functional programming

axel sarlin

plan

1. background and basics
2. functions, functions, functions
3. side-effects
4. areas of application

1. background

```
501 FORMAT(3I5)
601 FORMAT(4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2,
   $13H SQUARE UNITS)
602 FORMAT(10HNORMAL END)
603 FORMAT(23HINPUT ERROR, ZERO VALUE)
    INTEGER A,B,C
 10 READ(5,501) A,B,C
    IF(A.EQ.0 .AND. B.EQ.0 .AND. C.EQ.0) GO TO 50
    IF(A.EQ.0 .OR.  B.EQ.0 .OR.  C.EQ.0) GO TO 90
    S = (A + B + C) / 2.0
    AREA = SQRT( S * (S - A) * (S - B) * (S - C) )
    WRITE(6,601) A,B,C,AREA
    GO TO 10
 50 WRITE(6,602)
    STOP
 90 WRITE(6,603)
    STOP
    END
```

fortran (1957)

in the beginning, there were punch cards

later, magnetic tape

higher-level programming languages evolved

paradigms

complexity brings need for organisational concepts

a number of different ideas arise:

- imperative
- object-oriented
- functional

```java
/**
 * This is an example of a Javadoc comment;
 * from this text. Javadoc comments must imm
 */
public class FibCalculator extends Fibonacci
    private static Map<Integer, Integer> mem

    /*
     * The main method written as follows is
     */
    public static void main(String[] args) {
        memoized.put(1, 1);
        memoized.put(2, 1);
        System.out.println(fibonacci(12)); /
    }

    /**
```

java (1995): imperative and object oriented

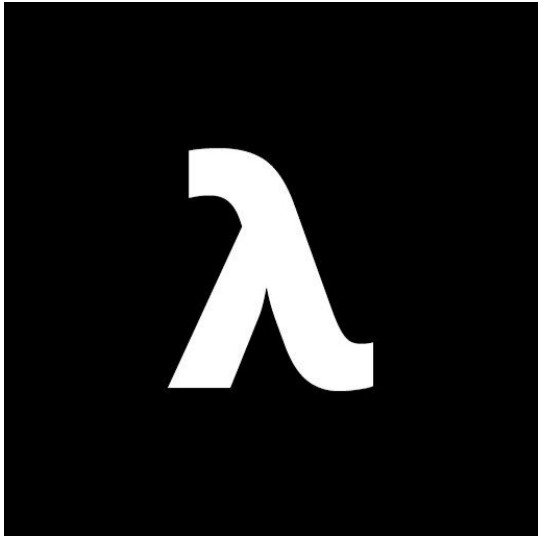
functional programming

roots: lambda calculus - what is computation?

focus on functions:

- composable - easy to combine parts
- transparent - clear flows of data
- reliable - same output every time*

easy to make parallel and concurrent!



a neat lambda

In Java every program consists of a list of instructions that are executed in a particular order when the program is run.

A Haskell program is a collection of equations declaring what the result of running the program should be.

2. functional programming

immutable values

declarations, not assignments

```
r = 5
r = 2
```

not allowed here!

pure functions

always the same output for an input

```
x = 3
y = x * 2
```

```
y = x * 2
x = 3
```

order does not matter!

