

# A Journey in Functional Programming

## An introduction to Haskell

Davide Spataro<sup>1</sup>

<sup>1</sup>Department of mathematics And Computer Science  
Univeristy of Calabria

April 18, 2015

# Table of contents

- 1 Introduction - Syntax and Types
  - Functional Programming
  - Tools and Installation
  - Hello world(s)
- 2 Basics - Syntax
  - Arithmetic And Boolean algebra
  - Guards, where, let
  - if and case construct
  - Ranges
  - List Comprehension
- 3 Basics - List Functions
  - List Functions - length, ++

## Section 1

### Introduction - Syntax and Types

# Functional Programming

## Definition and Intuitive idea

- Computation is just **function evaluation**  $\neq$  **program state manipulation**.
- Based on  $\lambda$ -calculus that is an alternative (to set theory) and convenient formalization of logic and mathematics for expressing **computation**
- Logic deduction  $\Leftrightarrow$   $\lambda$ -calculus thanks to the Curry-Howard correspondence.
- A program is a proof!



Figure:  
Alonzo-Church,  
father of  $\lambda$ -calculus

# Imperative vs Functional

- Imperative

- Focus on low-level **how!**
- A program is an ordered sequence of instructions
- Modifies/track the program's state

- Functional

- Focus on High level **what!**
- Specify high-level transformation/constraint on the desired result description.

Imperative, suffer from the so called **indexitis**

```
unsigned int sum=0;  
for(int i=1;i<100;i++)  
    sum+=i;
```

Functional

```
sum [1..99]
```

## What does this code do?

```
void function (int *a, int n) {  
    int i, j, p, t;  
    if (n < 2)  
        return;  
    p = a[n / 2];  
    for (i = 0, j = n - 1;; i++, j--) {  
        while (a[i] < p)  
            i++;  
        while (p < a[j])  
            j--;  
        if (i >= j)  
            break;  
        t = a[i];  
        a[i] = a[j];  
        a[j] = t;  
    }  
    function(a, i);  
    function(a + i, n - i);  
}
```

## ...and this?

---

```
function ::(Ord a) =>  [a] -> [a]
function [] = []
function (x:xs) = (function l) ++ [x] ++ (function g)
  where
    l = filter (<x) xs
    g = filter (>=x) xs
```

---

- No indices
- No memory/pointer management
- No variable assignment

# Imperative vs Functional

Characteristic	Imperative	Functional
Programmer focus	Algorithm design	What the output look like?
State changes	Fundamental	Non-existent
Order of execution	Important	Low importance (compilers may do much work on this)
Primary flow control	Loops, conditionals	Recursion and Functions
Primary data unit	structures or classes	Functions

- Other pure/quasi-pure languages: Erlang, Scala, F, LISP.



# Why Functional Programming? Why Haskell?

- ① Haskell's expressive power can improve productivity/understandability/maintanibility
  - Get best from compiled and interpreted languages
  - Can understand what complex library does
- ② Strong typed - Catches bugs at *compile time*
- ③ Powerful type inference engine
- ④ New Testing metologies
- ⑤ Automatic parallelization due to code purity

# Haskell platform

A full comprehensive, development environment for Haskell<sup>12</sup>.

## Installation

- `$sudo apt-get install haskell-platform`

**GHC** (Great Glasgow Compiler): State of the art

**GHCi** A read-eval-print loop interpreter

**Cabal** Build/distribute/retrieve libraries

**Haddock** A high quality documentation generation tool for Haskell

---

<sup>1</sup><https://www.haskell.org/platform/index.html>

<sup>2</sup><http://tryhaskell.org/>

# What really is Haskell?

## Purely Functional language

- Functions are first-class object (same things as data)
- Deterministic - No Side Effect- same function call  $\Rightarrow$  same Output, EVER!  
This *referential transparency* leaves room for compiler optimization and allow to mathematically prove correctness.
- safely replace expressions by its (unique) result value
- **Evaluate expression** rather than execute instruction
- Function describes what data are, not what what to do to...
- Everything (variables, data structures...) is immutable
- Multi-parameters function simply does not exists.

# Haskell is Lazy

It won't execute anything until it is *really needed*

- It is possible to define and work with infinite data structures
- Define new control structure just by defining a function.
- Reasoning about time/space complexity much more complicated



## Understanding laziness

```
lazyEval 0 b = 1
```

```
lazyEval _ b = b
```

- b never computed if the first parameter is zero
- this call is safe:

```
lazyEval 0 (2^123123123123123123123)
```

- this is not

```
lazyEval 1 (2^123123123123123123123)
```

Strict evaluation: parameter are evaluated **before** to be passed to functions

```
int cont=0;
auto fcall = [] (int a, int b)
{if(a==0) return 1; else return b;};
auto f1 = [] () { cont++; return 1;};
auto f2 = [] () { cont+=10; return 2;};
fcall (f1(),f2()));
```

fcall will always increments *cont* twice!

