# Haskell: From Basic to Advanced

## Part 1 – Basic Language

## Haskell buzzwords

- Functional
- Pure
- Lazy
- Strong static typing
- Type polymorphism
- Type classes
- Monads
- Haskell 98 / Haskell 2010
- GHC
  - Glasgow Haskell Compiler
- GADTs
  - Generalized Algebraic Data Types
- STM
- Hackage

# History

- Named after the logician Haskell B. Curry
- Designed by a committee aiming to
  - consolidate (lazy) FP languages into a common one
  - develop a language basis for FP language research
- Well crafted and designed pure FP language
  - concise and expressive
  - strong theoretical basis (λ-calculus)
  - sophisticated type system
  - evaluation on demand, at most once (laziness)

## Hello, World!

```haskell
-- File: hello.hs
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

Not the most representative Haskell program...

- '`--`' starts a one-line comment
- '`::`' denotes a type declaration
- '=' defines a function clause
- All but the last line are optional
- Source file names end in "`.hs`"

# Quick sort over lists

```haskell
-- File: qsort.hs
qsort [] = []
qsort (p:xs) =
  qsort [x | x <- xs, x < p] ++
  [p] ++
  qsort [x | x <- xs, x >= p]
```

- `[]` for the empty list
- `(h:t)` notation for a list with head **h** and tail **t**
- Very compact and easy to understand code
- Small letters for variables
- Simpler list comprehensions
- No parentheses or punctuations needed

```erlang
%% Erlang version
qsort([]) -> [];
qsort([P|Xs]) ->
  qsort([X || X <- Xs, X < P]) ++
  [P] ++ % pivot element
  qsort([X || X <- Xs, X >= P]).
```

# Another quick sort program

```haskell
-- File: qsort2.hs
qsort [] = []
qsort (p:xs) =
  qsort lt ++ [p] ++ qsort ge
    where lt = [x | x <- xs, x < p]
          ge = [x | x <- xs, x >= p]
```

- Equivalent to the previous definition
- Which version to prefer is a matter of taste

```haskell
-- File: qsort.hs
qsort [] = []
qsort (p:xs) =
  qsort [x | x <- xs, x < p] ++
  [p] ++
  qsort [x | x <- xs, x >= p]
```

## Running the Haskell interpreter

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/ :? for help
Loading package ...  <SNIP>
Loading package base ... linking ... done.
Prelude> 6*7
42
Prelude> :quit
Leaving GHCi.
$
```

- The Glasgow Haskell interpreter is called ‘`GHCi`'
- The interactive shell lets you write any Haskell expressions and run them
- The “**`Prelude>`**” means that this library is available
- To exit the interpreter, type “**`:quit`**” (or “**`:q`**” or “**`^D`**”)

# Loading and running a program

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/ :? for help
Loading package ...  <SNIP>
Loading package base ... linking ... done.
Prelude> :load qsort.hs
[1 of 1] Compiling Main             ( qsort.hs, interpreted )
Ok, modules loaded: Main.
*Main> qsort [5,2,1,4,2,5,3]
[1,2,2,3,4,5,5]
```

- Use "**`:load`**" (or "**`:l`**") to load a file in the interpreter

# Functions and values

```
len [] = 0
len (x:xs) = len xs + 1

nums = [17,42,54]
n = len nums
```

As we will soon see, functions *are* values!

- Functions are written as equations (no **`fun`** keywords)
- Their definitions can consist of several clauses
- Function application is written without parentheses
- We can define values and apply functions to them
- Local definitions using **`let`** expressions or **`where`** clauses

```
nums = [17,42,54]
n = let len [] = 0
        len (x:xs) = len xs + 1
    in  len nums
```

```
nums = [17,42,54]
n = len nums
  where len [] = 0
        len (x:xs) = len xs + 1
```

# Layout matters!

- Note the spaces: all clauses of a function need to be aligned

```
nums = [17,42,54]
n = let len [] = 0
        len (x:xs) = len xs + 1
    in  len nums
```

- On the other hand, the following is not legal

```
nums = [17,42,54]
n = let len [] = 0
      len (x:xs) = len xs + 1
    in len nums
```

- One can also write

```
nums = [17,42,54]
n = let { len [] = 0; len (x:xs) = len xs + 1 }
    in  len nums
```

## Pattern matching

```haskell
len [] = 0
len (x:xs) = len xs + 1
```

- Function clauses are chosen by pattern matching
- Pattern matching also available using **case** expressions

```haskell
len ls = case ls of
           [] -> 0
           x:xs -> len xs + 1
```

- Strong static typing ensures the above is equivalent to

```haskell
len ls = case ls of
           x:xs -> len xs + 1
           _ -> 0
```

# Pattern matching (cont.)

```
-- take first N elements from a list
take 0 ls = []
take n [] = []
take n (x:xs) = x : take (n-1) xs
```

- Pattern matching can involve ‘multiple’ arguments
- But no repeated variables in patterns (as in ML)
- Pattern matching can also be expressed with **`case`**

```
-- equivalent definition using case
take n ls =
  case (n, ls) of
    (0, _)      -> []
    (_, [])     -> []
    (n, x:xs) -> x : take (n-1) ls
```

**Note:** All branches of a **`case`** have to return the same type

## Pattern matching and guards

- Pattern matching can also involve guards

```haskell
-- a simple factorial function
fac 0 = 1
fac n | n > 0 = n * fac (n-1)
```

This clause will match only for positive numbers

```
Prelude> :l factorial.hs
[1 of 1] Compiling Main             ( factorial.hs, interpreted )
Ok, modules loaded: Main.
*Main> fac 3
6
*Main> fac 42
1405006117752879898543142606244511569936384000000000
*Main> fac (-42)
*** Exception: factorial.hs:(2,1)-(3.31):
        Non-exhaustive patterns in function fac
```

No “match non exhaustive” warnings; runtime errors instead

# More on guards

- More than one clauses can contain guards

```
-- returns the absolute value of x
abs x | x >= 0 = x
abs x | x < 0 = -x
```

- We can abbreviate repeated left hand sides

```
-- returns the absolute value of x
abs x | x >= 0 = x
      | x < 0 = -x
```

- Haskell also has **`if-then-else`**

```
-- returns the absolute value of x
abs x = if x >= 0 then x else -x
```

# Type annotations

```
len :: [a] -> Integer
len [] = 0
len (x:xs) = len xs + 1

nums :: [Integer]
nums = [17,42,54]

n :: Integer
n = len nums
```

- Every function and value has an associated type
- This type can be (optionally) supplied by the programmer in the form of an annotation
- Note the variable in the type of **`len`** (a polymorphic type)

# Type notation

- Integer, String, Float, Double, Char, ... Base types
- [X] A list of values of type X
- X -> Y A function from X values to Y values
- (X,Y,Z) A 3-tuple with an X, a Y and a Z value
- ...

