Introduction to Functional Programming, or:

How I Learned to Stop Worrying and Love Referential Transparency

George Wilson

Ephox

george.wilson@ephox.com

March 22, 2017

Me

- Graduated from Griffith in 2014
- Working at Ephox ever since
- Assistant organiser of Brisbane FP Group

This talk

- Beginner talk - you should be able to understand it!
- Ask lots of questions
- Let's hang out after the talk

This talk

- You won't learn functional programming tonight
- You should learn *what it is* and hopefully *why you should care*
- The aim is to motivate you and provide the tools to teach yourself

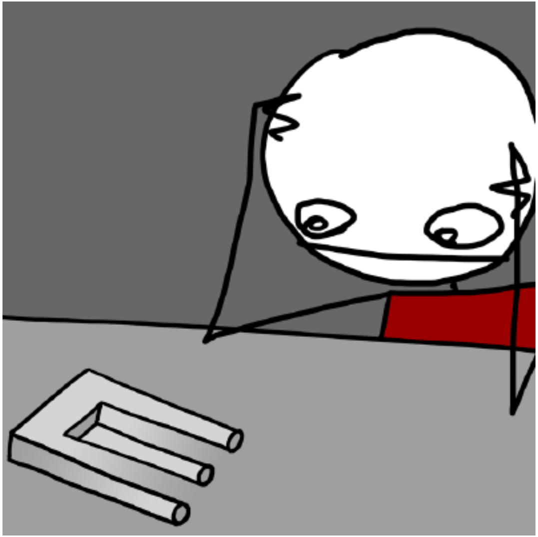
Why should we care?

Programming is hard

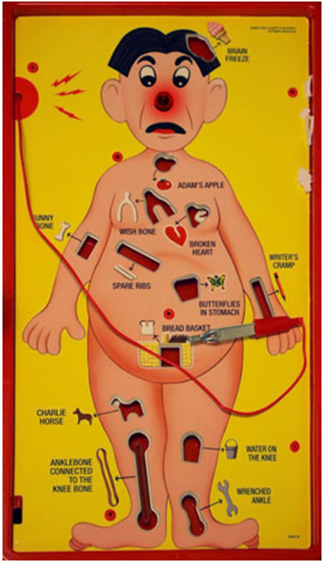
Programming is hard

Why?

Programs are difficult to understand



Programs are difficult to modify



It is difficult to build new programs from old ones

Functional programming helps with these things

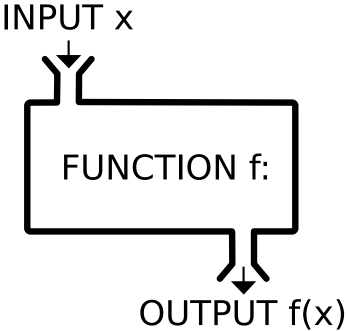
What is functional programming?

What is functional programming?

Functional programming is programming with functions.

What is a function?

A function relates its input to a result *and does nothing else*.


INPUT x
FUNCTION f:
OUTPUT f(x)

$$f(x) = x + 5$$

By definition, a function can not:

- Modify or depend on external state
- Perform I/O (such as reading a file or printing to the screen)
- Throw an exception

We call those things *side effects*.

Some people call functions *pure functions*
to differentiate them from “functions” with side effects

Useful properties of functions:

- Passing the same input *always* gives the same output

Useful properties of functions:

- Passing the same input *always* gives the same output
- You do not need to understand the environment in which the function is run to understand its result

Useful properties of functions:

- Passing the same input *always* gives the same output
- You do not need to understand the environment in which the function is run to understand its result
- Functions can be composed to build bigger functions

Examples of functions

```python
def add5(x):
  return x + 5
```

Examples of functions

```python
def add5(x):
  return x + 5
```

```python
def greet(name):
  return f"Hello, {name}"
```

Example of not a function

```
count = 0

def countedGreet(name):
  global count
  count = count + 1
  s = f"Hello, {name}. I've greeted {count} people!"
  return s
```

Example of not a function

```python
count = 0

def countedGreet(name):
  global count
  count = count + 1
  s = f"Hello, {name}. I've greeted {count} people!"
  return s

g = countedGreet("George")
h = countedGreet("George")

# g = "Hello, George. I've greeted 1 people!"
# h = "Hello, George. I've greeted 2 people!"
```