referential transparency

we can always substitute a variable for its value

functions

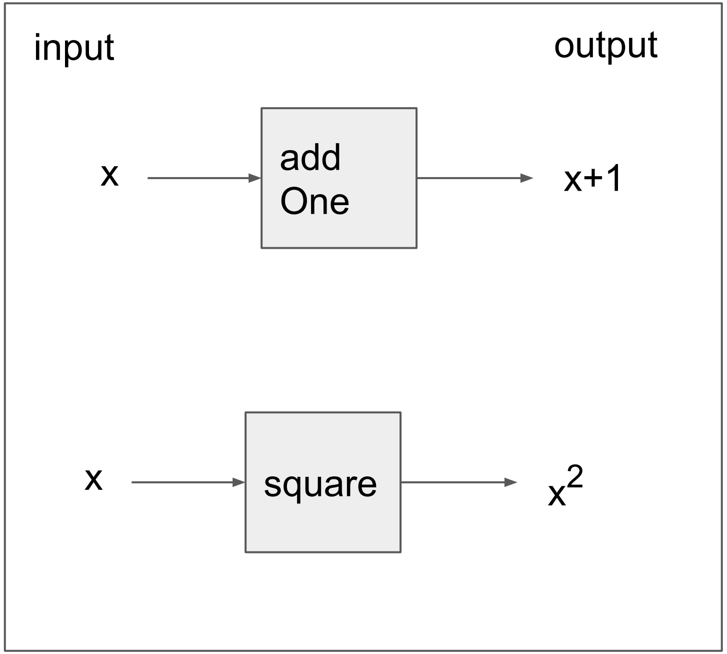
(example code in Haskell)

```
addOne :: Int -> Int
addOne x = x + 1



square :: Int -> Int
square x = x^2
```

note: no parentheses needed



magic black boxes

common features:
pattern matching

define a function piece by piece

the compiler puts it together!

```
f :: Int -> Int
f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1
```

here _ is short for “anything else”

recursive definitions

list syntax:
`[a,b] = a:[b] = a:b:[]`

functions defined in terms of themselves

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

```
sum :: [Integer] -> Integer
sum []     = 0
sum (x:xs) = x + sum xs
```

higher-order functions

we are free to apply functions to other functions

plenty of standard tools:

- **`map`**
- **`foldr`**
- **`scanr`**

```
square x = x^2

> square 4
16

> map square [1,2,3]
[1,4,9]
```

example using map

folds

“instead of for loops”

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc []     = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

```
GHCi> foldr (-) 6 [1, 2, 3] == 1 - (2 - (3 - 6))
True
```

```
   f
  / \
 a   f
    / \
   b   f
      / \
     c   acc
```

conceptual illustration

iterated application of a function to output of previous step with new inputs given from a list

scans

similar to folds, but returns a list

example: summing and keeping the partial sums:

```
> scanl1 (+) [1,2,3]
[1,3,6]
```

running totals

example

another advanced function

```
inits :: [a] -> [[a]]
inits = map reverse . scanl (flip (:)) []

 > inits [1,2,3]
[[],[1],[1,2],[1,2,3]]
```

returning the initial segments of a list

example

another less boring example:

```
capitalise :: String -> String
capitalise x = let
                    capWord [] = []
                    capWord (x:xs) = toUpper x : xs
                in
                    unwords $ map capWord (words x)

> capitalise "hello analytics team"
"Hello Analytics Team"
```

features

recap:

- type declarations
- pattern matching
- recursive definitions
- higher order functions

```
capitalise :: String -> String
capitalise x = let
                   capWord [] = []
                   capWord (x:xs) = toUpper x : xs
               in
                   unwords $ map capWord (words x)


> capitalise "hello analytics team"
"Hello Analytics Team"
```

advantages over imperative

easier to make **parallel** and **concurrent**!

reusable code, **less boilerplate** - more like Lego

less risk for **low-level errors**

compact, high-level **code** - easy to read and write

powerful type system - “if your program compiles, it works”

3. handling the real world

this is all very good - but can it do anything useful?

side-effects:

- input/output
- state
- random numbers

are these compatible with “functional purity”?

yes!

but we need a “wrapper” abstraction...