# Hello World

## Our First Program

Create a file *hello.hs* and compile with the followings

```
main = putStrLn "Hello World with Haskell"  
$ghc -o hello hello.hs
```

## GHCi

Execute and play with GHCi by simply typing

```
reverse [1..10]  
:t foldl  
[1..]  
(filter (even) .reverse) [1..100]
```

# Hello World - 2

Another example, the  $k^{th}$  Fibonacci number (type in GHCi):

```
let f a b k = if k==0 then a else f b (a+b) (k-1)
```

- Defines a recursive function  $f$  that takes  $a, b, k$  as parameters:
- Spaces are important. Are like function call operator  $()$  in C-like languages.
- Wait, three space in  $f a b k$ : 3 function calls? YES!. **Every function in Haskell officially only takes one parameter.**
- $f$  infact has type

```
f :: Integer -> (Integer -> (Integer -> Integer))
```

i.e. a function that takes an integer and return (the  $->$ ) a function that takes an integer and return ...

```
f 0 :: Integer -> (Integer -> Integer)
```

```
f 0 1 :: Integer -> Integer
```

```
f 0 1 10 :: Integer
```

## Hello World - 3

### Number of distinct powers counting (Project Euler 29)

Consider all integer combinations of  $a, b$  for  $2 \leq a, b \leq 100$ : how many distinct terms are in the sequence generated by  $a^b$ ?

$$2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32$$

$$3^2 = 9, 3^3 = 27, 3^4 = 81, 3^5 = 243$$

$$4^2 = 16, 4^3 = 64, 4^4 = 256, 4^5 = 1024$$

$$5^2 = 25, 5^3 = 125, 5^4 = 625, 5^5 = 3125$$

### Naïve solution

```
np :: Integer -> Integer -> Int
np a b = let l = [a^b | a<-[2..a], b<-[2..b]]
  in length (remDup l)
  where
    remDup = (map head . group . sort)
```



# Hello World - 3

## Number of distinct powers counting (Project Euler 29)

Consider all integer combinations of  $a, b$  for  $2 \leq a, b \leq 100$ : how many distinct terms are in the sequence generated by  $a^b$ ?

$$2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32$$

$$3^2 = 9, 3^3 = 27, 3^4 = 81, 3^5 = 243$$

$$4^2 = 16, 4^3 = 64, 4^4 = 256, 4^5 = 1024$$

$$5^2 = 25, 5^3 = 125, 5^4 = 625, 5^5 = 3125$$

## Naïve solution

```
np :: Integer -> Integer -> Int
np a b = let l = [a^b | a<-[2..a], b<-[2..b]]
          in length (remDup l)
          where
            remDup = (map head . group . sort)
```

# Statically Typed

- Haskell is strictly typed
- Helps in thinking and express program structure
- **Turns run-time errors into compile-time errors.** If it compiles, it must be correct is mostly true<sup>3</sup>.

Abstraction: Every idea, algorithm, and piece of data should occur exactly once in your code.

Haskell features as parametric polymorphis, typeclasses high-order functions greatly aid in fighting repetition.

---

<sup>3</sup>It is still quite possible to have errors in logic even in a type-correct program

# What really is Haskell?

## C-like vs Haskell

Code as the one that follows

```
int acc = 0;
for ( int i = 0; i < lst.length; i++ )
    acc = acc + 3 * lst[i];
```

is full of low-level details of iterating over an array by keeping track of a current index. It much elegantly translates in:

```
sum (map (*3) lst)
```

Other examples:

```
partition (even) [49, 58, 76, 82, 83, 90]
```

```
--prime number generation
```

```
let pgen (p:xs) = p : pgen [x|x <- xs, x`mod`p > 0]
```

```
take 40 (pgen [2..])
```

## Section 2

### Basics - Syntax

# Syntax Basics

- Arithmetic and Boolean algebra works as expected

```
v1 = 12
```

```
v2 = mod (v1+3) 10
```

```
v3 = not $ True || (v2>=v1) --not (True || (v2>=v1))
```

- Function definition is made up of two part: type and body.

The body is made up of several *clause* that are evaluated (pattern matched) **top to bottom**.

```
1 exp _ 0 = 1
```

```
2 exp 0 _ = 0
```

```
3 exp a b = a * (exp a (b-1))
```

What if we swap line 2 and 3?

