Haskell Introduction

Jiang Wu

Install

- ✓ apt-get install haskell-platform
- ✓ latest version: 2013.2.0.0

Hello, world?

```haskell
-- ghc --make main.hs
{-
Multiple line comments
-}

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

Hello, world!

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
      | otherwise = undefined
```

GHCi

- ✓ $ ghci
- ✓ some utilities: :m, :t, :i

Why Haskell Matters

- ✓ Type System
- ✓ High Order Function
- ✓ Lazy Evaluation

Basic Types (1)

Bool

True, False

Char

'a', 'A', '3', '\t'

String

"abc", "1+2=3", ""

Basic Types (2)

Int

fixed-precision integers

Integer

arbitrary-precision integers

Float

single precision floating-point number

List Types

```
[False, True, False] :: [Bool]
['a', 'b', 'c', 'd'] :: [Char]
["One", "Two", "Three"] :: [String]
```

Tuple Types

```
(False, True) :: (Bool, Bool)
(False, 'a', True) :: (Bool, Char, Bool)
("Yes", True, 'a') :: (String, Bool, Char)
```

Function Types

```
add' :: Int -> Int -> Int
add' x y = x + y

add1 = add' 1
add1 :: Int -> Int
```

Polimorphic types

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

length' = foldr add1 0
```

Overloaded types

```
(+) :: Num a => a -> a -> a

sort :: Ord a => [a] -> [a]

sortTree :: Ord a => Tree a -> Tree a
```

Algebra Data (1)

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

Algebra Data (2)

```haskell
data Person = Person {
  name :: String,
  age :: Int
}

Person :: String -> Int -> Person
```

Algebra Data (3)

```
data Maybe a = Just a | Nothing

data Either a b = Left a | Right b
```

Class Eq

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

  x /= y                   = not (x == y)
  x == y                   = not (x /= y)
```

Class Ord

```
class  (Eq a) => Ord a  where
  compare                 :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min                :: a -> a -> a

  compare x y = if x == y then EQ
                else if x <= y then LT
                else GT
```

High Order Function(1)

```haskell
data List a = Nil | Cons a (List a)

sum Nil = 0
sum (Cons n list) = n + sum list

product Nil = 1
product (Cons n list) = n * product list
```

High Order Function(2)

```haskell
sum :: (Num a) => [a] -> a
product :: (Num a) => [a] -> a

fold :: (a -> b -> b) -> b -> [a] -> b
fold f e Nil = e
fold f e (Cons n list) = f n (fold f e list)

sum = fold (+) 0
product = fold (*) 1
```

Lazy Evaluation(1)

```haskell
{-
  No matter you belive or not,
  I believe.
-}

const :: a -> b -> a
uBelieve :: Bool

-- uBelieve won't be computed
iBelieve = const True uBelieve
```

Lazy Evaluation(2)

```haskell
neturalNumbers = [1..]
positiveOdds = [1,3..]
positiveEvens = [2,4..]

-- List Comprehension
squares = [ x*x | x <- [1..] ]
```

