Outline

1. GHCI
2. Basic expressions and primitive data types. Integers, reals, characters, unit.
3. Tuples
4. Lists, list comprehension
5. Basic functions
6. Patterns
7. Type variables. When Haskell does not care what the type is, a type variable is used to stand for any type
8. Higher-order functions, partial application
9. Map, filter, fold
10. Curry, uncurry, isomorphism
11. Data types
12. Main program, interact, compilation
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

`> :help
Commands available from the prompt:

  <statement>           evaluate/run <statement>
  :                   repeat last command
  :{\n    ..lines.. \n}:\n  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
:repeat last command

:{
 ..lines.. 
:}
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

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

: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
We typically view programming as a combination of algorithms (procedures) and data structures.

Functional programming presents us with a grand unified theory of data and procedures. This may be unsettling at first.

Integers, lists, tuples, functions: they are all data.
Data

Integers

\[ \langle integer \rangle ::= \langle digit \rangle^+ \]

Lists

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

Tuples

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

Functions

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

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

0

42

3

7

5429723947

567
The Type of Integers
Integer literals are canonical values.
Integer expressions might be evaluated/reduced to integer canonical values.
Integer literals are canonical values.

Integer expressions might be evaluated/reduced to integer canonical values.

More of the usual details about integers later, but now quickly on to tuples.
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)
(Integer,Integer)
(Char,Integer,Integer,Integer)

(Boolean,Integer[Boolean,Integer)

(Char[Boolean)

(Integer,(Char[Boolean),Char)

(Integer,Integer,Integer,Integer)
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] = String

[Integers]

[Char] = String

[Bool]

[Bool]
Lists Have Special Syntax

\[ [1,1] \]
\[ ['a','b','c'] = "abc" \]
\[ [(),()] \]
\[ [3,5,3+6,7] \]
\[ [False,False,True] \]
\[ [True,False,True] \]
Functions

\begin{align*}
\langle \text{function} \rangle & ::= \lambda \langle \text{name} \rangle \rightarrow \langle \text{expr} \rangle \\
\hline
1 & \text{lambda (backslash); punctuation, keyword} \\
2 & \text{formal parameter; an identifier (really a pattern)} \\
3 & \text{“arrow” (\rightarrow); punctuation, keyword} \\
4 & \text{body; arbitrary expression possibly using the identifier}
\end{align*}
\[ \text{xs} \rightarrow 2: \text{xs} \]

\[ \text{i} \rightarrow \text{max i 0} \]

\[ \text{s} \rightarrow \text{s++", "++s} \]

\[ \text{i} \rightarrow \text{i+2} \]

