Programming Languages and Paradigms – Part 2

Carol V. Alexandru-Funakoshi, Dr. sc.
Software Evolution and Architecture Lab
University of Zurich

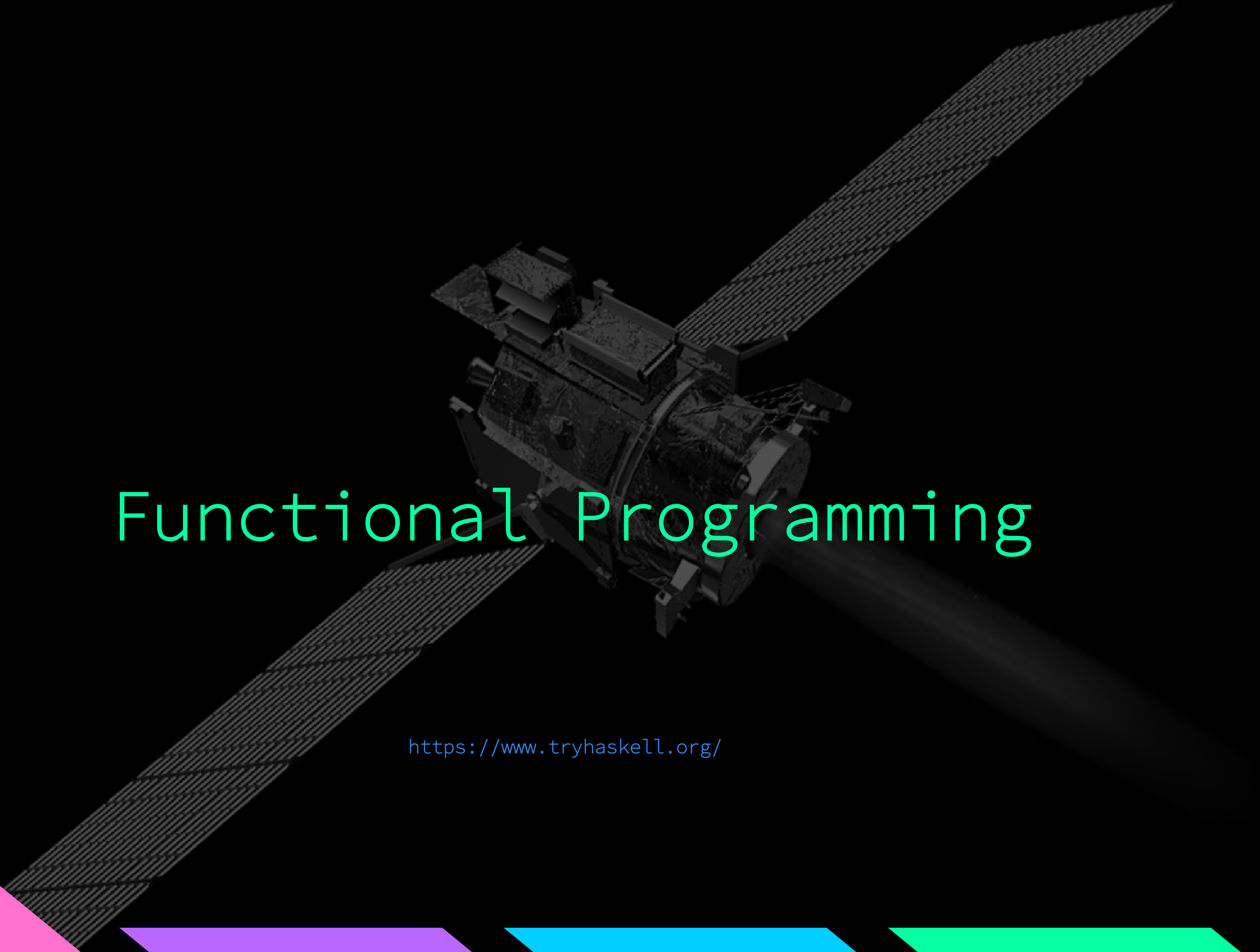
Seminar – Spring 2022

Today's agenda

- **Functional programming at a glance**
  - Functions, composition, higher-level functions, immutability, referential transparency, purity, lazy evaluation, lambda calculus, Hindley–Milner type system, type variables/classes
- **Briefly**
  - Reflection & Macros
  - LLVM, Common Language Interface (CLI), JVM, etc.
  - How **not** to design a language: PHP
  - Esoteric programming languages
- **How we go from here / pick a programming language**

What’s the Take-Away from Today?

- **Recognize and know the terms**
- **Get a feel for functional programming**
  - Don’t worry: understanding takes practical experience
- **Get a tiny bit excited about the concepts?**
- **Do you want to dip your toes into Functional Programming?**
  - Pick a functional programming language to learn



Functional Programming

https://www.tryhaskell.org/

Amuse Bouche

Hello.hs

```
main = putStrLn "Hello, World!"
```

shell

```
sh> ghc --make Hello.hs && ./Hello
[1 of 1] Compiling Main
( Hello.hs, Hello.o )
Linking Hello ...
Hello, World!
```

ghci

```
h> 5 + 1
6
h> 5 + 1.5
6.5
h> 5 / 2
2.5
h> 2 % 4 + 2 % 8
3 % 4
h> 2 ^ 8
256
```

Amuse Bouche

Hello.hs

```
main = putStrLn "Hello, World!"
```

shell

```
sh> ghc --make Hello.hs && ./Hello
[1 of 1] Compiling Main
( Hello.hs, Hello.o )
Linking Hello ...
Hello, World!
```

ghci

```
h> 5 + 1
6
h> 5 + 1.5
6.5
h> 5 / 2
2.5
h> 2 % 4 + 2 % 8
3 % 4
h> 2 ^ 8
256
```

ghci

```
h> [1,2] ++ [3,4]
[1,2,3,4]
h> 1 : [2,3,4]
[1,2,3,4]
h> 1 : 2 : 3 : []
[1,2,3]
h> [1..4]
[1,2,3,4]
h> ['o','k'] ++ "!"
"ok!"
```

Amuse Bouche

Hello.hs

```
main = putStrLn "Hello, World!"
```

shell

```
sh> ghc --make Hello.hs && ./Hello
[1 of 1] Compiling Main
( Hello.hs, Hello.o )
Linking Hello ...
Hello, World!
```

ghci

```
h> 5 + 1
6
h> 5 + 1.5
6.5
h> 5 / 2
2.5
h> 2 % 4 + 2 % 8
3 % 4
h> 2 ^ 8
256
```

ghci

```
h> [1,2] ++ [3,4]
[1,2,3,4]
h> 1 : [2,3,4]
[1,2,3,4]
h> 1 : 2 : 3 : []
[1,2,3]
h> [1..4]
[1,2,3,4]
h> ['o','k'] ++ "!"
"ok!"
```

ghci

```
h> let list = [0..4]
h> list
[0,1,2,3,4]
```

Amuse Bouche

Hello.hs

```haskell
main = putStrLn "Hello, World!"
```

shell

```
sh> ghc --make Hello.hs && ./Hello
[1 of 1] Compiling Main
( Hello.hs, Hello.o )
Linking Hello ...
Hello, World!
```

ghci

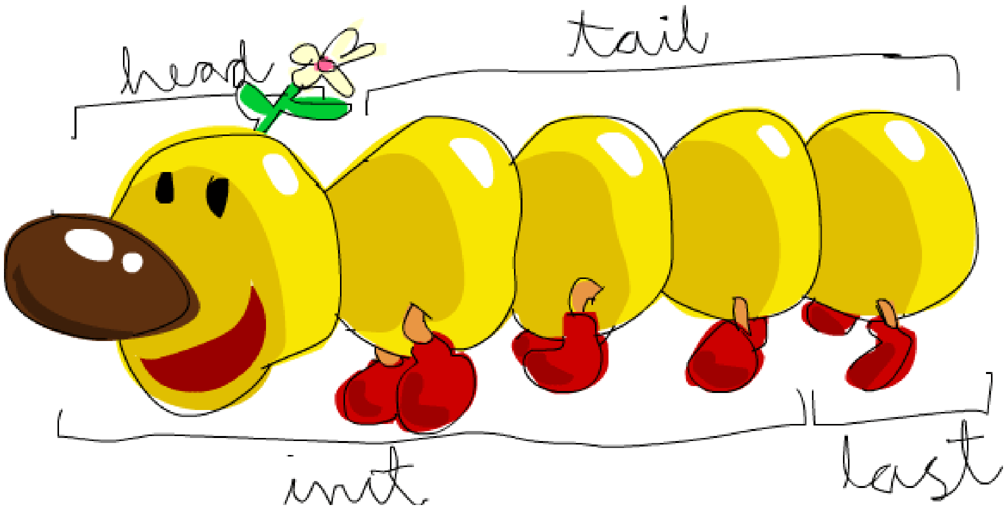
```
h> 5 + 1
6
h> 5 + 1.5
6.5
h> 5 / 2
2.5
h> 2 % 4 + 2 % 8
3 % 4
h> 2 ^ 8
256
```

ghci

```
h> [1,2] ++ [3,4]
[1,2,3,4]
h> 1 : [2,3,4]
[1,2,3,4]
h> 1 : 2 : 3 : []
[1,2,3]
h> [1..4]
[1,2,3,4]
h> ['o','k'] ++ "!"
"ok!"
```

ghci

```
h> let list = [0..4]
h> list
[0,1,2,3,4]
h> head list
0
h> head "hello " : last "hello" : []
"ho"
```



CC-BY-NC-SA 3.0 http://learnyouahaskell.com/starting-out#an-intro-to-lists

Amuse Bouche

Hello.hs

```haskell
main = putStrLn "Hello, World!"
```

shell

```
sh> ghc --make Hello.hs && ./Hello
[1 of 1] Compiling Main
( Hello.hs, Hello.o )
Linking Hello ...
Hello, World!
```

ghci

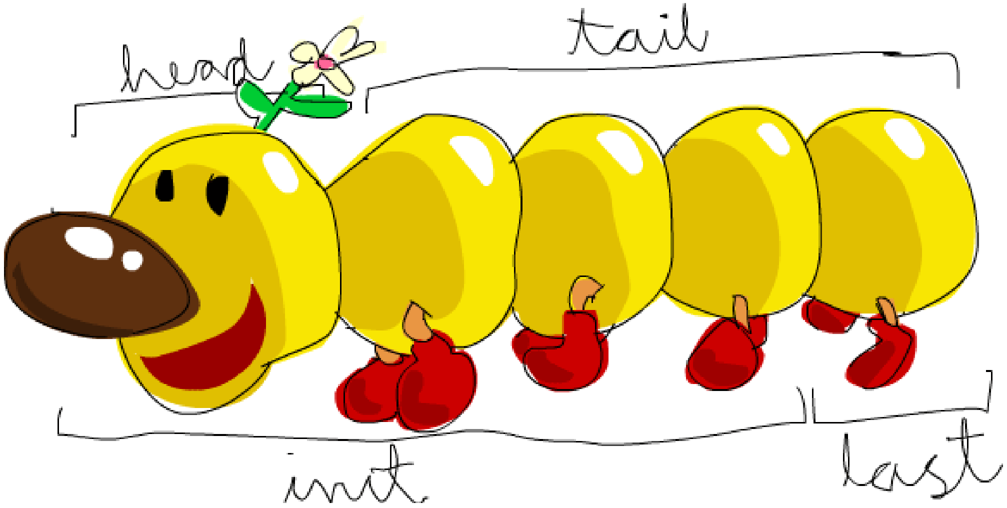
```
h> 5 + 1
6
h> 5 + 1.5
6.5
h> 5 / 2
2.5
h> 2 % 4 + 2 % 8
3 % 4
h> 2 ^ 8
256
```

ghci

```
h> [1,2] ++ [3,4]
[1,2,3,4]
h> 1 : [2,3,4]
[1,2,3,4]
h> 1 : 2 : 3 : []
[1,2,3]
h> [1..4]
[1,2,3,4]
h> ['o','k'] ++ "!"
"ok!"
```

ghci

```
h> let list = [0..4]
h> list
[0,1,2,3,4]
h> head list
0
h> head "hello " : last "hello" : []
"ho"
h> tail "hello"
"ello"
h> head (reverse "hello")
'o'
```



