Introduction to Functional Programming

Chris Purdy (PhD student, RHUL Computer Lab)

Follow along with the code samples at:
https://replit.com/@ChrisPurdy1/IntroToFP

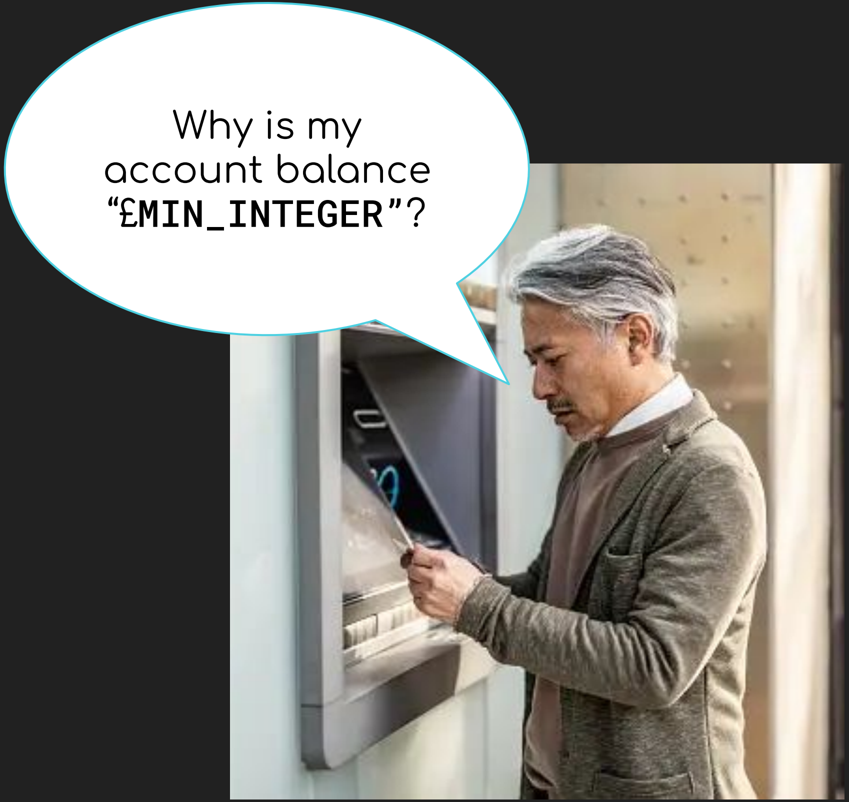
What is FP?

A way of programming that emphasises correctness of, and ability to reason about program behaviour.

FP is mostly declarative - it describes what a program should do.

The opposite is imperative - describing how a program should do it.

Functional programming is often synonymous with heavy use of functions.