```
pair_sum :: (Integer,Integer) -> Integer
pair_sum (a,b) = a + b

triple :: (Integer,(String,Integer),[Char])
triple = (17,("foo",42),['b','a','r'])
```

# Type inference

- A type annotation is a contract between the author and the user of a function definition
- In Haskell, writing type annotations is *optional*
  - the compiler will infer types and detect inconsistencies
  - in fact, it will infer the best possible type (principal type)
- Still, providing type annotations is recommended
  - to enhance readability of programs
  - especially when the intended meaning of functions is not “immediately obvious”
- But, as we will see, often Haskell infers better types than those we would normally write by hand

# User defined types

We can create new types by enumerating constants and constructors (they need to start with uppercase)

```haskell
data Color = Green | Yellow | Red

next Green = Yellow
next Yellow = Red
next Red = Green
```

```haskell
data Shape = Rectangle Double Double
           | Circle Double

area (Rectangle x y) = x * y
area (Circle r) = 3.14159265 * r * r
```

A type used in another type (such as `Double` above) has to be wrapped in a constructor -- why?

# Constructors vs. pattern matching

- Constructors are a special kind of functions that construct values

  e.g. **`Rectangle 3.0 2.0`** constructs a **`Shape`** value

- Constructors have types!

  e.g. **`Rectangle :: Double -> Double -> Shape`**

- Pattern matching can be used to “destruct” values

  e.g. below we define a function that can extract the first (**`x`**) component of a **`Rectangle`** value

```
getX (Rectangle x y) = x
```

# Recursive data types

- Type definitions can be recursive

```haskell
data Expr = Const Double
          | Add Expr Expr
          | Neg Expr
          | Mult Expr Expr

eval :: Expr -> Double
eval (Const c) = c
eval (Add e1 e2) = eval e1 + eval e2
eval (Neg e) = - eval e
eval (Mult e1 e2) = eval e1 * eval e2
```

```
eval (Mult (Const 6.0) (Add (Const 3.0) (Const 4.0)))
  ⇒ ... ⇒ 42.0
```

# Parameterized types

```
data Expr = Const Double
          | Add Expr Expr
          | Neg Expr
          | Mult Expr Expr
```

- Type definitions can also be parameterized

```
data Expr a = Const a
            | Add (Expr a) (Expr a)
            | Neg (Expr a)
            | Mult (Expr a) (Expr a)

type DoubleExpr = Expr Double
```

- Now **`Expr`** is a parameterized type:
  - It takes a type as “argument” and “returns” a type

## Parameterized types (cont.)

- Another parameterized type definition

```
data Tree a = Empty | Node a (Tree a) (Tree a)

Empty :: Tree a
Node  :: a -> Tree a -> Tree a -> Tree a

depth :: Tree a -> Integer
depth Empty = 0
depth (Node x l r) = 1 + max (depth l) (depth r)
```

- Types can be parameterized on more type variables

```
type Map a b = [(a,b)]
```

# Type synonyms

- Synonyms for types are just abbreviations
- Defined for convenience

```
type String = [Char]

type Name = String
data OptAddress = None | Addr String
type Person = (Name,OptAddress)
```

**A note on names:** The naming style we have been using is mandatory
- Type names and constructor names begin with an uppercase letter
- Value names (and type variables) begin with a lowercase letter

# Higher order functions

- Functions are first class values
- They can take functions as arguments and return functions as results

Type variables

```
map :: (a -> b) -> [a] -> [b]
map f [] = 0
map f (x:xs) = f x : map f xs

nums = [17,42,54]
inc x = x + 1
more_nums = map inc nums
```

- Function application associates to the left

```
f x y = (f x) y
```

# Currying

```haskell
add_t :: (Integer,Integer) -> Integer
add_t (x,y) = x + y

add_c :: Integer -> Integer -> Integer
add_c x y = x + y

add42 = add_c 42
```

- **add_t** takes a *pair* of integers as argument and returns their sum
- **add_c** takes one integer as argument and returns a function that takes another integer as argument and returns their sum (*curried* version)

# Anonymous functions

- A **λ-abstraction** is an anonymous function
- Math syntax:

  *λx.exp*    where *x* is a variable name and
  *exp* is an expression that may use *x*

- Haskell syntax:

  `\`*x* `->` *exp*

- Two examples:

```
inc42 x = x + 42    ≈  inc42 = \x -> x + 42
add x y = x + y     ≈  add = \x -> \y -> x + y
                    ≈  add = \x y -> x + y
```

# Infix operators

- Infix operators (e.g. **+** or **++**) are just "binary" functions

```
x + y
```
≈
```
(+) x y
```

- "Binary" functions can be written with an infix notation

```
add x y
```
≈
```
x `add` y
```

- Apart from the built-in operators, we can define our own
  - Infix operators are built from non-alphanumeric characters

```
[] @@ ys = ys
(x:xs) @@ ys = x : (ys @@ xs)
```

  - Operator precedence and associativity can be specified with "*fixity declarations*"

Strictly, there are no binary functions in Haskell as all functions have only one argument...

## Infix operators & partial application

Even infix operators can be applied partially

```
Prelude> map (42 +) [1,2,3]
[43,44,45]
Prelude> map (+ 42) [1,2,3]
[43,44,45]
Prelude> map ("the " ++) ["dog","cat","pig"]
["the dog","the cat","the pig"]
Prelude> map (++ " food") ["dog","cat","pig"]
["dog food","cat food","pig food"]
```

Notice that for a non-commutative operator order matters!

