Haskell4Life: Introduction

Sergiu Ivanov

sergiu.ivanov@lacl.fr

Slides and code examples available online:

http://lacl.fr/~sivanov/doku.php?id=en:haskell_for_life

Overlaps Between Programming and Cooking

https://wiki.haskell.org/wikiupload/8/86/HaskellBCurry.jpg

Overlaps Between Programming and Cooking

1. Cooking receipt = program

https://wiki.haskell.org/wikiupload/8/86/HaskellBCurry.jpg

Overlaps Between Programming and Cooking

1. Cooking receipt = program
2. 

Haskell B. Curry

https://wiki.haskell.org/wikiupload/8/86/HaskellBCurry.jpg

Overlaps Between Programming and Cooking

1. Cooking receipt = program
2. 

Haskell B. Curry

https://wiki.haskell.org/wikiupload/8/86/HaskellBCurry.jpg

Overlaps Between Programming and Cooking

1. Cooking receipt = program

2. 

Haskell B. Curry

(joke)

https://wiki.haskell.org/wikiupload/8/86/HaskellBCurry.jpg

Outline

1. The Basics
2. The Types
3. The Functions
4. The Real Functional Programming
5. The Fun

Outline

1. The Basics
2. The Types
3. The Functions
4. The Real Functional Programming
5. The Fun

Getting Started

- Install Haskell: `https://www.haskell.org/platform/`
- Run interactively: `ghci`
- Compile: `ghc source.hs -o executable`
- Get all kinds of help: `https://www.haskell.org/`

Getting Started

- Install Haskell: `https://www.haskell.org/platform/`
- Run interactively: `ghci`
- Compile: `ghc source.hs -o executable`
- Get all kinds of help: `https://www.haskell.org/`

There are Haskell syntax highlighting modules in all major IDEs and Haskell modes in Emacs and Vim.

Saying Hello

```
λ> putStrLn "Hello World!"
Hello World
```

Saying Hello

```
λ> putStrLn "Hello World!"
Hello World
```

- call the function `putStrLn` with with one argument

Saying Hello

```
λ> putStrLn "Hello World!"
Hello World
```

- call the function `putStrLn` with with one argument
- no parentheses
- no commas
- no semicolons

Haskell Calculator

```
λ> 1 + 2
```

Haskell Calculator

```
λ> 1 + 2
3
```

Haskell Calculator

```
λ> 1 + 2
3
```

```
λ> 2 * 3
```

Haskell Calculator

```
λ> 1 + 2
3
```

```
λ> 2 * 3
6
```

Haskell Calculator

```
λ> 1 + 2
3
```

```
λ> 2 * 3
6
```

```
λ> (+) 1 2
```

Haskell Calculator

```
λ> 1 + 2
3
```

```
λ> 2 * 3
6
```

```
λ> (+) 1 2
3
```

Haskell Calculator

```
λ> 1 + 2
3
```

```
λ> 2 * 3
6
```

```
λ> (+) 1 2
3
```

(+) is a function of two arguments.

Haskell Calculator

```
λ> 1 + 2
3
```

```
λ> 2 * 3
6
```

```
λ> (+) 1 2
3
```

`(+)` is a function of two arguments.

Normal functions are usually written in prefix form.

- `putStrLn "Hello"`

Haskell Calculator

```
λ> 1 + 2
3
```

```
λ> 2 * 3
6
```

```
λ> (+) 1 2
3
```

`(+)` is a function of two arguments.

Normal functions are usually written in prefix form.

- `putStrLn "Hello"`

Operators are usually written in infix form.

- `1 + 2`

Haskell Calculator

```
λ> 1 + 2
3
```

```
λ> 2 * 3
6
```

```
λ> (+) 1 2
3
```

`(+)` is a function of two arguments.

Normal functions are usually written in prefix form.

- `putStrLn "Hello"`

Operators are usually written in infix form.

- `1 + 2`

Parentheses are used to write operators in prefix form.

- `(+) 1 2`

Negative Numbers

```
λ> 2 + -3
```

Negative Numbers

```
λ> 2 + -3
<interactive>:12:1-6:
    Precedence parsing error
        cannot mix '+' [infixl 6] and prefix '-' [infixl 6]
        in the same infix expression
```

Negative Numbers

```
λ> 2 + -3
<interactive>:12:1-6:
    Precedence parsing error
        cannot mix '+' [infixl 6] and prefix '-' [infixl 6]
        in the same infix expression
```

Parsing ambiguity!

How does one parse `function - 3`:

- apply `function` to -3?
- subtract 3 from `function`?

Negative Numbers

```
λ> 2 + -3
<interactive>:12:1-6:
    Precedence parsing error
        cannot mix '+' [infixl 6] and prefix '-' [infixl 6]
        in the same infix expression
```

Parsing ambiguity!