example: Maybe

```haskell
data Maybe a = Just a | Nothing

printMaybe :: Maybe String -> IO ()
printMaybe (Just x) = print x
printMaybe Nothing  = print "error"

> printMaybe (Just "hello")
"hello"

> printMaybe Nothing
"error"
```

motivation

```
phone2name :: Int -> String

name2income :: String -> Int
```

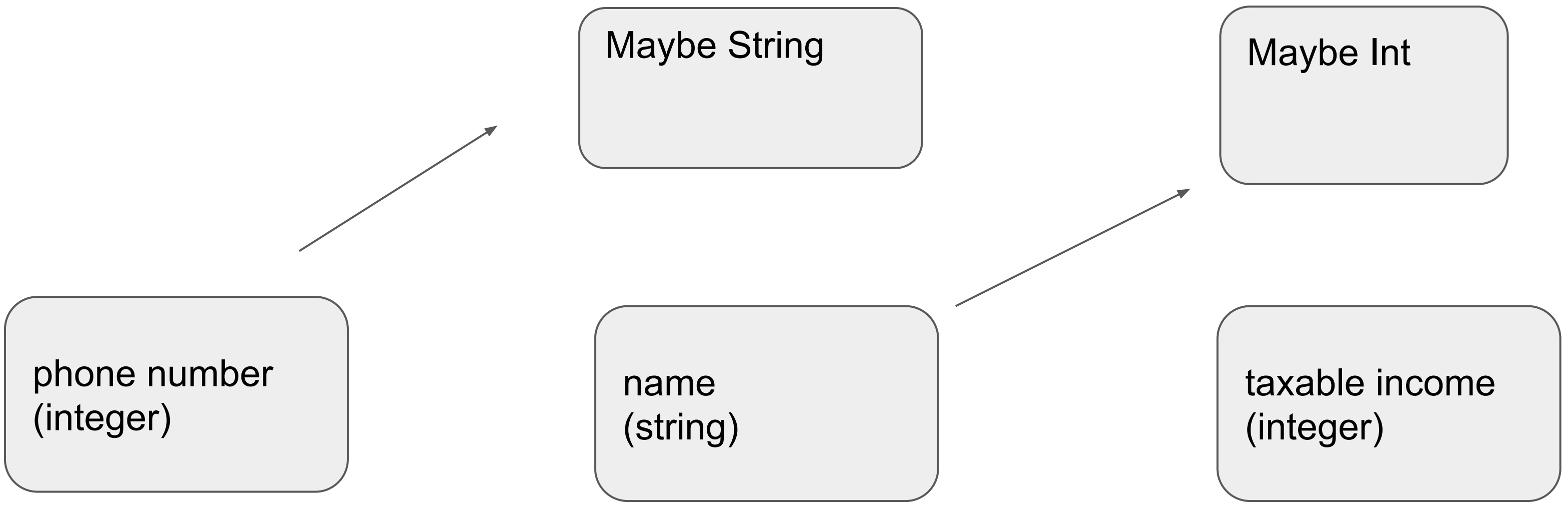
our dream would be to have functions like this:

motivation

```
phone2name :: Int -> Maybe String

name2income :: String -> Maybe Int
```