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

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

Integer -> Integer

String -> String

Integer -> Integer

Double -> Double

Char -> Integer
Why do people name things? So that they can refer to the later. Data, all data, exists apart from their name.

```haskell
theMeaningOfLife = 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

\[
\text{cons2 } xs = 2:xs
\]
\[
\text{cutOff } i = \max i 0
\]
\[
\text{doubleWord } s = s ++ ", " ++ s
\]
\[
\text{add2 } i = i+2
\]
\[
f x = \sin (x+\pi)
\]
\[
\text{toOrd } c = \text{ord } c
\]
The special syntax for function is so obvious and familiar and using in reductions so obvious and familiar, it hardly needs mentioning. But we wait for later to elaborate.
The special syntax for function is so obvious and familiar and using in reductions so obvious and familiar, it hardly needs mentioning. But we wait for later to elaborate.

What is the canonical value of a function?
Function Declarations Have Special Syntax

The special syntax for function is so obvious and familiar and using in reductions so obvious and familiar, it hardly needs mentioning. But we wait for later to elaborate.

What is the canonical value of a function? Itself!
Function Declarations Have Special Syntax

The special syntax for function is so obvious and familiar and using in reductions so obvious and familiar, it hardly needs mentioning. But we wait for later to elaborate.

What is the canonical value of a function? Itself!

We by observing that these four kinds of data and be combined in endless without constraint.
Complex Canonical Values

(2, ('a', 5), "abc", True) :: (Integer, (Char, Integer), "abc", True) :: (Integer, (Char, Integer), [Char], Bool)

([1], ('a', 5), "abc", True, "xyz") :: ([Integer], (Char, Integer), [Char], Bool, [Char])

(\i -> i + 2, \x -> sin(x + pi)) :: (Integer -> Integer, Double -> Double)

([(1, True, "abc"), (2, True, "mno"), (3, True, "xyz")]) :: [(Integer, Book, [Char])]

\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: lists of character
143
succ 34
3+4
5*7
9-4
negate 8

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

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

quot 36  5
    -->  7

div 36  5
    -->  7

quot 36 (-5)
    -->  -7

div 36 (-5)
    -->  -8

rem 36  5
    -->  1

mod 36  5
    -->  1

rem 36 (-5)
    -->  1

mod 36 (-5)
    -->  -4
Integer

\[
\text{quotRem} \; 36 \; (-5) \\
\quad \rightarrow (-7, 1)
\]

\[
\text{divMod} \; 36 \; (-5) \\
\quad \rightarrow (-8, -4)
\]
Floating-Point

--- 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
Boolean Implication

Because boolean values are ordered we use the less-than-or-equals relation as the boolean connective for implication!

\[
\begin{align*}
\text{False} & \leq \text{False} \quad -- \quad F \rightarrow F = T \\
\text{False} & \leq \text{True} \quad -- \quad F \rightarrow T = T \\
\text{True} & \leq \text{False} \quad -- \quad T \rightarrow F = F \\
\text{True} & \leq \text{True} \quad -- \quad T \rightarrow T = T \\
\end{align*}
\]
Tuples

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

fst (2,'b')    -- pair projection
snd (1,'a')    -- pair projection

-- pairs of pairs
(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.

\[
\begin{align*}
\text{[]} & \quad \text{[]} & \quad \text{[]} \\
1:[] & \quad 1:[] & \quad 1:[] \\
1:(2:[]) & \quad 1:2:[] & \quad [1] \\
1:(2:(3:[])) & \quad 1:2:3:[] & \quad [1,2] \\
1:(2:(3:(4:[]))) & \quad 1:2:3:4:[] & \quad [1,2,3,4]
\end{align*}
\]
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
Efficiency not determined by the letters in the name.
Arithmetic Sequences

\[-1, -1 \ldots 0\]
\[3, 2 \ldots 8\]
\[0, 2 \ldots 1\]
\[1, 1 \ldots 1\]
\[3, 3 \ldots (-4)\]
\[2, 1 \ldots (-4)\]
\[3, 3 \ldots (-12)\]
\[-1, -1 \ldots 8\]
\[-6, -6 \ldots 12\]
\[1, 2 \ldots (-12)\]
\[-6, -5 \ldots (-4)\]
\[1, 0 \ldots 5\]
\[-6, -6 \ldots 8\]
\[-1, -2 \ldots (-4)\]
\[1, 2 \ldots 5\]
“Good stuff comes from Math”
“Sets are not computationally good”
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:

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

where \( 1 \geq n \)

There are three types qualifiers

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

See the section on List Comprehensions in the Haskell 2010 Language Report.
Also, there are intriguing language extensions to list comprehension that resemble more complex database query languages.

- **ParallelListComp.** Syntax: 
  \[
  [ \text{expr} \mid \text{qualifier}, \ldots \mid \text{qualifier}, \ldots ]
  \]

- **TransformListComp.** Three new keywords: group, by, and using.

See a paper by Simon Peyton Jones.
Upcoming script

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

```
λ> 
```

List Comprehension

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

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

```
λ> [ n | n<-[1..5] ]
[1,2,3,4,5]

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)]
```

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

