



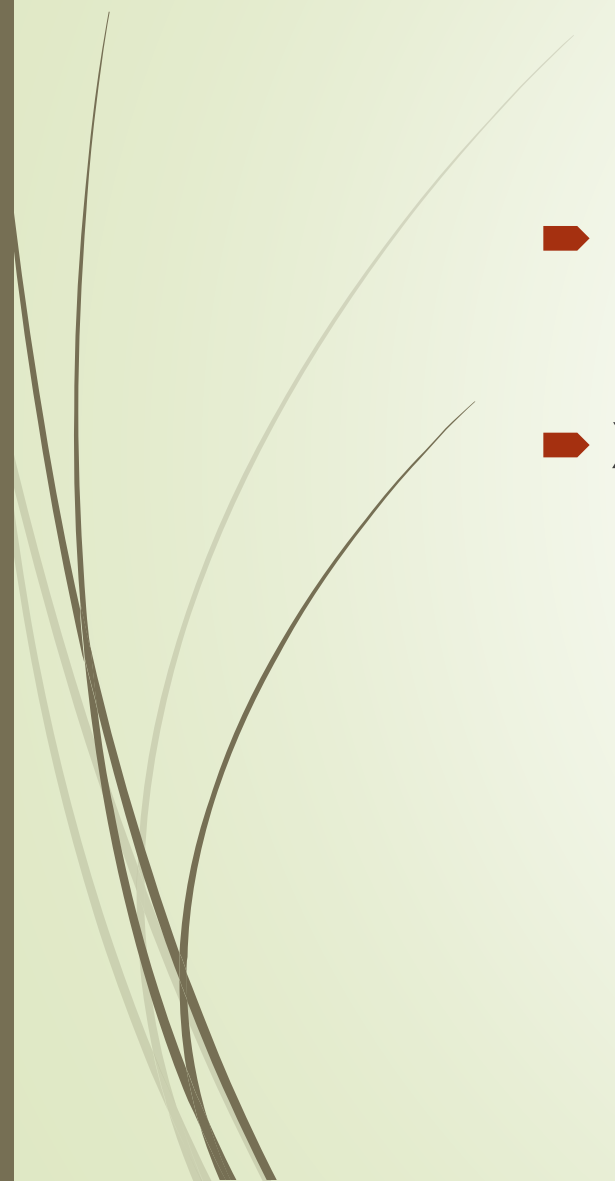
# Haskell and the Software Industry

Νίκος Μαυρογεώργης

03113087



# A two-part presentation

- Εισαγωγή στη Haskell και τα χαρακτηριστικά της
  - Χρήση της στην Τεχνολογία Λογισμικού
- 



# Εισαγωγή - Βασικά χαρακτηριστικά

- Η Haskell είναι μία “αμιγώς” (pure) συναρτησιακή γλώσσα.
- Δεν λέμε στον υπολογιστή τι να κάνει, με τη μορφή ακολουθίας εντολών (if, for, while κλπ.).
- Αντίθετα, **περιγράφουμε** στον υπολογιστή τι είναι το κάθετι, με τη μορφή **συνάρτησης**.
- Δεν υπάρχουν side effects (i++ κλπ.).
- Οι συναρτήσεις επιστρέφουν πάντα το ίδιο αποτέλεσμα, όταν κληθούν με τα ίδια ορίσματα.

# Εισαγωγή - Βασικά χαρακτηριστικά

- Η Haskell είναι **lazy**.
- Δεν υπολογίζει τίποτα, αν δεν χρειαστεί.
- Παράδειγμα: Έστω η λίστα `xs = [1, 2, 3]` και η συνάρτηση `doubleMe` που παίρνει μία λίστα και διπλασιάζει τα στοιχεία της.
- Τι θα συμβεί με την κλήση `doubleMe (doubleMe xs)` ;

# Εισαγωγή - Βασικά χαρακτηριστικά

- Σε μία διαδικασιακή γλώσσα, η λίστα θα περνούσε ως παράμετρος στη συνάρτηση `doubleMe` και αποτέλεσμα (η λίστα `[2, 4, 6]`, θα περνούσε επίσης ως παράμετρος στην εξωτερική συνάρτηση `doubleMe`.
- Σε μία *lazy* γλώσσα, αν καλέσουμε τη συνάρτηση, χωρίς να την αναγκάσουμε να δείξει το αποτέλεσμά της, θα μας πει: "Yeah yeah, I'll do it later!"
- Αν την αναγκάσουμε να δείξει το αποτέλεσμά της, τότε η **εξωτερικότερη `doubleMe`**, θα ζητήσει το αποτέλεσμα από την εσωτερική. Και μετά η εσωτερική θα αναγκαστεί να το υπολογίσει (`[2, 4, 6]`) και να το στείλει στην εξωτερική, η οποία θα επιστρέψει `[4, 8, 12]`.