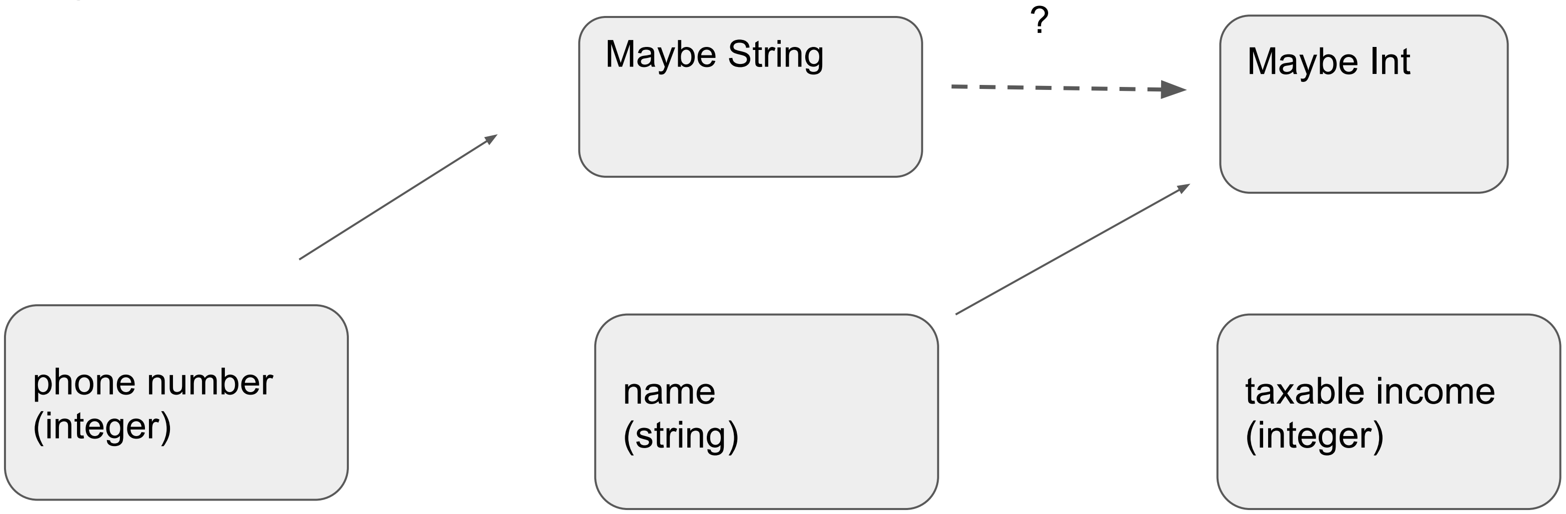
but reality is more like this:



motivation

```
phone2income :: Int -> Maybe Int
```

so how do we compose them?



composing - the hard way

for Maybe, we can do this with logical conditions...

```
phone2income :: Int -> Maybe Int
phone2income x =
    case phone2name x of
      Nothing -> Nothing
      Just name -> name2income name
```

composing - the cool way

… or with abstract wrapper formalism:

```
phone2income x = phone2name x >>= name2income
```

or equivalently

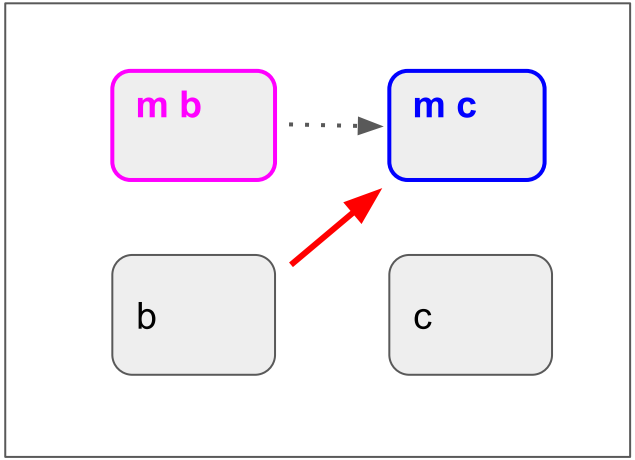
```
phone2income x = do
          y <- phone2name x
          name2income y
```