How does one parse `function - 3`:

- apply `function` to -3?
- subtract `3` from `function`?

```
λ> 2 + (-3)
-1
```

Boolean Values and Operators

```
λ> True && False
False
```

```
λ> True || False
True
```

Boolean Values and Operators

```
λ> True && False
False
```

```
λ> True || False
True
```

No direct coercion to/from numbers:

```
λ> True && 1
```

... won't work :-)

Comparison Operators

Typical comparison operators:

```
λ> 1 == 2
False
```

```
λ> 1 <= 2
True
```

Comparison Operators

Typical comparison operators:

```
λ> 1 == 2
False
```

```
λ> 1 <= 2
True
```

Unusual "not equals":

```
λ> 1 /= 2
True
```

Variables

```
λ> let a = 1
```

Variables

```
λ> let a = 1
λ> a
1
```

Variables

```
λ> let a = 1
λ> a
1
λ> a + 1
2
```

Variables

```
λ> let a = 1
λ> a
1
λ> a + 1
2
λ> let e = exp 1
λ> e
2.718281828459045
```

Lists

```
λ> [1,2,3]
[1,2,3]
```

Lists

```
λ> [1,2,3]
[1,2,3]

λ> [1..5]
[1,2,3,4,5]
```

Lists

```
λ> [1,2,3]
[1,2,3]

λ> [1..5]
[1,2,3,4,5]

λ> [True,"three"]
```

... type error

Lists

```
λ> [1,2,3]
[1,2,3]

λ> [1..5]
[1,2,3,4,5]

λ> [True,"three"]
```

... type error

All elements of a list must be of the same type.

Lists

```
λ> [1,2,3]
[1,2,3]
```

```
λ> 1:[2,3]
[1,2,3]
```

```
λ> [1..5]
[1,2,3,4,5]
```

```
λ> [True,"three"]
```

... type error

All elements of a list must be of the same type.

Lists

```
λ> [1,2,3]
[1,2,3]
```

```
λ> 1:[2,3]
[1,2,3]
```

```
λ> [1..5]
[1,2,3,4,5]
```

```
λ> [1,2,3]++[4,5]
[1,2,3,4,5]
```

```
λ> [True,"three"]
```

... type error

All elements of a list must be of the same type.

Lists

```
λ> [1,2,3]
[1,2,3]
```

```
λ> 1:[2,3]
[1,2,3]
```

```
λ> [1..5]
[1,2,3,4,5]
```

```
λ> [1,2,3]++[4,5]
[1,2,3,4,5]
```

```
λ> [True,"three"]
```

... type error

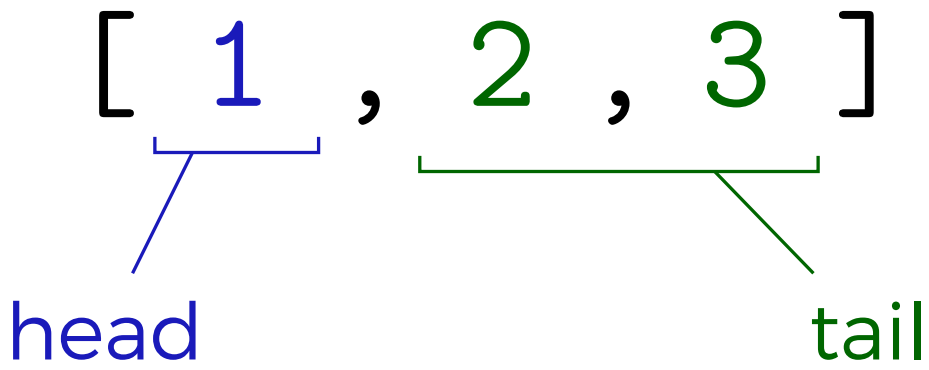
```
λ> length [4,5]
2
```

All elements of a list must be of the same type.