# Syntax Basics

- Arithmetic and Boolean algebra works as expected

```
v1 = 12
```

```
v2 = mod (v1+3) 10
```

```
v3 = not $ True || (v2>=v1) --not (True || (v2>=v1))
```

- Function definition is made up of two part: type and body.

The body is made up of several *clause* that are evaluated (pattern matched) **top to bottom**.

```
4 exp _ 0 = 1
```

```
5 exp 0 _ = 0
```

```
6 exp a b = a * (exp a (b-1))
```

**What if we swap line 2 and 3?**

- Comments:

```
--this is an inline comment
```

```
{-
```

```
All in here is comment
```

```
-}
```

# Syntax Basics

- Arithmetic and Boolean algebra works as expected

```
v1 = 12
```

```
v2 = mod (v1+3) 10
```

```
v3 = not $ True || (v2>=v1) --not (True || (v2>=v1))
```

- Function definition is made up of two part: type and body.

The body is made up of several *clause* that are evaluated (pattern matched) **top to bottom**.

```
7 exp _ 0 = 1
```

```
8 exp 0 _ = 0
```

```
9 exp a b = a * (exp a (b-1))
```

**What if we swap line 2 and 3?**

- Comments:

```
--this is an inline comment
```

```
{-
```

```
All in here is comment
```

```
-}
```

## Guards, where, let

- Guards, let and where constructs

```
1 fastExp :: Integer -> Integer -> Integer
2 fastExp _ 0 = 1
3 fastExp a 1 = a
4 fastExp a b
5   | b < 0 = undefined
6   | even b = res * res
7   | otherwise = let next = (fastExp a (b-1)) in (a * next)
8   where res = (fastExp a (div b 2))
```

Suppose we execute *fastExp 2 7*. The call stack would be

- *fastExp 2 7* line 7 pattern match
- *fastExp 2 6* line 6 pattern match

---

<sup>4</sup>Here for more informations: [https://wiki.haskell.org/Let\\_vs\\_Where](https://wiki.haskell.org/Let_vs_Where)



# Guards, where, let

- Guards, let and where constructs

```

1 fastExp :: Integer -> Integer -> Integer
2 fastExp _ 0 = 1
3 fastExp a 1 = a
4 fastExp a b
5   | b < 0 = undefined
6   | even b = res * res
7   | otherwise = let next = (fastExp a (b-1)) in (a * next)
8   where res = (fastExp a (div b 2))

```

Suppose we execute *fastExp 2 7*. The call stack would be

- *fastExp 2 7* line 7 pattern match
- *fastExp 2 6* line 6 pattern match
- *fastExp 2 3* line 7 pattern match

---

<sup>4</sup>Here for more informations: [https://wiki.haskell.org/Let\\_vs\\_Where](https://wiki.haskell.org/Let_vs_Where)

# Guards, where, let

- Guards, let and where constructs

```

1 fastExp :: Integer -> Integer -> Integer
2 fastExp _ 0 = 1
3 fastExp a 1 = a
4 fastExp a b
5   | b < 0 = undefined
6   | even b = res*res
7   | otherwise = let next=(fastExp a (b-1)) in (a * next)
8   where res=(fastExp a (div b 2))

```

Suppose we execute *fastExp 2 7*. The call stack would be

- fastExp 2 7 line 7 pattern match
- fastExp 2 6 line 6 pattern match
- fastExp 2 3 line 7 pattern match
- fastExp 2 2 line 6 pattern match

---

<sup>4</sup>Here for more informations: [https://wiki.haskell.org/Let\\_vs\\_Where](https://wiki.haskell.org/Let_vs_Where)

# Guards, where, let

- Guards, let and where constructs

```

1 fastExp :: Integer -> Integer -> Integer
2 fastExp _ 0 = 1
3 fastExp a 1 = a
4 fastExp a b
5   | b < 0 = undefined
6   | even b = res*res
7   | otherwise = let next=(fastExp a (b-1)) in (a * next)
8   where res=(fastExp a (div b 2))

```

Suppose we execute *fastExp 2 7*. The call stack would be

- fastExp 2 7 line 7 pattern match
- fastExp 2 6 line 6 pattern match
- fastExp 2 3 line 7 pattern match
- fastExp 2 2 line 6 pattern match
- fastExp 2 1 line 3 pattern match, STOP RECURSION

---

<sup>4</sup>Here for more informations: [https://wiki.haskell.org/Let\\_vs\\_Where](https://wiki.haskell.org/Let_vs_Where)

# Guards, where, let

- Guards, let and where constructs

```

1 fastExp :: Integer -> Integer -> Integer
2 fastExp _ 0 = 1
3 fastExp a 1 = a
4 fastExp a b
5   | b < 0 = undefined
6   | even b = res*res
7   | otherwise = let next=(fastExp a (b-1)) in (a * next)
8   where res=(fastExp a (div b 2))

```

Suppose we execute *fastExp 2 7*. The call stack would be

- fastExp 2 7 line 7 pattern match
- fastExp 2 6 line 6 pattern match
- fastExp 2 3 line 7 pattern match
- fastExp 2 2 line 6 pattern match
- fastExp 2 1 line 3 pattern match, STOP RECURSION

In contrast to where, let are expressions and can be used anywhere<sup>4</sup>.