```
Prelude> map (/ 2) [1,2,3]
[0.5,1.0,1.5]
Prelude> map (2 /) [1,2,3]
[2.0,1.0,0.6666666666666666]
```

# Function composition

- Function composition is easy (and built-in)

```haskell
-- same as the built-in operator . (dot)
compose f g = \x -> f (g x)
```

```
*Main> compose fac length "foo"
6
*Main> (fac . length) "foobar"
720
```

- Composition is not commutative
- What is the type of function composition?

```
*Main> :type compose
compose :: (b -> c) -> (a -> b) -> a -> c
```

## Haskell standard **Prelude**

- A library containing commonly used definitions
- Examples:

```haskell
type String = [Char]
```

```haskell
data Bool = False | True
```

```haskell
True  && x = x
False && _ = False
```

```haskell
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- The core of Haskell is quite small
- In theory, everything can be reduced to λ-calculus

# List comprehensions

- Lists are pervasive in Haskell (as in all FP languages...)
- List comprehensions are a convenient notation for list manipulation
- Recall

```
lt = [y | y <- xs, y < x]
```

which means the same as

```
lt = concatMap f xs
       where
         f y | y < x = [y]
             | otherwise = []
```

(**`concatMap`** is defined in the **`Prelude`**)

# List comprehensions (cont.)

- List comprehensions can have multiple generators

```
-- finds all Pythagorian triples up to n
pythag :: Int -> [(Int,Int,Int)]
pythag n =
  [(x,y,z) | x <- [1..n], y <- [x..n],
             z <- [y..n], x^2 + y^2 == z^2]
```

```
*Main> pythag 13
[(3,4,5),(5,12,13)(6,8,10)]
*Main> pythag 17
[(3,4,5),(5,12,13)(6,8,10),(8,15,17),(9,12,15)]
```

- Note that any list-producing expression can be used as a generator, not just explicit lists
- Similarly, any Boolean expression can be used as a filter

## The lists zip operation

- The function **`zip`** takes two lists as input (curried) and returns a list of corresponding pairs

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip [] ys = []
zip xs [] = []
```

- Two examples:

```
Prelude> zip [17,42,54] ['a','b','c']
[(17,'a'),(42,'b'),(54,'c')]
Prelude> zip [1,2,3,4] ['A'..'Z']
[(1,'A'),(2,'B'),(3,'C'),(4,'D')]
```

# Abstractions using HO functions

- These two functions perform a similar traversal of the list, but apply different operations to elements

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

prod [] = 1
prod (x:xs) = x * prod xs
```

very common technique in FP languages

- We can abstract the traversal part and separate it from the operations

```
foldr op init [] = init
foldr op init (x:xs) = x `op` foldr op init xs

sum  = foldr (+) 0
prod = foldr (*) 1
```

foldr op init [x1,x2,...,x42] ⇒
(x1 \`op\` (x2 \`op\` ... (x42 \`op\` init) ...

## More **foldr** fun

Using **foldr** we can obtain very concise definitions of many common list functions

```
and = foldr (&&) True
concat = foldr (++) []
```

```
xs ++ ys = foldr (:) ys xs
```

```
reverse = foldr (\y ys -> ys ++ [y]) []
```

```
maximum (x:xs) = foldr max x xs
```

## Syntactic redundancy

| **Expression style** | vs. | **Declaration style** |
|---|---|---|
| each function is defined as one expression | ⇔ | each function is defined as a series of equations |
| **`let`** | ⇔ | **`where`** |
| λ | ⇔ | arguments on the left hand side of **`=`** |
| **`case`** | ⇔ | function level pattern matching |
| **`if`** | ⇔ | guards |

## Terminology review

**Higher-order function**: a function that takes another function as argument and/or returns one as a result

**Polymorphic function**: a function that works with arguments of many possible types

**Type scheme**: a type that involves type variables

- the type of a polymorphic function is a type scheme

**Parameterized type**: a type that takes another type as “argument” and “returns” a type

- their constructors are often polymorphic functions

## Haskell: From Basic to Advanced

Part 2 – Type Classes, Laziness, IO, Modules

## Qualified types

- In the types schemes we have seen, the type variables were *universally quantified*, e.g.

```
++ :: [a] -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
```

- In other words, the code of **`++`** or **`map`** could assume *nothing* about the corresponding input

- What is the (principal) type of **`qsort`**?
  - we want it to work on *any list whose elements are comparable*
  - but nothing else
- The solution: qualified types

## The type of **qsort**

```haskell
-- File: qsort2.hs
qsort [] = []
qsort (p:xs) =
  qsort lt ++ [p] ++ qsort ge
    where lt = [x | x <- xs, x < p]
          ge = [x | x <- xs, x >= p]
```

```
Prelude> :l qsort2.hs
[1 of 1] Compiling Main             ( qsort2.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t qsort
qsort :: Ord a => [a] -> [a]
```

- The type variable a is *qualified* with the type class **`Ord`**
- **`qsort`** works only with any list whose elements are instances of the **`Ord`** type class

# Type classes and instances

```
class Ord a where
  (>)  :: a -> a -> Bool
  (<=) :: a -> a -> Bool
```

defines a *type class* named **`Ord`**

```
data Student = Student Name Score
type Name = String
type Score = Integer

better :: Student -> Student -> Bool
better (Student n1 s1) (Student n2 s2) = s1 > s2
```

Note that we can use the same name for a new data declaration and a constructor

```
instance Ord Student where
  x > y = better x y
  x <= y = not (better x y)
```

makes **`Student`** an *instance* of **`Ord`**

**Note**: The actual **`Ord`** Class in the standard **`Prelude`** defines more functions than these two

# Type classes

- Haskell's type class mechanism has some parallels to Java's interface classes
- **Ad-hoc polymorphism** (also called **overloading**)
  - for example, the **>** and **<=** operators are overloaded
  - the **`instance`** declarations control how the operators are implemented for a given type