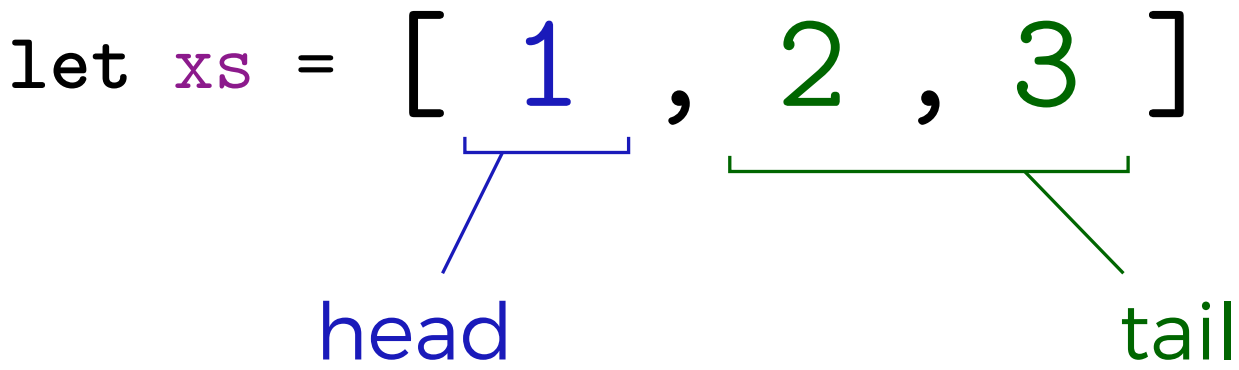
Lists: Heads and Tails

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

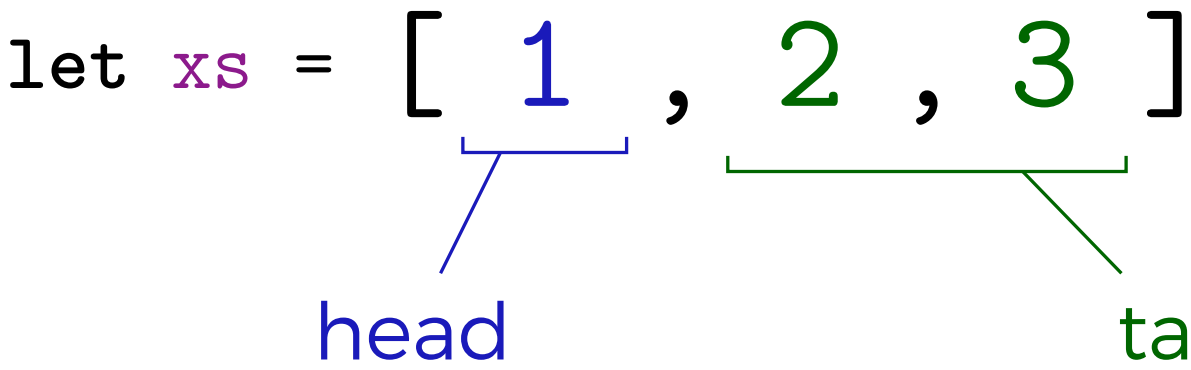
Lists: Heads and Tails



Lists: Heads and Tails



Lists: Heads and Tails




```
λ> head xs
1
```

```
λ> tail xs
[2,3]
```

Lists: Heads and Tails

```
let xs = [ 1 , 2 , 3 ]
```

head

tail

```
λ> head xs
1
```

```
λ> tail xs
[2,3]
```

```
∀ list. list == (head list) : (tail list)
```

```
[1,2,3] ==      1       :     [2,3]
```

Strings as Lists

```
λ> putStrLn "String"
String
```

Strings as Lists

```
λ> putStrLn "String"
String

λ> "String" == ['S','t','r','i','n','g']
True
```

Strings as Lists

```
λ> putStrLn "String"
String

λ> "String" == ['S','t','r','i','n','g']
True

λ> "" == []
True
```

Strings as Lists

```
λ> putStrLn "String"
String

λ> "String" == ['S','t','r','i','n','g']
True

λ> "" == []
True

λ> 'a':"bc"
abc
```

Tuples

```
λ> (1,2,3)
(1,2,3)
```

```
λ> (True,"three")
(True,"three")
```

Tuples can contain values of different types.

Tuples

```
λ> (1,2,3)
(1,2,3)
```

```
λ> (True,"three")
(True,"three")
```

Tuples can contain values of different types.

Tuples are very "strict": no concatenation, no indexing.

Tuples

```
λ> (1,2,3)
(1,2,3)
```

```
λ> (True,"three")
(True,"three")
```

Tuples can contain values of different types.

Tuples are very "strict": no concatenation, no indexing.

```
λ> fst (1,2)
1
```

```
λ> fst (1,2,3)
... type error
```

Tuples

```
λ> (1,2,3)
(1,2,3)
```

```
λ> (True,"three")
(True,"three")
```

Tuples can contain values of different types.

Tuples are very "strict": no concatenation, no indexing.

```
λ> fst (1,2)
1
```

```
λ> fst (1,2,3)
... type error
```

Attention: `fst` is a function applied to one argument: the (two-element) tuple `(1,2)`.

A Simple Program (a reading exercise)

A Simple Program (a reading exercise)

`input`

`input` : : string

A Simple Program (a reading exercise)

```
                                                    lines input
```

```
input          :: string
lines input    :: list of strings (lines)
```

A Simple Program (a reading exercise)

```
                                        length (lines input)
input                   :: string
lines input             :: list of strings (lines)
length (lines input)    :: number of lines
```

A Simple Program (a reading exercise)

```
lineCount input = length (lines input)
```

`input`	:: string
`lines input`	:: list of strings (lines)
`length (lines input)`	:: number of lines
`lineCount input`	:: function of one argument (input)

A Simple Program (a reading exercise)

```
      interact lineCount
where lineCount input =       length (lines input)
```

```
input                 :: string
lines input           :: list of strings (lines)
length (lines input)  :: number of lines
lineCount input       :: function of one argument (input)
```

`interact` is function that "makes `lineCount` interact with the input".