---

<sup>4</sup>Here for more informations: [https://wiki.haskell.org/Let\\_vs\\_Where](https://wiki.haskell.org/Let_vs_Where)

# Guards, where, let

- Guards, let and where constructs

```

1 fastExp :: Integer -> Integer -> Integer
2 fastExp _ 0 = 1
3 fastExp a 1 = a
4 fastExp a b
5   | b < 0 = undefined
6   | even b = res*res
7   | otherwise = let next=(fastExp a (b-1)) in (a * next)
8   where res=(fastExp a (div b 2))

```

Suppose we execute *fastExp 2 7*. The call stack would be

- fastExp 2 7 line 7 pattern match
- fastExp 2 6 line 6 pattern match
- fastExp 2 3 line 7 pattern match
- fastExp 2 2 line 6 pattern match
- fastExp 2 1 line 3 pattern match, STOP RECURSION

In contrast to where, let are expressions and can be used anywhere<sup>4</sup>.

---

<sup>4</sup>Here for more informations: [https://wiki.haskell.org/Let\\_vs\\_Where](https://wiki.haskell.org/Let_vs_Where)

# If, case

- if construct works as expected

```
1 div' n d = if d==0 then Nothing else Just (n/d)
```

- case construct

Useful when we don't wish to define a function every time we need to do pattern matching.

```
f p11 ... p1k = e1
```

```
...
```

```
f pn1 ... pnk = en
```

--where each  $p_{ij}$  is a pattern, is semantically equivalent

```
f x1 x2 ... xk = case (x1, ..., xk) of
```

```
(p11, ..., p1k) -> e1
```

```
...
```

```
(pn1, ..., pnk) -> en
```

All patterns of a function return the same type hence all the RHS of the case have the same type

## case construct: example

### case construct example

Pattern match “outside” the function definition. Note that all the cases return the same type (a list of  $b$ ’s in this case)

```
cE :: (Ord a) => a -> a -> [b]
cE a b xs = case (a `compare` b, xs) of
    (_, []) -> []
    (LT, xs) -> init xs
    (GT, xs) -> tail xs
    (EQ, xs) -> [head xs]
```

# Ranges

## ranges

Shortcut for listing stuff that can be enumerated. What if we need to test if a string contains a letter up to the lower case `j`? (Explicitly list all the letters is not the correct answer).

```
['a'..'j'] -- results in "abcdefghij" (String are [Char])
```

It work even in construction infinite list

```
[1,3..] -- results in [1,3,5,7,9,11,13,15.....]
```

and because of laziness we can (safely) do

```
take 10 [1,3..]
```



# List Comprehension

## list comprehension

It is a familiar concept for those who already have some experience in python. It resembles the mathematical set specification. For instance, let's compute the list of the factorial of the natural numbers

```
[product [2..x] | x<-[1..]]
```

More examples:

```
[[2..x*2] | x<-[1..]]
```

```
[filter (even) [2..x] | x<-[1..]]
```

```
--:m Data.Char (ord)
```

```
[let p=y*x in if even p then (negate p) else  
 (p*2) | x<-[1..10], y<-(map ord ['a'..'z'])]
```

```
--:m Data.List (nub)
```

```
nub $ map (\(x,y,z) -> z) [(a,b,c) | a<-[1..20], b<-[1..20],  
 c<-[1..20], a^2+b^2==c^2, a+b+c>10]
```

## Section 3

### Basics - List Functions

# Lists

List is the most used Data structure in Haskell

- Homogenous - Only objects of the same type
- Denoted by [ CONTENT OF THE LIST ]
- [ ["passions"], ["poetry"], ["and"], ["the"], ["ego"] ["have"], ["been"], ["seen"], ["as"], ["perpetual"] ["explosions?"] ]
- String are **List of Char**. We can use list function of strings

## length

length is a function that return the length of a List

```
length [1,2,3,4]
```

```
length "Hi guys"
```

# Concat

A common task is to merge two list. Done using the `++` operator

- `[1..3] ++ [4..10], "Hi" ++ "Guys"`