Turning it into a function

```python
def countedGreet2(name, count):

    newCount = count + 1
    s = f"Hello, {name}. I've greeted {count} people!"
    return (s, newCount)
```

Turning it into a function

```python
def countedGreet2(name, count):

  newCount = count + 1
  s = f"Hello, {name}. I've greeted {count} people!"
  return  (s, newCount)


(g,n) = countedGreet2("George", 0)
(h,m) = countedGreet2("George", 0)

# g = "Hello, George. I've greeted 1 people!"
# h = "Hello, George. I've greeted 1 people!"
```

Turning it into a function

```python
def countedGreet2(name, count):

  newCount = count + 1
  s = f"Hello, {name}. I've greeted {count} people!"
  return  (s, newCount)


(g,n) = countedGreet2("George", 0)
(h,m) = countedGreet2("George", n)

# g = "Hello, George. I've greeted 1 people!"
# h = "Hello, George. I've greeted 2 people!"
```

Removing side effects makes our types reveal more information

```
countedGreet  :: String -> String

countedGreet2 :: (String, Int) -> (String, Int)
```

You've seen how to remove mutable state by passing more parameters
Can we remove exceptions somehow? What about `null`?

Say we're writing a function to parse a String into an Int

```
parse("34") ==> 34
```

Say we're writing a function to parse a String into an Int

```
parse("34") ==> 34

parse("nope") ==> ?
```

Say we're writing a function to parse a String into an Int

```
parse("34") ==> 34
```

```
parse("nope") ==> ?
```

This function takes a `String` and *either* returns an `Int` or throws an exception

So the type is
`String -> Int`

```
let x: Int = parse("nope")

x + 5
```

Instead, let's use a type that is either an error or an `Int`

```
String -> Result<Int>
```

Instead, let's use a type that is either an error or an `Int`

```
String -> Result<Int>
```

```
enum Result<A> {
  case Error(String)
  case Success(A)
}
```

```
let r: Result<Int> = parse("34")

r + 5
```

```
let r: Result<Int> = parse("34")

r + 5
```

That doesn't compile! We can't forget to check for the exception!

```
let r: Result<Int> = parse("34")

switch r {
  case Error(message):
    // handle the message somehow
  case Success(n):
    n + 5
}
```

We can do the same thing to get rid of `null`

We can do the same thing to get rid of `null`

```
enum Optional<A> {
  case Nothing
  case Something(A)
}
```

Sir Tony Hoare called `null` his Billion Dollar Mistake

Let’s talk about one of the coolest benefits of FP

```
def p():
  proc(expression, expression)
```

```
def p():
  x = expression
  proc(x, x)
```

```
def print2(s, t):
  print(s)
  print(t)

def strpopthen():
  s = ['a','b','c','d','e','f']

  print2(s[5],s[5])

# 'f'
# 'f'
```

```
def print2(s, t):
  print(s)
  print(t)

def strpopthen():
  s = ['a','b','c','d','e','f']
  x = s[5]
  print2(x,x)

# 'f'
# 'f'
```

```
def print2(s, t):
  print(s)
  print(t)

def listpopthen():
  s = ['a','b','c','d','e','f']

  print2(s.pop(),s.pop())

# 'f'
# 'e'
```

```python
def print2(s, t):
  print(s)
  print(t)

def listpopthen():
  s = ['a','b','c','d','e','f']
  x = s.pop()
  print2(x,x)

# 'f'
# 'f'
```

Referential Transparency

An expression e in a program is *referentially transparent* if and only if
replacing all occurrences of e with its value
does not change the observable behaviour of the program

Referential Transparency

Referential transparency enables *equational reasoning*

```
x + y === y + x

reverse(reverse(list)) === list
```

Referential Transparency

Referential transparency enables *equational reasoning*

```
x + y === y + x

reverse(reverse(list)) === list
```

Equational reasoning is the process of substituting equals for equals and knowing that you are not altering the result of the program.

We can fearlessly modify our programs!

But how do we build our programs in the first place?

But how do we build our programs in the first place?