CC-BY-NC-SA 3.0 http://learnyouahaskell.com/starting-out#an-intro-to-lists

Amuse Bouche

Hello.hs

```haskell
main = putStrLn "Hello, World!"
```

shell

```
sh> ghc --make Hello.hs && ./Hello
[1 of 1] Compiling Main
( Hello.hs, Hello.o )
Linking Hello ...
Hello, World!
```

ghci

```
h> 5 + 1
6
h> 5 + 1.5
6.5
h> 5 / 2
2.5
h> 2 % 4 + 2 % 8
3 % 4
h> 2 ^ 8
256
```

ghci

```
h> [1,2] ++ [3,4]
[1,2,3,4]
h> 1 : [2,3,4]
[1,2,3,4]
h> 1 : 2 : 3 : []
[1,2,3]
h> [1..4]
[1,2,3,4]
h> ['o','k'] ++ "!"
"ok!"
```

ghci

```
h> let list = [0..4]
h> list
[0,1,2,3,4]
h> head list
0
h> head "hello " : last "hello" : []
"ho"
h> tail "hello"
"ello"
h> head (reverse "hello")
'o'
h> toUpper 'h'
'H'
h> map toUpper "hello"
"HELLO”
h> [odd 5, even 3]
[True, False]
h> filter odd list
[1,3]
h> filter isNumber "s-07-123-456"
"07123456"
```

What is “Functional” Programming

- **Computation is constructed by**
  - Applying functions to values
  - Binding values to variables
  - Combinding functions & substituting values

λ

What is “Functional” Programming

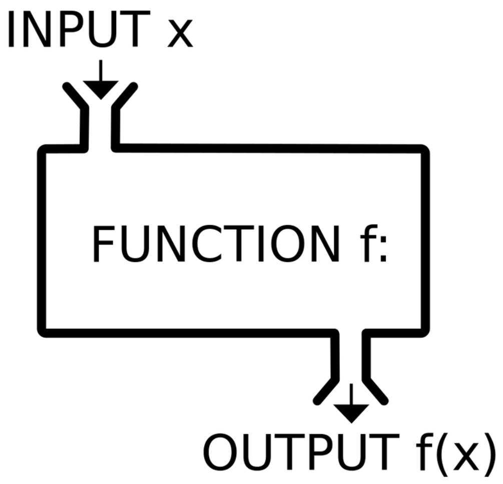
- **Computation is constructed by**
  - Applying functions to values
  - Binding values to variables
  - Combinding functions & substituting values

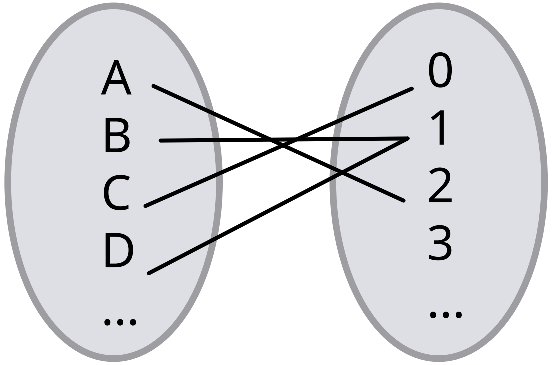
λ

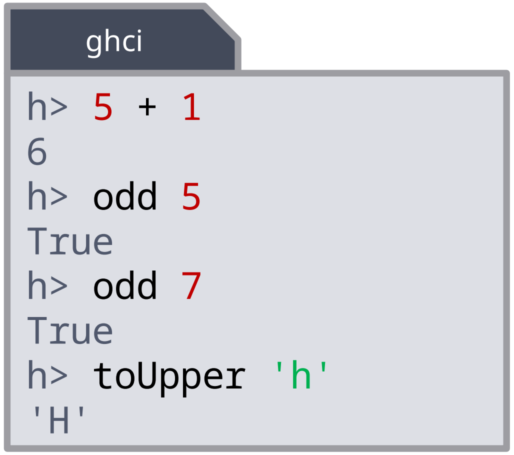
- **Functional peogramming languages typically**
  - Support 1st-class and higher-order functions
  - Avoid state (use immutable data structures)
  - Are declarative and based on lambda calculus

What is a Function?

- **A <u>mapping</u> of input to output**
  - $c = \sqrt{a^2 + b^2}$
  - `hypothenuse(a,b) = sqrt(pow(a,2)+pow(b,2))`
- **Each input mapped to <u>exactly 1</u> output**
  - Multiple inputs can produce the same output



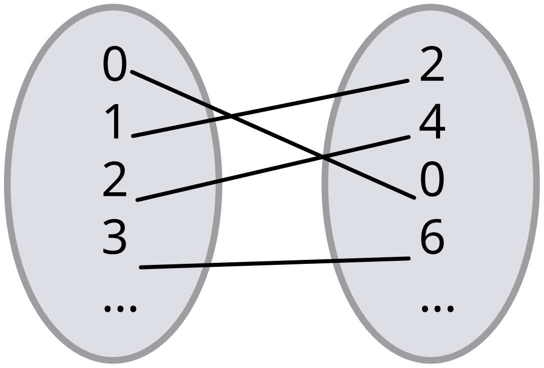




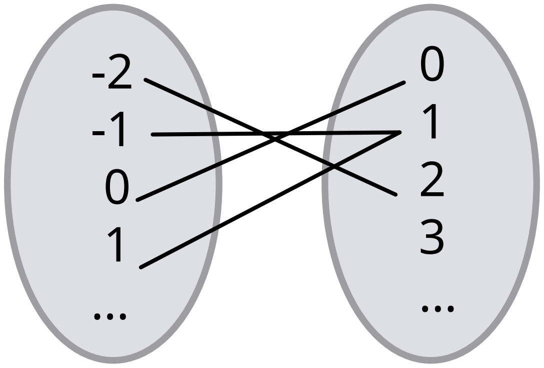
```
h> 5 + 1
6
h> odd 5
True
h> odd 7
True
h> toUpper 'h'
'H'
```

Functions Can Be Composed

f(x) = 2x

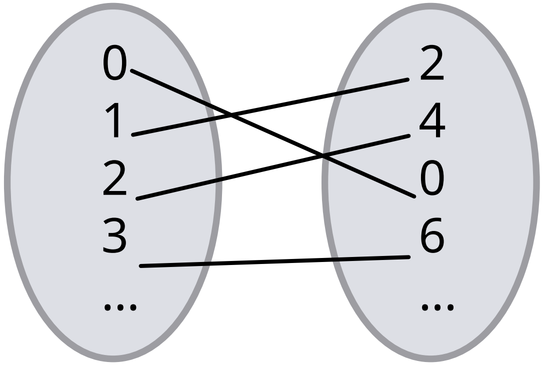


g(x) = |x|

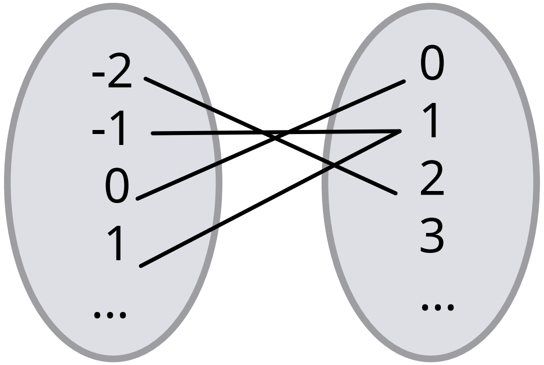


Functions Can Be Composed

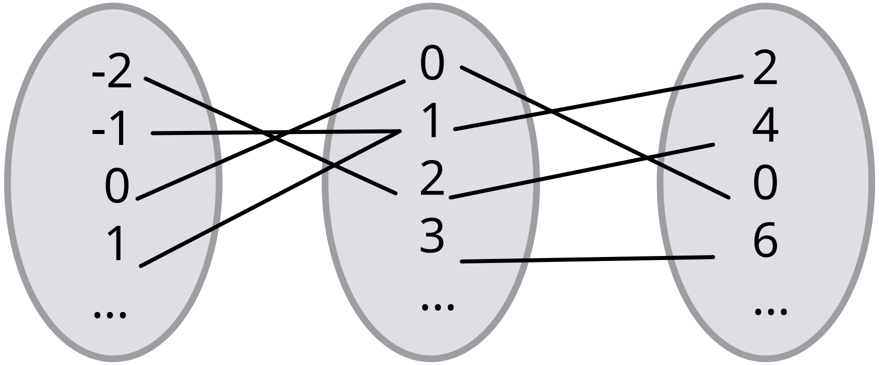
f(x) = 2x



g(x) = |x|

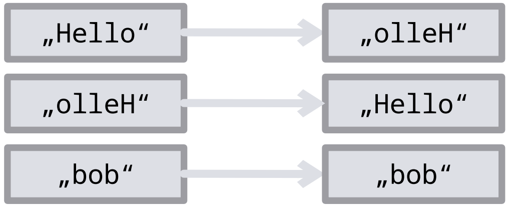


$f(g(x)) = (f \circ g)(x) = 2|x|$



Functions Can Be Composed

„f(x) =~ reverse x“

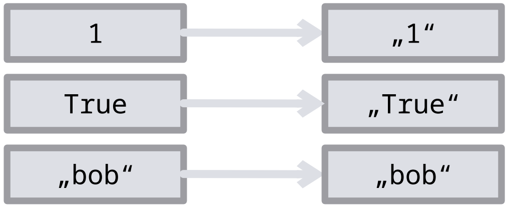


„Hello“ → „olleH“

„olleH“ → „Hello“

„bob“ → „bob“

„g(x) =~ show x“

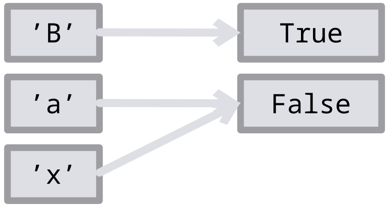


1 → „1“

True → „True“

„bob“ → „bob“

„h(x) =~ isUpper x“



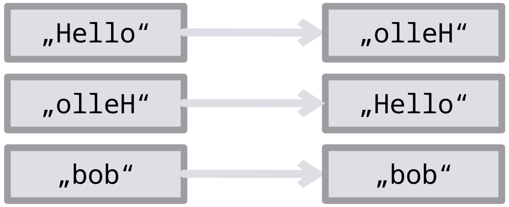
'B' → True

'a' → False

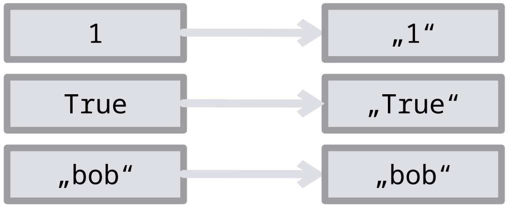
'x' → False

Functions Can Be Composed

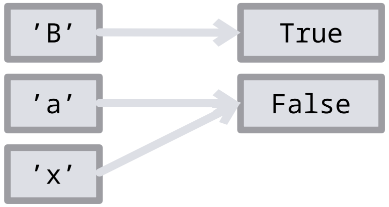
„f(x) =~ reverse x“


„Hello“
„olleH“
„olleH“
„Hello“
„bob“
„bob“

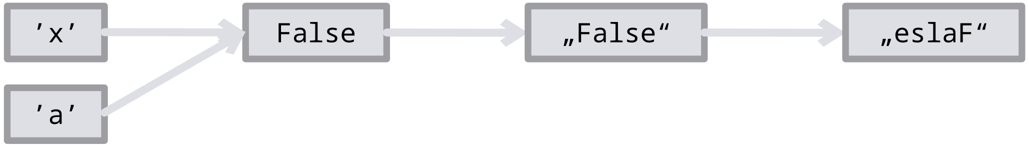
„g(x) =~ show x“


1
„1“
True
„True“
„bob“
„bob“

„h(x) =~ isUpper x“

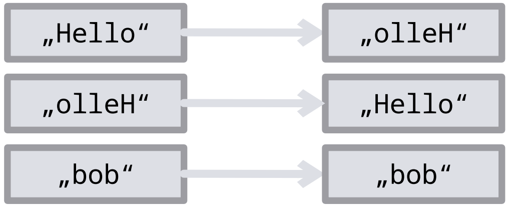

'B'
True
'a'
False
'x'