**Some standard type classes**

| | |
|---|---|
| **`Ord`** | used for totally ordered data types |
| **`Show`** | allow data types to be printed as strings |
| **`Eq`** | used for data types supporting equality |
| **`Num`** | functionality common to all kinds of numbers |

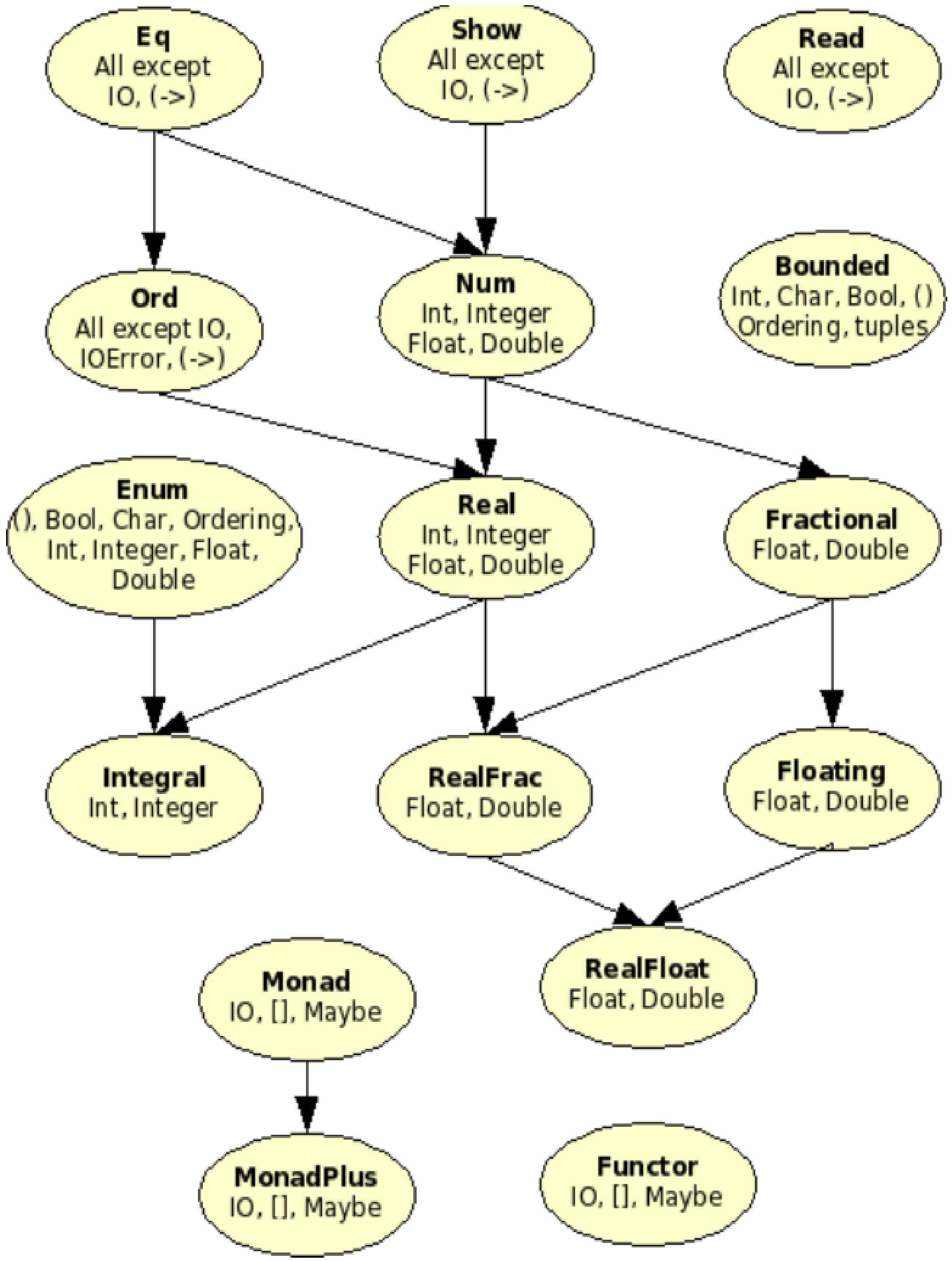
# Example: equality on Booleans

```haskell
data Bool = True | False
```

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```haskell
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
  x /= y = not (x == y)
```

# Predefined classes and instances



## Referential transparency

- *Purely functional* means that *evaluation has no side-effects*
  - A function maps an input to an output value and does nothing else (i.e., is a "real mathematical function")

- **Referential transparency**:

"*equals can be substituted with equals*"

We can disregard evaluation order and duplication of evaluation

`f x + f x` is always same as `let y = f x in y + y`

Easier for the programmer (and compiler!) to reason about code

## Lazy evaluation

```haskell
-- a non-terminating function
loop x = loop x
```

```
Prelude> :l loop
[1 of 1] Compiling Main             ( loop.hs, interpreted )
Ok, modules loaded: Main.
*Main> length [fac 42,loop 42,fib 42]
3
```

- We get a “correct” answer immediately
- Haskell is lazy: computes a value only when needed
  - none of the elements in the list are computed in this example
  - functions with undefined arguments might still return answers
- Lazy evaluation can be
  - efficient since it evaluates a value at most once
  - surprising since evaluation order is not “the expected”

# Lazy and infinite lists

- Since we do not evaluate a value until it is asked for, there is no harm in defining and manipulating infinite lists

```
from n = n : from (n + 1)

squares = map (\x -> x * x) (from 0)

even_squares = filter even squares

odd_squares = [x | x <- squares, odd x]
```

```
Prelude> :l squares
[1 of 1] Compiling Main             ( squares.hs, interpreted )
Ok, modules loaded: Main.
*Main> take 13 even_squares
[0,4,16,36,64,100,144,196,256,324,400,484,576]
*Main> take 13 odd_squares
[1,9,25,49,81,121,169,225,289,361,441,529,625]
```

- Avoid certain operations such as printing or asking for the length of these lists...

# Programming with infinite lists

- The (infinite) list of all Fibonacci numbers

