A Journey in Functional Programming

An introduction to Haskell

Davide Spataro[1]

[1]Department of mathematics And Computer Science
Univeristy of Calabria

April 21, 2015

Table of contents I

Table of contents II

Section 1

Introduction - Functional Programming Haskell

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 correnspondence.
- 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 desidered 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[1][2].

Installation

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

GHC (Great Glasgow Compiler): State of the art

GHCi A read-eval-print loop interpreter

Cabal Build/distribuite/retrieve libraries

Haddock A high quality documentation generation tool for Haskell

[1] https://www.haskell.org/platform/index.html
[2] 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 Ouput, EVER!
  This *referial 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 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 (2^123123123123123123)
  ```
- this is not
  ```
  lazyEval 1  (2^123123123123123123)
  ```

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 Currying

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 Currying - 2

Currying directly and naturally address the high-order functions support Haskell machinery.

High-order function:

- Take function as parameter
- returns a function

zipwith

- Combines two list of type *a* and *b* using a function f that takes a parameter of type a and one of type b and return a value of type *c*, producing a list of elements of type *c*.
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

Hello Currying - 2

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ _ [] = []
zipWith _ [] _ = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

usage examples

```
zipWith (+) [1,2,3] [4,5,6] = [5,7,9]
zipWith (*) [1,2,3] [4,5,6] = [4,10,18]
zipWith (\a b ->(+).(2*))) [1..] [1..]
```

What about this call? (missing one parameter)

```
let l = zipWith (*) [1,2,3]
l [3,2,1]
```

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?

$2^2 = 4, 2^3 = 8, \boxed{2^4 = 16}, 2^5 = 32$

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

$\boxed{4^2 = 16}, 4^3 = 64, 4^4 = 256, 4^5 = 1024$

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

Naïve solution

```
np a b = length $ nub l
where l = [c^d | c<-[2..a],d<-[2..b]]
```

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?

$2^2 = 4, 2^3 = 8, \boxed{2^4 = 16}, 2^5 = 32$

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

$\boxed{4^2 = 16}, 4^3 = 64, 4^4 = 256, 4^5 = 1024$

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

Naïve solution

```
np a b = length $ nub l
where l = [c^d | c<-[2..a],d<-[2..b]]
```

Statically Typed

- Haskell is stricly typed
- Helps in thinking and express program structure
- **Turns run-time errors into compile-time errors**. If it compiles, it must be correct is moslty true[3].

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.

[3] 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 elegantely 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

[4]Here for more informations: 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

[4]Here for more informations: 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

[4]Here for more informations: 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

[4] Here for more informations: 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[4].

[4] Here for more informations: 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[4].

[4] Here for more informations: https://wiki.haskell.org/Let_vs_Where

If, case

- if construct works as expected

```
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 pij is a pattern,
--is semantically equivalent to:
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 are useful!

- Colletcion of elements of the **SAME TYPE**.
- Delimited by square brackets and elements separated by commas.
- List che be *consed*. The **cons** operator (:) is used to incrementally build list putting an element at its head.
- empty list is []
- cons is a function that takes two parameter

```
(:) :: a -> [a] -> [a]
1:2:3:4:[]
```

List Comprehension

list comprehension

It is a familiar concept for those who already have some experience in python It resambles 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]
```

Lambda functions - The Idea

- Anonymous functions i.e. no need to give it a name
- $\lambda yx \rightarrow 2x + x^y$ translates in
```
(\x y -> 2*x + x^y)
```
- Usually used withing high order function context.
```
map (\x -> x*x-3) [1,10..300]
map (\x -> let p = ord x in if even p then p else p^2)
        "Lambda functions are cool!"
```
- $f = (\backslash x_1..x_n - > exp(x_1..x_n))(v_1, .., v_k)$ substitute each occurence of the free variable x_i with the value v_i. If $k < n$ f is again a function.
-
```
let f = (\x y z -> x+y+z)
let sum3 = f 2 3 = (\z -> 2+3+z) --again a function
sum23z 4 -> = 9
```

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

lenght

`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"`
- When possible use (:) instead of (++), the list concatenation operator. It's much more faster!

Section 4

Coding - Problems on Lists

Last element

Problem Statement

Given a polymorphic list *l* of type [*a*], find the last element of l (not using function *last*, I'm sorry).

Examples:

```
_last [1,2,3,4] = 4
_last ["programming","haskell","is","cool"]= "cool"
```

Last element

Problem Statement

Given a polymorphic list *l* of type [*a*], find the last element of l (not using function *last*, I'm sorry).

Examples:

```
_last [1,2,3,4] = 4
_last ["programming","haskell","is","cool"]= "cool"
```

Last element

Problem Statement

Given a polymorphic list *l* of type [*a*], find the last element of l (not using function *last*, I'm sorry).

Examples:

```
_last [1,2,3,4] = 4
_last ["programming","haskell","is","cool"]= "cool"
```

Solution

```
_last :: [a] -> a
_last []     = error "Undefined operation"
_last (x:[]) = x
_last (x:xs) = _last xs
```

Last element

Problem Statement

Given a polymorphic list *l* of type [*a*], find the last element of l (not using function *last*, I'm sorry).

Examples:

```
_last [1,2,3,4] =  4
_last ["programming","haskell","is","cool"]= "cool"
```

Solution

```
_last :: [a] -> a
_last []    = error "Undefined operation"
_last (x:[]) = x
_last (x:xs) = _last xs
```

k'th element of a list

Problem Statement

Find the *k*'th element of a list where the first element has index 1

Examples:

```
elementAt 2 [3,35,32,33]  =  35
elementAt 3 [('a',97),('b',98),('c',99)] = ('c',99)
elementAt 4 [('a',97),('b',98),('c',99)] = error "Index out
```

k'th element of a list

Problem Statement

Find the k'th element of a list where the first element has index 1

Examples:

```
elementAt 2 [3,35,32,33]  =  35
elementAt 3 [('a',97),('b',98),('c',99)] = ('c',99)
elementAt 4 [('a',97),('b',98),('c',99)] = error "Index out
```

k'th element of a list

Problem Statement

Find the *k*'th element of a list where the first element has index 1

Examples:

```
elementAt 2 [3,35,32,33]  =  35
elementAt 3 [('a',97),('b',98),('c',99)] = ('c',99)
elementAt 4 [('a',97),('b',98),('c',99)] = error "Index out
```

Solution

```
elementAt :: Integer -> [a] -> a
elementAt  _ []     = error "index out of bound"
elementAt 1 (x:_)   = x
elementAt n (_:xs)    = elementAt (n-1) xs
```

k'th element of a list

Problem Statement

Find the k'th element of a list where the first element has index 1

Examples:

```
elementAt 2 [3,35,32,33]  =  35
elementAt 3 [('a',97),('b',98),('c',99)] = ('c',99)
elementAt 4 [('a',97),('b',98),('c',99)] = error "Index out
```

Solution

```
elementAt :: Integer -> [a] -> a
elementAt  _ []    = error "index out of bound"
elementAt 1 (x:_)   = x
elementAt n (_:xs)    = elementAt (n-1) xs
```

Palindromic List

Problem Statement

Write a function that returns a boolean value tha indicates whether the input list is palindromic or not. 1

Examples:

```
palindrome "itopinonavevanonipoti" = True
palindrome "[1,2,3,3,1] = False
```

Palindromic List

Problem Statement

Write a function that returns a boolean value tha indicates whether the input list is palindromic or not. 1

Examples:

```
palindrome "itopinonavevanonipoti" = True
palindrome "[1,2,3,3,1] = False
```

Solution