```
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

```
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> [ (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

```haskell
[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"
```

Learning Haskell

April 9, 2020 53 / 86
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> 
```
List Comprehension (Multiple Generators)

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

Ryan Stansifer  (CS, Florida Tech)  Learning Haskell  April 9, 2020  54 / 86
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" ]
```
List Comprehension (Multiple Generators)

\[
\lambda\rightarrow [ (i,j) \mid i\llbracket1..3\rbracket, j\llbracket1..2\rbracket ]
\]

\[
\lambda\rightarrow [ (i,c) \mid i\llbracket1..2\rbracket, c\llbracket\text{"abc"}\rbracket ]
\]
List Comprehension (Multiple Generators)

\[
\lambda \Rightarrow [ (i, j) \mid i \leftarrow [1..3], j \leftarrow [1..2] ]
\]

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

\[
\lambda \Rightarrow [ (i, c) \mid i \leftarrow [1..2], c \leftarrow "abc" ]
\]

\[
[ (1,'a'), (1,'b'), (1,'c'), (2,'a'), (2,'b'), (2,'c') ]
\]

\[
\lambda \Rightarrow [ (i, c) \mid c \leftarrow "abc", j \leftarrow [1..2] ]
\]
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", i<-[1..2] ]
<interactive>:3:4: error: Variable not in scope: i

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

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

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

\[
\lambda> \ [ (i,c) \mid c\in "abc", j\in [1..2] ] \\
<interactive>:3:4: \mathbf{error}: \text{Variable not in scope}: i
\]

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

\[
\lambda>
\]
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>
```
List Comprehension (Multiple Generators)

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

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

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

```haskell
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)

```
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)

```
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),(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..] ]
```
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)]

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..] ]
[(1,1),(2,1),(1,2),(2,2),(1,3),(2,3),(1,4),(2,4)]
List Comprehension (Guards aka Filters)

```
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 (Guards aka Filters)

```
λ> [ n | n <- [50..100], n `mod` 7 == 3 ]
```
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]
```

Ryan Stansifer  (CS, Florida Tech)  Learning Haskell  April 9, 2020  57 / 86
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]
```
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)

```
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)

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 ]
List Comprehension (Guards aka Filters)

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

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

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

\[
\lambda> \ [ \ (n,n\times n) \mid n\in[-3..3], \ n\leq n\times n \ ]
\]
\[
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)]
\]
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)

\[
\begin{align*}
\lambda & \rightarrow [ n \mid n \leftarrow [50..100], \, n \; \text{`mod`} \; 7 \; == \; 3 ] \\
& \qquad \{52, 59, 66, 73, 80, 87, 94\}
\\
\lambda & \rightarrow [ n*\!n \mid n \leftarrow [1..22], \, \text{even} \; n ] \\
& \qquad \{4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484\}
\\
\lambda & \rightarrow [ n*\!n \mid n \leftarrow [2,4..22] ] \\
& \qquad \{4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484\}
\\
\lambda & \rightarrow [ (n, n*\!n) \mid n \leftarrow [-3..3], \, n < n*\!n ] \\
& \qquad \{(-3,9), (-2,4), (-1,1), (2,4), (3,9)\}
\\
\lambda & \rightarrow [ n \mid n \leftarrow [10..20], \, n /= 13, \, n /= 15, \, n /= 19 ] \\
& \qquad \{10, 11, 12, 14, 16, 17, 18, 20\}
\\
\lambda & \rightarrow
\end{align*}
\]
List Comprehension (Guards aka Filters)

\[
\lambda \Rightarrow [ n \mid n<-[50..100], n \ `mod` \ 7 == 3 ]
\]

[52, 59, 66, 73, 80, 87, 94]

\[
\lambda \Rightarrow [ n*n \mid n<-[1..22], \ even \ n ]
\]

[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484]

\[
\lambda \Rightarrow [ n*n \mid n<-[2,4..22] ]
\]

