begin{align*}
\text{min} &\quad 17 & 34 \\
\text{max} &\quad 17 & 34 \\
\text{div} &\quad 24 & 7 \\
24 &\div 7 \\
\text{rem} &\quad 24 & 7 \\
\text{mod} &\quad 36 & 5 \\
&\quad \rightarrow & 1 \\
\text{quot} &\quad 36 & 5 \\
&\quad \rightarrow & 7 \\
\text{div} &\quad 36 & 5 \\
&\quad \rightarrow & 7 \\
\text{quot} &\quad 36 & (-5) \\
&\quad \rightarrow & -7 \\
\text{div} &\quad 36 & (-5) \\
&\quad \rightarrow & -8
\end{align*}
\]
Integer

\texttt{quotRem} 36 (-5) \\
\hspace{1cm} \longrightarrow (-7, 1)

\texttt{divMod} 36 -5 \\
\hspace{1cm} \longrightarrow (-8, -4)
--- pi, exp, sqrt, log, (**), sin, tan, cos, truncate, ceiling, round, truncate, floor
3.2 + 43.1
5.2 * 43.2345236
9.9 - 2.3
2345.2345 / 34.34
34.4 ^ 34
4254 ** 4.345
sqrt (4.59)
sin (1.7172)
Char

'a'

```
--- :browse Data.Char
ord :: Char -> Int
chr :: Int -> Char
digitToInt :: Char -> Int
intToDigit :: Int -> Char
toLower :: Char -> Char
toUpper :: Char -> Char
isAlphaNum :: Char -> Bool
isDigit :: Char -> Bool
isAlpha :: Char -> Bool
isLower :: Char -> Bool
isUpper :: Char -> Bool
isSpace :: Char -> Bool
```
Bool

False
True
False && True
False || True
not True
-- otherwise is defined to be the value True
-- for the purposes of making guards more readable
otherwise

if True then 4 else 5

-- important predicates
-- (==), (/=), (<), (>=), (>), (<=), compare
Tuples

(2’a’)
(4.34, 'a', 456)
(True, (), 1)
(2, "ab", 2*2, 5.0)
(2) -- not a tuple
(((()))) -- unit

fst (2,’b’)
snd (1,’a’)

(2,(’a’,3,4),"abcd")
((2,3,4), (True, 3.3))
What is a List?
How do you make a List in Haskell?

Two constructors “nil” and “cons”

[]

1 : []

1 : (2 : [])

1 : (2 : (3 : []))

1 : (2 : (3 : 4 : []))
How do you make a List in Haskell?

“Cons” is right associative; and, anyway, lists have special syntax.

[]
1:
1 : [ ]
1 : (2 : [ ])
1 : (2 : (3 : [ ]))
1 : (2 : (3 : 4 : [ ]))

[]
1:
1:2:
1:2:3:
1:2:3:4:

[ ]
1:
1:2:
1:2:3:
1:2:3:4:

[]
1:
1:2:
1:2:3:
1:2:3:4:

[1]
[1,2]
[1,2,3]
[1,2,3,4]

Ryan Stansifer  (CS, Florida Tech)        Haskell Lists        March 28, 2019           34 / 76
Cons 1 (Cons 2 Nil)

1:2:Nil

[1,2]
\[ [1, 2, 3, 4] \]
What can you do with a list?

head [1,2,3,4]
tail [1,2,3,4]
null [1,2,3,4]
Lists are polymorphic but homogeneous.

[1, 2, 3, 4]

[‘a’, ‘b’, ‘c’]

[(1, ‘a’), (2, ‘b’), (3, ‘c’)]

[[], [1], [1, 2, 3, 4], [2, 3]]
There are no mutable arrays in Haskell, so lists are the “go to” data structure for collections.

Every list function one can think up has been pre-defined in Haskell. See lists at Wiki Haskell.

```
length
(++)   -- append
elem   -- member
 (!! ) -- get element
concat -- flatten
last init
splitAt
take drop
sort nub
reverse
sum product
minimum maximum
and or
all any
```
List and Arithmetic Sequences

[-1, -1 .. 0]
[3, 2 .. 8]
[0, 2 .. 1]
[1, 1 .. 1]
[3, 3 .. (-4)]
[2, 1 .. (-4)]
[3, 3 .. (-12)]
[-1, -1 .. 8]
[-6, -6 .. 12]
[1, 2 .. (-12)]
[-6, -5 .. (-4)]
[1, 0 .. 5]
[-6, -6 .. 8]
[-1, -2 .. (-4)]
[1, 2 .. 5]
List Comprehension

Video; 31 minutes
Presenter: E Meijer
based on Hutton’s book, chapter 5
Channel 9 lectures at YouTube
List Comprehension

Defining sets by their properties is sometimes known as set comprehension and found often in mathematical texts. For example,

\[ \{ x \in \mathbb{R} \mid x > 0 \} \]

In mathematical notation we write

\[ \{ x^2 \mid x \in \{1, 2, \ldots, 5\} \} \]

to mean the set \( \{1, 4, 9, 16, 25\} \).

In Haskell a similar syntax is used for lists. The special square brackets syntax is expanded to include generators and filters.
Haskell List Comprehension

A list comprehension has the form:

\[
[expr | qual_1, \ldots, qual_n]
\]

where \( 1 \geq n \)

There are three types qualifiers

- generators of the form \( pat \leftarrow expr \), where \( p \) is a pattern (see Section 3.17) of type \( t \) and \( e \) is an expression of type \([t]\)
- local bindings that provide new definitions for use in the generated expression \( expr \) or subsequent boolean guards and generators
- boolean guards, which are arbitrary expressions of type \( \text{Bool} \).

See the section on List Comprehensions \( \uparrow \) in the Haskell 2010 Language Report \( \uparrow \).

Also, there are intriguing language extensions:

- ParallelListComp. Syntax: \[ expr | \text{qualifier}, \ldots | \text{qualifier}, \ldots \]
- TransformListComp. Three new keywords: group, by, and using.

See a paper \( \uparrow \) by Simon Peyton Jones.
Upcoming script

\[
\begin{align*}
[ & n \mid n \leftarrow [1..5] ] \\
[ & (n+2, 5 \times n) \mid n \leftarrow [1..3] ] \quad -- \text{list of pairs} \\
: & \text{mod} + \text{Data.Char} \\
[ & \text{toUpper} \; c \mid c \leftarrow "\text{one fish}" ]
\end{align*}
\]
List Comprehension

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> 
```
List Comprehension

```
λ> [ n | n<-[1..5] ]
```
List Comprehension

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> [ n  |  n<-[1..5] ]
[1,2,3,4,5]

λ>  
```
List Comprehension

```haskell
λ> [ n | n<-[1..5] ]
[1,2,3,4,5]

λ> [ (n+2,5*n) | n<-[1..3] ]
[5,13,27]
```
List Comprehension

```haskell
λ> [ n | n<-[1..5] ]
[1,2,3,4,5]

λ> [(n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]
```
List Comprehension

```
λ> [ n | n<-[1..5] ]
[1,2,3,4,5]

λ> [(n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> [(n+2,5*n) | n<-[1..3] ]
```
List Comprehension

```
λ> [ n | n<-[1..5] ]
[1,2,3,4,5]

λ> [ (n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> [ (n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]
```
List Comprehension

```
λ> [ n | n<-[1..5] ]
[1,2,3,4,5]

λ> [(n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> [(n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> [(n+2,5*n) | n<-[1..3] ]
```

List Comprehension

```
λ> [ n | n<-[1..5] ]
[1,2,3,4,5]

λ> [ (n+2, 5*n) | n<-[1..3] ]
[(3,5), (4,10), (5,15)]

λ> [ (n+2, 5*n) | n<-[1..3] ]
[(3,5), (4,10), (5,15)]

λ> [ (n+2, 5*n) | n<-[1..3] ]
[(3,5), (4,10), (5,15)]
```

List Comprehension

```
λ> [ n | n<-[1..5] ]
[1,2,3,4,5]

λ> [ (n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> [ (n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> [ (n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> :mod + Data.Char

λ> [ toUpper c | c <- "one fish" ]
```
List Comprehension

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

λ> [ (n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> [ (n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> [ (n+2,5*n) | n<-[1..3] ]
[(3,5),(4,10),(5,15)]

λ> :mod + Data.Char

λ> [ toUpper c | c <- "one fish" ]
"ONE FISH"

λ>
```
List Comprehension (Multiple Generators)

```haskell
λ> 
```

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci
```
List Comprehension (Multiple Generators)

```
λ> [(i,j) | i<-[1..3], j<-[1..2]]
```
List Comprehension (Multiple Generators)

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> [(i,j) | i<-[1..3], j<-[1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]

λ> 
```
List Comprehension (Multiple Generators)

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> [(i,j) | i<-[1..3], j<-[1..2] ]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]

λ> [(i,c) | i<-[1..2], c<-'abc' ]