A Simple Program (a reading exercise)

```
      interact lineCount
where lineCount input = show (length (lines input))
```

```
input                  :: string
lines input            :: list of strings (lines)
length (lines input)   :: number of lines
lineCount input        :: function of one argument (input)
```

`interact` is function that "makes `lineCount` interact with the input".

`show` is function that converts "anything" to a string.

A Simple Program (a reading exercise)

```
main = interact lineCount
  where lineCount input = show (length (lines input))
```

`input` :: string

`lines input` :: list of strings (lines)

`length (lines input)` :: number of lines

`lineCount input` :: function of one argument (input)

`interact` is function that "makes `lineCount` interact with the input".

`show` is function that converts "anything" to a string.

`main` is the entry point of the program.

Outline

Types

In Haskell, every expression has a type.

The type of an expression is specified using `::`

Types

In Haskell, every expression has a type.

The type of an expression is specified using `::`

```
λ> :type True
True :: Bool
```

Types

In Haskell, every expression has a type.

The type of an expression is specified using `::`

```
λ> :type True
True :: Bool

λ> :type "Nightwish"
```

Types

In Haskell, every expression has a type.

The type of an expression is specified using `::`

```
λ> :type True
True :: Bool

λ> :type "Nightwish"
"Nightwish" :: [Char]
```

Types

In Haskell, every expression has a type.

The type of an expression is specified using `::`

```
λ> :type True
True :: Bool

λ> :type "Nightwish"
"Nightwish" :: [Char]

λ> :t 2 + 3
```

Types

In Haskell, every expression has a type.

The type of an expression is specified using `::`

```
λ> :type True
True :: Bool

λ> :type "Nightwish"
"Nightwish" :: [Char]

λ> :t 2 + 3
2 + 3 :: Num a => a
```

Types

In Haskell, every expression has a type.

The type of an expression is specified using `::`

```
λ> :type True
True :: Bool

λ> :type "Nightwish"
"Nightwish" :: [Char]

λ> :t 2 + 3
2 + 3 :: Num a => a
```

typeclass

Haskell's Types

Strong

- granular types (lots of details)
- no automatic coercion

Haskell's Types

Strong

- granular types (lots of details)
- no automatic coercion

Static

- the type of anything is known at compile time

Haskell's Types

Strong

- granular types (lots of details)
- no automatic coercion

Static

- the type of anything is known at compile time

Inferred

- the type of anything is deduced automatically
  - almost anything

The Basic Types

`Char`	a Unicode character
`Bool`	`True` or `False`
`Int`	signed, 32- or 64-bit integer values, depending on the architecture
`Integer`	signed integers of unbounded size (more expensive that `Int`)
`Double`	floating-point numbers, typically 64 bits long, uses native floating-point representation
`Float`	narrower than `Double`, but much slower that `Double`, so its use is discouraged

Basic Composite Types: Lists and Tuples

```
λ> [1,2,3] :: [Int]
[1,2,3]
λ> (1,2,3) :: (Int,Int,Int)
(1,2,3)
```

Basic Composite Types: Lists and Tuples

```
λ> [1,2,3] :: [Int]
[1,2,3]
λ> (1,2,3) :: (Int,Int,Int)
(1,2,3)
λ> (True,"three") :: (Bool,String)
(True,"three")
```

A tuple type describes the type of each component.

Basic Composite Types: Lists and Tuples

```
λ> [1,2,3] :: [Int]
[1,2,3]
λ> (1,2,3) :: (Int,Int,Int)
(1,2,3)
λ> (True,"three") :: (Bool,String)
(True,"three")
```

A tuple type describes the type of each component.

`[1,2]` and `[1,2,3]` are of the same type

- `[1,2] :: [Int]` and `[1,2,3] :: [Int]`

`(1,2)` and `(1,2,3)` are of different types

- `(1,2) :: (Int, Int)` and `(1,2,3) :: (Int, Int, Int)`

No Functions Working on All Tuples

`(1,2)` and `(1,2,3)` are of different types

- `(1,2) :: (Int, Int)` and `(1,2,3) :: (Int, Int, Int)`

No Functions Working on All Tuples

`(1,2)` and `(1,2,3)` are of different types

- `(1,2) :: (Int, Int)` and `(1,2,3) :: (Int, Int, Int)`

Conclusion: it is not (directly) possible to define a function accepting tuples of any length.