(f·g·h)(x) =~ (reverse . show . isUpper) x


'x'
'a'
False
„False“
„eslaF“

Functions Can Be Composed

„f(x) =~ reverse x“

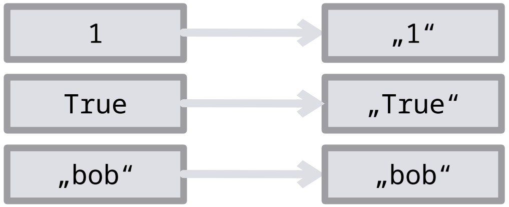


„Hello“ → „olleH“

„olleH“ → „Hello“

„bob“ → „bob“

„g(x) =~ show x“

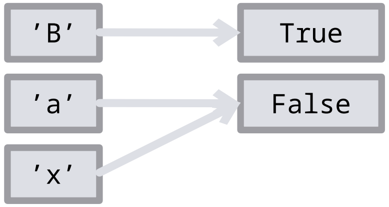


1 → „1“

True → „True“

„bob“ → „bob“

„h(x) =~ isUpper x“

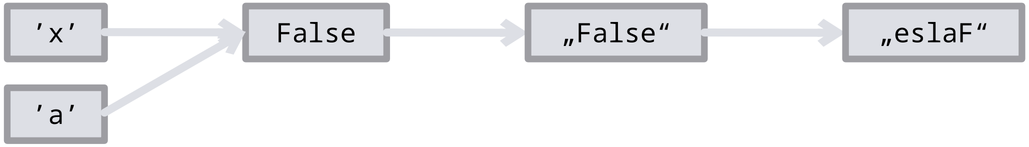


'B' → True

'a' → False

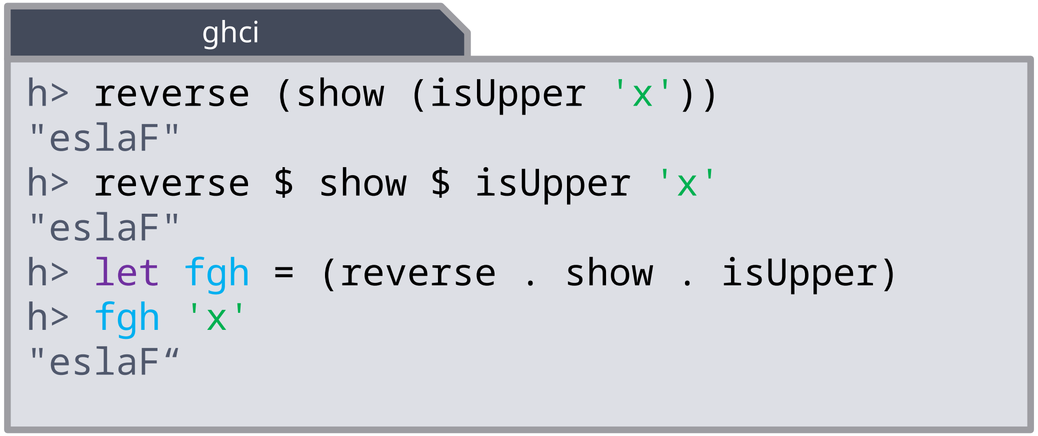
'x' → False

(f·g·h)(x) =~ (reverse . show . isUpper) x



'x' → False → „False“ → „eslaF“

'a' → False



ghci

```
h> reverse (show (isUpper 'x'))
"eslaF"
h> reverse $ show $ isUpper 'x'
"eslaF"
h> let fgh = (reverse . show . isUpper)
h> fgh 'x'
"eslaF“
```

1st-Class and Higher-Order Functions

- **1st-class functions**
  - A function can be treated “like any other value”

python

```
>>> def pow2(x): return x*x
...
>>> a = pow2
>>> e = [1, "hoi", a]
>>> e[2](2)
4
```

1st-Class and Higher-Order Functions

- **1st-class functions**
  - A function can be treated “like any other value”
- **Higher-order functions**
  - Accept or return functions

python

```
>>> def pow2(x): return x*x
...
>>> a = pow2
>>> e = [1, "hoi", a]
>>> e[2](2)
4
```

python

```
>>> def multiplier(x):
...   def inside(param):
...     return param*x
...   return inside
...
>>> m = multiplier(3)
>>> m
<function multiplier.<locals>.inside at 0x7fc8a63dfca0>
>>> m(2)
6
```

python

```
>>> l = [1, 2, 3, 4]
>>> list(map(pow2, l))
[1, 4, 9, 16]
>>> def mymap(f, l):
...   res = []
...   for e in l:
...     res.append(f(e))
...   return res
...
>>> mymap(pow2, l)
[1, 4, 9, 16]
```

Immutability

- A value that has been assigned cannot be changed
- To ‘change’ something, a new value has to be created
- In many languages (including Python), strings are immutable

State.hs

```
x = 1
x = 2
```

shell

```
sh> ghc --make State.hs
[1 of 1] Compiling Main ( State.hs, State.o )

State.hs:2:1: error:
    Multiple declarations of ‘x’
    Declared at: State.hs:1:1
                 State.hs:2:1
  |
2 | x = 2
  | ^
```

python

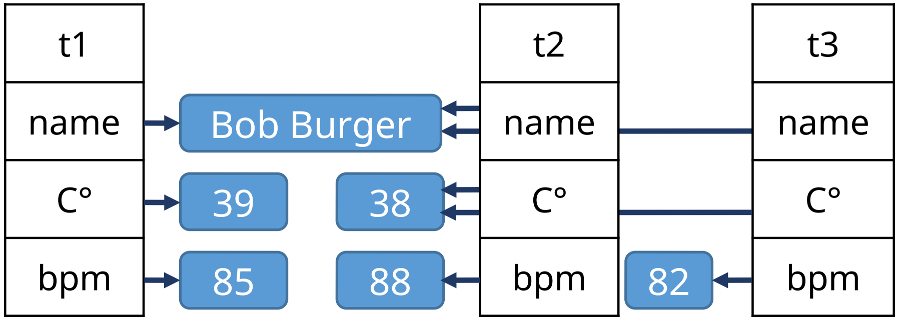
```
>>> s = "abc"
>>> s[1] = 'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
assignment
```

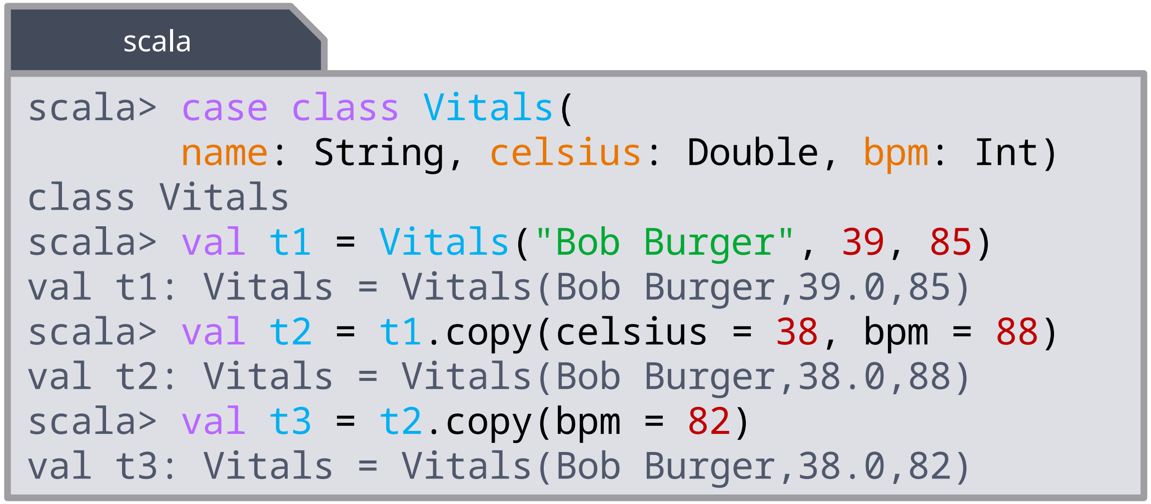
Immutability and Changes to Data

- **Copy on write:**
  - Existing references refer to old data
  - When data is changed, a copy is made for the new reference
- **Monads (change expressed as a pipeline)**

Immutability and Changes to Data

- **Copy on write:**
  - Existing references refer to old data
  - When data is changed, a copy is made for the new reference
- **Monads (change expressed as a pipeline)**
- **References / Pointers (e.g. Scala case classes)**





```
scala> case class Vitals(
       name: String, celsius: Double, bpm: Int)
class Vitals
scala> val t1 = Vitals("Bob Burger", 39, 85)
val t1: Vitals = Vitals(Bob Burger,39.0,85)
scala> val t2 = t1.copy(celsius = 38, bpm = 88)
val t2: Vitals = Vitals(Bob Burger,38.0,88)
scala> val t3 = t2.copy(bpm = 82)
val t3: Vitals = Vitals(Bob Burger,38.0,82)
```

Referential Transparency (rt)

- **An expression is rt if it can be replaced with its value (or vice versa)**
  - Always return the same thing for a given set of inputs
- **Reassignment is not rt**

Referential Transparency (rt)

- **An expression is rt if it can be replaced with its value (or vice versa)**
  - Always return the same thing for a given set of inputs
- **Reassignment is not rt**

python

```
>>> def transparent(x):
...     return x + 1
...
>>> transparent(2)
3
>>> transparent(2)
3
>>> def opaque(x):
...     r = random.randint(1,9)
...     return x + r
...
>>> opaque(2)
3
>>> opaque(2)
5
>>> opaque(2)
8
>>> transparent(2) +
transparent(2) == 3 + 3
True
```

Referential Transparency (rt)

- **An expression is rt if it can be replaced with its value (or vice versa)**
  - Always return the same thing for a given set of inputs
- **Reassignment is not rt**
- **rt has many advantages:**
  - Memoization, common subexpression-elimination, parallelization, lazy evaluation

python

```python
>>> def transparent(x):
...   return x + 1
...
>>> transparent(2)
3
>>> transparent(2)
3
>>> def opaque(x):
...   r = random.randint(1,9)
...   return x + r
...
>>> opaque(2)
3
>>> opaque(2)
5
>>> opaque(2)
8
>>> transparent(2) +
transparent(2) == 3 + 3
True
```

Lazy Evaluation

- **Evaluate function or expression only when needed**
- **Advantages:**
  - Allows for infinite data structures
  - Don’t compute unecessarily
- **Disadvantages:**
  - Hard to reason about memory
- **Opposite: eager evaluation**
- **A.k.a. non-strict evaluation / semantics**

Lazy Evaluation

- **Evaluate function or expression only when needed**

ghci

```
h> let list = [1..]
h> head list
1
h> take 5 list
[1,2,3,4,5]
```

Lazy Evaluation

- **Evaluate function or expression only when needed**

ghci

```
h> let list = [1..]
h> head list
1
h> take 5 list
[1,2,3,4,5]
```

ghci

```
h> take 2 (filter odd [5,2,3,1,4])
[5,3]
h> take 2 (filter odd [0..])
[1,3]
```

Lazy Evaluation

- **Evaluate function or expression only when needed**

ghci

```
h> let list = [1..]
h> head list
1
h> take 5 list
[1,2,3,4,5]
```

python

```
>>> def pow2(x): return x*x
>>> l = [1, 2, 3, 4]
>>> map(pow2, l)
<map object at 0x7f8bb4ef2130>
>>> list(map(pow2, l))
[1, 4, 9, 16]
```

ghci

```
h> take 2 (filter odd [5,2,3,1,4])
[5,3]
h> take 2 (filter odd [0..])
[1,3]
```

Lazy Evaluation

- **Evaluate function or expression only when needed**

ghci

```
h> let list = [1..]
h> head list
1
h> take 5 list
[1,2,3,4,5]
```

python

```
>>> def pow2(x): return x*x
>>> l = [1, 2, 3, 4]
>>> map(pow2, l)
<map object at 0x7f8bb4ef2130>
>>> list(map(pow2, l))
[1, 4, 9, 16]
```

ghci

```
h> take 2 (filter odd [5,2,3,1,4])
[5,3]
h> take 2 (filter odd [0..])
[1,3]
```

Filecopy.hs

```
import System.Environment
main = do
   [inName, outName] <- getArgs
   content <- readFile inName
   writeFile outName content
```

shell

```
s> ghc --make Copy.hs && ./Copy a b
```

Lazy Evaluation

- **Evaluate function or expression only when needed**
- **Advantages:**
  - Allows for infinite data structures
  - Don’t compute unecessarily
- **Disadvantages:**
  - Hard to reason about memory
- **Opposite: eager evaluation**
- **A.k.a. non-strict evaluation / semantics**

ghci

```
h> let list = [1..]
h> head list
1
h> take 5 list
[1,2,3,4,5]
```

python

```
>>> def pow2(x): return x*x
>>> l = [1, 2, 3, 4]
>>> map(pow2, l)
<map object at 0x7f8bb4ef2130>
>>> list(map(pow2, l))
[1, 4, 9, 16]
```

ghci

```
h> take 2 (filter odd [5,2,3,1,4])
[5,3]
h> take 2 (filter odd [0..])
[1,3]
```