```

Ryan Stansifer  (CS, Florida Tech)  Haskell Lists  March 28, 2019  46 / 76
List Comprehension (Multiple Generators)

```
λ> [(i,j) | i<-[1..3], j<-[1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]

λ> [(i,c) | i<-[1..2], c<-'abc']
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]
```
List Comprehension (Multiple Generators)

```haskell
λ> [(i,j) | i<-[1..3], j<-[1..2]]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]

λ> [(i,c) | i<-[1..2], c<"abc"]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]}

λ> [(i,c) | c<"abc", j<-[1..2]]
```
List Comprehension (Multiple Generators)

```
λ> [ (i,j) | i<-[1..3], j<-[1..2] ]
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]

λ> [ (i,c) | i<-[1..2], c<-"abc" ]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c')]  

λ> [ (i,c) | c<-"abc", j<-[1..2] ]
<interactive>:3:4: error: Variable not in scope: i

λ> [ (i,c) | c<-"abc", i<-[1..2] ]
```
List Comprehension (Multiple Generators)

\[
\lambda> \ [ (i,j) \mid i\in[1..3], j\in[1..2] ]
[ (1,1),(1,2),(2,1),(2,2),(3,1),(3,2) ]
\]

\[
\lambda> \ [ (i,c) \mid i\in[1..2], c\in"abc" ]
[ (1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c') ]
\]

\[
\lambda> \ [ (i,c) \mid c\in"abc", j\in[1..2] ]
\]

<interactive>:3:4: **error**: Variable not in scope: i

\[
\lambda> \ [ (i,c) \mid c\in"abc", i\in[1..2] ]
[ (1,'a'),(2,'a'),(1,'b'),(2,'b'),(1,'c'),(2,'c') ]
\]
List Comprehension (Multiple Generators)

```haskell
λ> 
```

```bash
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci
```
List Comprehension (Multiple Generators)

```haskell
λ> nouns = ["hobo","frog","pope"]
λ> adjectives = ["lazy","grouchy","scheming"]
λ> 
```
List Comprehension (Multiple Generators)

```haskell
λ> nouns = ["hobo","frog","pope"]
λ> adjectives = ["lazy","grouchy","scheming"]
λ> [adjective++" ++noun | adjective<-adjectives, noun<-nouns]
```
List Comprehension (Multiple Generators)

```haskell
λ> nouns = ["hobo","frog","pope"]
λ> adjectives = ["lazy","grouchy","scheming"]
λ> [adjective++" +noun | adjective<-adjectives, noun<-nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog","grouchy pope","scheming hobo","scheming frog","scheming pope"]
λ> 
```
List Comprehension (Infinite Lists, Nested)

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ>
```
List Comprehension (Infinite Lists, Nested)

```
λ> take 8 [ (i,j) | i<-[1,2], j<-[i..] ]
```
List Comprehension (Infinite Lists, Nested)

```
λ> take 8 [ (i,j) | i<-[1,2], j<-[i..] ]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]
```

```
List Comprehension (Infinite Lists, Nested)

```
λ> take 8 [ (i,j) | i<-[1,2], j<-[i..] ]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]

λ> take 8 [ (i,j) | j<-[3..], i<-[1,2] ]
```
List Comprehension (Infinite Lists, Nested)

```
λ> take 8 [(i,j) | i<-[1,2], j<-[i..]]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]

λ> take 8 [(i,j) | j<-[3..], i<-[1,2]]
[(1,3),(2,3),(1,4),(2,4),(1,5),(2,5),(1,6),(2,6)]
```
List Comprehension (Infinite Lists, Nested)

```haskell
λ> take 8 [ (i,j) | i<-[1,2], j<-[i..] ]
([(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]

λ> take 8 [ (i,j) | j<-[3..], i<-[1,2] ]
([(1,3),(2,3),(1,4),(2,4),(1,5),(2,5),(1,6),(2,6)]

λ> take 4 [ [ (i,j) | i <- [1,2] ] | j <- [1..] ]
```
List Comprehension (Infinite Lists, Nested)

```haskell
λ> take 8 [ (i,j) | i<-[1,2], j<-[i..] ]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]

λ> take 8 [ (i,j) | j<-[3..], i<-[1,2] ]
[(1,3),(2,3),(1,4),(2,4),(1,5),(2,5),(1,6),(2,6)]

λ> take 4 [ [ (i,j) | i <- [1,2] ] | j <- [1..] ]
[[[1,1),(2,1)],[[1,2),(2,2)],[[1,3),(2,3)],[[1,4),(2,4)]]

λ>  
```
List Comprehension (Infinite Lists, Nested)

```haskell
λ> take 8 [ (i,j) | i <- [1..2], j <- [i..] ]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]

λ> take 8 [ (i,j) | j <- [3..], i <- [1..2] ]
[(1,3),(2,3),(1,4),(2,4),(1,5),(2,5),(1,6),(2,6)]

λ> take 4 [ [ (i,j) | i <- [1..2] ] | j <- [1..] ]
[[[1,1],[1,2]],[[1,2],[2,2]],[[1,3],[2,3]],[[1,4],[2,4]]]

λ> take 8 $ concat [ [ (i,j) | i <- [1..2] ] | j <- [1..] ]
```

List Comprehension (Infinite Lists, Nested)

```
λ> take 8 [ (i,j) | i<-[1,2], j<-[1..] ]
[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8)]

λ> take 8 [ (i,j) | j<-[3..], i<-[1,2] ]
[(1,3),(2,3),(1,4),(2,4),(1,5),(2,5),(1,6),(2,6)]

λ> take 4 [ [ (i,j) | i <- [1,2] ] | j <- [1..] ]
[[[(1,1),(2,1)],[1,2),(2,2)],[1,3),(2,3)],[1,4),(2,4)]]

λ> take 8 $ concat [ [ (i,j) | i <- [1,2] ] | j <- [1..] ]
[(1,1),(2,1),(1,2),(2,2),(1,3),(2,3),(1,4),(2,4)]
```
List Comprehension (Guards aka Filters)

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> 
```
List Comprehension (Guards aka Filters)

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> [ n | n<-[50..100], n `mod` 7 == 3 ]
```
List Comprehension (Guards aka Filters)

```
λ> [ n | n<-[50..100], n `mod` 7 == 3 ]
[52,59,66,73,80,87,94]
```

List Comprehension (Guards aka Filters)

```
λ> [ n | n <- [50..100], n `mod` 7 == 3 ]
[52,59,66,73,80,87,94]

λ> [ n*n | n <- [1..22], even n ]
```
List Comprehension (Guards aka Filters)

```
λ> [ n | n<-[50..100], n `mod` 7 == 3 ]
[52,59,66,73,80,87,94]

λ> [ n*n | n<-[1..22], even n ]
[4,16,36,64,100,144,196,256,324,400,484]
```

Ryan Stansifer (CS, Florida Tech)  Haskell Lists  March 28, 2019  49 / 76
List Comprehension (Guards aka Filters)

```
λ> [ n | n<-[50..100], n `mod` 7 == 3 ]
[52,59,66,73,80,87,94]

λ> [ n*n | n<-[1..22], even n ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ n*n | n<-[2,4..22] ]
```
List Comprehension (Guards aka Filters)

```haskell
λ> [ n | n<-[50..100], n `mod` 7 == 3 ]
[52,59,66,73,80,87,94]

λ> [ n*n | n<-[1..22], even n ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ n*n | n<-[2,4..22] ]
[4,16,36,64,100,144,196,256,324,400,484]

λ>  
```
List Comprehension (Guards aka Filters)

```
ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCI configuration from /home/ryan/.ghci

λ> [ n | n<-[50..100], n `mod` 7 == 3 ]
[52,59,66,73,80,87,94]

λ> [ n*n | n<-[1..22], even n ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ n*n | n<-[2,4..22] ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ (n,n*n) | n<-[3..3], n<n*n ]
```

Ryan Stansifer  (CS, Florida Tech)  Haskell Lists  March 28, 2019
List Comprehension (Guards aka Filters)

```
λ> [ n | n<-[50..100], n `mod` 7 == 3 ]
[52,59,66,73,80,87,94]

λ> [ n*n | n<-[1..22], even n ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ n*n | n<-[2,4..22] ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ (n,n*n) | n<-[3..3], n<n*n ]
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)]
```