```
palindrome1 l = l== reverse l

palindrome2 [] = True --empty list is palindrome
palindrome2 (_:[]) = True --one element is palindrome
palindrome2 l
     | head l /= last l = False
     | otherwise = palindrome2 ((tail . init) l)
```

Palindromic List

Problem Statement

Write a function that returns a boolean value tha indicates whether the input list is palindromic or not. 1

Examples:

```
palindrome  "itopinonavevanonipoti"  =  True
palindrome  "[1,2,3,3,1]  =  False
```

Solution

```
palindrome1 l = l== reverse l

palindrome2 [] = True --empty list is palindrome
palindrome2 (_:[]) = True --one element is palindrome
palindrome2 l
     | head l /= last l = False
     | otherwise  = palindrome2 ((tail . init) l)
```

Section 5

Coding - Project Euler Problem 1

Problems 1

Problem Statement

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

How would you solve it using Haskell?

Problems 1

Problem Statement

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

How would you solve it using Haskell?

```
problem1' = sum .
            filter (\x -> x `mod` 3==0 || x `mod` 5 ==0)
```

Problems 1

Problem Statement

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

How would you solve it using Haskell?

```
problem1' = sum .
            filter (\x -> x `mod` 3==0 || x `mod` 5 ==0)
```

Section 6

Coding - Project Euler Problem 26

Problems 26

Problem Statement

A unit fraction contains 1 in the numerator. Where 0.1(6) means 0.166666..., and has a 1-digit recurring cycle. It can be seen that $1/7$ has a 6-digit recurring cycle. **Find the value of $d < 1000$ for which $1/d$ contains the longest recurring cycle in its decimal fraction part.**

- $1/2 = 0.5$ - 0-recur
- $1/3 = 0.(3)$ - 1-recur
- $1/4 = 0.25$ - 0-recur
- $1/5 = 0.2$ - 0-recur
- $1/6 = 0.1(6)$ - 1-recur
- $1/7 = 0.(142857)$ - 6-recur
- $1/8 = 0.125$ - 0-recur
- $1/9 = 0.(1)$ - 1-recur
- $1/10 = 0.1$ - 0-recur

Problems 26 - Solution

Key idea: Find the order of 10 in $\mathbb{N}/p\mathbb{N}$

The length of the repetend (period of the repeating decimal) of $1/p$ is equal to the order of 10 modulo p. If 10 is a primitive root modulo p, the repetend length is equal to $p-1$; if not, the repetend length is a factor of $p-1$. This result can be deduced from Fermat's little theorem, which states that $10p-1 \equiv 1\ (mod\ p)$.(Wikipedia)

Reminder: order of an element g in $\mathbb{N}/p\mathbb{N}$

The smallest power n **of** g **s.t.** $g^n \equiv 1\ (mod\ p)$.

Problems 26 - Order finding example

Find the order of 10 in $\mathbb{N}/13\mathbb{N}$

$$10^1 \equiv 10 \; (mod\ 13)$$

$$10^2 \equiv 9 \; (mod\ 13)$$

$$10^3 \equiv 12 \; (mod\ 13)$$

$$10^4 \equiv 3 \; (mod\ 13)$$

$$10^5 \equiv 4 \; (mod\ 13)$$

$$\mathbf{10^6 \equiv 1 \; (mod\ 13)}$$

- 6 is the order of 10 (modulo 13)
- `map (\a -> mod (10^a) 13) [1..12]`

Problems 26 - Order finding example

So now the problem is. Compute the order of numbers $n < 1000$ and return the one that have maximum order

```haskell
--modulo, current order
order :: Integer -> Integer -> Integer
order a ord
| mod (10^ord) a == 1 = ord
| ord > a      = 0
| otherwise     = order a (ord+1)
```

```haskell
maxo = fst $  maximumBy comparing $ pp
  where
    comparing  = (\(m,n) (p,q) -> n `compare` q)
     pp  = map (\x->(x,order x 1))
          (filter (\x-> mod x 10 > 0 ) [1,3..1000])
```