[4, 16, 36, 64, 100, 144, 196, 256, 324, 400, 484]

\[
\lambda \Rightarrow [ (n,n*n) \mid n<-[-3..3], n<n*n ]
\]

[(-3,9), (-2,4), (-1,1), (2,4), (3,9)]

\[
\lambda \Rightarrow [ n \mid n<-[10..20], n /= 13, n /= 15, n /= 19 ]
\]

[10, 11, 12, 14, 16, 17, 18, 20]

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

\[
\lambda> [ \ n*n \mid n<-[1..22], \text{even } n \ ] \\
[4,16,36,64,100,144,196,256,324,400,484] \\
\lambda> [ \ n*n \mid n<-[2,4..22] \ ] \\
[4,16,36,64,100,144,196,256,324,400,484] \\
\lambda> [ \ (n,n*n) \mid n<-[-3..3], \text{ n<n*n } \ ] \\
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)] \\
\lambda> [ \ n \mid n <- [10..20], \text{ n /} = 13, \text{ n /} = 15, \text{ n /} = 19 \ ] \\
[10,11,12,14,16,17,18,20] \\
\lambda> [ \ c \mid c <- \ "one fish", \ c \ `\text{elem`} \ "aeiou"\ ] \\
"oei" \\
\lambda>
\]
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 with guards: `even n` implies `even` is a type that only selects even numbers.
- Guards in list comprehension can be used to filter the list based on conditions.
- The `if` condition in list comprehension allows for conditional elements in the list.
List Comprehension (Guards aka Filters)

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

\[
\lambda > \ [ \ (n,n\ast n) \ | \ n<-\{-3..3\}, \ n<n\ast n \ ]
[(-3,9),(-2,4),(-1,1),(2,4),(3,9)]
\]

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

\[
\lambda > \ [ \ c \ | \ c<-"one fish", \ c `\text{elem}` "aeiou"\]
"oei"
\]

\[
\lambda > \ [\text{if} \ x<10 \ \text{then} \ "BOOM!" \ \text{else} \ "BANG!" \ | \ x<-\{7..13\}, \ odd \ x]\]
["BOOM!","BOOM!","BANG!","BANG!"]
\]

\[
\lambda > \]

List Comprehension and Functions

```
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 and Functions

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

```
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

```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> 
```
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)  Learning Haskell  April 9, 2020  58 / 86
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> spaces 8
  "          

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]
λ>
```
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]

λ>
List Comprehension and Functions

```haskell
<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]

λ> doublePos [1,4..13]
```
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]
List Comprehension and Functions

```
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]
```

Perhaps you meant ‘spaces’ (line 1)
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
```

```
List Comprehension and Functions

```haskell
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> doublePos [1,0.. (-78)]
```
List Comprehension and Functions

\[
\text{doublePos } \text{xs} = [2 \times x \mid x \leftarrow \text{xs}, x > 0]
\]

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

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

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

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

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

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

\[
\lambda> \text{doublePos } xs = [2 \times x \mid x \leftarrow xs, x > 0]
\]

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

[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]

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

\[
\lambda> \text{doublePos} \ [-1,2,1,2,3] \\
[4,2,4,6] \\
\lambda> \text{doublePos} \ [1,4..13] \\
[2,8,14,20,26] \\
\lambda> \text{doublePos} \ [1,0..-78] \\
\text{<interactive>}:8:15: \text{error:} \\
\text{Variable not in scope: } (...) :: \text{Integer} \to \text{Integer} \to a \\
\lambda> \text{doublePos} \ [1,0.. (-78)] \\
[2] \\
\lambda> \text{factors n} = [x | x <- [1..n], n `mod` x == 0] \\
\lambda> \]
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

```
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]
```
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

```

Ryan Stansifer  (CS, Florida Tech)  Learning Haskell  April 9, 2020  58 / 86
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]
```

List Comprehension and Functions