saved by the bind

here `(>>=)` is a function (called “bind”) that takes

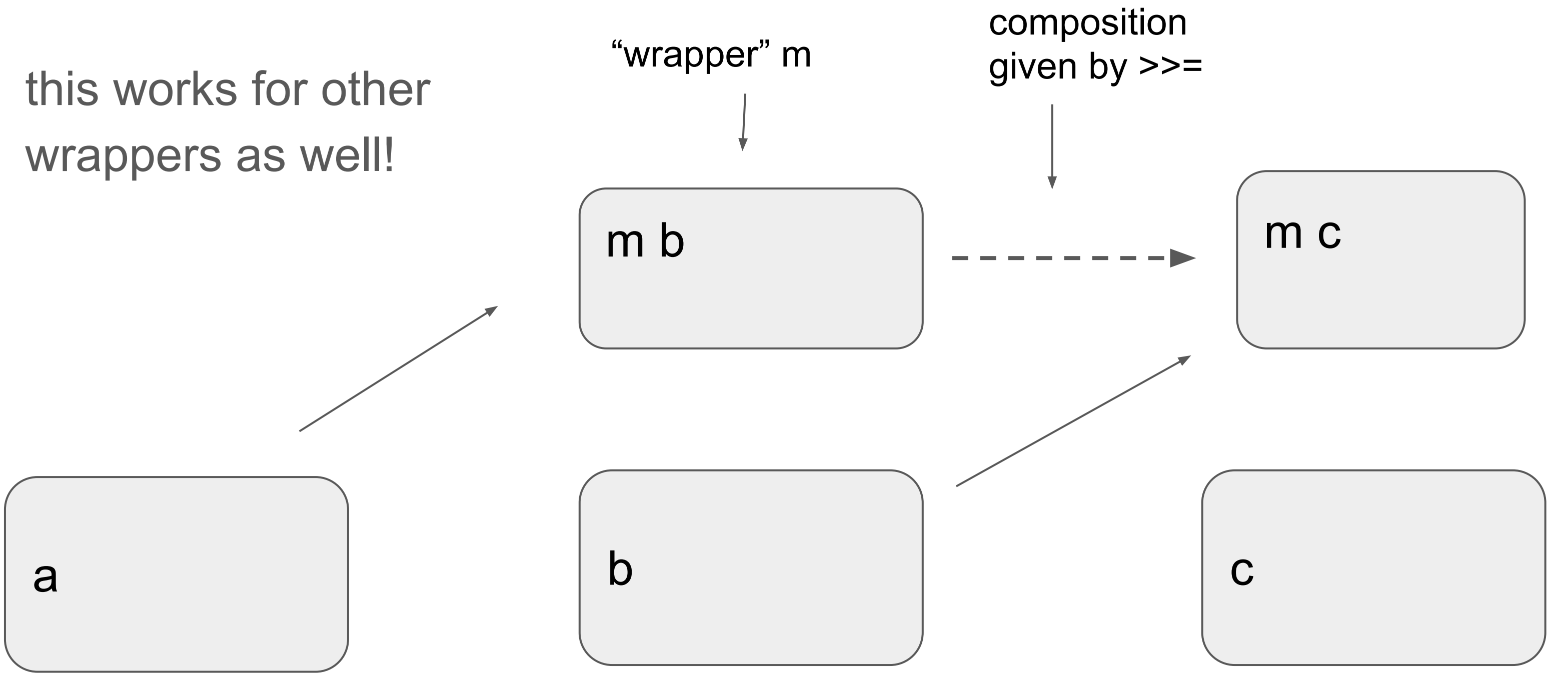
1. some “thing” of type `Maybe b`
2. some function of type `b -> Maybe c`

...and gives an output of `Maybe c`



catharsis

this works for other wrappers as well!



examples

other "wrappers" include

- input/output
- lists
- error handling
- random
- state (e.g. count of iterations)
- quantum computation
- SQL

```
main :: IO ()
main = getLine >>= putStrLn
```

...you just need a `(>>=)` to make your own class into one!

example with IO and Maybe

our wrappers can also interact in neat ways:

```
interactiveSumming = do
    putStrLn "Choose two numbers:"
    sx <- getLine
    sy <- getLine
    let mx = readMaybe sx :: Maybe Double
        my = readMaybe sy
    case (+) <$> mx <*> my of
        Just z -> putStrLn ("The sum of your numbers is " ++ show z)
        Nothing -> do
            putStrLn "Invalid number. Retrying..."
            interactiveSumming
```

moral

1. we can have the cookie and eat it:

treating `IO String` and `Maybe String` differently from a `String`, we can have the advantages of “pure” functions whilst also handling side-effects!