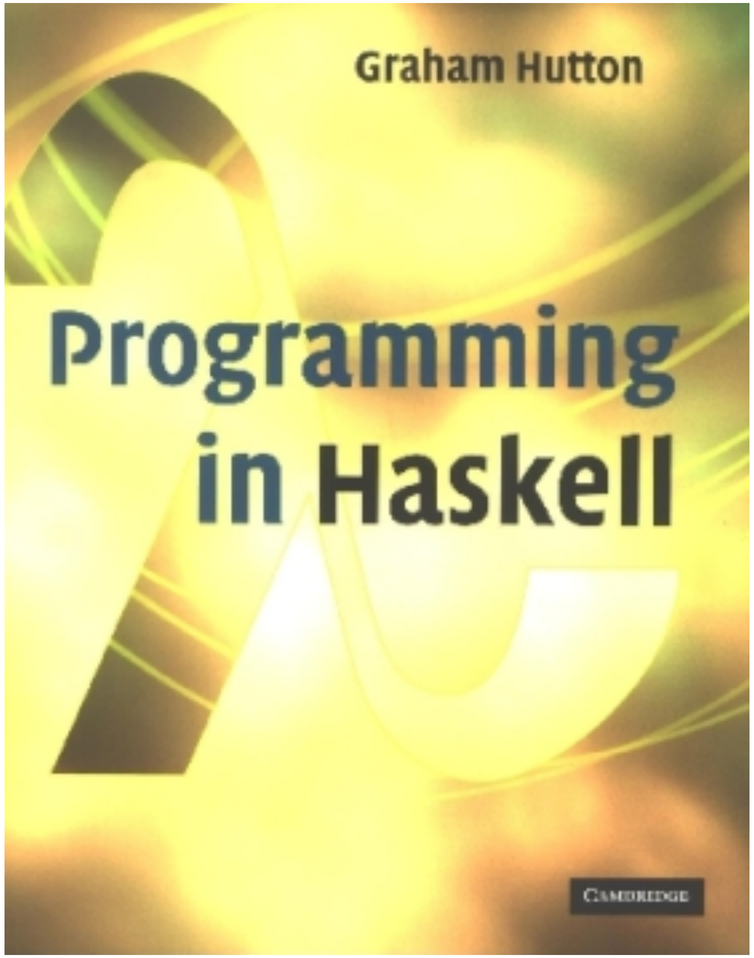
Lazy Evaluation(3)

```haskell
-- Newton's method for finding roots

squareRoots :: Double -> [Double]
squareRoots x | x >=0 = squareRoots' 1 x


squareRoots' :: Double -> Double -> [Double]
squareRoots' r n = let r' = (r + n/r)/2 in
                       r' : squareRoots' r' n

takeWhile (\x -> abs(x*x-10)>1e-10 ) $ squareRoots 10
```