Outline

Function Types

```
λ> head [1,2,3]
1

λ> tail [1,2,3]
[2,3]
```

Function Types

```
λ> head [1,2,3]
1

λ> tail [1,2,3]
[2,3]

λ> :type head
```

Function Types

```
λ> head [1,2,3]
1

λ> tail [1,2,3]
[2,3]

λ> :type head
head :: [a] -> a
```

Function Types

```
λ> head [1,2,3]
1

λ> tail [1,2,3]
[2,3]

λ> :type head
head :: [a] -> a

λ> :type tail
```

Function Types

```
λ> head [1,2,3]
1

λ> tail [1,2,3]
[2,3]

λ> :type head
head :: [a] -> a

λ> :type tail
tail :: [a] -> [a]
```

Function Types: Polymorphism

```
head :: [a] -> a
```

Function Types: Polymorphism

```
head :: [a] -> a
```

`a` is a type variable : `a` can be of any type.

Function Types: Polymorphism

```
head :: [a] -> a
```

`a` is a type variable : `a` can be of any type.

`head` is a function taking a list of any type and returning an element of that type.

Function Types: Polymorphism

```
head :: [a] -> a
```

`a` is a type variable : `a` can be of any type.

`head` is a function taking a list of any type and returning an element of that type.

`head` is a polymorphic function.

Function Types: Polymorphism

```
head :: [a] -> a
```

`a` is a type variable : `a` can be of any type.

`head` is a function taking a list of any type and returning an element of that type.

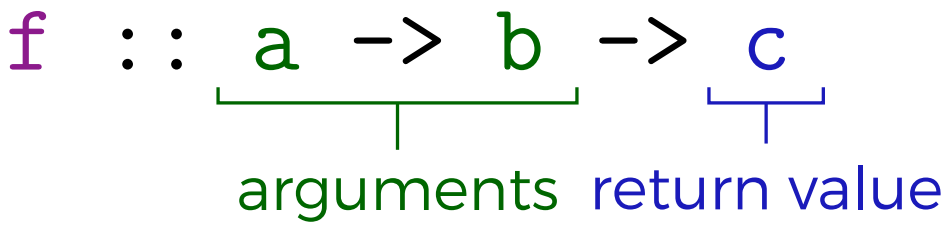
`head` is a polymorphic function.

Types give a rather detailed description of what a function may do.

Function Types: Multiple Arguments

```
f :: a -> b -> c
```

Function Types: Multiple Arguments



Function Types: Multiple Arguments

```
f :: a -> b -> c
```

arguments return value

```
f x y :: c
```

Function Types: Multiple Arguments

```
f :: a -> b -> c
```

arguments return value

```
f x y :: c

f x :: b -> c
```

`f x` is a function of one argument.

Function Types: Multiple Arguments

```
f :: a -> b -> c
```

arguments return value

```
f x y :: c

f x :: b -> c
```

`f x` is a function of one argument.

```
f :: a -> (b -> c)
```

`f` is a function of one argument returning a function of one argument.

Function Types: Multiple Arguments

```
f :: a -> b -> c
```

arguments return value

```
f x y :: c

f x :: b -> c
```

`f x` is a function of one argument.

```
f :: a -> (b -> c)
```

`f` is a function of one argument returning a function of one argument.

`f x` is called partially applying `f`.

Lists: `take` and `drop`

```
λ> :t take
take :: Int -> [a] -> [a]
```

Lists: take and drop

```
λ> :t take
take :: Int -> [a] -> [a]

λ> take 3 [1..5]
```

Lists: take and drop

```
λ> :t take
take :: Int -> [a] -> [a]
λ> take 3 [1..5]
[1,2,3]

λ> :t drop
drop :: Int -> [a] -> [a]
λ> drop 3 [1..5]
```

Lists: take and drop

```
λ> :t take
take :: Int -> [a] -> [a]
λ> take 3 [1..5]
[1,2,3]

λ> :t drop
drop :: Int -> [a] -> [a]
λ> drop 3 [1..5]
[4,5]
```

Higher-order Functions

Higher-order functions take other functions as arguments.

```
dup :: a -> (a -> a -> b) -> b
```

that's one argument

```
dup x f = f x x
```

Outline

if statement

```
if condition
  then expression
  else expression
```

The newlines are not required.

```
λ> if True then putStrLn "yes" else putStrLn "no"
True
```

Recursion: the Factorial

Recursion: the Factorial

```
factorial :: Int -> Int
```

Recursion: the Factorial

```
factorial :: Int -> Int
factorial x = if x == 0
              then 1
              else x * factorial (x-1)
```

Recursion: the Fibonacci Series

Recursion: the Fibonacci Series

```
fibonacci :: Int -> Int
```

Recursion: the Fibonacci Series

```
fibonacci :: Int -> Int
fibonacci n = if n == 1
              then 1
              else if n == 2
                   then 1
                   else fibonacci (n-1)
                        + fibonacci (n-2)
```