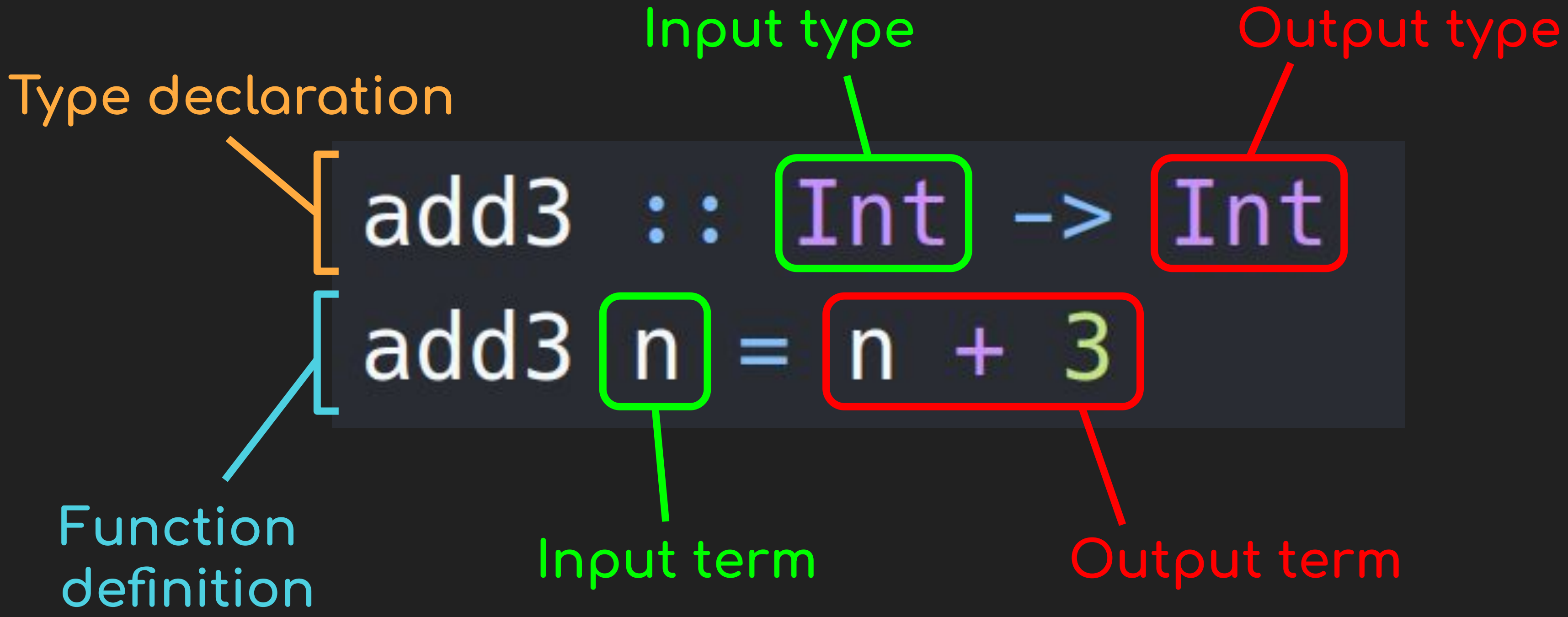
“FP languages”

You can do FP in most programming languages, but there are some languages that enforce it more.

There are many “functional programming languages”, and some are being heavily used in industry. For example, Haskell has been used at Meta to program software that protects users on their social media platforms from malware.

Today I’m going to introduce Haskell, a statically typed, purely functional language based on the lambda calculus.

Anatomy of a function



Anatomy of a constant

Type

Type declaration

```
hello :: String
hello = "Hello World!"
```

Anonymous (lambda) functions

```
\x -> 5 * x
```

"A function that takes x and returns 5 * x"

The function is a term!

Functions, revisited I

```haskell
secondLetter :: String -> Char
secondLetter = \s -> s !! 1
```

List/string indexing function (`s[1]` in Python)

...is equivalent to...

```haskell
secondLetter :: String -> Char
secondLetter s = s !! 1
```

Functions, revisited II

```haskell
addSomeMore :: Int -> (Int -> Int)
addSomeMore = \a -> (\b -> a + b + 5)
```

||

...is equivalent to...

```haskell
addSomeMore :: Int -> (Int -> Int)
addSomeMore a b = a + b + 5
```

Partial application

```haskell
addSomeMore :: Int -> (Int -> Int)
addSomeMore a b = a + b + 5
```

In GHCI

```
ghci> :t addSomeMore 4
addSomeMore 4 :: Int -> Int
ghci> x = addSomeMore 4
ghci> x 5
14
```

Guess the type I

```haskell
1 :: Int
```

```haskell
False :: Bool
```

```haskell
'c' :: Char
```

```haskell
"hi" :: String
```

```haskell
(5, True) :: (Int, Bool)
```

Guess the type II

```haskell
prefixWithTitle :: String -> String
prefixWithTitle s = "Dr. " ++ s
```

```haskell
andGate :: Bool ->(Bool -> Bool)
andGate a b = if a then b else False
```

```haskell
positive :: Int -> Bool
positive = \n -> n > 0
```

```haskell
myFavNumbers :: [Int]
myFavNumbers = [12, 42, 7, 121]
```

Higher-order functions

```haskell
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)

add6 :: Int -> Int
add6 = twice add3


add6 :: Int -> Int
add6 = twice (\n -> n + 3)
```

f is a function!