```haskell
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]

λ> factors 0
```

Variable not in scope: (..-): Integer -> Integer -> a
List Comprehension and Functions

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

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

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

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

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

\[
\lambda\mapsto
\]
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]
```
List Comprehension and Functions

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

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

> factors 6
[1,2,3,6]

> factors 17
[1,17]

> factors 0
[]

> 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> prime 12
```
List Comprehension and Functions

```haskell
[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> prime 12
False
λ>
```
List Comprehension and Functions

```
[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> prime 12
False

λ> prime 53
```
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>
```
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

```
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> 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
```
List Comprehension, Pythagorean Triples

```
λ> 
```

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

```
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> 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

```haskell
λ> pythagorean n = [(x,y,z) | x <- [1..n], y <- [x..n], z <- [y..n], x^2 + y^2 == z^2]
λ> pythagorean 12
```
List Comprehension, Pythagorean Triples

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

\[
\text{pythagorean 12} = [(3, 4, 5), (6, 8, 10)]
\]
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

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

\[
\lambda> \text{pythagorean 100}
\]
List Comprehension, Pythagorean Triples

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

\[
\begin{align*}
\lambda> & \text{ pythagorean 100} \\
& [(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15), (9, 40, 41), (10, 24, 26), (11, 60, 61), (12, 16, 20), (12, 35, 37), (13, 84, 85), (14, 48, 50), (15, 20, 25), (15, 36, 39), (16, 30, 34), (16, 63, 65), (18, 24, 30), (18, 80, 82), (20, 21, 29), (20, 48, 52), (21, 28, 35), (21, 72, 75), (24, 32, 40), (24, 45, 51), (24, 70, 74), (25, 60, 65), (27, 36, 45), (28, 45, 53), (28, 96, 100), (30, 40, 50), (30, 72, 78), (32, 60, 68), (33, 44, 55), (33, 56, 65), (35, 84, 91), (36, 48, 60), (36, 77, 85), (39, 52, 65), (39, 80, 89), (40, 42, 58), (40, 75, 85), (42, 56, 70), (45, 60, 75), (48, 55, 73), (48, 64, 80), (51, 68, 85), (54, 72, 90), (57, 76, 95), (60, 63, 87), (60, 80, 100), (65, 72, 97)]
\end{align*}
\]
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:
  theMeaningOfLife = 42

  NB. Not assignment!

• Control of scope using let expression.
  
  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>
</tbody>
</table>

| symbolc | (+) | + |

```
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:

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

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
Using GHCi with Functions

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

```
$ 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> 
```
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

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

```
<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

λ> :mod + Text.Show.Functions

λ> \ x -> x + 1

<function>
```

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

\begin{verbatim}
:type 'A'
:type 3
:type \ b -> not b
:type \ x -> not x
:set +t
\ x -> x + 1
\end{verbatim}
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> :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
Using GHCi with Functions

```
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
```
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>
```
Using GHCi with Functions

```
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

\> :mod + Text.Show.Functions

\> :type 'A'
'A' :: Char

\> :type 3
3 :: Num p => p

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

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

\>
Using GHCi with Functions

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

\[ \lambda x \cdot x + t \]

<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

\[ \lambda x \cdot x + t \]

unknown option: 't'
Some flags have not been recognized: t

\[ \lambda x \cdot x + t \]
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

:mod + Text.Show.Functions Data.Char

\ x -> x + 1
\ i -> max i 0
\ x -> sin (x+pi)
\ c -> prd c
\ s -> s ++ ", " ++ s
\ xs -> 2:xs
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

$ 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>
Functions Definitions

Terminal output:

```
$ 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>

λ> 
```
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)
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>