Pattern Matching

How do I write `not` ? (for Booleans)

Pattern Matching

How do I write `not` ? (for Booleans)

```
not :: Bool -> Bool
not True = False
not False = True
```

Better Factorial

Better Factorial

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Better Fibonacci Series

Better Fibonacci Series

```
fibonacci :: Int -> Int
fibonacci 1 = 1
fibonacci 2 = 1
fibonacci n = fibonacci (n-1)
            + fibonacci (n-2)
```

Pattern Guards

Pattern Guards

```
fibonacci :: Int -> Int
fibonacci n | n <= 2 = 1
fibonacci n | n > 2 = fibonacci (n-1)
                    + fibonacci (n-2)
```

The first equation will only be picked if `n <= 2`.

The Catch-all Pattern

How to define `isZero`?

The Catch-all Pattern

How to define `isZero`?

```
isZero :: Int -> Bool
isZero 0 = True
isZero _ = False
```

The Catch-all Pattern

How to define `isZero`?

```
isZero :: Int -> Bool
isZero 0 = True
isZero _ = False
```

`_` matches anything, without giving it a name.

Pattern Matching on Lists and Tuples

```
λ> let list = [1,2,3]
λ> let [a,b,c] = list
λ> [c,a,b]
[3,1,2]
```

Pattern Matching on Lists and Tuples

```
λ> let list = [1,2,3]
λ> let [a,b,c] = list
λ> [c,a,b]
[3,1,2]

λ> let (one,_) = (1,2)
λ> one
1
```

Deconstructing Lists

```
λ> let (x:xs) = [1,2,3]
λ> (x,xs)
(1,[2,3])
```

Deconstructing Lists

```
λ> let (x:xs) = [1,2,3]
λ> (x,xs)
(1,[2,3])
```

```
head :: [a] -> a
```

```
tail :: [a] -> [a]
```

Deconstructing Lists

```
λ> let (x:xs) = [1,2,3]
λ> (x,xs)
(1,[2,3])
```

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
```

Deconstructing Lists

```
λ> let (x:xs) = [1,2,3]
λ> (x,xs)
(1,[2,3])
```

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

Outline

Anonymous Functions (Lambda Abstraction)

```
(\ arguments -> expression)
```

Anonymous Functions (Lambda Abstraction)

```
(\ arguments -> expression)
```

```
(\ x y -> x + y)
```

Anonymous Functions (Lambda Abstraction)

```
(\ arguments -> expression)
```

```
(\ x y -> x + y)
```

```
λ> let f = (\ x y -> x + y)
```

Anonymous Functions (Lambda Abstraction)

```
(\ arguments -> expression)
```

```
(\ x y -> x + y)
```

```
λ> let f = (\ x y -> x + y)
λ> f 1 2
3
```

Anonymous Functions (Lambda Abstraction)

```
(\ arguments -> expression)
```

```
(\ x y -> x + y)
```

```
λ> let f = (\ x y -> x + y)
λ> f 1 2
3
```

Same thing as `f x y = x + y`

Sections

Partial application of operators.

```
(+) :: Int -> Int -> Int
```

Sections

Partial application of operators.

```
(+)  :: Int -> Int -> Int
(+ 2) :: Int -> Int
```

Sections

Partial application of operators.

```
   (+)  ::  Int -> Int -> Int
(+ 2)  ::  Int -> Int
(+ 2)  ==  (\ x -> x + 2)
```

Sections

Partial application of operators.

```
(+)  :: Int -> Int -> Int
(+ 2) :: Int -> Int
(+ 2) == (\ x -> x + 2)
```

Note: sections preserve positions of the arguments:

```
(- 2)  ==  (\ x -> x - 2)
```

Sections

Partial application of operators.

```
(+)  :: Int -> Int -> Int
(+ 2) :: Int -> Int
(+ 2) == (\ x -> x + 2)
```

Note: sections preserve positions of the arguments:

```
(- 2) == (\ x -> x - 2)
(2 -) == (\ x -> 2 - x)
```

Folds

Fold $=$ apply a function to every element of the list, accumulating a result on the way.

Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Folds

Fold $=$ apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```

Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```

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

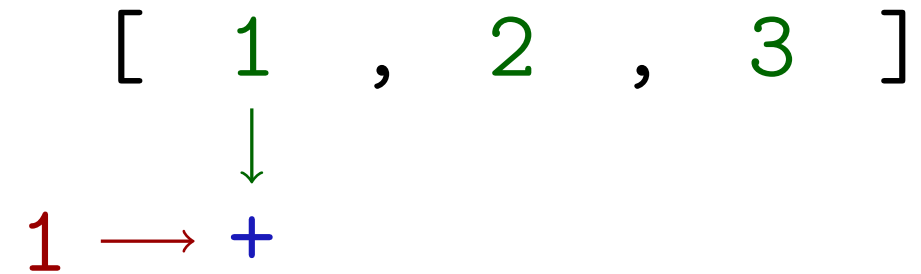
```
1 +
```

Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```

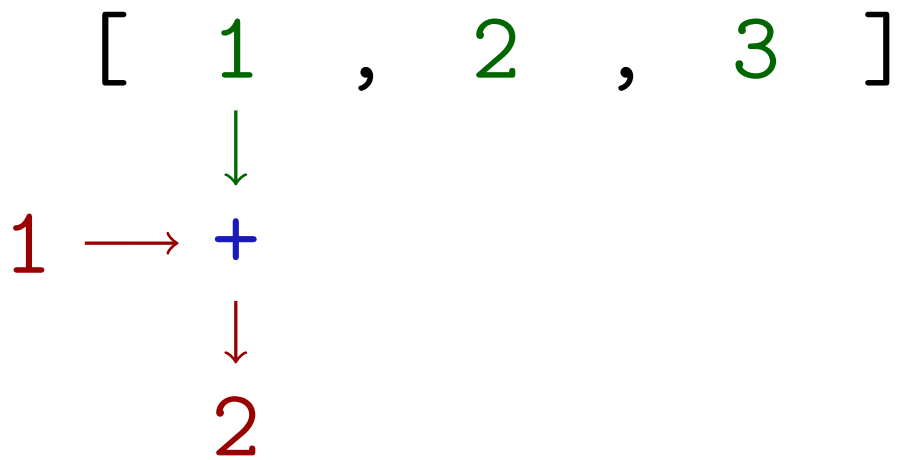


Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```

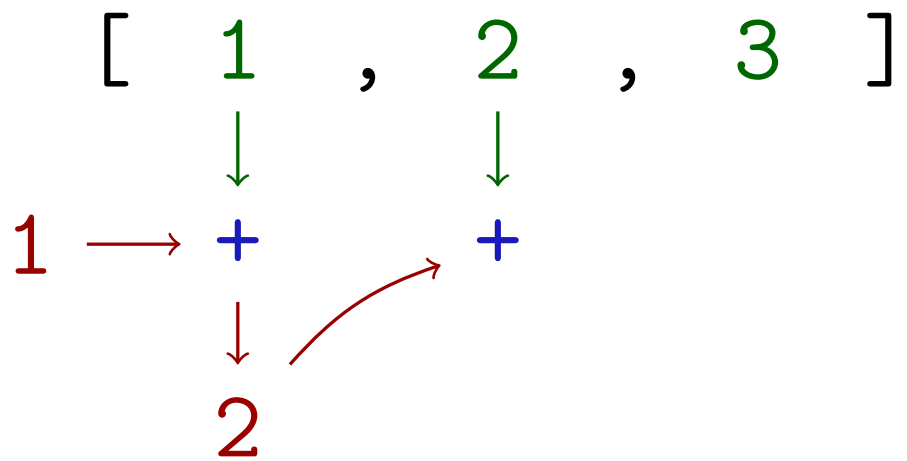


Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```