List Comprehension (Guards aka Filters)

```
λ> [ n | n<-[50..100], n `mod` 7 == 3 ]
[52,59,66,73,80,87,94]

λ> [ n*n | n<-[1..22], even n ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ n*n | n<-[2,4..22] ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ (n,n*n) | n<-[-3..3], n<n*n ]
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)]

λ> [ n | n <- [10..20], n /= 13, n /= 15, n /= 19 ]
```
List Comprehension (Guards aka Filters)

\[
\lambda\> [ \ n \mid n\text{-}[50..100], n \mod 7 == 3 ]
\]
[52, 59, 66, 73, 80, 87, 94]

\[
\lambda\> [ n*n \mid n\text{-}[1..22], \text{even n} ]
\]
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484]

\[
\lambda\> [ n*n \mid n\text{-}[2,4..22] ]
\]
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484]

\[
\lambda\> [ (n, n*n) \mid n\text{-}[-3..3], n<n*n ]
\]
[(-3, 9), (-2, 4), (-1, 1), (2, 4), (3, 9)]

\[
\lambda\> [ n \mid n \text{-}[10..20], n /= 13, n /= 15, n /= 19 ]
\]
[10, 11, 12, 14, 16, 17, 18, 20]
List Comprehension (Guards aka Filters)

\[
\lambda > \ [ \ n \mid n<-[50..100], \ n \ `mod` \ 7 == 3 ] \\
[52,59,66,73,80,87,94]
\]

\[
\lambda > \ [ \ n*n \mid n<-[1..22], \ even \ n ] \\
[4,16,36,64,100,144,196,256,324,400,484]
\]

\[
\lambda > \ [ \ n*n \mid n<-[2,4..22] ] \\
[4,16,36,64,100,144,196,256,324,400,484]
\]

\[
\lambda > \ [ \ (n,n*n) \mid n<-[-3..3], \ n<n*n ] \\
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)]
\]

\[
\lambda > \ [ \ n \mid n<-[10..20], \ n /= 13, \ n /= 15, \ n /= 19 ] \\
[10,11,12,14,16,17,18,20]
\]

\[
\lambda > \ [ \ c \mid c <- \ "one fish", \ c `elem` \ "aeiou"]
\]
List Comprehension (Guards aka Filters)

λ> [ n*n | n<-[1..22], even n ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ n*n | n<-[2,4..22] ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ (n,n*n) | n<-[-3..3], n<n*n ]
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)]

λ> [ n | n <- [10..20], n /= 13, n /= 15, n /= 19 ]
[10,11,12,14,16,17,18,20]

λ> [ c | c <- "one fish", c `elem` "aeiou"]
"oei"
List Comprehension (Guards aka Filters)

λ> [ n*n | n<-[1..22], even n ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ n*n | n<-[2,4..22] ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ (n,n*n) | n<-[-3..3], n<n*n ]
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)]

λ> [ n | n <- [10..20], n /= 13, n /= 15, n /= 19 ]
[10,11,12,14,16,17,18,20]

λ> [ c | c <- "one fish", c `elem` "aeiou"]
"oei"

λ> [if x<10 then "BOOM!" else "BANG!" | x<-[7..13], odd x]
List Comprehension (Guards aka Filters)

```
λ> [ n*n | n<- [2..22] ]
[4,16,36,64,100,144,196,256,324,400,484]

λ> [ (n,n*n) | n<- [-3..3], n<n*n ]
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)]

λ> [ n | n <- [10..20], n /= 13, n /= 15, n /= 19 ]
[10,11,12,14,16,17,18,20]

λ> [ c | c <- "one fish", c `elem` "aeiou"]
"oei"

λ> [if x<10 then "BOOM!" else "BANG!" | x<- [7..13], odd x]
"BOOM!","BOOM!","BANG!","BANG!"

λ> 
```
List Comprehension and Functions

```haskell
λ> 
```
List Comprehension and Functions

```
λ> spaces n = [ ' ' | i <- [1..n] ]
```
List Comprehension and Functions

```haskell
λ> spaces n = [ ' ' | i <- [1..n] ]
λ> space 3
```

List Comprehension and Functions

```
λ> spaces n = [ ' ' | i <- [1..n] ]
λ> space 3

<interactive>:2:1: error:
  • Variable not in scope: space :: Integer -> t
  • Perhaps you meant ‘spaces’ (line 1)

λ> spaces 3
```
List Comprehension and Functions

```
λ> spaces n = [ ' ' | i <- [1..n] ]
λ> space 3

<interactive>:2:1: error:
  • Variable not in scope: space :: Integer -> t
  • Perhaps you meant ‘spaces’ (line 1)

λ> spaces 3
"   
λ> 
```
List Comprehension and Functions

```
λ> spaces n = [ '' | i <- [1..n] ]
λ> space 3

<interactive>:2:1:  error:
  • Variable not in scope: space :: Integer -> t
  • Perhaps you meant ‘spaces’ (line 1)
λ> spaces 3

λ> spaces 8
```

Ryan Stansifer  (CS, Florida Tech)  Haskell Lists  March 28, 2019  50 / 76
List Comprehension and Functions

```
Loaded GHCi configuration from /home/ryan/.ghci

λ> spaces n = [ ' ' | i <- [1..n] ]
λ> space 3
<interactive>:2:1: error:
  • Variable not in scope: space :: Integer -> t
  • Perhaps you meant ‘spaces’ (line 1)

λ> spaces 3
  "    "

λ> spaces 8
  "    "

λ>
```
List Comprehension and Functions

```
Loaded GHCi configuration from /home/ryan/.ghci

λ> spaces n = [ ' ' | i <- [1..n] ]
λ> space 3

<interactive>:2:1: error:
    • Variable not in scope: space :: Integer -> t
    • Perhaps you meant ‘spaces’ (line 1)

λ> spaces 3

```

```
λ> doublePos xs = [2*x | x<-xs, x>0]
```

List Comprehension and Functions

λ> spaces n = [ ' ' | i <- [1..n] ]

λ> space 3

<interactive>:2:1: error:
  • Variable not in scope: space :: Integer -> t
  • Perhaps you meant ‘spaces’ (line 1)

λ> spaces 3
  "   "

λ> spaces 8
  "   "

λ> doublePos xs = [2*x | x <- xs, x>0]
List Comprehension and Functions

```haskell
λ> spaces n = [ ' ' | i <- [1..n] ]
λ> space 3

<interactive>:2:1:  error:
  • Variable not in scope: space :: Integer -> t
  • Perhaps you meant ‘spaces’ (line 1)

λ> spaces 3
  "    

λ> spaces 8
  "    

λ> doublePos xs = [2*x | x<-xs, x>0]
λ> doublePos [-1,2,1,2,3]
```
List Comprehension and Functions

```
<interactive>:2:1: error:
  • Variable not in scope: space :: Integer -> t
  • Perhaps you meant ‘spaces’ (line 1)

λ> spaces 3
  "  "

λ> spaces 8
  "  "

λ> doublePos xs = [2*x | x<-xs, x>0]

λ> doublePos [-1,2,1,2,3]
  [4,2,4,6]
```

Ryan Stansifer (CS, Florida Tech)
List Comprehension and Functions

```haskell
λ> spaces 3
   "   "
λ> spaces 8
   "   "
λ> doublePos xs = [2*x | x<-xs, x>0]
λ> doublePos [-1,2,1,2,3]
   [4,2,4,6]
λ> doublePos [1,4..13]
   [2,8,10,12]  
```
List Comprehension and Functions

• Perhaps you meant ‘spaces’ (line 1)

> spaces 3

> spaces 8

> doublePos xs = [2*x | x<-xs, x>0]

> doublePos [-1,2,1,2,3]

> doublePos [1,4..13]

> doublePos [2,8,14,20,26]

>
List Comprehension and Functions

• Perhaps you meant ‘spaces’ (line 1)

λ> spaces 3
  "   
λ> spaces 8
  "   
λ> doublePos xs = [2*x | x<-xs, x>0]
λ> doublePos [-1,2,1,2,3]
  [4,2,4,6]
λ> doublePos [1,4..13]
  [2,8,14,20,26]
λ> doublePos [1,0..-78]

Terminal - ryan@hc210-059044:~
List Comprehension and Functions

```
λ> spaces 8
   
λ> doublePos xs = [2*x | x <- xs, x>0]
λ> doublePos [-1,2,1,2,3]
   [4,2,4,6]
λ> doublePos [1,4..13]
   [2,8,14,20,26]
λ> doublePos [1,0..-78]
<interactive>:8:15: error: Variable not in scope: (..-) :: Integer -> Integer -> a
λ> 
```
List Comprehension and Functions

\[
\lambda> \text{spaces 8} \\
\qquad " \quad " \\
\lambda> \text{doublePos} \; \text{xs} = [2\times x \mid x<-\text{xs}, x>0] \\
\lambda> \text{doublePos} \; [-1,2,1,2,3] \\
\qquad [4,2,4,6] \\
\lambda> \text{doublePos} \; [1,4..13] \\
\qquad [2,8,14,20,26] \\
\lambda> \text{doublePos} \; [1,0..-78] \\
\lambda> \text{doublePos} \; [1,0..(-78)] \\
\text{<interactive>:8:15: error: Variable not in scope: (..) :: Integer -> Integer -> a} \\
\lambda> \text{doublePos} \; [1,0..(-78)] \]
List Comprehension and Functions

```haskell
λ> doublePos xs = [2*x | x<-xs, x>0]
λ> doublePos [-1,2,1,2,3]
[4,2,4,6]
λ> doublePos [1,4..13]
[2,8,14,20,26]
λ> doublePos [1,0..-78]
<interactive>:8:15: error:
  Variable not in scope: (..-) :: Integer -> Integer -> a
λ> doublePos [1,0.. (-78)]
[2]
λ> 
```
List Comprehension and Functions

\[
\lambda\text{> doublePos } xs = [2*x \mid x<-xs, x>0]
\]

\[
\lambda\text{> doublePos } [-1,2,1,2,3]
\]
\[
[4,2,4,6]
\]