Filecopy.hs

```
import System.Environment
main = do
   [inName, outName] <- getArgs
   content <- readFile inName
   writeFile outName content
```

shell

```
s> ghc --make Copy.hs && ./Copy a b
```

Unpure vs. Pure / Side-Effect-Free

- **Pure functions have no side-effects!**
- **Pure functions are stateless, immutable and rt**
  - Easier parallelism, easier debugging
  - Compiler has many guarantees at compile time

Unpure.java

```java
public class Unpure {
  private static int state = 0;
  public static int factorial(int n) {
    state += 666; // Side effects!
    if (n == 1) { return 1; }
    return n * factorial(n-1);
  }
}
```

Unpure vs. Pure / Side-Effect-Free

- **Pure functions have no side-effects!**
- **Pure functions are stateless, immutable and rt**
  - Easier parallelism, easier debugging
  - Compiler has many guarantees at compile time

Unpure.java

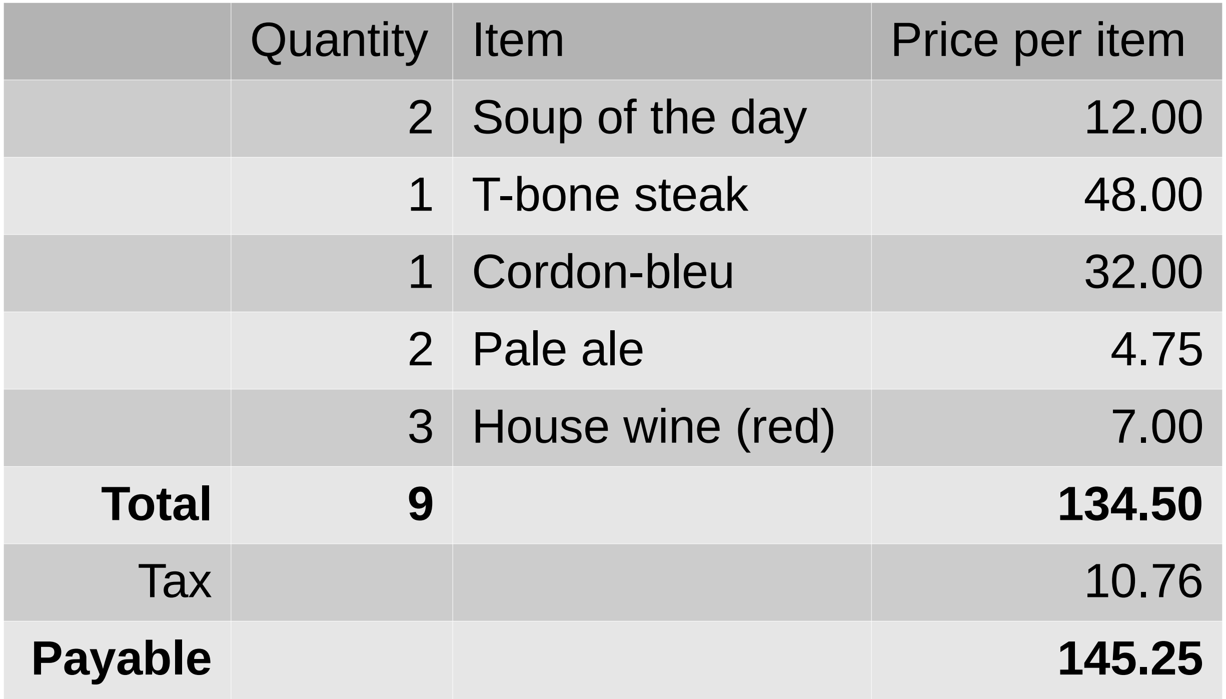
```java
public class Unpure {
  private static int state = 0;
  public static int factorial(int n) {
    state += 666; // Side effects!
    if (n == 1) { return 1; }
    return n * factorial(n-1);
  }
}
```

Pure.hs

```haskell
module Pure (factorial) where

factorial n =
  if n > 0
  then n * factorial (n-1)
  else 1

factorial' n
  | n > 0     = factorial' (n-1)
  | otherwise = 1

factorial'' n = product [1..n]
```

Intuition of Functional Programming

- **Have you ever used Microsoft Excel?**
- **Focus on what should be computed, not how**



	Quantity	Item	Price per item
	2	Soup of the day	12.00
	1	T-bone steak	48.00
	1	Cordon-bleu	32.00
	2	Pale ale	4.75
	3	House wine (red)	7.00
Total	**9**		**134.50**
Tax			10.76
Payable			**145.25**

=sumproduct(B2:B6,D2:D6)

=D7*0.08

=mround(D7+D8,0.05)

Intuition of Functional Programming

- **Have you ever used Microsoft Excel?**
- **Focus on what should be computed, not how**
  - Define each thing as a composition of other things
  - Or as compositions of itself (a.k.a. recursion!)

ghci

```
h> let oddDigits = filter odd [0..9]
h> oddDigits
[1,3,5,7,9]
```

Intuition of Functional Programming

- **Have you ever used Microsoft Excel?**
- **Focus on <u>what</u> should be computed, not <u>how</u>**
  - Define each thing as a composition of other things
  - Or as compositions of itself (a.k.a. recursion!)

ghci

```
h> let oddDigits = filter odd [0..9]
h> oddDigits
[1,3,5,7,9]
```

Simple.hs

```
even' x = x `mod` 2 == 0
odd' x  = not even' x
```

Intuition of Functional Programming

- **Have you ever used Microsoft Excel?**
- **Focus on what should be computed, not how**
  - Define each thing as a composition of other things
  - Or as compositions of itself (a.k.a. recursion!)

ghci

```
h> let oddDigits = filter odd [0..9]
h> oddDigits
[1,3,5,7,9]
```

Simple.hs

```
even' x = x `mod` 2 == 0
odd' x  = not even' x

filter' cond (x:xs) = if cond x then x : rest else rest
        where rest = filter' cond xs
```

Lambda Calculus (1930, Alonzo Church)

- **Simple semantics for computation via function application and binding**
- **Turing complete**
- **Can by typed or untyped**
  - Typed λ-calculus puts restrictions on what types of inputs a function can accept
- **Built on two "simplifications":**
  - Anonymous functions
  - Currying

λ-Calculus: Anonymous Functions

- **Assume the following named functions:**
  - `square_sum(a,b) = ` $a^2 + b^2$
  - `id(x) = x`
- **Functions in λ-calculus have no name, they are "anonymous":**
  - $(a, b) \mapsto a^2 + b^2$
  - $(x) \mapsto x$

λ-Calculus: Currying

- **Functions can only ever take a single parameter but can return functions**
  - $a \mapsto (b \mapsto a^2 + b^2)$
- **Evaluating the original term:**
  - $((a, b) \mapsto a^2 + b^2)(2,3) = 2^2 + 3^2 = 13$

λ-Calculus:
Currying

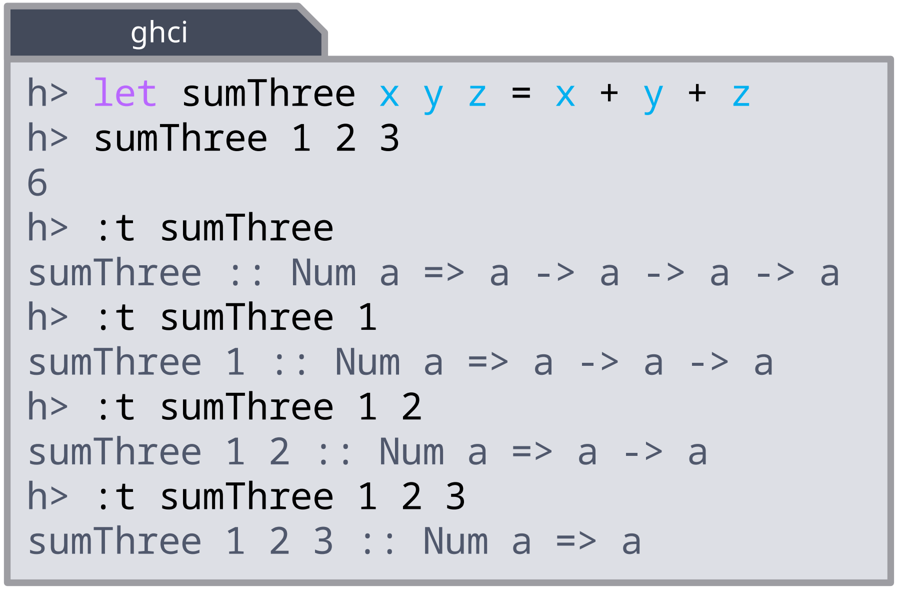
- **Functions can only ever take a single parameter but can return functions**
  - $a \mapsto (b \mapsto a^2 + b^2)$
- **Evaluating the original term:**
  - $((a, b) \mapsto a^2 + b^2)(2,3) = 2^2 + 3^2 = 13$
- **Evaluating the single-parameter functions:**
  - $((a \mapsto (b \mapsto a^2 + b^2))(2))(3) =$
  - $(b \mapsto 2^2 + b^2)(3) = 13$
  - Each of these steps is called a “β-reduction”

λ-Calculus:
Lambda terms

- **A variable: `x`**
- **An abstraction: `λx.t`**
  - "Binds the variable x in the λ-term t"
  - $\lambda x.x^2+5$ corresponds to $f(x) = x^2+5$
- **An application: `ts`**
  - "Calls function t on input s"
  - $(\lambda x.x^2+5)3$ corresponds to $f(3) = 3^2+5$
  - t and s are both λ-terms

λ-Calculus:
What to remember?

- **Identity function:**
  - λx.x / f(x) = x
- **Constant function:**
  - λx.y / f(x) = y
- **Anonymous functions have no name**



```
h> let sumThree x y z = x + y + z
h> sumThree 1 2 3
6
h> :t sumThree
sumThree :: Num a => a -> a -> a -> a
h> :t sumThree 1
sumThree 1 :: Num a => a -> a -> a
h> :t sumThree 1 2
sumThree 1 2 :: Num a => a -> a
h> :t sumThree 1 2 3
sumThree 1 2 3 :: Num a => a
```

- **Functions only ever take 1 parameter, multiple parameters are processed through currying**

Hindley-Milner Type System

- **a.k.a. Damas–Hindley–Milner after**
  - J. Roger Hindley, Robin Milner and Luis Damas
- **Able to infer the most general type of a "program"**
  - Formally defined in "Algorithm W"
- **First implemented in ML, later Haskell**
  - Not all (actually few) functional languages use strong typing.