2. we get a neat unified syntax for dealing with things like `IO` and `Maybe`

note

these wrappers are called **monads**

monads are tools that generalise **containers** and **computation**

there are other ways to handle side-effects, but monads can be used in

- FP languages like Haskell, Clojure, OCaml,
- others like Scala, Perl, Ruby, Python, Javascript, C#, PHP

4. haskell in industry

Alcatel

[..] used Haskell to prototype narrowband software radio systems, running in real-time.

AT&T

Haskell is being used in the Network Security division to automate processing of internet abuse complaints. Haskell has allowed us to easily meet very tight deadlines with reliable results.

Deutsche Bank

The Directional Credit Trading group used Haskell as the primary implementation language for its software infrastructure.

Facebook

[uses] Haskell internally for tools, [..] a tool for programmatically manipulating a PHP code base via Haskell.

Microsoft

[..] uses Haskell for its production serialization system [which] is broadly used at Microsoft in high scale services.

overview of FP languages

old:	lisp, scheme, ML, Erlang, miranda..
modern:	haskell, clojure, ocaml, idris, agda..
can be used functionally:	F#, scala, R, JS, kotlin, python, perl, php...
catching up:	java, C++, C#..

some notable FP languages

Clojure

“industry grade LISP” that runs on the JVM
used by Spotify, Netflix, Walmart…

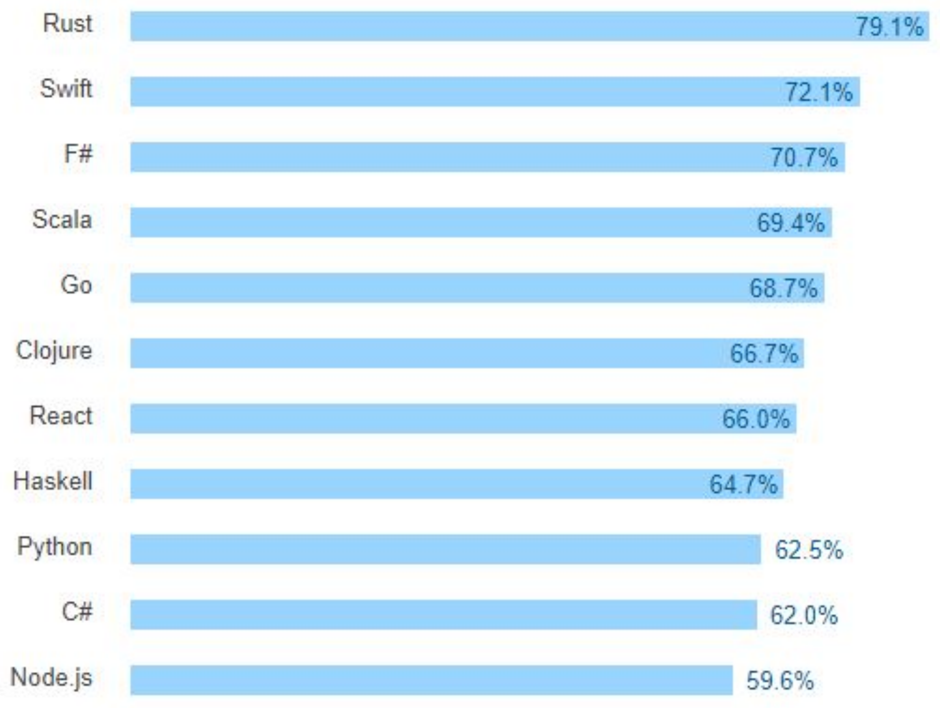
Erlang

industry language developed and used by Ericsson
also used by Amazon, Yahoo! and formerly Facebook

Scala

imperative/OOP/FP hybrid running on the JVM - “java with folds and monads”
rising in popularity, especially with Spark
used by Twitter, Sony, Siemens, Linkedin...

stack overflow developer survey
most loved languages



% of developers who are developing with the language or tech and have expressed interest in continuing to develop with it

end