```
fibs = 0 : 1 : sumlists fibs (tail fibs)
  where sumlists (x:xs) (y:ys) = (x + y) : sumlists xs ys
```

```
Prelude> :l fibs
[1 of 1] Compiling Main             ( fibs.hs, interpreted )
Ok, modules loaded: Main.
*Main> take 15 fibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]
*Main> take 15 (filter odd fibs)
[1,1,3,5,13,21,55,89,233,377,987,1597,4181,6765,17711]
*Main> take 13 (filter even fibs)
[0,2,8,34,144,610,2584,10946,46368,196418,832040,3524578,14930352]
```

- Two more ways of defining the list of Fibonacci numbers using variants of **`map`** and **`zip`**

```
fibs2 = 0 : 1 : map2 (+) fibs2 (tail fibs2)
  where map2 f xs ys = [f x y | (x,y) <- zip xs ys]
-- the version above using a library function
fibs3 = 0 : 1 : zipWith (+) fibs3 (tail fibs3)
```

# Lazy and infinite lists

**`[n..m]`** shorthand for a list of integers from **n** to **m** (inclusive)

**`[n..]`** shorthand for a list of integers from **n** upwards

We can easily define the list of all prime numbers

```
primes = sieve [2..]
  where sieve (p:ns) = p : sieve [n | n <- ns, n `mod` p /= 0]
```

```
Prelude> :l primes
[1 of 1] Compiling Main             ( primes.hs, interpreted )
Ok, modules loaded: Main.
*Main> take 13 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41]
```

# Infinite streams

- A *producer* of an infinite stream of integers:

```
fib  = 0 : fib1
fib1 = 1 : fib2
fib2 = add fib fib1
  where add (x:xs) (y:ys) = (x+y) : add xs ys
```

- A *consumer* of an infinite stream of integers:

```
consumer stream n =
  if n == 1 then show head
  else show head ++ ", " ++ consumer tail (n-1)
    where head:tail = stream
```

```
consumer fib 10 ⇒ ... ⇒ "0, 1, 1, 2, 3, 5, 8, 13, 21, 34"
```

## Drawbacks of lazy evaluation

- More difficult to reason about performance
  - especially about space consumption
- Runtime overhead

**The five symptoms of laziness:**
**1.**

## Side-effects in a pure language

- We really need side-effects in practice!
  - I/O and communication with the outside world (user)
  - exceptions
  - mutable state
  - keep persistent state (on disk)
  - ...

- How can such *imperative* features be incorporated in a *purely functional language*?

## Doing I/O and handling state

- When doing I/O there are some desire properties
  - It should be done. Once.
  - I/O statements should be handled in sequence

- Enter the world of **Monad**s* which
  - encapsulate the state, controlling accesses to it
  - effectively model *computation* (not only sequential)
  - clearly separate pure functional parts from the impure

* A notion and terminology adopted from **category theory**

## The **IO** type class

- **Action**: a special kind of value
  - e.g. reading from a keyboard or writing to a file
  - must be ordered in a well-defined manner for program execution to be meaningful
- **Command**: expression that evaluates to an action
- **`IO T`**: a type of command that yields a value of type **T**
  - `getLine :: IO String`
  - `putStr  :: String -> IO ()`
- Sequencing IO operations (the *bind* operator):

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

current state second action new state

# Example: command sequencing

- First read a string from input, then write a string to output

```haskell
getLine >>= \s -> putStr ("Simon says: " ++ s)
```

- An alternative, more convenient syntax:

```haskell
do s <- getLine
   putStr ("Simon says: " ++ s)
```

- This looks very “imperative”, but all side-effects are controlled via the **IO** type class!
  - **IO** is merely an instance of the more general type class **Monad**

```haskell
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

  - Another application of **Monad** is simulating mutable state

# Example: copy a file

- We will employ the following functions:

```
Prelude> :info writeFile
writeFile :: FilePath -> String -> IO ()    -- Defined in `System.IO’
Prelude> :i FilePath
type FilePath = String                      -- Defined in `GHC.IO’
Prelude>:i readFile
readFile :: FilePath -> IO String           -- Defined in `System.IO’
```

- The call **`readFile "my_file"`** is not a String, and no String value can be extracted from it
- But it can be used as part of a more complex sequence of instructions to compute a String

```
copyFile fromF toF =
  do contents <- readFile fromF
     writeFile toF contents
```

# Monads

- As we saw, Haskell introduces a **do** notation for working with monads, i.e. introduces sequences of computation with an implicit state

```
do expr1; expr2; ...
```

- An “assignment” “expands” to

```
do x <- action1; action2
```

```
action1 >>= \x -> action2
```

- A monad also requires the `return` operation for returning a value (or introducing it into the monad)
- There is also a sequencing operation that does not take care of the value returned from the previous operation

  Can be defined in terms of bind: `x >> y = x >>= (\_ -> y)`

# Modules

- Modularization features provide
  - *encapsulation*
  - *reuse*
  - *abstraction*

  (separation of name spaces and information hiding)
- A module *requires* and *provides* functionality

```
module Calculator (Expr,eval,gui) where
import Math
import Graphics
...
```

- It is possible to export everything by omitting the export list

# Modules: selective export

- We need not export all constructors of a type
- Good for writing ADTs: supports hiding representation

```
module AbsList (AbsList, empty, isempty,
                cons, append, first, rest) where

data AbsList a = Empty
               | Cons a (AbsList a)
               | App (AbsList a) (AbsList a)

empty = Empty
cons x l = Cons x l
append l1 l2 = App l1 l2
...
```

- Here we export only the type and abstract operations

## Modules: import

- We can use **`import`** to use entries from another module

```
module MyMod (...) where
import Racket (cons, null, append)
import qualified Erlang (send, receive, spawn)

foo pid msg queue = Erlang.send pid (cons msg queue)
```