Types

- **Strong typing, all types inferred/resolved at compile time**

ghci

```
h> 'H'
'H'
h> True
True
h> :t 'H'
'H' :: Char
h> :t True
True :: Bool
h> :t "Hello"
"Hello" :: [Char]
h> :t toUpper
toUpper :: Char -> Char
h> :t isUpper
isUpper :: Char -> Bool
```

Types

- **Strong typing, all types inferred/resolved at compile time**

ghci

```
h> 'H'
'H'
h> True
True
h> :t 'H'
'H' :: Char
h> :t True
True :: Bool
h> :t "Hello"
"Hello" :: [Char]
h> :t toUpper
toUpper :: Char -> Char
h> :t isUpper
isUpper :: Char -> Bool
```

ghci

```
h> let yell s = s ++ "!!!“
h> :t yell
yell :: [Char] -> [Char]
```

Types

- **Strong typing, all types inferred/resolved at compile time**

ghci

```
h> 'H'
'H'
h> True
True
h> :t 'H'
'H' :: Char
h> :t True
True :: Bool
h> :t "Hello"
"Hello" :: [Char]
h> :t toUpper
toUpper :: Char -> Char
h> :t isUpper
isUpper :: Char -> Bool
```

ghci

```
h> let yell s = s ++ "!!!“
h> :t yell
yell :: [Char] -> [Char]
```

Explicit.hs

```
isPalindrome :: [Char] -> Bool
isPalindrome s = s == (reverse s)
```

Types

- **Strong typing, all types inferred/resolved at compile time**

ghci

```
h> 'H'
'H'
h> True
True
h> :t 'H'
'H' :: Char
h> :t True
True :: Bool
h> :t "Hello"
"Hello" :: [Char]
h> :t toUpper
toUpper :: Char -> Char
h> :t isUpper
isUpper :: Char -> Bool
```

ghci

```
h> let yell s = s ++ "!!!“
h> :t yell
yell :: [Char] -> [Char]
```

Explicit.hs

```
isPalindrome :: [Char] -> Bool
isPalindrome s = s == (reverse s)
```

ghci

```
h> isPalindrome "racecar"
True
> :t isPalindrome
isPalindrome :: [Char] -> Bool
```

Type Variables

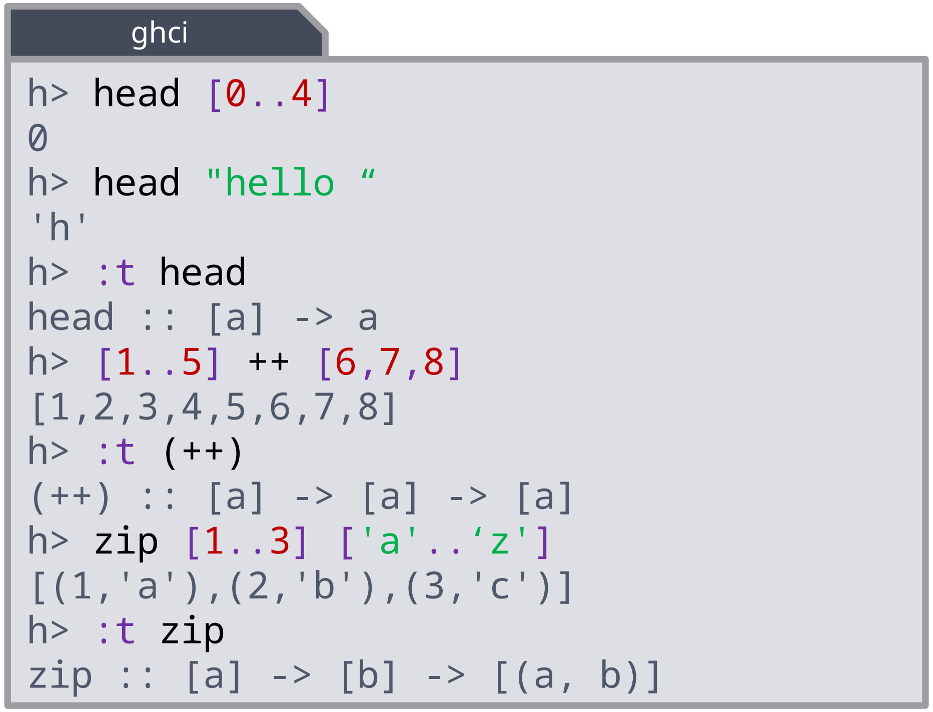
- **Like “Generics” in Java / C++ etc.**

ghci

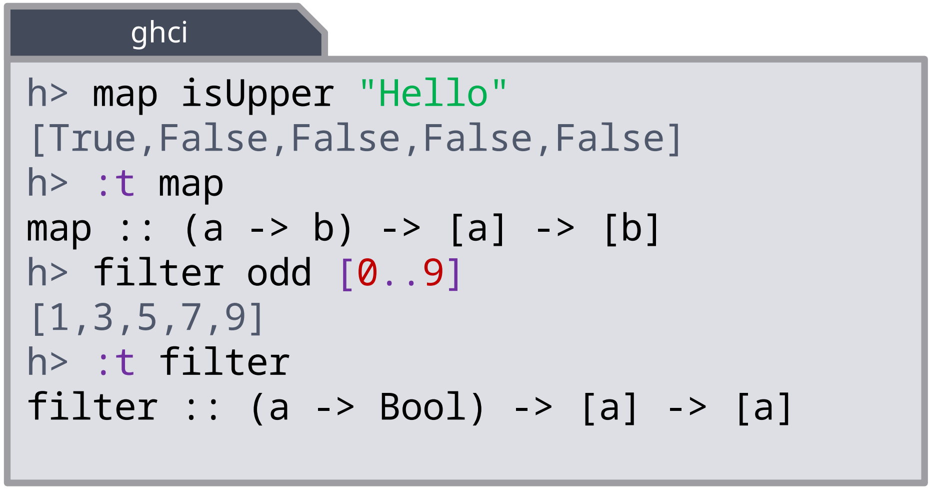
```
h> head [0..4]
0
h> head "hello “
'h'
h> :t head
head :: [a] -> a
h> [1..5] ++ [6,7,8]
[1,2,3,4,5,6,7,8]
h> :t (++)
(++) :: [a] -> [a] -> [a]
h> zip [1..3] ['a'..‘z']
[(1,'a'),(2,'b'),(3,'c')]
h> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Type Variables

- **Like “Generics” in Java / C++ etc.**



```
h> head [0..4]
0
h> head "hello “
'h'
h> :t head
head :: [a] -> a
h> [1..5] ++ [6,7,8]
[1,2,3,4,5,6,7,8]
h> :t (++)
(++) :: [a] -> [a] -> [a]
h> zip [1..3] ['a'..‘z']
[(1,'a'),(2,'b'),(3,'c')]
h> :t zip
zip :: [a] -> [b] -> [(a, b)]
```



```
h> map isUpper "Hello"
[True,False,False,False,False]
h> :t map
map :: (a -> b) -> [a] -> [b]
h> filter odd [0..9]
[1,3,5,7,9]
h> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

Type Variables

- **Like “Generics” in Java / C++ etc.**

ghci

```
h> head [0..4]
0
h> head "hello “
'h'
h> :t head
head :: [a] -> a
h> [1..5] ++ [6,7,8]
[1,2,3,4,5,6,7,8]
h> :t (++)
(++) :: [a] -> [a] -> [a]
h> zip [1..3] ['a'..‘z']
[(1,'a'),(2,'b'),(3,'c')]
h> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

ghci

```
h> map isUpper "Hello"
[True,False,False,False,False]
h> :t map
map :: (a -> b) -> [a] -> [b]
h> filter odd [0..9]
[1,3,5,7,9]
h> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

Invalid.hs

```
~~isPalindrome :: [Char] -> Bool~~
isPalindrome :: [a] -> Bool
isPalindrome s = s == (reverse s)
```

Type Variables

- **Like “Generics” in Java / C++ etc.**

ghci

```
h> head [0..4]
0
h> head "hello “
'h'
h> :t head
head :: [a] -> a
h> [1..5] ++ [6,7,8]
[1,2,3,4,5,6,7,8]
h> :t (++)
(++) :: [a] -> [a] -> [a]
h> zip [1..3] ['a'..‘z']
[(1,'a'),(2,'b'),(3,'c')]
h> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Invalid.hs

```
~~isPalindrome :: [Char] -> Bool~~
isPalindrome :: [a] -> Bool
isPalindrome s = s == (reverse s)
```

ghci

```
h> map isUpper "Hello"
[True,False,False,False,False]
h> :t map
map :: (a -> b) -> [a] -> [b]
h> filter odd [0..9]
[1,3,5,7,9]
h> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

shell

```
sh> ghc --make Invalid.hs
Invalid.hs:2:18: error:
    • No instance for (Eq a) arising
from a use of ‘==’
      [...]
  |
2 | isPalindrome s = s == (reverse s)
  |                  ^^^^^^^^^^^^^^^^
```

Type Classes

- **Like “Interfaces”**

Inferred.hs

```haskell
isPalindrome s = s == (reverse s)
```

ghci

```
h> isPalindrome "racecar"
True
h> isPalindrome [1,2,3,2,2]
False
h> isPalindrome [True, False, True]
True
```

Type Classes

- **Like "Interfaces"**

Inferred.hs

```
isPalindrome s = s == (reverse s)
```

ghci

```
h> isPalindrome "racecar"
True
h> isPalindrome [1,2,3,2,2]
False
h> isPalindrome [True, False, True]
True
h> :t isPalindrome
isPalindrome :: Eq a => [a] -> Bool
```

The HM type system did this!

Type Classes

- **Like "Interfaces"**

Inferred.hs

```haskell
isPalindrome s = s == (reverse s)
```

ghci

```
h> isPalindrome "racecar"
True
h> isPalindrome [1,2,3,2,2]
False
h> isPalindrome [True, False, True]
True
h> :t isPalindrome
isPalindrome :: Eq a => [a] -> Bool
```

The HM type system did this!

ghci

```
h> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Type Classes

- **Like "Interfaces"**

Inferred.hs

```
isPalindrome s = s == (reverse s)
```

ghci

```
h> isPalindrome "racecar"
True
h> isPalindrome [1,2,3,2,2]
False
h> isPalindrome [True, False, True]
True
h> :t isPalindrome
isPalindrome :: Eq a => [a] -> Bool
```

The HM type system did this!

ghci

```
h> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

ghci

```
h> :i Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

ghci

```
> :i Char
instance Bounded Char
instance Enum Char
instance Eq Char
instance Ord Char
instance Read Char
instance Show Char
```

Side-effects in a Pure Language?

- **E.g. Haskell:** do **blocks and the IO Type**

Converse.hs

```haskell
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  putStrLn $ "Welcome, " ++ name ++ "!"
```

shell

```
s> ghc --make Converse.hs
[1 of 1] Compiling Main
    ( Converse.hs, Converse.o )
Linking Converse ...
s> ./Converse
Hello!  What is your name?
Spock
Welcome, Spock!
```

Side-effects in a Pure Language?

- **E.g. Haskell:** do **blocks and the IO Type**

Converse.hs

```
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  putStrLn $ "Welcome, " ++ name ++ "!"
```

ghci

```
h> :t putStrLn
putStrLn :: String -> IO ()
h> :t getLine
getLine :: IO String
```

shell

```
s> ghc --make Converse.hs
[1 of 1] Compiling Main
    ( Converse.hs, Converse.o )
Linking Converse ...
s> ./Converse
Hello!  What is your name?
Spock
Welcome, Spock!
```

Side-effects in a Pure Language?

- **E.g. Haskell:** do **blocks and the IO Type**

Converse.hs

```
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  putStrLn $ "Welcome, " ++ name ++ "!"
```

ghci

```
h> :t putStrLn
putStrLn :: String -> IO ()
h> :t getLine
getLine :: IO String
```

shell

```
s> ghc --make Converse.hs
[1 of 1] Compiling Main
    ( Converse.hs, Converse.o )
Linking Converse ...
s> ./Converse
Hello!  What is your name?
Spock
Welcome, Spock!
```

Converse.hs

```
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  putStrLn $ greet name

greet n = "Welcome, " ++ n ++ "!"
```

ghci

```
h> :t greet
greet :: [Char] -> [Char]
```

Side-effects in a Pure Language?

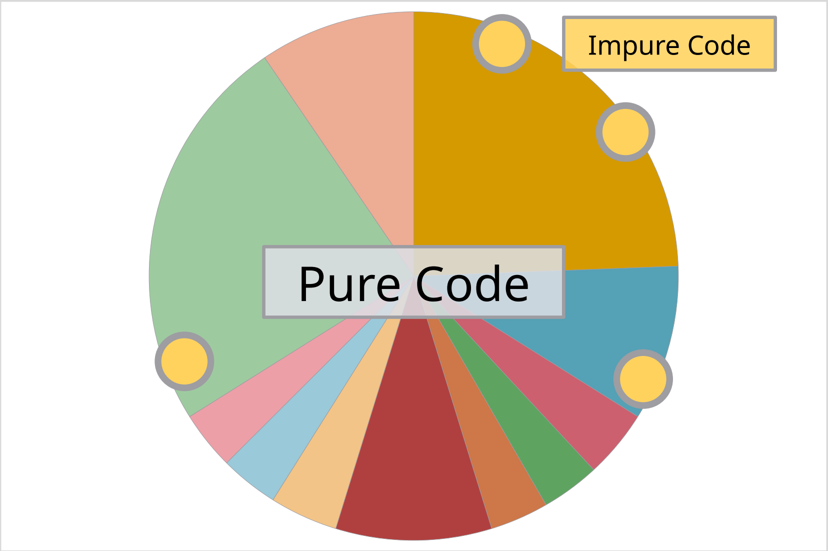
- **E.g. Haskell:** do **blocks and the IO Type**

Converse.hs