\[
\lambda\text{> doublePos } [1,4..13]
\]
\[
[2,8,14,20,26]
\]

\[
\lambda\text{> doublePos } [1,0..-78]
\]

\[
<\text{interactive}>:8:15: \text{error:}
  \text{Variable not in scope: (.-) :: Integer -> Integer -> a}
\]

\[
\lambda\text{> doublePos } [1,0..(-78)]
\]
\[
[2]
\]

\[
\lambda\text{> factors } n = [x \mid x<-[1..n], n \ `mod` x == 0]
\]
List Comprehension and Functions

```haskell
λ> doublePos [-1,2,1,2,3]
[4,2,4,6]

λ> doublePos [1,4..13]
[2,8,14,20,26]

λ> doublePos [1,0..-78]

<interactive>:8:15: error: Variable not in scope: (...) :: Integer -> Integer -> a

λ> doublePos [1,0..(-78)]
[2]

λ> factors n = [x | x<-[1..n], n `mod` x == 0]

λ>
List Comprehension and Functions

λ> doublePos [-1,2,1,2,3]
   [4,2,4,6]

λ> doublePos [1,4..13]
   [2,8,14,20,26]

λ> doublePos [1,0..-78]

<interactive>:8:15: error:
   Variable not in scope: (..-) :: Integer -> Integer -> a

λ> doublePos [1,0.. (-78)]
   [2]

λ> factors n = [x | x<-[1..n], n `mod` x == 0]

λ> factors 12
List Comprehension and Functions

\begin{verbatim}
λ> doublePos [1,4..13]
[2,8,14,20,26]

λ> doublePos [1,0..-78]
<interactive>:8:15: error:
    Variable not in scope: (..-) :: Integer -> Integer -> a

λ> doublePos [1,0.. (-78)]
[2]

λ> factors n = [x | x<-[1..n], n `mod` x == 0]

λ> factors 12
[1,2,3,4,6,12]
\end{verbatim}
List Comprehension and Functions

```haskell
λ> doublePos [1,0..-78]
<interactive>:8:15: error:  
    Variable not in scope: (..-) :: Integer -> Integer -> a
λ> doublePos [1,0.. (-78)]
[2]
λ> factors n = [x | x<-[1..n], n `mod` x == 0]
λ> factors 12
[1,2,3,4,6,12]
λ> factors 6
[1,2,3,6]
λ> 
```
List Comprehension and Functions

```
λ> doublePos [1,0..-78]

<interactive>:8:15: error: Variable not in scope: (..-) :: Integer -> Integer -> a
```

```
λ> doublePos [1,0.. (-78)]
[2]
```

```
λ> factors n = [x | x<-[1..n], n `mod` x == 0]
```

```
λ> factors 12
[1,2,3,4,6,12]
```

```
λ> factors 6
[1,2,3,6]
```

```
λ> factors 17
```

List Comprehension and Functions

```
λ> doublePos [1,0.. (-78)]
[2]

λ> factors n = [x | x<-[1..n], n `mod` x == 0]

λ> factors 12
[1,2,3,4,6,12]

λ> factors 6
[1,2,3,6]

λ> factors 17
[1,17]
```

Variable not in scope: (..-) :: Integer -> Integer -> a
Variable not in scope: (..-) :: Integer -> Integer -> a

\[ \lambda \text{doublePos} \, [1,0..(-78)] \]
\[ [2] \]

\[ \lambda \text{factors} \, n = [x \mid x<-[1..n], \, n \, `\text{mod}` \, x \, == \, 0] \]

\[ \lambda \text{factors} \, 12 \]
\[ [1,2,3,4,6,12] \]

\[ \lambda \text{factors} \, 6 \]
\[ [1,2,3,6] \]

\[ \lambda \text{factors} \, 17 \]
\[ [1,17] \]

\[ \lambda \text{factors} \, 0 \]
List Comprehension and Functions

```
[2]

λ> factors n = [x | x⇐[1..n], n `mod` x == 0]

λ> factors 12
[1,2,3,4,6,12]

λ> factors 6
[1,2,3,6]

λ> factors 17
[1,17]

λ> factors 0
[]
```
List Comprehension and Functions

```
[2]

λ> factors n = [x | x<-[1..n], n `mod` x == 0]

λ> factors 12
[1,2,3,4,6,12]

λ> factors 6
[1,2,3,6]

λ> factors 17
[1,17]

λ> factors 0
[]

λ> prime n = factors n == [1,n]
```
List Comprehension and Functions

```
λ> factors n = [x | x<-[1..n], n `mod` x == 0]
λ> factors 12
[1,2,3,4,6,12]
λ> factors 6
[1,2,3,6]
λ> factors 17
[1,17]
λ> factors 0
[]
λ> prime n = factors n == [1,n]
λ> 
```
List Comprehension and Functions

\[
\lambda > \text{factors } n = [x \mid x \leftarrow [1..n], n \mod x == 0]
\]

\[
\lambda > \text{factors } 12
\]
[1,2,3,4,6,12]

\[
\lambda > \text{factors } 6
\]
[1,2,3,6]

\[
\lambda > \text{factors } 17
\]
[1,17]

\[
\lambda > \text{factors } 0
\]
[]

\[
\lambda > \text{prime } n = \text{factors } n == [1,n]
\]

\[
\lambda > \text{prime } 12
\]
List Comprehension and Functions

\[
[1,2,3,4,6,12]
\]

\[
\lambda>\text{factors } 6
\]
\[
[1,2,3,6]
\]

\[
\lambda>\text{factors } 17
\]
\[
[1,17]
\]

\[
\lambda>\text{factors } 0
\]
\[
[]
\]

\[
\lambda>\text{prime } n = \text{factors } n == [1,n]
\]
\[
\lambda>\text{prime } 12
\]
False

\[
\lambda>
\]
List Comprehension and Functions

[1,2,3,4,6,12]

\lambda \text{ factors } 6
\text{[1,2,3,6]}

\lambda \text{ factors } 17
\text{[1,17]}

\lambda \text{ factors } 0
\text{[]}

\lambda \text{ prime } n \equiv \text{ factors } n \equiv [1,n]

\lambda \text{ prime } 12
\text{False}

\lambda \text{ prime } 53
List Comprehension and Functions

```haskell
[1,2,3,6]

λ> factors 17
[1,17]

λ> factors 0
[]

λ> prime n = factors n == [1,n]

λ> prime 12
False

λ> prime 53
True
```

Haskell Lists

March 28, 2019
List Comprehension and Functions

```
[1,2,3,6]

λ> factors 17
[1,17]

λ> factors 0
[]

λ> prime n = factors n == [1,n]

λ> prime 12
False

λ> prime 53
True

λ> primes n = [ i | i<-[2..n], prime i]
```
List Comprehension and Functions

```
λ> factors 17
[1,17]
λ> factors 0
[]
λ> prime n = factors n == [1,n]
λ> prime 12
False
λ> prime 53
True
λ> primes n = [ i | i<-[2..n], prime i]
λ>
```
List Comprehension and Functions

\[\lambda> \text{factors } 17\]
\[\text{[1,17]}\]

\[\lambda> \text{factors 0}\]
\[\text{[]}\]

\[\lambda> \text{prime n = factors n == [1,n]}\]

\[\lambda> \text{prime 12}\]
\[\text{False}\]

\[\lambda> \text{prime 53}\]
\[\text{True}\]

\[\lambda> \text{primes n = [ i | i<[2..n], prime i]}\]

\[\lambda> \text{primes 101}\]
List Comprehension and Functions

```
λ> factors 0
[]

λ> prime n = factors n == [1,n]

λ> prime 12
False

λ> prime 53
True

λ> primes n = [ i | i<-[2..n], prime i]

λ> primes 101

λ>
```
List Comprehension, Pythagorean Triples

ryan@hc210-059044:~$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ>
List Comprehension, Pythagorean Triples

```
λ> pythagorean n = [(x,y,z) | x<-[1..n], y<-[x..n], z<-[y..n], x*x+y*y==z*z]
```
List Comprehension, Pythagorean Triples

```haskell
λ> pythagorean n = [(x,y,z) | x<-[1..n], y<-[x..n], z<-[y..n], x*x+y*y==z*z]
λ>  
```
List Comprehension, Pythagorean Triples

\[
\text{pythagorean } n = \left\{ (x, y, z) \mid x \in [1..n], y \in [x..n], z \in [y..n], x^2 + y^2 = z^2 \right\}
\]

\[
\text{pythagorean } 12
\]
List Comprehension, Pythagorean Triples

```
λ> pythagorean n  = [ (x,y,z) | x<-[1..n],y<-[x..n],z<-[y..n],x*x+y*y==z*z ]
λ> pythagorean 12
[(3,4,5),(6,8,10)]
```
List Comprehension, Pythagorean Triples