- Unqualified import allows to use exported entries as is
  - + shorter symbols
  - − risk of name collision
  - − not clear which symbols are internal or external
- Qualified import means we need to include module name
  - − longer symbols
  - + no risk of name collision
  - + easy distinction of external symbols

# A better quick sort program

- Recall the **`qsort`** function definition

```
qsort [] = []
qsort (p:xs) = qsort lt ++ [p] ++ qsort ge
    where lt = [x | x <- xs, x < p]
          ge = [x | x <- xs, x >= p]
```

- We can avoid the two traversals of the list by using an appropriate function from the **`List`** library

```
import Data.List (partition)

qsort [] = []
qsort (p:xs) = qsort lt ++ [p] ++ qsort ge
    where (lt,ge) = partition (<p) xs
```

# Exercise: sort a file (with its solution)

- Write a module defining the following function:

```
sortFile :: FilePath -> FilePath -> IO ()
```

- **`sortFile file1 file2`** reads the lines of **`file1`**, sorts them, and writes the result to **`file2`**

- The following functions may come handy

```
lines   :: String -> [String]
unlines :: [String] -> String
```

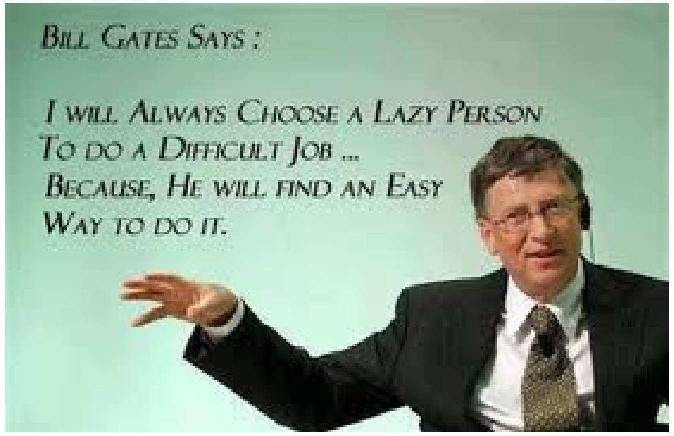
```
module FileSorter (sortFile) where
import Data.List (sort)      -- or use our qsort

sortFile f1 f2 =
  do str <- readFile f1
     writeFile f2 ((unlines . sort . lines) str)
```

## Summary so far

- **Higher-order functions**, **polymorphic functions** and **parameterized types** are useful for building abstractions
- **Type classes** and **modules** are useful mechanisms for structuring programs
- **Lazy evaluation** allows programming with infinite data structures
- Haskell is a **purely** functional language that can avoid redundant and repeated computations
- Using **monads**, we can control side-effects in a purely functional language

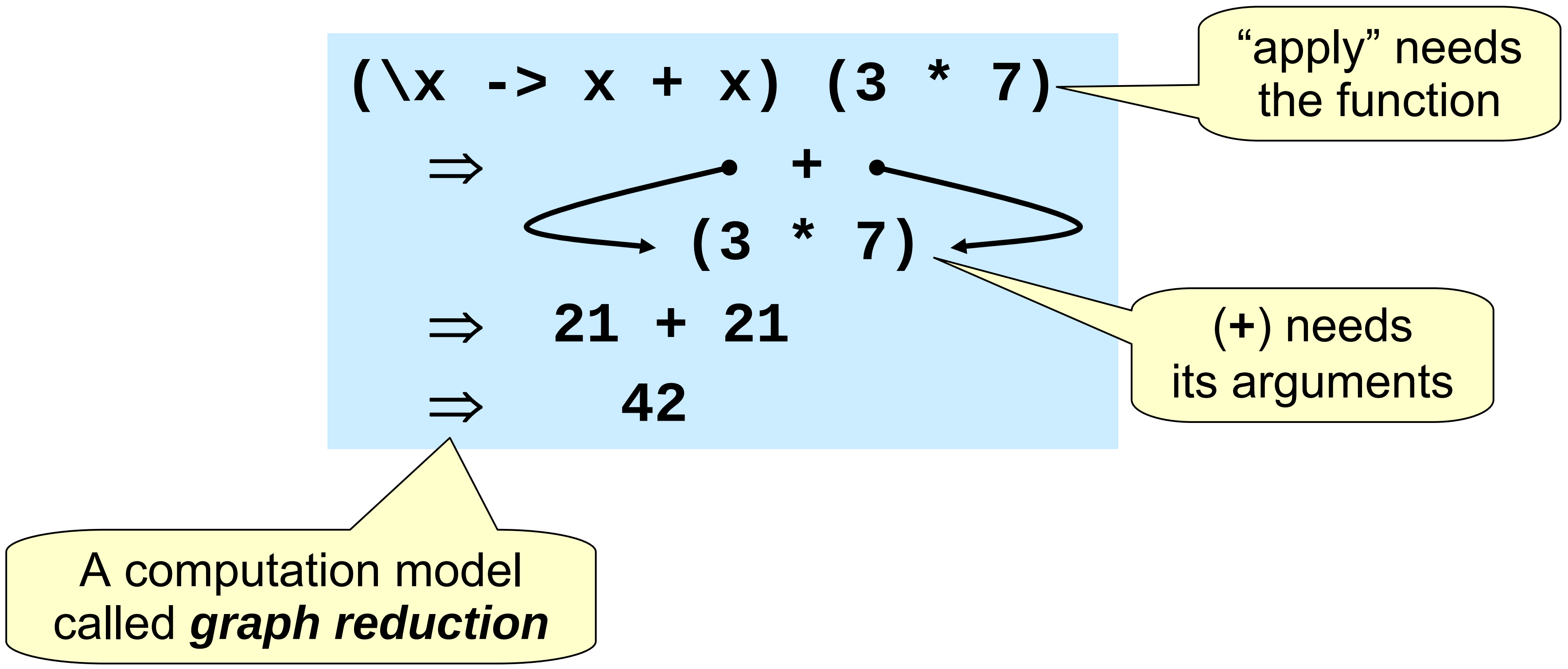
# Haskell: From Basic to Advanced

## Part 3 – A Deeper Look into Laziness


BILL GATES SAYS :
I WILL ALWAYS CHOOSE A LAZY PERSON
TO DO A DIFFICULT JOB ...
BECAUSE, HE WILL FIND AN EASY
WAY TO DO IT.