```
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  putStrLn $ "Welcome, " ++ name ++ "!"
```

ghci

```
h> :t putStrLn
putStrLn :: String -> IO ()
h> :t getLine
getLine :: IO String
```



Converse.hs

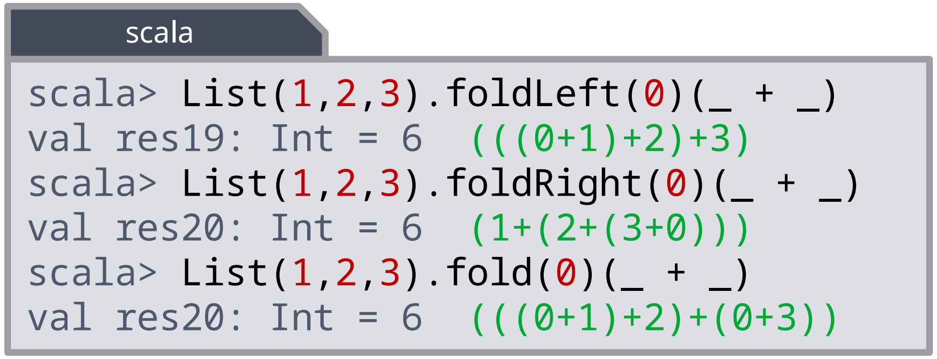
```
main = do
  putStrLn "Hello! What is your name?"
  name <- getLine
  putStrLn $ greet name

greet n = "Welcome, " ++ n ++ "!"
```

ghci

```
h> :t greet
greet :: [Char] -> [Char]
```

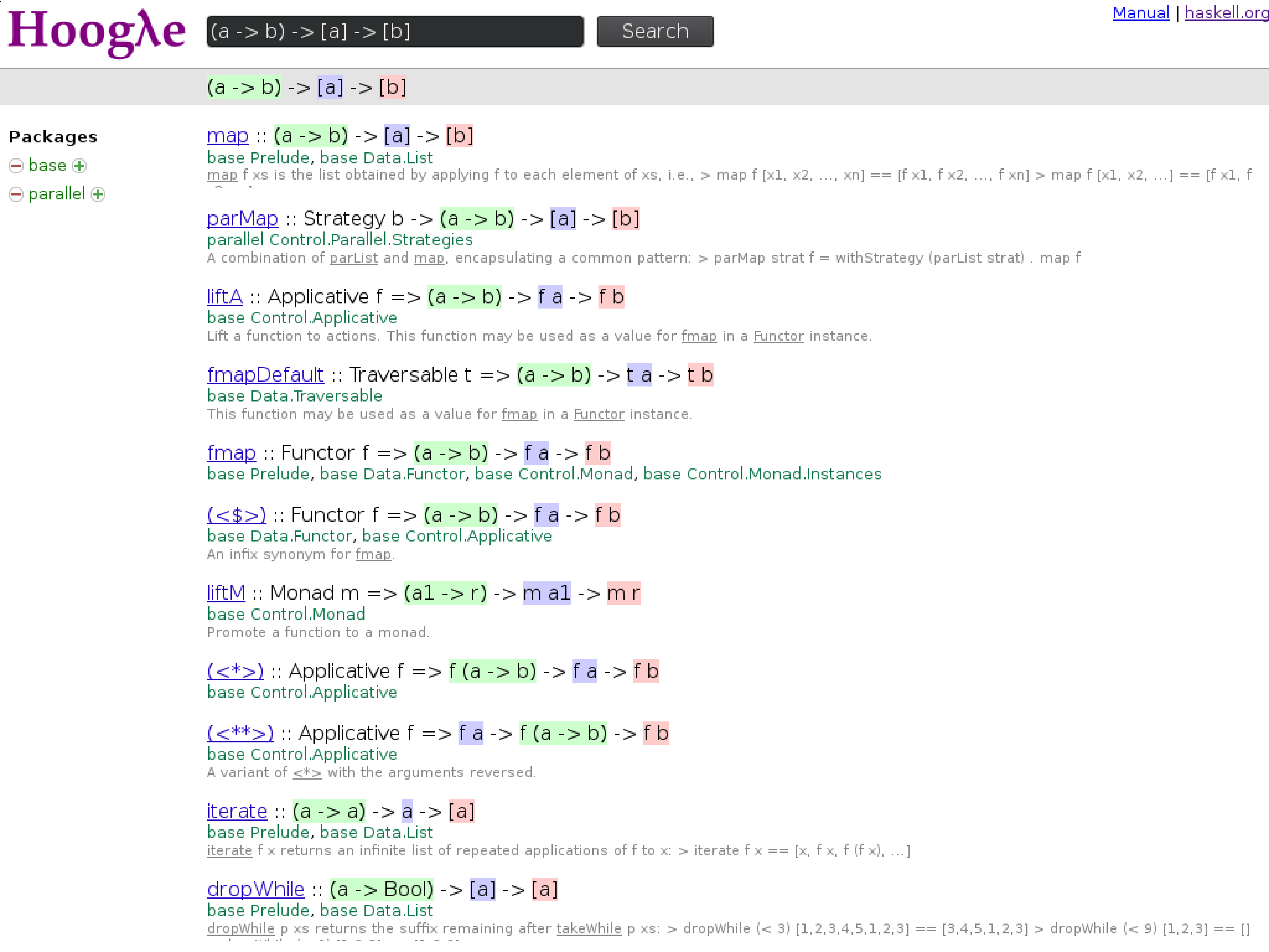
Rich Vocab in Functional Programming

- **Folds / Folding**
  - Left, Right, Parallel


scala

```
scala> List(1,2,3).foldLeft(0)(_ + _)
val res19: Int = 6  (((0+1)+2)+3)
scala> List(1,2,3).foldRight(0)(_ + _)
val res20: Int = 6  (1+(2+(3+0)))
scala> List(1,2,3).fold(0)(_ + _)
val res20: Int = 6  (((0+1)+2)+(0+3))
```

- **Zippers**
  - How to “navigate” data structures in a functional way
- **Functors**
  - Things that can be “mapped over”
- **Monads**
  - “Pipelines”, formally pure state and state change



Hoogλe (a -> b) -> [a] -> [b] Search

Manual | haskell.org

(a -> b) -> [a] -> [b]

Packages

⊖ base ⊕

⊖ parallel ⊕

map :: (a -> b) -> [a] -> [b]
base Prelude, base Data.List
map f xs is the list obtained by applying f to each element of xs, i.e., > map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn] > map f [x1, x2, ...] == [f x1, f

parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parallel Control.Parallel.Strategies
A combination of parList and map, encapsulating a common pattern: > parMap strat f = withStrategy (parList strat) . map f

liftA :: Applicative f => (a -> b) -> f a -> f b
base Control.Applicative
Lift a function to actions. This function may be used as a value for fmap in a Functor instance.

fmapDefault :: Traversable t => (a -> b) -> t a -> t b
base Data.Traversable
This function may be used as a value for fmap in a Functor instance.

fmap :: Functor f => (a -> b) -> f a -> f b
base Prelude, base Data.Functor, base Control.Monad, base Control.Monad.Instances

(<$>) :: Functor f => (a -> b) -> f a -> f b
base Data.Functor, base Control.Applicative
An infix synonym for fmap.

liftM :: Monad m => (a1 -> r) -> m a1 -> m r
base Control.Monad
Promote a function to a monad.

(<*>) :: Applicative f => f (a -> b) -> f a -> f b
base Control.Applicative

(<**>) :: Applicative f => f a -> f (a -> b) -> f b
base Control.Applicative
A variant of <*> with the arguments reversed.

iterate :: (a -> a) -> a -> [a]
base Prelude, base Data.List
iterate f x returns an infinite list of repeated applications of f to x: > iterate f x == [x, f x, f (f x), ...]

dropWhile :: (a -> Bool) -> [a] -> [a]
base Prelude, base Data.List
dropWhile p xs returns the suffix remaining after takeWhile p xs: > dropWhile (< 3) [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3] > dropWhile (< 9) [1,2,3] == []

Why Functional Programming?

- **Declarative, non-procedural programming style has many advantages:**
  - Easy to reason about side-effects (usually none)
  - When programming, you ask yourself “What **is** an x/y/z?” instead of “How do I get to an x/y/z?”
- **What’s the catch?**
  - A couple of “strange” concepts need to be learned
  - Some things are slightly harder to express than in imperative languages – but can be worth it!

Some more things...

Reflection

- **Program can inspect and change itself at runtime**

python

```
>>> type(1)
<class 'int'>
>>> type(int)
>>> dir("hi")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
>>> def pow2(x): return x*x
...
>>> pow2.__code__.co_argcount
1
```

Reflection

- **Program can inspect and change itself at runtime**

python

```
>>> type(1)
<class 'int'>
>>> type(int)
>>> dir("hi")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
>>> def pow2(x): return x*x
...
>>> pow2.__code__.co_argcount
1
```

python

```
>>> def say(x): print(x)
...
>>> say("hi")
hi
>>> say.__code__ = (lambda x: print("no!")).__code__
>>> say("hi")
no!
```

Reflection

- **Program can inspect and change itself at runtime**

python

```python
>>> type(1)
<class 'int'>
>>> type(int)
>>> dir("hi")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
>>> def pow2(x): return x*x
...
>>> pow2.__code__.co_argcount
1
```

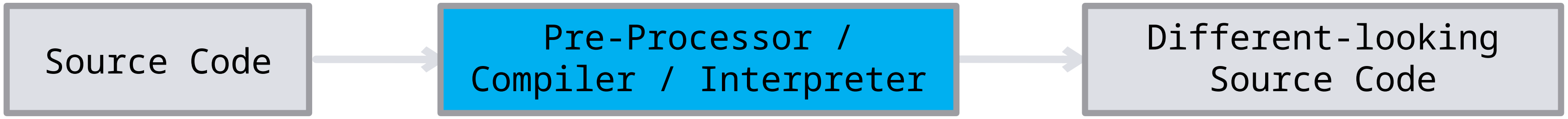
python

```python
>>> def say(x): print(x)
...
>>> say("hi")
hi
>>> say.__code__ = (lambda x: print("no!")).__code__
>>> say("hi")
no!
>>> MyClass = type("MyClass", (), {"prop": 10})
>>> m = MyClass()
>>> m.prop
10
```

- **Powerful tool, but can make debugging trickier**

Macros

- **Generally: Substitute one (simple) piece of code with another (more complex) piece of code**
- **Fancy: Syntactic extensions to a language**



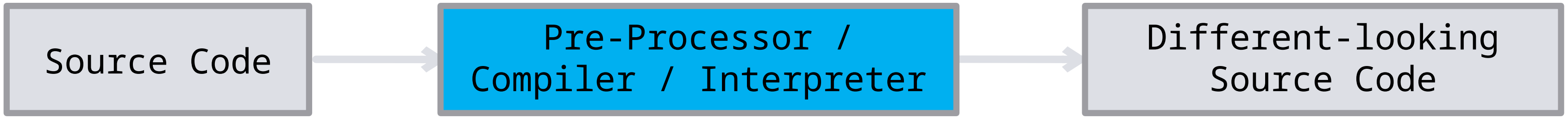
Source Code

Pre-Processor / Compiler / Interpreter

Different-looking Source Code

Macros

- **Generally: Substitute one (simple) piece of code with another (more complex) piece of code**
- **Fancy: Syntactic extensions to a language**


Source Code
Pre-Processor / Compiler / Interpreter
Different-looking Source Code

example.c

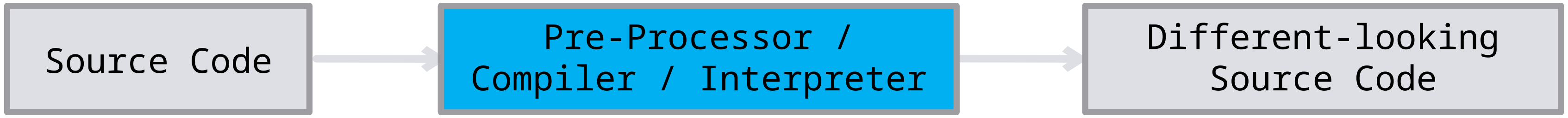
```
#define BUFFER_SIZE 1024
foo = (char *) malloc (BUFFER_SIZE);
```

What the C compiler actually sees

```
foo = (char *) malloc (1024);
```

Macros

- **Generally: Substitute one (simple) piece of code with another (more complex) piece of code**
- **Fancy: Syntactic extensions to a language**



Source Code

Pre-Processor / Compiler / Interpreter

Different-looking Source Code

example.c