We define data-types (like `Result<A>`)
and define functions on them

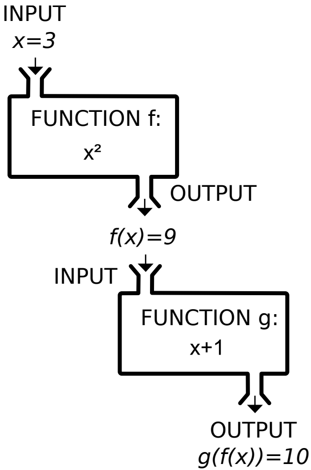
and then just combine the functions

Higher-order functions let us build new functions from others

A higher-order function:

- takes a function as input
- produces a function as output
- or both!

An example is function *composition*



```
def compose(f,g):
  return lambda x: f(g(x))
```

```python
def compose(f,g):
  return lambda x: f(g(x))


def excited(s):
  return s + "!"

def loud(s):
  return s.upper()
```

```python
def compose(f,g):
  return lambda x: f(g(x))


def excited(s):
  return s + "!"

def loud(s):
  return s.upper()


talkingAboutFunctionalProgramming = compose(excited, loud)
```

Another higher-order function is `map`

`map` runs a function on every element of a list

Another higher-order function is `map`
`map` runs a function on every element of a list

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

def embiggen(x):
  return x * 10

bigNumbers = map(embiggen, numbers)
# [10,20,30,40,50]
```

Higher-order functions let us build larger programs from smaller ones
We call this *modularity*

Summary

Pure functions give benefits

- It's easier to understand what a function does
- It's easier to refactor functions (and not break the program)
- Pure functions can always be run in parallel
- The type system helps you more

Referential transparency and higher-order functions give rise to modularity and reuse.

Necessary evil?

“we only use side effects when they are necessary”

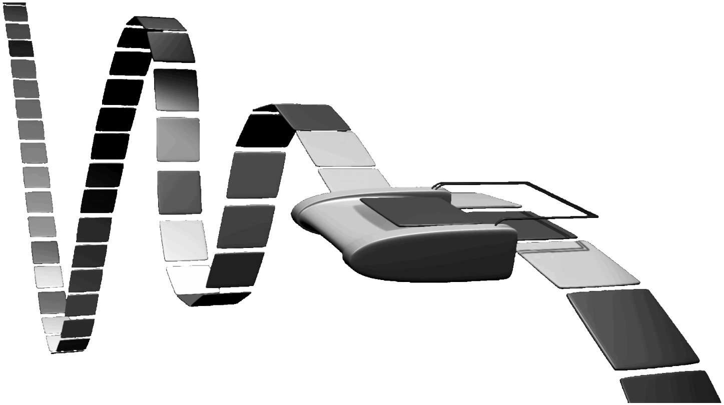
“the algorithm we’re using is inherently stateful”

What is an algorithm?

In the 1930's, **Alan Turing** invented

Turing Machines

to answer this question

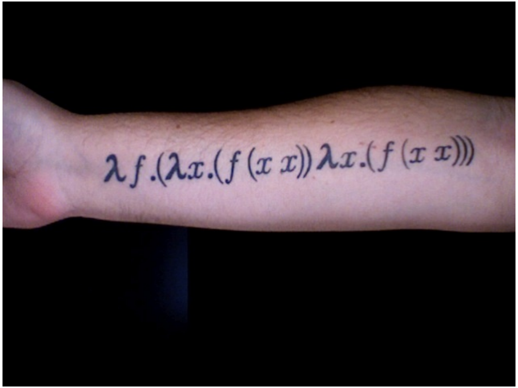


Turing machines compute by reading and writing to a tape

Computation is done by this *modification* of the tape

At the same time, **Alonzo Church** invented

λ calculus



Lambda calculus is pure functions *and nothing else*

Computation is done by function application

Church and Turing discovered that λ calculus and Turing Machines

are equivalent

Anything computable with side effects
can be computed with only pure functions

How can you learn functional programming?

I recommend learning Haskell

Haskell lectures

`https://www.seas.upenn.edu/%7Ecis194/spring13`

`https://github.com/bfpg/cis194-yorgey-lectures`

Books

`http://www.haskellbook.com` `http://http://learnyouahaskell.com/`

Nothing in this talk is specific to Haskell!

Other languages with excellent support for pure functional programming:

- Purescript
- Idris

and other languages with support for functional programming:

- OCaml
- Scala
- Rust
- Swift
- C#
- Java
- Many others!

Next time side effects are ruining your day, remember you don't need them


ICE CUBE

"If you are scared [. . .] go to Church!"
- Ice Cube

Thanks for listening!