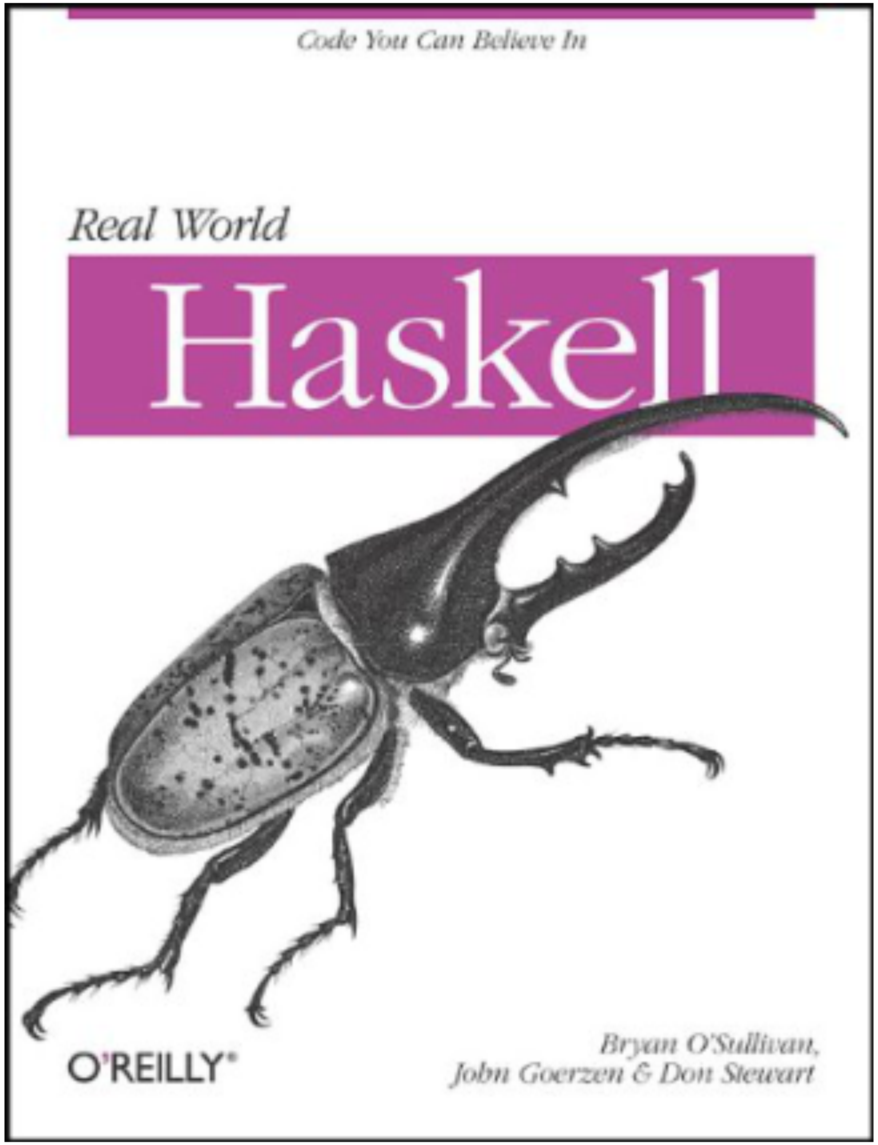
Conclusion

- ✓ Functional
- ✓ Lazy evaluation
- ✓ Type safety

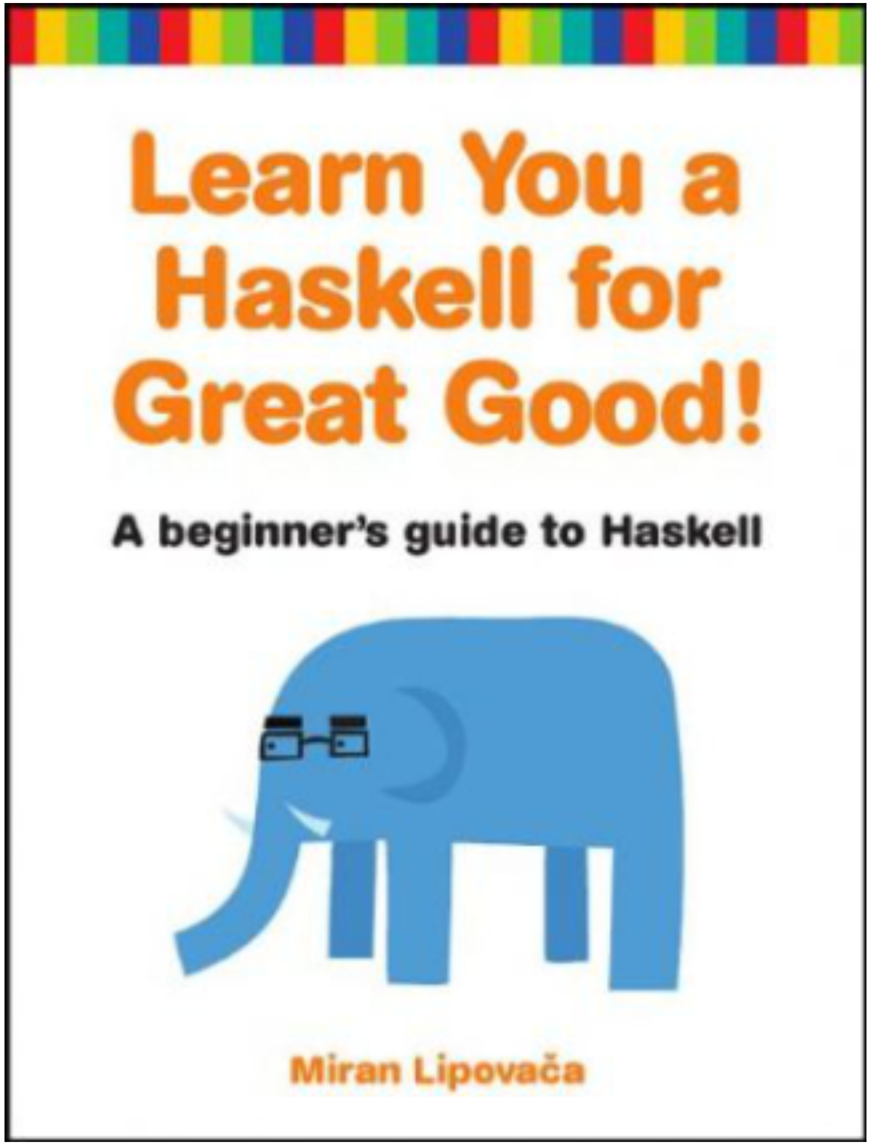