```c
#define BUFFER_SIZE 1024
foo = (char *) malloc (BUFFER_SIZE);
```

What the C compiler actually sees

```c
foo = (char *) malloc (1024);
```

In Lisp

```lisp
;; setf sets a variable to some value
(setf x whatever)
;; rplaca sets the first elem of a list
(rplaca somelist whatever)
;; car gets the first element of a list
(car somelist)
;; setf is actually a macro! When called:
(setf (car somelist) whatever)
;; the setf macro automatically rewrites to
(rplaca somelist whatever)
```

JVM, CLR, ART, LLVM, ...

- **“Virtual” Machines for portability**
- **Java Virtual Machine (JVM, 1994):**
  - Multiple Implementations (OpenJDK, Oracle, OpenJ9)

    Languages: Java, Groovy, Scala, Kotlin, Clojure, but also: COBOL, Lisp, Haskell, JavaScript, Ocaml, Pascal, PHP, Prolog, Python, R, Simula, Visual Basic…
- **Common Language Runtime (CLR, 1998)**
  - Any .NET Language
- **Android Runtime (ART, ~2013)**
  - Replacement for Dalvik
- **LLVM (2003)**
  - Becoming an ever-more popular alternative for all of the above

How not to design a language: PHP

- **PHP is full of bad/random design decisions**
  - Scoping issues, no modules, namespaces are “new”
  - Bad failure modes, inconsistent (or non-existent) error reporting and handling
  - Full of global and implicit state
  - No threading support
  - Inconsistent naming and behavior

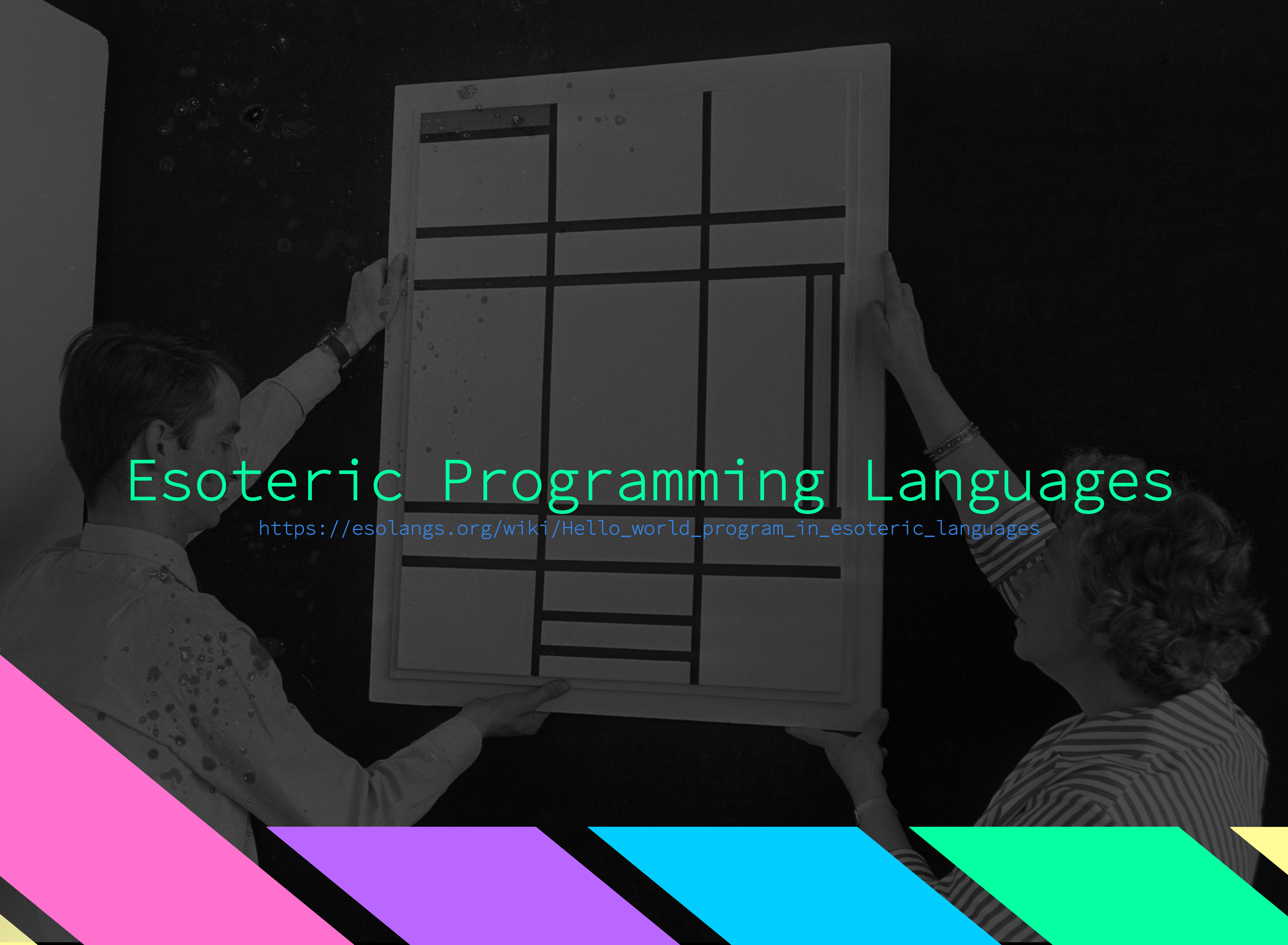
PHP

```
“foo” == True
“foo” == 0
“True” != 0
123 == “123foo”
“6” == “ 6”
“4.2” == “4.20”
NULL < -1
NULL == 0
“123” < “0124”
“4779” == “0x12AB”
```

- **It fails in every way a language should not**

For a most entertaining read: https://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design/

http://phpsadness.com/



Esoteric Programming Languages

https://esolangs.org/wiki/Hello_world_program_in_esoteric_languages

Brainfuck (1993) / P''(1964)

- **Minimalist, but Turing complete**
  - 240-byte compiler
  - "Turing tarpit"
- **1-dimensional array + pointer + ops:**
  - Move <> Byte +- Input , Output . Loop []
- **"Hello World!":**

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.
+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

Brainfuck (1993) / P''(1964)

- **Move <> Byte +- Input , Output . Loop [ ]**

```
++++++++
[>++++
  [>++>+++>+++>+<<<<-]>+>+>->>+[<]<-
]
>>.>---.+++++++..+++.>>.



<-.<.+++.------.--------.>>+.>++.
```

0	1	2	3	4	5	6
8	0	0	0	0	0	0
8	4	0	0	0	0	0

Brainfuck (1993) / P''(1964)

- **Move <> Byte +- Input , Output . Loop [ ]**

```
++++++++
[>++++
  [>++>+++>+++>+<<<<-]>+>+>->>+[<]<-
]
>>.>---.+++++++..+++.>>.



<-.<.+++.------.--------.>>+.>++.
```

0	1	2	3	4	5	6
8	0	0	0	0	0	0
8	4	0	0	0	0	0

Brainfuck (1993) / P''(1964)

- **Move <> Byte +- Input , Output . Loop []**

```
++++++++
[>++++
 [>++>+++>+++>+<<<<-]>+>+>->>+[<]<-
]
>>.>---.+++++++..+++.>>.
  H    e       ll   o  _



<-.<.+++.------.--------.>>+.>++.
```

0	1	2	3	4	5	6
8	0	0	0	0	0	0
8	4	0	0	0	0	0
0	0	72	104	88	32	8
0	0	72	101 108 111	88	32	8

Brainfuck (1993) / P''(1964)

- **Move <> Byte +- Input , Output . Loop [ ]**

```
++++++++
[>++++
 [>++>+++>+++>+<<<<-]>+>+>->>+[<]<-
]
>>.>---.+++++++..+++.>>.
  H    e       ll   o  _



<-.<.+++.------.--------.>>+.>++.
  W o   r      l        d   !   \n
```

0	1	2	3	4	5	6
8	0	0	0	0	0	0
8	4	0	0	0	0	0
0	0	72	104	88	32	8
0	0	72	101 108 111	88	32	8
0	0	72	111 114 108 100	87	33	10

Shakespeare, Chef

SPL

```
The Infamous Hello World Program.

Romeo, a young man with a remarkable patience.
Juliet, a likewise young woman of remarkable grace.
Ophelia, a remarkable woman much in dispute with Hamlet.
Hamlet, the flatterer of Andersen Insulting A/S.

        Act I: Hamlet's insults and flattery.
        Scene I: The insulting of Romeo.

[Enter Hamlet and Romeo]

Hamlet:
 You lying stupid fatherless big smelly half-witted
coward!
 You are as stupid as the difference between a handsome
rich   brave hero and thyself! Speak your mind!
(...)
```

SPL

```
(...)
Juliet:
 Am I better than you?

Hamlet:
 If so, let us proceed to scene III.
(...)
```

Chef

```
Hello World Cake with Chocolate sauce.

(...)

Ingredients.
33 g chocolate chips
100 g butter
54 ml double cream
2 pinches baking powder
114 g sugar
111 ml beaten eggs
119 g flour
32 g cocoa powder
0 g cake mixture

Cooking time: 25 minutes.

Pre-heat oven to 180 degrees Celsius.

Method.
Put chocolate chips into the mixing bowl.
Put butter into the mixing bowl.
Put sugar into the mixing bowl.
Put beaten eggs into the mixing bowl.
Put flour into the mixing bowl.
Put baking powder into the mixing bowl.
Put cocoa  powder into the mixing bowl.
Stir the mixing bowl for 1 minute.
Combine double cream into the mixing bowl.
Stir the mixing bowl for 4 minutes.
Liquefy the contents of the mixing bowl.
Pour contents of the mixing bowl into the baking
dish.
bake the cake mixture.
Wait until baked.
Serve with chocolate sauce.

chocolate sauce.