# Εισαγωγή - Βασικά χαρακτηριστικά

- Τέλος, η Haskell είναι **statically typed**, έχει δηλαδή αυστηρό σύστημα τύπων (σε αντίθεση με τη C για παράδειγμα).
- Δεν υπάρχουν εντολές όπως `c = c + 'a'` (στη C το `'a'` γίνεται 97 μέσω type conversion). Οι τύποι πρέπει να συμφωνούν οπωσδήποτε κατά τη μεταγλώττιση.
- Οποιοδήποτε λάθος σε τύπους εντοπίζεται στο compile-time.

# Εισαγωγή - Λίστες

- Υποστηρίζει εκτεταμένη διαχείριση λιστών.

- Ranges όπως `[1..10]` == `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

- List comprehension.

```
[x*2 | x <- [1..10], x*2 >= 12]
```

- Άπειρες λίστες όπως η `[1 2 ..]` (Lazy!)

- Και πολλά άλλα π.χ. Η `repeat n` επαναλαμβάνει επ'άπειρον το `n`, και η `take k` παίρνει τα πρώτα `k` στοιχεία.

```
take 10 (repeat 5) == [5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```



# Εισαγωγή - Τύποι

- Όμοιοι τύποι με τις συναρτησιακές γλώσσες όπως ML κλπ.
- Int, Char, Bool, Float, Double κλπ.
- `[t]` : σημαίνει λίστα από αντικείμενα τύπου t
- `(t1, t2)` : tuple από δύο ξεχωριστούς τύπους
- Integral : Οσωδήποτε μεγάλος ακέραιος (Lazy!)
- `t1 -> t2` : Συνάρτηση από t1 σε t2
- Παραμετρικοί τύποι: `head :: [a] -> a`



# Εισαγωγή - Type classes

- Τα **Typeclasses** μοιάζουν λίγο με τα interfaces σε μία γλώσσα όπως η Java, στο ότι ορίζουν μία συμπεριφορά (όπως σύγκριση για ισότητα, σύγκριση για διάταξη κλπ).
- Αν ένας τύπος είναι instance ενός type class, τότε μπορούμε να χρησιμοποιήσουμε τις “μεθόδους” του type class σε αυτόν τον τύπο.

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

- Ορίζεται ο τύπος των τελεστών (“μεθόδων”), και ακολουθεί η υλοποίησή τους.
- Ο τύπος Int ανήκει στο Eq type class, οπότε μπορούμε να χρησιμοποιήσουμε για παράδειγμα τον τελεστή `==` μεταξύ δύο ακεραίων.

# Εισαγωγή - Custom types

- Μπορούμε να ορίσουμε τους δικούς μας τύπους..

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

```
data Int = -2147483648 | -2147483647 | ... | -1 | 0 | 1 |  
          ... | 2147483647
```

- ... οι οποίοι μπορούν να είναι και παραμετρικοί

```
data Shape = Circle Float Float Float  
           | Rectangle Float Float Float Float
```

# Εισαγωγή - Συναρτήσεις

- ▶ Πρέπει να ορίζεται ρητά ο τύπος τους (statically typed!)

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

- ▶ Pattern matching
- ▶ Let expressions: Ορίζουμε ένα όνομα στο let το οποίο είναι διαθέσιμο στο in.
- ▶ Τι επιστρέφει το παρακάτω?

```
4 * (let a = 9 in a + 1) + 2
```

- ▶ Anonymous functions – Lambda abstractions

```
\x -> x + 1
```

# Εισαγωγή - Συναρτήσεις Υψηλότερης Τάξης

- Κάθε συνάρτηση παίρνει μόνο ένα όρισμα. Τότε πώς μπορούμε να τις δώσουμε παραπάνω?
- Πρώτη λύση: tuples
- Δεύτερη (πονηρή) λύση: **Curried** functions!