# Laziness again

- Haskell is a *lazy* language
  - A particular function argument is only evaluated when it is *needed*, and
  - if it is needed then it is evaluated *just once*



```
(\x -> x + x) (3 * 7)
   ⇒        • + •
          (3 * 7)
   ⇒   21 + 21
   ⇒     42
```

"apply" needs the function
(+) needs its arguments
A computation model called ***graph reduction***

# When is a value “needed”?

```
strange :: Bool -> Integer
strange True  = 42
strange False = 42
```

```
Prelude> strange undefined
*** Exception: Prelude.undefined
```

An argument is evaluated when a pattern match occurs

But also primitive functions evaluate their arguments

use **`undefined`** or **`error`** to test if an argument is evaluated

# Lazy programming style

- Clear separation between
  - Where the computation of a value is defined
  - Where the computation of a value happens

We naturally get modularity!

# At most once?

```haskell
fib n = head (drop n fibs)
```

```haskell
foo :: Integer -> Integer
foo n = (fib n)^2 + fib n + 42
```

```
Prelude> foo (6 * 7)
717780702690899544
```

**6 * 7** is evaluated once but **fib 42** is evaluated twice

```haskell
bar :: Integer -> Integer
bar n = foo 42 + n
```

```
Prelude> bar 17 + bar 54
1435561405381799979
```

**foo 42** is evaluated twice

**Quiz:** How to avoid such recomputation?

## At most once!

```haskell
foo :: Integer -> Integer
foo x = t^2 + t + 42
        where t = fib x
```

```haskell
bar :: Integer -> Integer
bar x = foo42 + x

foo42 :: Integer
foo42 = foo 42
```

The compiler might also perform these optimizations with

```
ghc -O
ghc –ffull-laziness
```

# Lazy iteration

```haskell
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

```
Prelude> take 13 (iterate (*2) 1)
[1,2,4,8,16,32,64,128,256,512,1024,2048,4096]
```

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

cycle ::[a] -> [a]
cycle xs = xs ++ cycle xs
```

Define these with **`iterate`**?

```haskell
repeat :: a -> [a]
repeat x = iterate id x

cycle ::[a] -> [a]
cycle xs = concat (repeat xs)
```

# Lazy replication and grouping

```haskell
replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)
```

```
Prelude> replicate 13 42
[42,42,42,42,42,42,42,42,42,42,42,42,42]
```

How to define this?

```haskell
group :: Int -> [a] -> [[a]]
group n =
          takeWhile (not . null)
        . map (take n)
        . iterate (drop n)
```

**.** connects stages like Unix pipe symbol **|**

```
Prelude> group 3 "abracadabra!"
["abr","aca","dab","ra!"]
```

## Lazy IO

- Even IO is done lazily!

```
headFile f = do
  c <- readFile f
  let c’ = unlines . take 5 . lines $ c
  putStrLn c’
```

Does not actually read in the whole file!

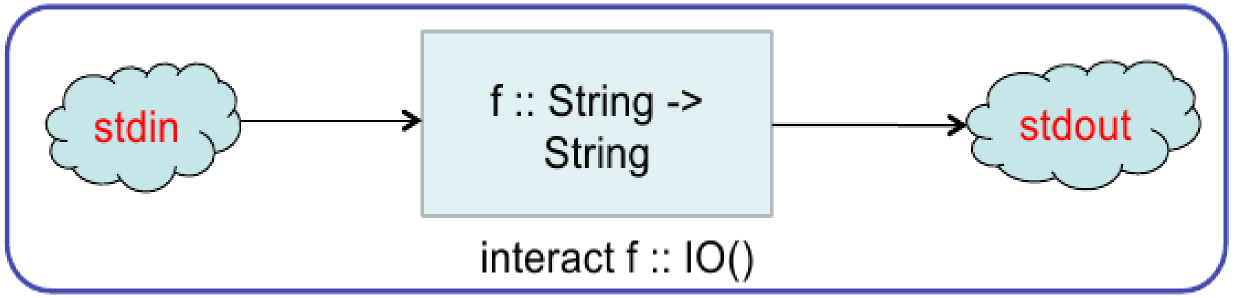
Need to print causes just 5 lines to be read

Aside: we can use names with ’ at their end

# Lazy IO

**Common pattern**: take a function from String to String, connect **stdin** to the input and **stdout** to the output

```
interact :: (String -> String) -> IO ()
```



```
import Network.HTTP.Base (urlEncode)

encodeLines = interact $
  unlines . map urlEncode . lines
```

```
Prelude> encodeLines
hello world
hello%20world
20+22=42
20%2B22%3D42
...
```

## Controlling laziness

- Haskell includes some features to reduce the amount of laziness, allowing us to decide *when* something gets evaluated
- These features can be used for performance tuning, particularly for controlling space usage
- Not recommended to mess with them unless you have to – hard to get right in general!

# Tail recursion

- A function is tail recursive if its last action is a recursive call to itself and that call produces the function's result
- Tail recursion uses no stack space; a tail recursive call can be compiled to an unconditional jump
- Important concept in non-lazy functional programming

- Recall **`foldr`**

```
foldr op init [] = init
foldr op init (x:xs) = x `op` foldr op init xs
```

**foldr op init [x1,x2,...,x42] ⇒**
**(x1 \`op\` (x2 \`op\` ... (x42 \`op\` init) ...**

- The tail recursive "relative" of **`foldr`** is **`foldl`**

```
foldl op init [] = init
foldl op init (x:xs) = foldl op (init `op` x) xs
```

**foldl op init [x1,x2,...,x42] ⇒**
**(...(init \`op\` x1) \`op\` x2) ... \`op\` x42**

# Tail recursion and laziness

- Recall **`sum`**

```
sum = foldr (+) 0
```

```
*Main> let big = 42424242 in sum [1..big]
*** Exception: stack overflow
*Main> let big = 42424242 in foldr (+) 0 [1..big]
*** Exception: stack overflow
```

- OK, we were expecting these, but how about **`foldl`**?

```
*Main> let big = 42424242 in foldl (+) 0 [1..big]
*** Exception: stack overflow
```

- What’s happening!?
- Lazy evaluation is too lazy!

```
foldl (+) 0 [1..big]
⇒  foldl (+) (0+1) [2..big]
⇒  foldl (+) (0+1+2) [3..big]
⇒  ...
```

Not computed until needed;
at the 42424242 recursive call!