```
λ> pythagorean n = [(x,y,z) | x<-[1..n], y<-[x..n], z<-[y..n], x*x+y*y==z*z]
λ> pythagorean 12
[(3,4,5),(6,8,10)]
λ> pythagorean 200
```
List Comprehension, Pythagorean Triples

\[
\{(24, 143, 145), (25, 60, 65), (26, 168, 170), (27, 36, 45), (27, 120, 123), (28, 45, 53), (28, 96, 100), (28, 195, 197), (30, 40, 50), (30, 72, 78), (32, 60, 68), (32, 126, 130), (33, 44, 55), (33, 56, 65), (33, 180, 183), (35, 84, 91), (35, 120, 125), (36, 48, 60), (36, 77, 85), (36, 105, 111), (36, 160, 164), (39, 52, 65), (39, 80, 89), (40, 42, 58), (40, 75, 85), (40, 96, 104), (42, 56, 70), (42, 144, 150), (44, 117, 125), (45, 60, 75), (45, 108, 117), (48, 55, 73), (48, 64, 80), (48, 90, 102), (48, 140, 148), (48, 189, 195), (49, 168, 175), (50, 120, 130), (51, 68, 85), (51, 140, 149), (52, 165, 173), (54, 72, 90), (55, 132, 143), (56, 90, 106), (56, 105, 119), (56, 192, 200), (57, 76, 95), (57, 176, 185), (60, 63, 87), (60, 80, 100), (60, 91, 109), (60, 144, 156), (60, 175, 185), (63, 84, 105), (64, 120, 136), (65, 72, 97), (65, 156, 169), (66, 88, 110), (66, 112, 130), (69, 92, 115), (70, 168, 182), (72, 96, 120), (72, 135, 153), (72, 154, 170), (75, 100, 125), (75, 180, 195), (78, 104, 130), (78, 160, 178), (80, 84, 116), (80, 150, 170), (81, 108, 135), (84, 112, 140), (84, 135, 159), (85, 132, 157), (87, 116, 145), (88, 105, 137), (88, 165, 187), (90, 120, 150), (93, 124, 155), (95, 168, 193), (96, 110, 146), (96, 128, 160), (99, 132, 165), (99, 168, 195), (100, 105, 145), (102, 136, 170), (104, 153, 185), (105, 140, 175), (108, 144, 180), (111, 148, 185), (114, 152, 190), (117, 156, 195), (119, 120, 169), (120, 126, 174), (120, 160, 200), (130, 144, 194)\}
\]

\[
\text{\texttt{\lambda\to pythagorean\;100}}
\]
List Comprehension, Pythagorean Triples

\[
(154, 170), (75, 100, 125), (75, 180, 195), (78, 104, 130), (78, 160, 178), (80, 84, 116), (80, 150, 170), (81, 108, 135), (84, 112, 140), (84, 135, 159), (85, 132, 157), (87, 116, 145), (88, 105, 137), (88, 165, 187), (90, 120, 150), (93, 124, 155), (95, 168, 193), (96, 110, 146), (96, 128, 160), (99, 132, 165), (99, 168, 195), (100, 105, 145), (102, 136, 170), (104, 153, 185), (105, 140, 175), (108, 144, 180), (111, 148, 185), (114, 152, 190), (117, 156, 195), (119, 120, 169), (120, 126, 174), (120, 160, 200), (130, 144, 194)
\]

\[
\lambda> \text{pythagorean 100}\\
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15), (9, 40, 41), (10, 24, 26), (11, 60, 61), (12, 16, 20), (12, 35, 37), (13, 84, 85), (14, 48, 50), (15, 20, 25), (15, 36, 39), (16, 30, 34), (16, 63, 65), (18, 24, 30), (18, 80, 82), (20, 21, 29), (20, 48, 52), (21, 28, 35), (21, 72, 75), (24, 32, 40), (24, 45, 51), (24, 70, 74), (25, 60, 65), (27, 36, 45), (28, 45, 53), (28, 96, 100), (30, 40, 50), (30, 72, 78), (32, 60, 68), (33, 44, 55), (33, 56, 65), (35, 84, 91), (36, 48, 60), (36, 77, 85), (39, 52, 65), (39, 80, 89), (40, 42, 58), (40, 75, 85), (42, 56, 70), (45, 60, 75), (48, 55, 73), (48, 64, 80), (51, 68, 85), (54, 72, 90), (57, 76, 95), (60, 63, 87), (60, 80, 100), (65, 72, 97)]\\
\lambda>
\]
Before continuing on the last and most important data value: functions, we consider the problem of giving data values names.

It is important psychologically to give the programmer a means of referring to data and not just creating data.
Every object of computation/value gets a name in the same way:

```haskell
theMeaningOfLife = 42
```

NB. Not assignment!

Control of scope using `let` expression.

```haskell
let <declaration> in <expression>
```

[We don’t need any examples now.]
Syntax of Names

Names consist of either letters and digits, or of all symbols. All symbolic names are treated as infix operators by the parser. All non-symbolic names are treated as prefix functions by the parser.

::
!!
++
*

div
quotRem
Integer
Syntax of Names

Enclosing a name with () tells the parser to drop the infix assumption. Enclosing a name with backquotes tells the parser to assume infix assumption.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Infix</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabetic</td>
<td>elem</td>
</tr>
<tr>
<td>symbolc</td>
<td>(+)</td>
</tr>
</tbody>
</table>

```
div 24 7
24 ‘div‘ 7

elem 5 [1,2,3,4,5,6]
5 ‘elem‘ [1,2,3,4,5,6]

(+) 2 3
2 + 3
```
Declarations

A name is given a value by a declaration of the simple form:

\[ \text{name} = \text{value} \]

In haskell this declaration is actually a specific case of a more general declaration of the form:

\[ \text{pattern "=" value} \]

Patterns (section 3.17.1 of the reference 2010 manual) are more naturally discussed later in the context of functions.

There are declarations for data types and classes. These will be discussed later.
Declarations

Many languages have declarations for many kinds of things: constants, variables, procedures, and functions.

Haskell does not talk about memory locations (however, see boxed versus unboxed), hence no need for variable declarations.

Fundamentally, given a datum a name in Haskell is the same for any value—functions included.
Overview of Simple Functions

1. syntax (lambda)
2. Two Haskell quirks
3. anonymous (functions as regular data)
4. special syntax
5. patterns (generalization of formal parameters)
6. definition by cases
7. guards [omit]
Syntax of Functions

\[ x \rightarrow 2 \times x \]
Syntax of Functions

- backslash; pronounced “lambda”
- identifier: name of formal parameter (but generalized to patterns later!)
- arrow separates the body of the function from the parameter
Before we can begin illustrating functions we must deal with two quirks in Haskell which totally distract and even obscure the main issues.

1. Haskell does not print anything reasonable to represent a function by default.

2. The types of many function contain class constraints, and there is no need to discuss classes in a simple introduction to Haskell. (*Use :type +d directive.*)
Using GHCi with Functions

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCI configuration from /home/ryan/.ghci

λ> \ x -> x + 1
```
Using GHCi with Functions

```haskell
λ> \ x -> x + 1
<interactive>:1:1: error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

λ> 
```
Using GHCi with Functions

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> \ x -> x + 1

<interactive>:1:1: error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

λ> :mod + Text.Show.Functions

λ> 
```
Using GHCi with Functions

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> \ x -> x + 1

<interactive>:1:1:  error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

λ> :mod + Text.Show.Functions

λ> \ x -> x + 1
```
Using GHCi with Functions

GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> \ x -> x + 1

<interactive>:1:1: error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven’t applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

λ> :mod + Text.Show.Functions

λ> \ x -> x + 1
<function>

λ>
Using GHCi with Functions

Haskell magical show function will show/print anything, *but not functions*.

In Haskell’s defense, what is the printable representation of a function? Is the Intel assembly code? The abstract syntax tree of the code? Should it just parrot back out the input source code?

One can get the Haskell interactive system to print the word `<function>` which seems like a really better idea than one of its inscrutable error messages.
Upcoming script

```
:type 'A'
:type 3
:type \ b -> not b
:type \ x -> not x
:set +t
\ x -> x + 1
```
Using GHCi with Functions

```bash
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> \x -> x + 1

<interactive>:1:1: error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

λ> :mod + Text.Show.Functions

λ> :type 'A'
```
Using GHCi with Functions

```
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> \x -> x + 1

<interactive>:1:1: error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
      (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

λ> :mod + Text.Show.Functions

λ> :type 'A'
'A' :: Char

λ> 
```
Using GHCi with Functions

GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> \x -> x + 1

<interactive>:1:1: error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

λ> :mod + Text.Show.Functions

λ> :type 'A'
'A' :: Char

λ> :typ e 3
Using GHCi with Functions

\[ \lambda \rightarrow x \rightarrow x + 1 \]

<interactive>:1:1: error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven’t applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

\[ \lambda \rightarrow \text{mod} + \text{Text.Show.Functions} \]

\[ \lambda \rightarrow \text{:type 'A'} \]
'A' :: Char

\[ \lambda \rightarrow \text{:type 3} \]
3 :: Num p => p

\[ \lambda \rightarrow \]
Using GHCi with Functions

```haskell
λ> \x -> x + 1

<interactive>:1:1: error:
  • No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’
    (maybe you haven't applied a function to enough arguments?)
  • In a stmt of an interactive GHCi command: print it

λ> :mod + Text.Show.Functions

λ> :type 'A'
'A' :: Char

λ> :type 3
3 :: Num p => p

λ> :type \b -> not b
```
Using GHCi with Functions