λ> 
```
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
Functions Definitions

Loaded GHCi configuration from /home/ryan/.ghci

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

\> \ x -> x + 1
<function>

\> \ i -> max i 0
<function>

\> \ x -> sin (x+pi)
<function>

\> \ c -> ord c
<function>

\>
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

\( \lambda \ x \rightarrow x + 1 \)
\( \lambda \ i \rightarrow \text{max} \ i \ 0 \)
\( \lambda \ x \rightarrow \sin(x + \pi) \)
\( \lambda \ c \rightarrow \text{ord} \ c \)
\( \lambda \ s \rightarrow s \ ++ \ "\,,\ " \ ++ \ s \)
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> xs -> 2:xs
```
Functions Definitions

```
λ> \ i -> max i 0
<function>

λ> \ x -> sin (x+pi)
<function>

λ> \ c -> ord c
<function>

λ> \ s -> s ++ "," ++ s
<function>

λ> \ xs -> 2:xs
<function>
```

```
What Can You Do With a Function?

This question is easy for the usual data types:

- Integer: get the next integer
- Pari: get the first and the second part
- List: get head and the rest of the list
What Can You Do With a Function?

This question is easy for the usual data types:

- Integer: get the next integer
- Pari: get the first and the second part
- List: get head and the rest of the list
- Function: use/apply it to an argument
We ask what is the type of the function, and then we apply it to some element of the domain.
Function Application

\> \ i \rightarrow \text{max} \ i \ 0
<function>

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

\> \ c \rightarrow \text{ord} \ c
<function>

\> \ s \rightarrow s ++ "," ++ s
<function>

\> \ xs \rightarrow 2:xs
<function>

\> :type +d \ i \rightarrow x \rightarrow x + 1
Function Application

\[
\textbf{Haskell Code:}
\]

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

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

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

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

\[
\lambda > \ :\text{type} \ +d \ \backslash \ x \rightarrow \ x + 1
\backslash \ x \rightarrow \ x + 1 :: \text{Integer} \rightarrow \text{Integer}
\]

\[
\lambda >
\]
Function Application

\> \ x \rightarrow \ sin \ (x+pi)
\textit{<function>}

\> \ c \rightarrow \ ord \ c
\textit{<function>}

\> \ s \rightarrow \ s ++ "," ++ s
\textit{<function>}

\> \ xs \rightarrow \ 2:xs
\textit{<function>}

\> :type +d \ x \rightarrow \ x + 1
\textit{\ x \rightarrow \ x + 1 :: Integer \rightarrow Integer}

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

```
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>
```
Function Application

```
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

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

\langle \text{function} \rangle

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

\langle \text{function} \rangle

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

\[ \lambda \ x \rightarrow x + 1 :: \text{Integer} \rightarrow \text{Integer} \]

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

235

\[ \lambda \ : \text{type} +d \ \ \lambda \ i \rightarrow \text{max} i \ 0 \]

\[ \lambda \ i \rightarrow \text{max} i \ 0 :: \text{Integer} \rightarrow \text{Integer} \]

\[ \lambda \]
Function Application

```haskell
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

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

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

235

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

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

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

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

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

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

235

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

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

<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

```
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>
```
Function Application

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

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

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

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

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

0

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

λ>
Function Application

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

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

<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

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

0

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

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

\( \lambda > :\text{type} \ +d \ \lambda \ c \rightarrow \text{ord } c \)
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>
```
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

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

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

```
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
\ s -> s ++ "," ++ s :: [Char] -> [Char]

λ> (\ s -> s ++ "," ++ s) "very"
"very, very"
```
How do functions get names?
The same way anything gets a name!