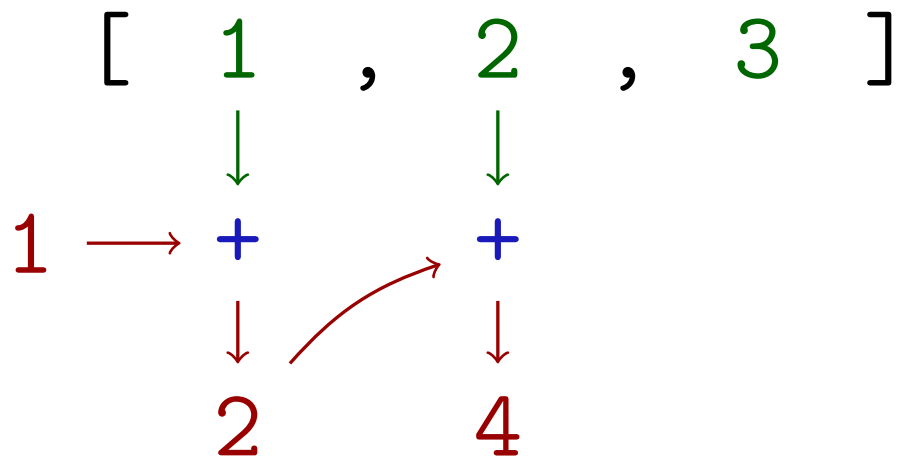


Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```

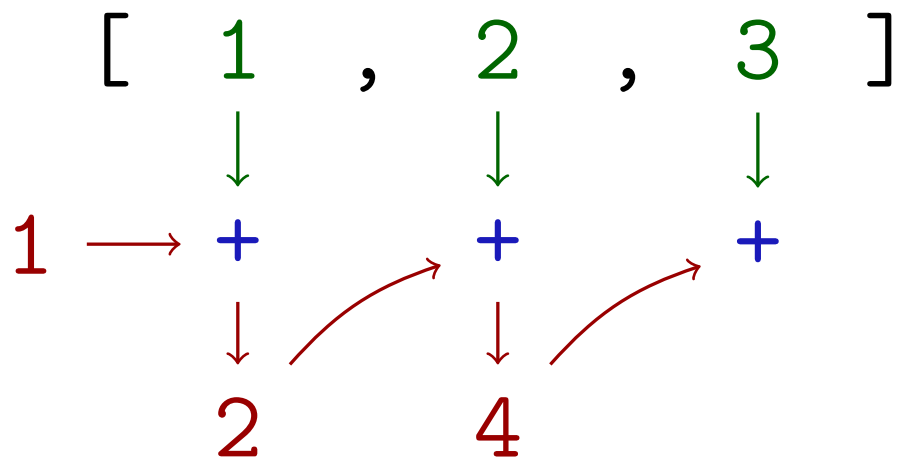


Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```

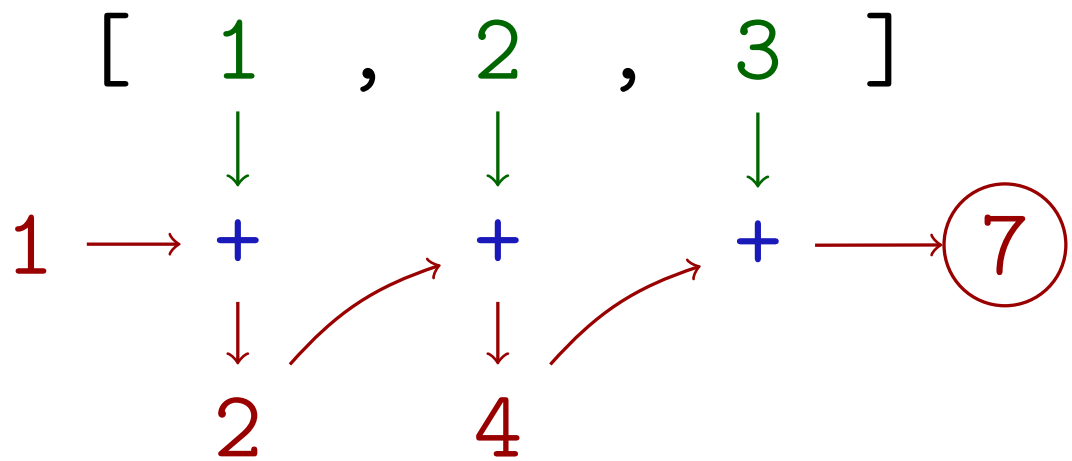


Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```

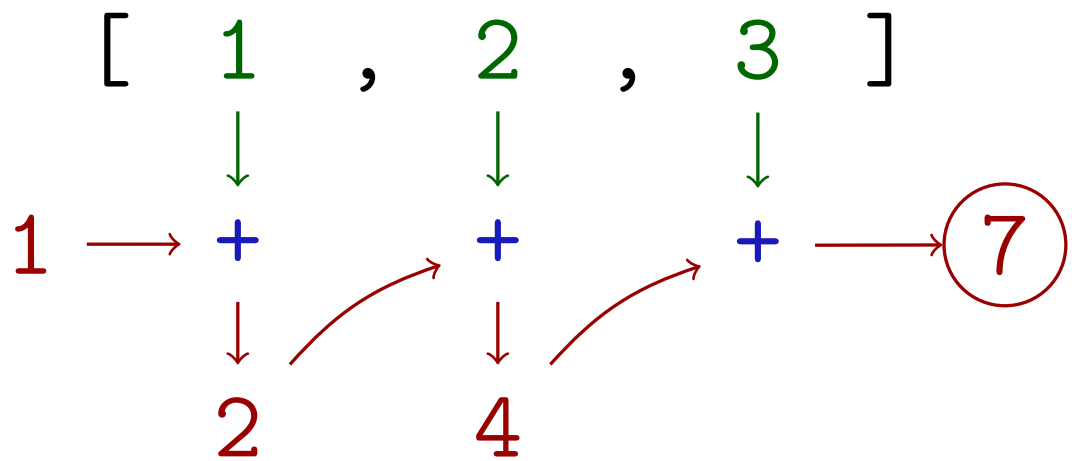


Folds

Fold = apply a function to every element of the list, accumulating a result on the way.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
λ> foldl (+) 1 [1,2,3]
```



`foldr` does the same thing from the right to the left.

The Rest of the Fun is Online

The code shown during The Fun Session is online!