Ingredients.
111 g sugar
108 ml hot water
(...)
```

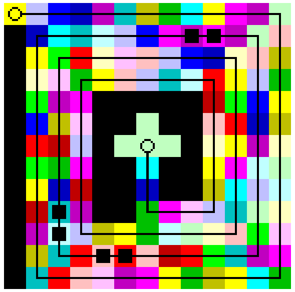
Malbolge

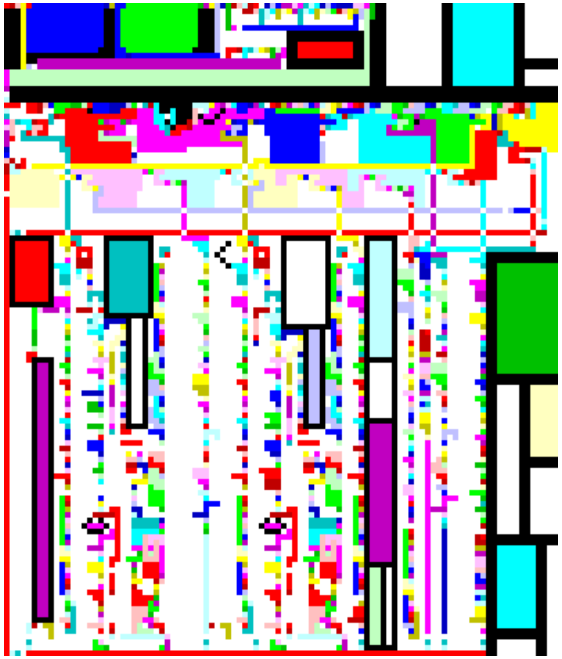
- **Designed to be “the most difficult programming language”**
- **Some “features” are:**
  - Self-modifying
  - Effect of instructions depends on their memory location
- **Hello World:**

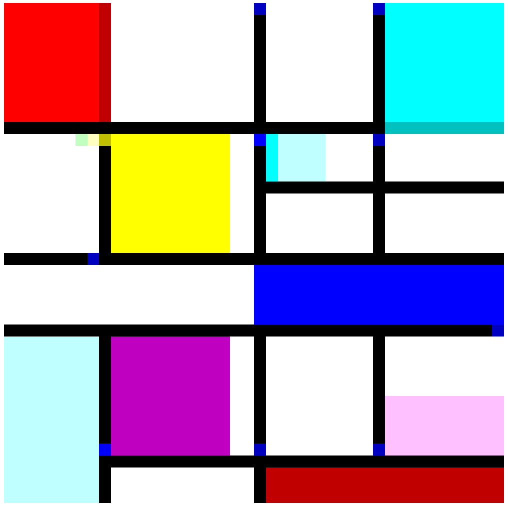
```
 (=<`#9]~6ZY32Vx/4Rs+0No-&Jk)"Fh}|Bcy?`=*z]Kw%oG4UUS0/@-
ejc(:'8dc
```

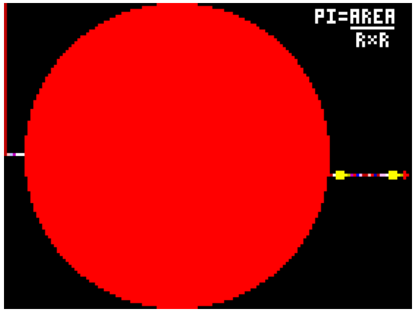
Piet

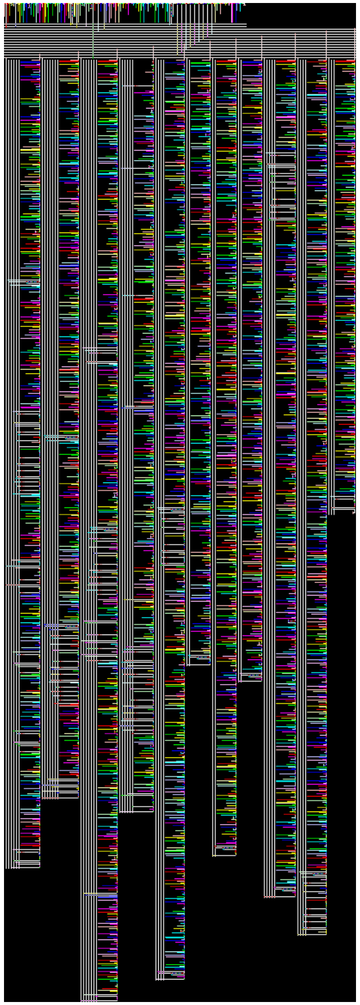
- **Actions determined by hue change steps**







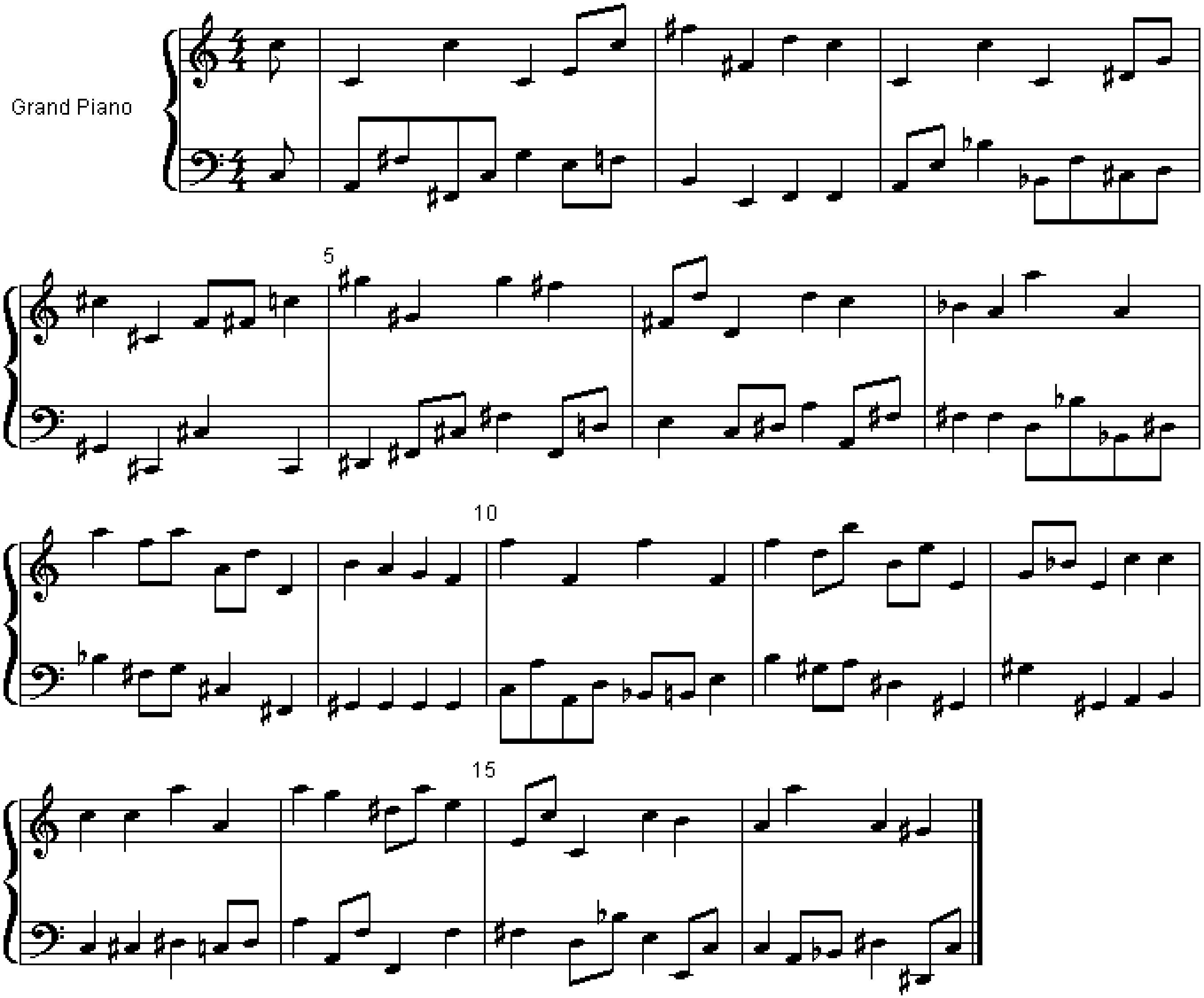




https://www.dangermouse.net/esoteric/piet/samples.html

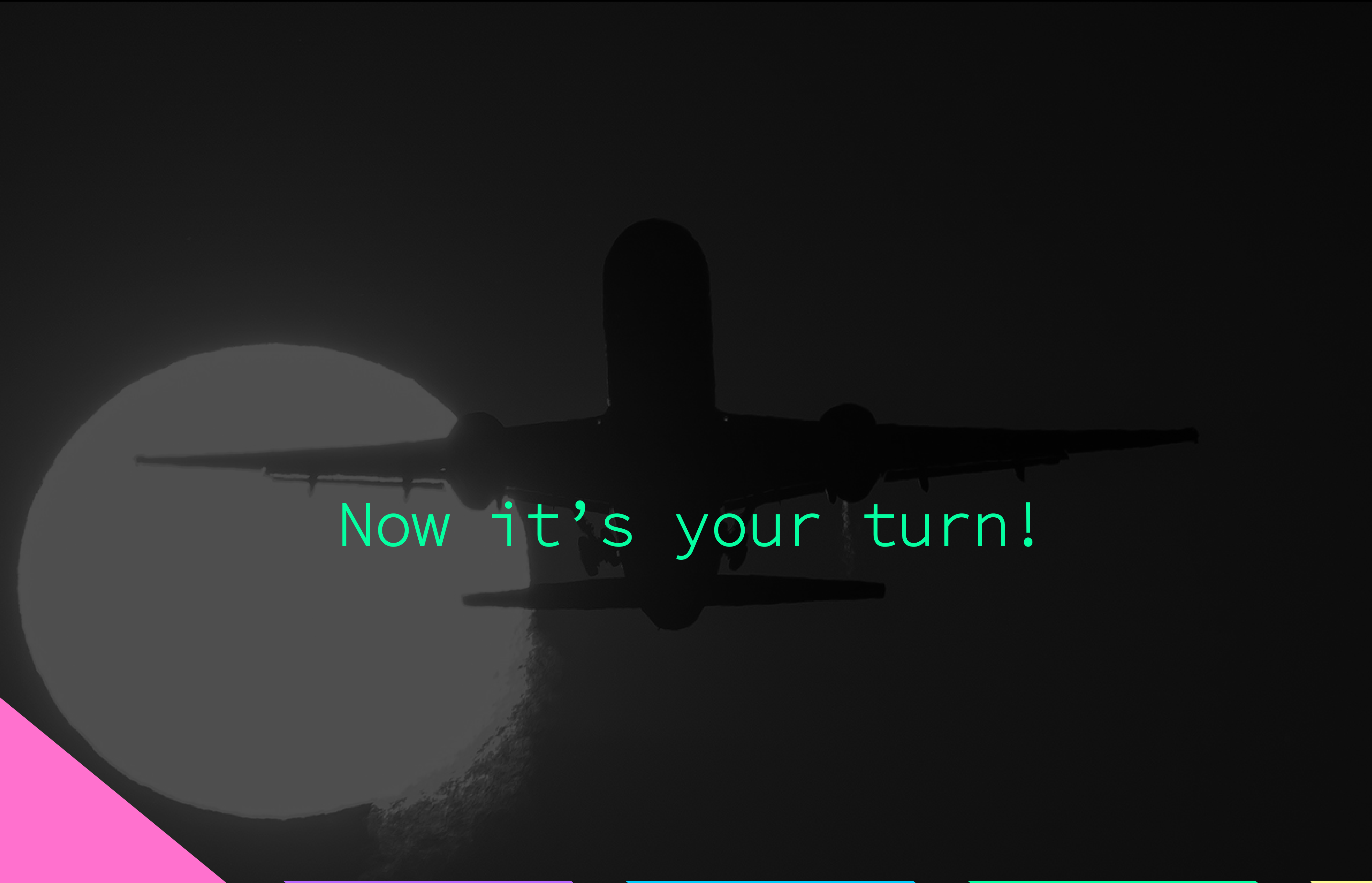
Prelude, Fugue

- **Intervals between notes make instructions**
- **Hello World ------------->**



Fun things

- **Code Golf:**
  - https://code.golf/
- **Rosetta Code:**
  - https://rosettacode.org/
- **Code obfuscation (for fun, not evil):**
  - https://codegolf.stackexchange.com/questions/22533/weirdest-obfuscated-hello-world



Now it’s your turn!

Pick a programming language!

- **Most cerebral and educational:**
  - Haskell, ML, Lisp / Scheme / Clojure, Erlang, Smalltalk/Pharo
- **Very useful, but very common:**
  - Bash, JavaScript, C, C++
- **Nice to know with potential for future growth:**
  - Rust, Go, Swift or Objective-C, TypeScript, Kotlin, Scala, Lua
- **Do you like a challenge?**
  - Assembly, esoteric languages, create your own language
- **Do you want to work at a bank and earn lots of $$$?**
  - COBOL, APL

How to pick / Tasks

- **Pick in OLAT:**
  - https://lms.uzh.ch/auth/RepositoryEntry/17190518785/CourseNode/105238826923143
  - The form will close tomorrow at 23:59!!!
- **Tasks (tentative, subject to change):**
  - 1st: Hello World, interactive + simple algorithm
  - 2nd: Various small scenarios
  - 3rd: Writing a simple game (text based, GUI optional)
- **Learning tips and tasks follow on Tuesday**

Course Schedule

	Online Session	Deadline (end of day)
Wed, 23.02.2022	Lecture	
Wed, 02.03.2022	Lecture	Pick your language
Wed, 09.03.2022		
Wed, 16.03.2022		Programming tasks #1
Wed, 23.03.2022		
Wed, 30.03.2022		
Wed, 13.04.2022	“Touch base”	Programming tasks #2
Wed, 20.04.2022		
Wed, 27.04.2022		Programming tasks #3
Wed, 04.05.2022	Discussion and Wrap-up	Seminar paper
Wed, 11.05.2022		Paper reviews
Wed, 18.05.2022	Student presentations	
Wed, 25.05.2022	Student presentations	
Wed, 01.06.2022	Student presentations & Wrap-up	Paper revision

Getting Help

- **Post in the OLAT Forum**
- **Try asking on IRC (freenode):**
  - Seriously, it's a fantastic resource! Here's some instructions:
    - https://libera.chat/guides/clients
    - https://libera.chat/guides/connect
  - Follow these rules not to disturb the nerds:
    - Type `/topic` to make sure you're in the right channel, read their rules
    - Make sure you tried finding the answer yourself before asking
    - Don't ask to ask, just ask your question
    - Wait patiently after asking. Just stay in the channel.
    - Try to be brief and precise in stating your problem.
- **Message me on Teams**