- No instance for (Show (Integer -> Integer))
  arising from a use of ‘print’
  (maybe you haven’t applied a function to enough arguments?)
- In a stmt of an interactive GHCi command: print it

λ> :mod + Text.Show.Functions

λ> :type 'A'
'A' :: Char

λ> :type 3
3 :: Num p => p

λ> :type \b -> not b
\b -> not b :: Bool -> Bool

λ> :type \x -> x + 1
Using GHCi with Functions

- In a stmt of an interactive GHCi command: print it

```haskell
λ> :mod + Text.Show.Functions

λ> :type 'A'
'A' :: Char

λ> :type 3
3 :: Num p => p

λ> :type \b -> not b
\b -> not b :: Bool -> Bool

λ> :type \x -> x + 1
\x -> x + 1 :: Num a => a -> a

λ>  ```
Using GHCi with Functions

\x \rightarrow \ x + 1 :: \text{Num} \ a \Rightarrow \ a \rightarrow \ a

\lambda> \text{set} + t

<interactive>:7:1: \text{error:}
  * Variable not in scope: set
  * Perhaps you meant ‘seq’ (imported from Prelude)

<interactive>:7:7: \text{error: Variable not in scope: t}

\lambda> :\text{set} + t
unknown option: ‘'
Some flags have not been recognized: t

\lambda> :\text{set} + t

\lambda>
Using GHCi with Functions

```
<interactive>:7:1: error:
  • Variable not in scope: set
  • Perhaps you meant ‘seq’ (imported from Prelude)

<interactive>:7:7: error: Variable not in scope: t

λ> :set + t
unknown option: ''
Some flags have not been recognized: t

λ> :set +t

λ> \x -> x + 1
<function>
it :: Num a => a -> a

λ>
```
Functions Are Data

Upcoming script

```haskell
mod + Text.Show.Functions Data.Char

\ x \rightarrow x + 1
\ i \rightarrow \text{max} \ i \ 0
\ x \rightarrow \text{sin} \ (x + \pi)
\ c \rightarrow \text{prd} \ c
\ s \rightarrow s ++ "", "" ++ s
\ xs \rightarrow 2 : xs
```

Ryan Stansifer (CS, Florida Tech)  Haskell Lists  March 28, 2019  66 / 76
Functions Definitions

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> :mod + Text.Show.Functions Data.Char

λ> \ x -> x + 1
```
Functions Definitions

```haskell
λ> \ x -> x + 1
<function>
```

$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> :mod + Text.Show.Functions Data.Char
Functions Definitions

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCI configuration from /home/ryan/.ghci

λ> :mod + Text.Show.Functions Data.Char

λ> \ x -> x + 1
<function>

λ> \ i -> max i 0
```
Functions Definitions

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> :mod + Text.Show.Functions Data.Char

λ> \ x -> x + 1
<function>

λ> \ i -> max i 0
<function>
```

Ryan Stansifer  (CS, Florida Tech)  Haskell Lists  March 28, 2019
Functions Definitions

```haskell
λ> :mod + Text.Show.Functions Data.Char

λ> \ x -> x + 1
<function>

λ> \ i -> max i 0
<function>

λ> \ x -> sin (x+pi)
```
Functions Definitions

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCI configuration from /home/ryan/.ghci

λ> :mod + Text.Show.Functions Data.Char

λ> \ x -> x + 1
<function>

λ> \ i -> max i 0
<function>

λ> \ x -> sin (x+pi)
<function>
```

Ryan Stansifer  (CS, Florida Tech)  Haskell Lists  March 28, 2019  67 / 76
Functions Definitions

$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/ryan/.ghci

λ> :mod + Text.Show.Functions Data.Char

λ> \ x -> x + 1
<function>

λ> \ i -> max i 0
<function>

λ> \ x -> sin (x+π)
<function>

λ> \ c -> ord c
loaded GHCi configuration from /home/ryan/.ghci

λ> :mod + Text.Show.Functions Data.Char

λ> \ x -> x + 1
<function>

λ> \ i -> max i 0
<function>

λ> \ x -> sin (x+π)
<function>

λ> \ c -> ord c
<function>

λ>
Functions Definitions

```
Loaded GHCi configuration from /home/ryan/.ghci

λ> :mod + Text.Show.Functions Data.Char

λ> \ x -> x + 1 <function>

λ> \ i -> max i 0 <function>

λ> \ x -> sin (x+pi) <function>

λ> \ c -> ord c <function>

λ> \ s -> s ++ ", " ++ s
```
Functions Definitions

λ> \ x -> x + 1
<function>

λ> \ i -> max i 0
<function>

λ> \ x -> sin (x+pi)
<function>

λ> \ c -> ord c
<function>

λ> \ s -> s ++ ", " ++ s
<function>

λ>
Functions Definitions

\[
\lambda > \ x \rightarrow x + 1 \\
\text{<function>}
\]

\[
\lambda > \ i \rightarrow \text{max } i \ 0 \\
\text{<function>}
\]

\[
\lambda > \ x \rightarrow \text{sin } (x+\pi) \\
\text{<function>}
\]

\[
\lambda > \ c \rightarrow \text{ord } c \\
\text{<function>}
\]

\[
\lambda > \ s \rightarrow \text{s ++ "", ++ s} \\
\text{<function>}
\]

\[
\lambda > \ xs \rightarrow 2:xs \\
\text{<function>}
\]
Functions Definitions

\[
\lambda \ x \rightarrow \sin (x + \pi)
\]

\[
\lambda \ c \rightarrow \text{ord } c
\]

\[
\lambda \ s \rightarrow \text{append } ”,” \text{ to } s
\]

\[
\lambda \ xs \rightarrow 2 : xs
\]
What Can You Do With a Function?

- Integer: get the next integer
- Tuple: get the first and the second part
- List: get the rest of the list
- Function: use/apply it to an argument
Functions Application

We ask what is the type of the function, and then we apply it to some element of the domain.
Function Application

\[ \lambda \, i \rightarrow \text{max } i \theta \]

\[ \lambda \, x \rightarrow \sin (x+\pi) \]

\[ \lambda \, c \rightarrow \text{ord } c \]

\[ \lambda \, s \rightarrow s ++ "\, " ++ s \]

\[ \lambda \, xs \rightarrow 2:xs \]

\[ \lambda \, :\text{type } +d \, \lambda \, x \rightarrow x + 1 \]
Function Application

\[
\lambda \ x \to \ \sin (x + \pi)
\]
<function>

\[
\lambda \ c \to \ \text{ord} \ c
\]
<function>

\[
\lambda \ s \to \ s \ ++ \ "\ , \ " \ ++ \ s
\]
<function>

\[
\lambda \ xs \to \ 2:xs
\]
<function>

\[
: \text{type} \ +d \ \lambda \ x \to \ x + 1
\]
\[
\lambda \ x \to \ x + 1 :: \text{Integer} \to \text{Integer}
\]
Function Application

\( \lambda \ x \rightarrow \sin(x + \pi) \)  
<function>

\( \lambda \ c \rightarrow \text{ord} \ c \)  
<function>

\( \lambda \ s \rightarrow s ++ ", " ++ s \)  
<function>

\( \lambda \ xs \rightarrow 2:xs \)  
<function>

\( \lambda \ : \text{type} \ +d \ \lambda \ x \rightarrow x + 1 \)  
\( \lambda \ x \rightarrow x + 1 :: \text{Integer} \rightarrow \text{Integer} \)

\( \lambda \ (\ \lambda \ x \rightarrow x + 1) \ 234 \)
Function Application

\[
\begin{align*}
\lambda & \rightarrow \text{ord } c \\
\lambda & \rightarrow s \rightarrow s ++ "," + s \\
\lambda & \rightarrow xs \rightarrow 2:xs \\
\lambda & \rightarrow \text{type } \text{+d } x \rightarrow x + 1 \\
\lambda & \rightarrow x \rightarrow x + 1 :: \text{Integer } \rightarrow \text{Integer} \\
\lambda & \rightarrow (\text{x } \rightarrow x + 1) \ 234 \\
& \quad 235
\end{align*}
\]
Function Application

```haskell
λ> \ c -> ord c
<function>

λ> \ s -> s ++ "," ++ s
<function>

λ> \ xs -> 2:xs
<function>

λ> :type +d \ x -> x + 1
\ x -> x + 1 :: Integer -> Integer

λ> (\ x -> x + 1) 234
235

λ> :type +d \ i -> max i 0
```
Function Application

```
λ> \ s -> s ++ "", " ++ s
<function>
λ> \ xs -> 2:xs
<function>
λ> :type +d \ x -> x + 1
\ x -> x + 1 :: Integer -> Integer
λ> (\ x -> x + 1) 234
235
λ> :type +d \ i -> max i 0
\ i -> max i 0 :: Integer -> Integer
λ>
```
Function Application