Programming in Haskell



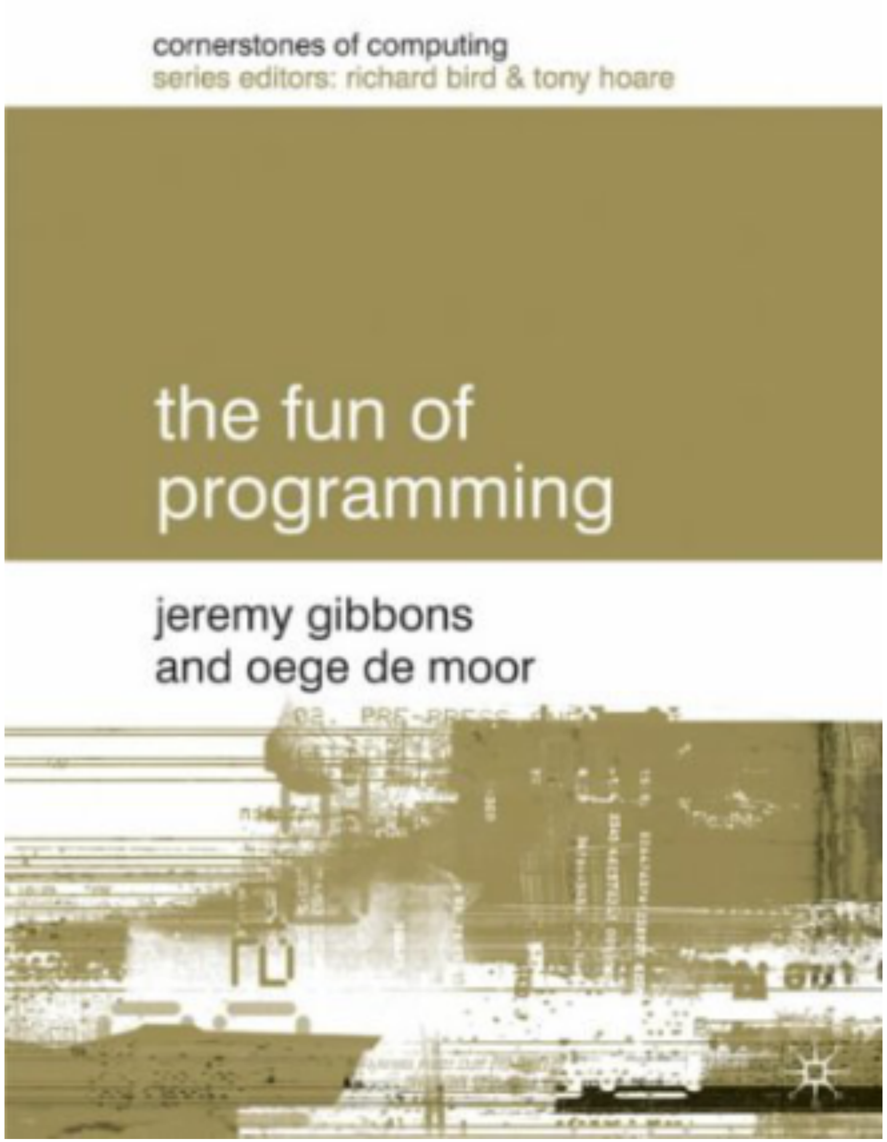
Real World Haskell



Learn You a Haskell



Haskell is Fun!



Thanks!

Questions?