# Controlling laziness using **seq**

- Haskell includes a primitive function

```
seq :: a -> b -> b
```

- It evaluates its first argument and returns the second

“strict” is used to mean the opposite of “lazy”

The **`Prelude`** also defines a strict application operation

```
($!) :: (a -> b) -> a -> b
f $1 x = x `seq` (f x)
```

## Strictness

- A tail recursive lists sum function

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

- When compiling with **ghc -O** the compiler looks for arguments which will eventually be needed and will insert \`seq\` calls in appropriate places

```
sum’ :: [Integer] -> Integer
sum’ = s 0
  where s acc [] = acc
        s acc (x:xs) = acc `seq` s (acc+x) xs
```

force **acc** to be simplified on each recursive call

## Strict tail recursion with `foldl’`

```
foldl’ :: (a -> b -> a) -> a -> [b] -> a
foldl’ op init [] = init
foldl’ op init (x:xs) = let a = (init `op` x)
                        in  a `seq` foldl’ op a xs
```

And now

```
*Main> let big = 42424242 in foldl’ (+) 0 [1..big]
899908175849403
```

Or even better, we can use the built-in one

```
*Main> import Data.List (foldl’)
*Main> let big = 42424242 in foldl’ (+) 0 [1..big]
899908175849403
```

# Are we there yet?

- One more example: average of a list of integers

```haskell
average :: [Integer] -> Integer
average xs = sum′ xs `div` fromIntegral (length xs)
```

needed due to the types of **sum′** and **length**

```
*Sum> let big = 42424242 in length [1..big]
42424242
*Sum> let big = 42424242 in sum′ [1..big]
899908175849403
*Sum> let big = 42424242 in average [1..big]
21212121
```

- Seems to work, doesn't it? Let's see:

```
*Sum> let bigger = 424242420 in length [1..bigger]
424242420
*Sum> let bigger = 424242420 in sum′ [1..bigger]
89990815675849410
*Sum> let bigger = 424242420 in average [1..bigger]
... CRASHES THE MACHINE DUE TO THRASHING!
```

WTF?

# Space leaks

- Making **`sum`** and **`length`** tail recursive and strict does not solve the problem ☹

- This problem is often called a space leak
  - **`sum`** forces us to build the whole **`[1..bigger]`** list
  - laziness (“at most once”) requires us to keep the list in memory since it is going to be used by **`length`**
  - when we compute either the length or the sum, as we go along, the part of the list that we have traversed so far is reclaimed by the garbage collector

# Fixing the space leak

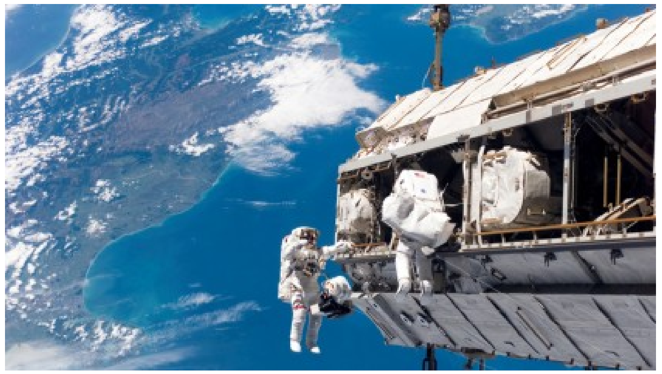
- The problem can be solved by making average tail recursive by computing sum and length at the same time

call to **`fromIntegral`** not needed anymore

```
average’ :: [Integer] -> Integer
average’ xs = av 0 0 xs where
  av sm len []       = sm `div` len
  av sm len (x:xs) = sm   `seq`
                     len `seq`
                     av (sm + x) (len + 1) xs
```

```
*Sum> let bigger = 424242420 in average [1..bigger]
212121210
```

fixing a space leak



# Gotcha: **seq** is still quite lazy!

- **seq** forces evaluation of its first argument, but *only as far as the first constructor!*

```
Prelude> undefined `seq` 42
*** Exception: Prelude.undefined
Prelude> (undefined,17) `seq` 42
42
```

the pair is already "evaluated", so a **seq** here would have no effect

```
sumlength = foldl' f (0,0)
  where f (s,l) a = (s+a,l+1)
```

```
sumlength = foldl' f (0,0)
  where f (s,l) a = let (s',l') = (s+a,l+1)
                    in  s' `seq` l' `seq` (s',l')
```

force the evaluation of components *before* the pair is constructed

# Laziness and IO

```
count :: FilePath -> IO Int
count f = do contents <- readFile f
             let n = read contents
             writeFile f (show (n+1))
             return n
```

for the time being this will do

**readFile** is not computed until it is needed

```
Prelude> count "some_file"
*** Exception: some_file: openFile: resource busy (file is locked)
```

- We sometimes need to control lazy IO
  - Here the problem is easy to fix (see below)
  - Some other times, we need to work at the level of file handles

```
count :: (Num b,Show b,Read b) => FilePath -> IO b
count f = do contents <- readFile f
             let n = read contents
             n `seq` writeFile f (show (n+1))
             return n
```

# Some lazy remarks

- Laziness
  - – Evaluation happens on demand and “at most once”
  - + Can make programs more “modular”
  - + Very powerful tool when used right
  - − Different programming style / approach
- We do not have to employ it everywhere!
- Some performance implications are very tricky
  - – Evaluation can be controlled by tail recursion and seq
  - – Best avoid their use when not really necessary