```
λ> \ s -> s ++ ", " ++ s
<function>

λ> \ xs -> 2:xs
<function>

λ> :type +d \ x -> x + 1
\ x -> x + 1 :: Integer -> Integer

λ> (\ x -> x + 1) 234
235

λ> :type +d \ i -> max i 0
\ i -> max i 0 :: Integer -> Integer

λ> (\ i -> max i 0) -8
```
Function Application

\( \lambda x \to x + 1 :: \text{Integer} \to \text{Integer} \)

\( \lambda x \to x + 1 \) 234

235

\( \lambda i \to \max i 0 :: \text{Integer} \to \text{Integer} \)

\( \lambda i \to \max i 0 \) -8

<interactive>:11:1: error:
- Non type-variable argument in the constraint: \( \text{Num} (a \to a) \)
  (Use \text{FlexibleContexts} to permit this)
- When checking the inferred type
  \( \forall a. (\text{Ord} a, \text{Num} a, \text{Num} (a \to a)) \Rightarrow a \to a \)
Function Application

\( \lambda \ x \rightarrow x + 1 : : \text{Integer} \rightarrow \text{Integer} \)

\( \lambda \rightarrow (\ \lambda \ x \rightarrow x + 1) \ 234 \)

\( 235 \)

\( \lambda \rightarrow \text{type } \ +d \ \ \lambda \ i \rightarrow \text{max } i \ 0 \)

\( \lambda \ i \rightarrow \text{max } i \ 0 : : \text{Integer} \rightarrow \text{Integer} \)

\( \lambda \rightarrow (\ \lambda \ i \rightarrow \text{max } i \ 0) \ -8 \)

<interactive>:11:1: error:

- Non type-variable argument in the constraint: Num (a -> a)
  (Use FlexibleContexts to permit this)
- When checking the inferred type
  it : forall a. (Ord a, Num a, Num (a -> a)) => a -> a

\( \lambda \rightarrow (\ \lambda \ i \rightarrow \text{max } i \ 0) \ (-8) \)
Function Application

```
\> :type (+) \ i -> max i 0
\ i -> max i 0 :: Integer -> Integer

\> (\ i -> max i 0) -8

<interactive>:11:1: error:
  • Non type-variable argument in the constraint: Num (a -> a)
    (Use FlexibleContexts to permit this)
  • When checking the inferred type
    it :: forall a. (Ord a, Num a, Num (a -> a)) => a -> a

\> (\ i -> max i 0) (-8)

0
```

Ryan Stansifer (CS, Florida Tech)  Haskell Lists  March 28, 2019  70 / 76
Function Application

```
λ> :type \ i -> max i 0
\ i -> max i 0 :: Integer -> Integer

λ> (\ i -> max i 0) -8

<interactive>:11:1: error:
  • Non type-variable argument in the constraint: Num (a -> a)
    (Use FlexibleContexts to permit this)
  • When checking the inferred type
    it :: forall a. (Ord a, Num a, Num (a -> a)) => a -> a

λ> (\ i -> max i 0) (-8)
0

λ> :type \ x -> sin (x+pi)
```
Function Application

\( i \rightarrow \text{max } i \ 0 :: \text{Integer} \rightarrow \text{Integer} \)

\( \lambda > (\backslash i \rightarrow \text{max } i \ 0) \ -8 \)

<interactive>:11:1: \textbf{error:}

- Non type-variable argument in the constraint: \( \text{Num } (a \rightarrow a) \)
  (Use FlexibleContexts to permit this)
- When checking the inferred type
  \( \text{it :: forall a. (Ord } a, \text{Num } a, \text{Num } (a \rightarrow a)) \rightarrow a \rightarrow a \)

\( \lambda > (\backslash i \rightarrow \text{max } i \ 0) \ (-8) \)

\( 0 \)

\( \lambda > :\text{type } +d \ \backslash x \rightarrow \text{sin } (x+\pi) \)
\( \backslash x \rightarrow \text{sin } (x+\pi) :: \text{Double} \rightarrow \text{Double} \)
Function Application

\( i \to \max i 0 :: \text{Integer} \to \text{Integer} \)

\( \lambda> (\ \lambda i \to \max i 0) \ -8 \)

<interactive>:11:1: error:
  - Non type-variable argument in the constraint: Num (a -> a)
  - (Use FlexibleContexts to permit this)
  - When checking the inferred type
    it :: forall a. (Ord a, Num a, Num (a -> a)) => a -> a

\( \lambda> (\ \lambda i \to \max i 0) \ (-8) \)

0

\( \lambda> :\text{type} +d \ \lambda x \to \sin (x+\pi) \)

\( \lambda> :\text{type} +d \ \lambda x \to \sin (x+\pi) :: \text{Double} \to \text{Double} \)

\( \lambda> :\text{type} +d \ \lambda c \to \text{ord c} \)
Function Application

Non type-variable argument in the constraint: Num (a -> a)
(Use FlexibleContexts to permit this)
When checking the inferred type
   it :: forall a. (Ord a, Num a, Num (a -> a)) => a -> a

\> (\ i -> max i 0) (-8)
0

\> :type +d \ x -> sin (x+pi)
\ x -> sin (x+pi) :: Double -> Double

\> :type +d \ c -> ord c
\ c -> ord c :: Char -> Int
\>
Function Application

```
<interactive>:11:1:  error:
  * Non type-variable argument in the constraint: Num (a -> a)
    (Use FlexibleContexts to permit this)
  * When checking the inferred type
    it :: forall a. (Ord a, Num a, Num (a -> a)) => a -> a