Executing Haskell programs

```
add6 4
= twice (\n -> n + 3) 4
= (\n -> n + 3) ((\n -> n + 3) 4)
= (\n -> n + 3) (4 + 3)
= (\n -> n + 3) 7
= 7 + 3
= 10
```

n is “substituted by” 4

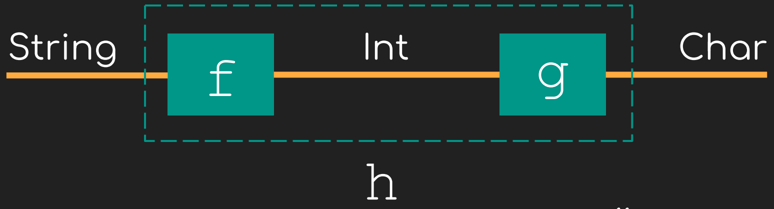
Function composition I

Think of a function as a box with a typed input and output wire.

```haskell
f :: String -> Int
```


```haskell
g :: Int -> Char
```

You can attach matching wires to get a new "composite" box:




```haskell
h :: String -> Char
h = g . f
```

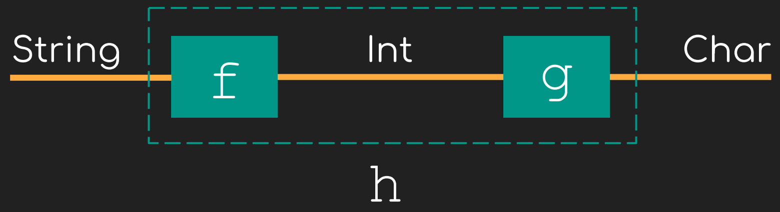
"h is g after f"

Function composition II

Composition is defined as `(g . f) x = g (f x)`

```
((add 5) . (twice (add 6))) 8

= add 5 (twice (add 6) 8)
  ...
= add 5 20
  ...
= 25
```



Loops... I

Haskell has no while or for loops... so how do we iterate things?

The answer: recursion!

"Calculate the sum of numbers from 0 to n?"

```haskell
sumBelow :: Int -> Int
sumBelow n = if n == 0
             then 0
             else n + sumBelow (n - 1)
```

Loops... II

```python
def sumBelow(n):
  res = 0
  for i in range(n):
    res += i
  return res
```

```python
def sumBelow(n):
  if n == 0:
    return 0
  else:
    return n + sumBelow(n - 1)
```

Python (╯°□°)╯

Haskell <3

```haskell
sumBelow :: Int -> Int
sumBelow n = if n == 0 then 0 else n + sumBelow (n - 1)
```

Practical session

Live coding time!

You can try out Haskell with the code snippets in this talk by forking my repl: https://replit.com/@ChrisPurdy1/IntroToFP

If you're feeling confident, you can try the exercises on the worksheet (or ask me for extra exercises).

Fibonacci sequence refresher

The Fibonacci sequence is the sequence generated by the equations:

$$x_0 = 1$$
$$x_1 = 1$$
$$x_n = x_{n-1} + x_{n-2}$$

n	0	1	2	3	4	5	6	7
x_n	1	1	2	3	5	8	13	21

```haskell
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Haskell <3

Python (╯°□°)╯

```python
def fib(n):
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return fib(n - 1) + fib(n - 2)
```

```haskell
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Haskell

Python

```python
def fib(n):
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return fib(n - 1) + fib(n - 2)
```

Which is closer to the equations?

$$x_0 = 1$$
$$x_1 = 1$$
$$x_n = x_{n-1} + x_{n-2}$$

Further resources

Learn You a Haskell for Great Good!
(link)

Why Functional Programming Matters - John Hughes
(link)

Haskell language wiki
(link)