```
multTwo :: (Num a) => a -> a -> a  
multTwo x y = x * y
```

- Η `multTwo` είναι μία συνάρτηση που παίρνει έναν αριθμό και επιστρέφει μία συνάρτηση που παίρνει έναν αριθμό και επιστρέφει έναν αριθμό

# Εισαγωγή - Monads

- Πώς μπορώ σε μία καθαρά συναρτησιακή γλώσσα να έχω I/O?
- Λύση: **Monads**
- Το Monad είναι ένα type class, το οποίο μας επιτρέπει να συνδέσουμε επιμέρους λειτουργίες (συναρτήσεις), με κάποιο τρόπο χρήσιμο για εμάς.
- Μαγικό συσταστικό: τελεστής **bind**. Τύπος:

```
m a -> (a -> m b) -> m b
```

# Εισαγωγή - Monads

- Το `bind` μπορεί να εκτελέσει συναρτήσεις διαδοχικά, με ό,τι σειρά θέλει:

```
putStrLn "What is your name?"
```

```
>>= (\_ -> getLine)
```

```
>>= (\name -> putStrLn ("Welcome, " ++ name ++ "!" ))
```

- Τύπωσε το `string`, μετά πάρε το `string` από το πληκτρολόγιο και τύπωσε το στην είσοδο.
- Syntactic sugar για πιο όμορφη και εύκολη χρήση του `bind`:

```
do
```

```
    putStrLn "What is your name?"
```

```
    name <- getLine
```


```
    putStrLn ("Welcome, " ++ name ++ "!" )
```



# Σκέψεις


- Το Monad είναι μία αρκετά σύνθετη έννοια (ιδιαίτερα σε σχέση με αυτό που περιγράψαμε)
- Ωστόσο είναι από τα πιο θεμελιώδη στοιχεία της Haskell, καθώς ακόμα και ένα απλό `print` απαιτεί τη χρήση του.
- Είναι από τα πρώτα πράγματα στο οποίο σκονταύτουν οι προγραμματιστές αρκετά νωρίς κατά την εκμάθηση της γλώσσας.





# Ερώτηση

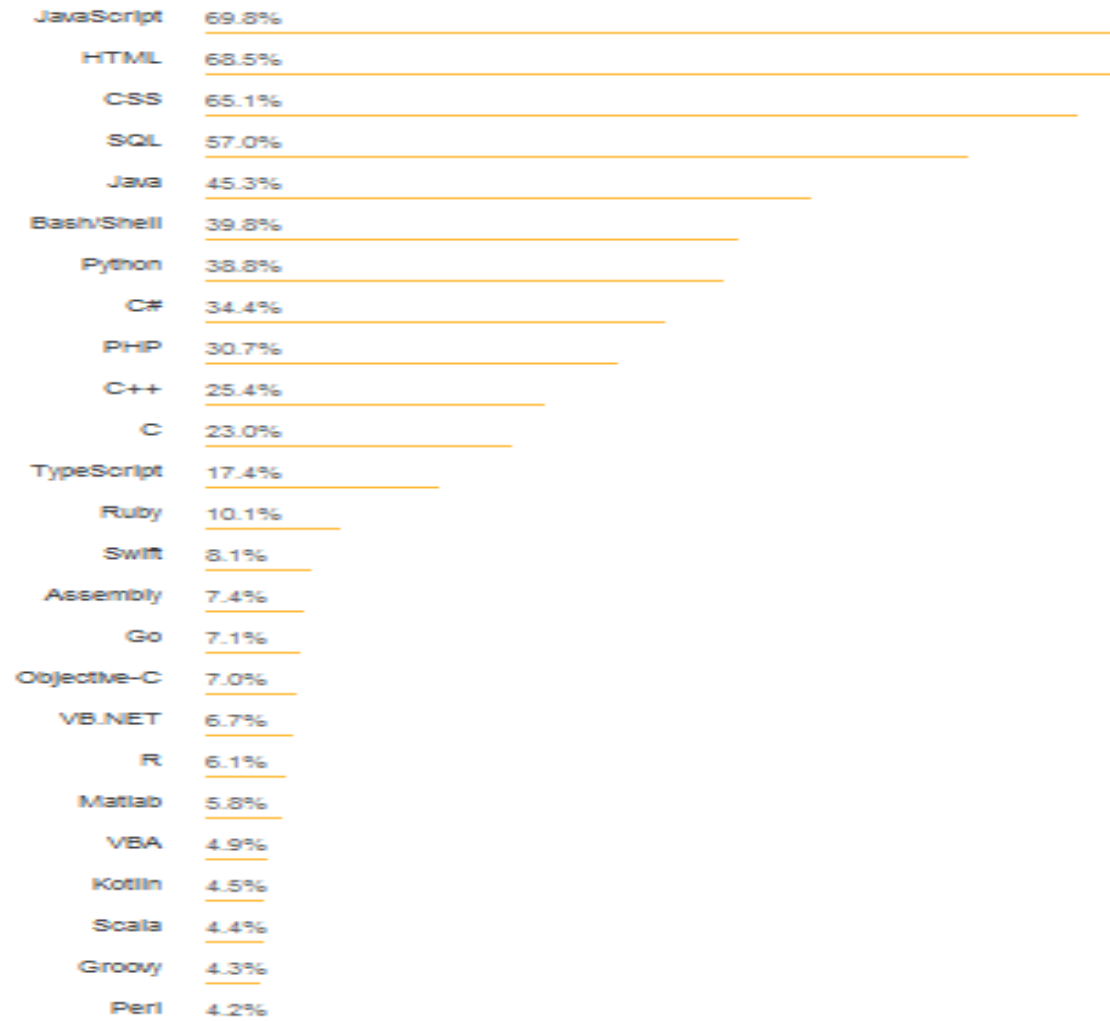
- Πώς γίνεται να γράψει κάποιος κάποιο μεγάλο και ολοκληρωμένο λογισμικό σε Haskell, όταν κάτι τόσο απλό όπως το I/O δείχνει να είναι “περίεργο”?



# Ερώτηση

- Πώς γίνεται να γράψει κάποιος κάποιο μεγάλο και ολοκληρωμένο λογισμικό σε Haskell, όταν κάτι τόσο απλό όπως το I/O δείχνει να είναι “περίεργο”?
- Μια επόμενη ερώτηση θα ήταν: κατά πόσο χρησιμοποιείται η συγκεκριμένη γλώσσα στην τεχνολογία λογισμικού για την παραγωγή μεγάλων έργων?

# Most popular Technologies



78,334 responses; select all that apply

➤ <https://insights.stackoverflow.com/survey/2018#most-popular-technologies>



# Εφαρμογή στην Τεχνολογία Λογισμικού

- Τα υπέρ και τα κατά της γλώσσας προκύπτουν από τα τρία βασικά χαρακτηριστικά της που εξετάσαμε στην αρχή: **pure functional, lazy, strict type system**



# Εφαρμογή στην Τεχνολογία Λογισμικού

## The Case Against Haskell

- High learning curve
- Ιδιαίτερα για κάποιον που δεν έχει ξαναπρογραμματίσει σε συναρτησιακή γλώσσα ή σε γλώσσα με αυστηρό σύστημα τύπων.
- Το laziness σημαίνει ότι οι τιμές δεν υπολογίζονται μέχρι να χρειαστούν. Έτσι η ανάλυση της χρήσης μνήμης δεν είναι πάντα προφανής (unintended laziness).
- Περιορισμένο community support σε σχέση με γλώσσες όπως η JavaScript ή η Python.



# Εφαρμογή στην Τεχνολογία Λογισμικού

## The Case Against Haskell

- Όπως είδαμε η Haskell δεν ανήκει στις δημοφιλείς γλώσσες του software industry (ωστόσο υπάρχει ένα up-to-date [repo](#) με εταιρείες που τη χρησιμοποιούν)
- Λίγα παραδείγματα από success-stories σε μεγάλα site όπως Facebook-PHP, Youtube-Python, Twitter-Ruby on Rails κλπ
- Είναι λίγο πιο counter-intuitive σε σχέση με τον imperative προγραμματισμό και την object-oriented λογική.



# Εφαρμογή στην Τεχνολογία Λογισμικού

## The Case For Haskell

- Εξαιτίας του static type system, πολλά bugs μπορούν να εντοπιστούν από το compile time.
- Ευκολότερο refactoring: Αν γίνουν λάθη με μεταβλητές που βγαίνουν εκτός scope ή interfaces που παραβιάζονται, εντοπίζονται νωρίς.
- Ασφαλέστερες αλλαγές στο codebase σε global επίπεδο.





# Εφαρμογή στην Τεχνολογία Λογισμικού

## The Case For Haskell

- Ευκολότερο testing
- Στη γλώσσα αυτή πιο πολύ περιγράφουμε ιδιότητες που πρέπει να έχει μια δομή, παρά υλοποιούμε.
- Introduce the invariant at the type level.
- Με σωστή περιγραφή της δομής, είμαστε βέβαιοι ότι το επόμενο δέντρο θα είναι δυαδικό. Ένα bug που αφορά τη δομή δύσκολα θα ξεφύγει από το compile time.

# Εφαρμογή στην Τεχνολογία Λογισμικού

## The Case For Haskell