λ> (\ i -> max i 0) (-8)
0

λ> :type +d \ x -> sin (x+pi)
\ x -> sin (x+pi) :: Double -> Double

λ> :type +d \ c -> ord c
\ c -> ord c :: Char -> Int

λ> :type +d \ s -> s ++ ", " ++ s
```
Function Application

(Use FlexibleContexts to permit this)
- When checking the inferred type
  it :: forall a. (Ord a, Num a, Num (a -> a)) => a -> a

\> (\ i -> max i 0) (-8)
0

\> :type +d \ x -> sin (x+pi)
\x -> sin (x+pi) :: Double -> Double

\> :type +d \ c -> ord c
\c -> ord c :: Char -> Int

\> :type +d \ s -> s ++ "," ++ s
\s -> s ++ "," ++ s :: [Char] -> [Char]
Function Application

(Use FlexibleContexts to permit this)
• When checking the inferred type
  it :: forall a. (Ord a, Num a, Num (a -> a)) => a -> a

\> (\ i -> max i 0) (-8)
0

\> :type +d \ x -> sin (x+pi)
\ x -> sin (x+pi) :: Double -> Double

\> :type +d \ c -> ord c
\ c -> ord c :: Char -> Int

\> :type +d \ s -> s ++ "," ++ s
\ s -> s ++ "," ++ s :: [Char] -> [Char]

\> (\ s -> s ++ "," ++ s) "very"
Function Application

\[
\lambda > (\ i \rightarrow \max \ i \ 0) \ (-8) \\
0
\]

\[
\lambda > \text{type } \text{d} \ \lambda \ x \rightarrow \sin \ (x+\pi) \\
\lambda \ x \rightarrow \sin \ (x+\pi) :: \text{Double} \rightarrow \text{Double}
\]

\[
\lambda > \text{type } \text{d} \ \lambda \ c \rightarrow \text{ord} \ c \\
\lambda \ c \rightarrow \text{ord} \ c :: \text{Char} \rightarrow \text{Int}
\]

\[
\lambda > \text{type } \text{d} \ \lambda \ s \rightarrow s ++ "", " ++ s \\
\lambda \ s \rightarrow s ++ "", " ++ s :: [\text{Char}] \rightarrow [\text{Char}]
\]

\[
\lambda > (\ s \rightarrow s ++ "", " ++ s) "\text{very}"
"\text{very, very}\"
\]

\[
\lambda >
\]
How do functions get names?
The same way anything gets a name!

\[
\begin{align*}
\text{add1} & = \lambda x \rightarrow x + 1 \\
\text{cutOff} & = \lambda i \rightarrow \text{max} \ i \ 0 \\
g & = \lambda x \rightarrow \sin (x+\pi) \\
f & = \lambda c \rightarrow \text{ord} \ c \\
\text{double} & = \lambda s \rightarrow s ++ " \, \, " ++ s \\
\text{cons2} & = \lambda xs \rightarrow 2:xs
\end{align*}
\]
Naming Functions

\>
(\ i \rightarrow \text{max } i \ 0) \ (-8)
0

\>
: \text{type } +d \ \ \ \ \ \ \ \ \ \ \ \ \ x \rightarrow \sin (x+\pi)
\ x \rightarrow \sin (x+\pi) :: \text{Double } \rightarrow \text{Double}

\>
: \text{type } +d \ \ \ \ \ \ \ \ \ \ \ \ \ c \rightarrow \text{ord } c
\ c \rightarrow \text{ord } c :: \text{Char } \rightarrow \text{Int}

\>
: \text{type } +d \ \ \ \ \ \ \ \ \ \ \ \ \ s \rightarrow s ++ \ "", \ "++ s
\ s \rightarrow s ++ \ "", \ "++ s :: [\text{Char}] \rightarrow [\text{Char}]

\>
(\ s \rightarrow s ++ \ "", \ "++ s) \ "\text{very}"
"\text{very, very}"

\>
\text{add1} = \ \ \ \ \ \ \ x \rightarrow x + 1
Naming Functions

\> \texttt{\textbackslash x \rightarrow \sin (x+\pi)}
\> \texttt{\textbackslash x \rightarrow \sin (x+\pi) :: Double \rightarrow Double}

\> \texttt{\textbackslash c \rightarrow \text{ord c}}
\> \texttt{\textbackslash c \rightarrow \text{ord c} :: Char \rightarrow Int}

\> \texttt{\textbackslash s \rightarrow s ++ "", " ++ s}
\> \texttt{\textbackslash s \rightarrow s ++ "", " ++ s :: [Char] \rightarrow [Char]}

\> (\texttt{\textbackslash s \rightarrow s ++ "", " ++ s}) "very"
\> "very, very"

\> \texttt{\textbackslash x \rightarrow x + 1}
Naming Functions

\x \rightarrow \sin (x + \pi) :: \text{Double} \rightarrow \text{Double}

\lambda c \rightarrow \text{ord} c
\lambda c \rightarrow \text{ord} c :: \text{Char} \rightarrow \text{Int}

\lambda s \rightarrow s ++ "", " ++ s
\lambda s \rightarrow s ++ "", " ++ s :: \text{[Char]} \rightarrow \text{[Char]}

\lambda s \rightarrow s ++ "", " ++ s) "very"
"very, very"

\lambda x \rightarrow x + 1

\lambda \text{add1} = \lambda x \rightarrow x + 1

\lambda \text{add1} :: \text{Integer} \rightarrow \text{Integer}
Naming Functions

\[
\text{x} \rightarrow \sin(x + \pi) :: \text{Double} \rightarrow \text{Double}
\]

\[
\text{\lambda > :type +d \ c \rightarrow \text{ord c}}
\]
\[
\text{\c \rightarrow \text{ord c} :: \text{Char} \rightarrow \text{Int}}
\]

\[
\text{\lambda > :type +d \ s \rightarrow s ++ "\", " ++ s}
\]
\[
\text{\s \rightarrow s ++ "\", " ++ s :: [Char] \rightarrow [Char]}
\]

\[
(\lambda \ s \rightarrow s ++ "\", " ++ s) \text{ "very"}
\]
\[
\text{"very, very"}
\]

\[
\text{\lambda > \text{add1 = \ x \rightarrow x + 1}}
\]

\[
\text{\lambda > :type +d \ \text{add1}}
\]
\[
\text{\text{add1} :: \text{Integer} \rightarrow \text{Integer}}
\]

\[
\text{\lambda > \text{add1 813}}
\]
Naming Functions

\ c \rightarrow \text{ord} \ c :: \text{Char} \rightarrow \text{Int}

\lambda \rightarrow \text{type +d} \ s \rightarrow s ++ "\", " ++ s
\ s \rightarrow s ++ "\", " ++ s :: \text{Char} \rightarrow \text{Char}

\lambda \ (\ s \rightarrow s ++ "\", " ++ s) "very"
"very, very"

\lambda \ \text{add1} = \ \lambda x \rightarrow x + 1

\lambda \ : \text{type +d} \ \text{add1}
\text{add1} :: \text{Integer} \rightarrow \text{Integer}

\lambda \ \text{add1} 813
814

\lambda \
Naming Functions

\[
\lambda> (\ \_ s -> s ++ ",\ " ++ s) \ "very"
"very, very"
\]

\[
\lambda> add1 = \ \_ x -> x + 1
\]

\[
\lambda> :type +d \ add1
add1 :: Integer -> Integer
\]

\[
\lambda> add1 813
814
\]

\[
\lambda> cutOff = \ \_ i -> max i 0
\]

\[
\lambda> :type +d \ cutOff
cutOff :: Integer -> Integer
\]

\[
\lambda> \]

Terminal - ryan@hc210-059044:~
Naming Functions

\[
\lambda \text{> } \text{add1} = \ \lambda x \rightarrow x + 1
\]

\[
\lambda \text{> } \text{cutOff} = \ \lambda i \rightarrow \text{max } i \ 0
\]

\[
\lambda \text{> } \text{add1} \ 813
814
\]

\[
\lambda \text{> } \text{cutOff} \ (-10)
0
\]
Naming Functions

```haskell
λ> cutOff (-10)
0

λ> double = \ x -> x ++ ", " ++ s

<interactive>:23:30: **error:** Variable not in scope: s :: [Char]

λ> double = \ s -> s ++ ", " ++ s

λ> :type +d double
double :: [Char] -> [Char]

λ> double "much"
"much, much"
```
Naming Functions

\[\lambda\] double = \ s \rightarrow s ++ "," ++ s

\[\lambda\] :type +d double
double :: [Char] -> [Char]

\[\lambda\] double "much"
"much, much"

\[\lambda\] cons2 = \ xs \rightarrow 2:xs

\[\lambda\] :type +d cons2
cons2 :: [Integer] -> [Integer]

\[\lambda\] cons2 [3,4,5]
[2,3,4,5]
Function Declaration Has Special Syntax

add1 = \ x -> x + 1  
add1 x = x + 1

cutOff = \ i -> max i 0  
cutOff i = max i 0

g = \ x -> sin (x+pi)  
g x = sin (x+pi)

f = \ c -> ord c  
f c = ord c

double=\s->s++", "++ s  
double s = s++", "++s

cons2 = \ xs -> 2:xs  
cons2 xs = 2:xs
Functions

\[
f(x) = \begin{cases} 
\text{"Empty!"} & \text{if \ null\ } x \\
\text{"Not Empty!"} & \text{else}
\end{cases}
\]

\[
\text{factorial}\ n = \begin{cases} 
1 & \text{if } n < 2 \\
\text{else } n \ast \text{factorial}\ (n-1)
\end{cases}
\]
Patterns

Think of a formal argument as a name that matches any value in the domain of the function.

Patterns as formal arguments use constructors to match against the value given to the function as the actual argument.

If the value does not match, you are out of luck.
Patterns

\[ p_3(x, y, z) = z \quad -- \text{definition of function } p_3 \]

\[ p_3(1, 2, 3) \quad -- \text{evaluates to } 3 \]

\[ t = (1, 2, 3) \]

\[ p_3(t) \quad -- \text{evaluates to } 3 \]

\[ f(x:2:y:rest) = x+y \quad -- \text{definition of } f \]

\[ f(1:2:3:9:\[]) \quad -- \text{evaluates to } 4 \]

\[ f[1, 2, 3, 9] \quad -- \text{evaluates to } 4 \]

\[ l = [12, 3, 9] \]

\[ f\ l \quad -- \text{evaluates to } 4 \]
Wildcard Patterns

\[ p_3 (_ , _ , z) = z \quad -- \text{definition of function } p_3 \]

\[ f (_ : 2 : _ : _) = x + y \quad -- \text{definition of } f \]
Definition by cases

\[
\begin{align*}
    f \ [\] & = "empty" \\
    f \ (x:[]) & = "single" \\
    f \ (x:y:[]) & = "small" \\
    f \ (x:y:z:[]) & = "medium" \\
    f \ (_) & = "large"
\end{align*}
\]
Overview

- Everything is digital (all files are binary)
- Unix streams and pipes (function composition is important)
- What is a program?
- GHC
  - writing a program
  - compiling a program (also -Wall)
  - running a program (also +RTS -RTS)
- interact: boilerplate to turn a function into a working program
Everything is Digital

Ryan Stansifer  (CS, Florida Tech)
Streams and Pipes

Along the Stream by Sharon France
We know what a function is: it maps an element in the domain to an element in the range.

When we think of a program as a function we generally think of something like the factorial function from numbers to numbers.

But this is not a “real” program. A real program reads input and writes output.

Limiting ourselves to the important case of programs from US-ASCII text to US-ASCII text, a real program is really a function from a stream/file/string of text to a stream/file/string of text.
Input and Output Stream
Using the Glasgow Haskell Compiler

In Haskell, just like numerous other programming languages, one creates a source program in a text file, one invokes the compiler to create an executable file, and then one commands the computer to run the executable file on some text input and observe or collect the text output.

$ ghc -Wall -o main Main.hs
$ main < input.txt > output.txt
Warning!

Ask the compiler to help you write a more beautiful program by turning on all warnings.

$ ghc -Wall -o main Main.hs

Don’t turn in a program with warnings.
How do we create a Haskell program that reads the input and writes the output?
Some programming languages have elaborate IO mechanisms which one must learn to simple process the input and produce the output. In Haskell the simplest approach is to use the function `interact` to transform the conceptually pure program (a function from the input stream to the output stream) to an actual, working program on the computer.
Standard input stream (domain)

a predefined Haskell function

interact :: (String -> String) -> IO ()

"has type"

Standard output stream (range)

[IO monad!]

ignore!
Examples On-Line

README