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

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

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

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

\[ \text{double} = \lambda s \rightarrow s ++ "\,", "++ s \]

\[ \text{cons2} = \lambda xs \rightarrow 2:xs \]
Naming Functions

\[
\lambda \ (\ i \to \ \text{max} \ i \ 0) \ (-8) \\
0
\]

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

\[
\lambda \ : \text{type} \ +d \ \ \backslash \ c \to \ \text{ord} \ c \\
\ c \to \ \text{ord} \ c :: \text{Char} \to \text{Int}
\]

\[
\lambda \ : \text{type} \ +d \ \ \backslash \ s \to \ s ++ \ "\", \ "++ \ s \\
\ s \to \ s ++ \ "\", \ "++ \ s :: \text{[Char]} \to \text{[Char]}
\]

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

\[
\lambda \ \text{add1} = \ \backslash \ x \to \ x + 1
\]
Naming Functions

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

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

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

(λ> (\ s -> s ++ "", " ++ s) "very"
  "very, very"

(λ> add1 = \ x -> x + 1
```
Naming Functions

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

\( \lambda \cdot \text{type } +d \ \cdot \ c \to \text{ord } c \\
\ c \to \text{ord } c :: \text{Char} \to \text{Int} \\
\)

\( \lambda \cdot \text{type } +d \ \cdot \ s \to s \ ++ \ " , \ " ++ s \\
\ s \to s \ ++ \ " , \ " ++ s :: [\text{Char}] \to [\text{Char}] \\
\)

\( \lambda \cdot (s \to s \ ++ \ " , \ " ++ s) \ "\text{very}" \\
\ "\text{very, very}" \\
\)

\( \lambda \cdot \text{add1 } = \ \cdot x \to x + 1 \\
\)

\( \lambda \cdot \text{type } +d \ \cdot \text{add1} \\
\text{add1 } :: \text{Integer} \to \text{Integer} \\
\)

\( \lambda \)
Naming Functions

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

\[ \lambda \text{c} \to \text{ord c} \]
\[ \text{c} \to \text{ord c} :\text{Char} \to \text{Int} \]

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

\[ \lambda \text{s} \to \text{s} ++ "", " ++ \text{s} \] "very"
"very, very"

\[ \lambda \text{x} \to \text{x} + 1 \]

\[ \lambda \text{add1} \]
\[ \text{add1} :\text{Integer} \to \text{Integer} \]

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

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

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

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

\( \lambda > \text{add1 } = \ \lambda x \rightarrow x + 1 \)

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

\( \lambda > \text{add1 } 813 \)
\( 814 \)
Naming Functions

```haskell
λ> (\ s -> s ++ ", " ++ s) "very"
"very, very"

λ> add1 = \ x -> x + 1

λ> :type +d  add1
add1 :: Integer -> Integer

λ> add1 813
814

λ> cutOff = \ i -> max i 0

λ> :type +d  cutOff
cutOff :: Integer -> Integer

λ>
```
Naming Functions

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

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

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

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

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

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

\[
\lambda \text{> cutOff (-10)} \\
0 \\
\lambda \text{> double = } \backslash \ x \rightarrow x \++ \ "", \ " \++ s \\
<\text{interactive}>:23:30: \text{error: Variable not in scope: s :: [Char]} \\
\lambda \text{> double = } \backslash \ s \rightarrow s \++ \ "", \ " \++ s \\
\lambda \text{> :type +d double} \\
\text{double :: [Char] -> [Char]} \\
\lambda \text{> double "much"} \\
\text{"much, much"} \\
\lambda \text{>}
\]
Naming Functions

```haskell
λ> double = \ s -> s ++ ", " ++ s
λ> :type +d double
double :: [Char] -> [Char]
λ> double "much"
"much, much"
λ> cons2 = \ xs -> 2:xs
λ> :type +d cons2
cons2 :: [Integer] -> [Integer]
λ> cons2 [3,4,5]
[2,3,4,5]
λ>
```
Function Declaration Has Special Syntax

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

\[ f \ x = \ \text{if null } x \ \text{then "Empty!" else "Not Empty!"} \]

\[ \text{factorial } n = \ \text{if } n < 2 \ \text{then 1 else } n \ \ast \ \text{factorial } (n-1) \]
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

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

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

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

\[ p_3 \ tr \quad -- \text{evaluates to } 3 \]

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

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

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

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

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