```
data BinaryTree a = EmptyTree | Node a (BinaryTree a)  
                  (BinaryTree a)
```

```
deriving (Show)
```

```
preorder :: Ord a => BinaryTree a -> [a]
```

```
preorder EmptyTree = []
```

```
preorder (Node a left right) = [a] ++ preorder left ++  
preorder right
```



# Εφαρμογή στην Τεχνολογία Λογισμικού

## The Case For Haskell

- Στο παράδειγμα πρέπει πάντα να κοιτάμε την περίπτωση του `EmptyTree`, όταν υλοποιούμε μία συνάρτηση. Αλλιώς, ο compiler θα μήνυμα σφάλματος.
- Αναγκαζόμαστε να ελέγξουμε όλες τις περιπτώσεις, οπότε ελαττώνονται τα bugs



# Εφαρμογή στην Τεχνολογία Λογισμικού

## The Case For Haskell

- Code reuse
- Στη C ή στη Java πρέπει να κοιτάμε διαρκώς για null pointers πχ στο διάβασμα αρχείου (ότι πέτυχε δηλαδή).
- Στη Haskell υπάρχουν έτοιμες μοντελοποιήσεις, που καθιστούν ομοιόμορφους αυτούς τους ελέγχους

# Εφαρμογή στην Τεχνολογία Λογισμικού

## The Case For Haskell

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

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
fmap f (Just x) = Just (f x)
```

```
fmap f Nothing = Nothing
```

```
fmap square (Just 2)
```

```
-- Just 4
```

```
fmap square Nothing
```

```
-- Nothing
```

- Η `fmap` μπορεί να χρησιμοποιηθεί με οποιαδήποτε άλλη συνάρτηση αντί για τη `square`



# Εφαρμογή στην Τεχνολογία Λογισμικού Συμπεράσματα

- Παρόλο που η γλώσσα είναι πάρα πολύ sophisticated και έχει το δικό της πυρήνα οπαδών, δεν χρησιμοποιείται στη βιομηχανία λογισμικού όπως άλλες.
- Πρώτος λόγος: η μη υιοθέτησή της από πολλές και μεγάλες εταιρείες.
- Δεύτερος λόγος(που βρίσκεται σε αλληλεπίδραση με τον πρώτο): για να γίνει μία γλώσσα ευρέως αποδεκτή από το community των προγραμματιστών, πρέπει να πείσει τον developer πως κάτι το οποίο αυτός κάνει συχνά στον κώδικα σε μία άλλη γλώσσα, γίνεται πολύ πιο εύκολα σε αυτήν



# Εφαρμογή στην Τεχνολογία Λογισμικού

## Πιθανές λύσεις

- Βελτίωση των βιβλιοθηκών. Οι εταιρείες είναι συνήθως της άποψης

*“it's not a differentiator, so we don't want to develop it in-house”*

- Περισσότερα εργαλεία (IDEs κλπ)
- Βελτιωμένο documentation
- Καλύτερο package management





## Further reading

- Lipovaca, M. (2011). **Learn You a Haskell for Great Good!: A Beginner's Guide**. No Starch Press
  - <https://www.reddit.com/r/haskell/>
  - [https://wiki.haskell.org/Haskell\\_in\\_industry](https://wiki.haskell.org/Haskell_in_industry)
  - <https://insights.stackoverflow.com/survey/2018>
- 