

# Σχεδίαση και Υλοποίηση της Γλώσσας Προγραμματισμού lambda-cases

Δημήτρης Σαριδάκης Μπίτος

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών  
Εθνικό Μετσόβιο Πολυτεχνείο

8 Ιουλίου 2024

# Μ' αρέσει πολύ η Haskell

Τα πάντα είναι τιμές και έχουν κάποιο τύπο:

- ▶ Σταθερές
- ▶ Συναρτήσεις
- ▶ Είσοδος/Έξοδος
  - ▶ Μπορούν να είναι ορίσματα συναρτήσεων, στοιχεία λίστας κτλ

Οι τύποι τα λένε όλα.

Βοηθητικός μεταγλωττιστής:

- ▶ Μεταγλωττίζεται; Δουλεύει! (Συνήθως)
- ▶ Δεν μεταγλωττίζεται; Οι τάδε τύποι δεν ταιριάζουν.

Γιατί να γράψω κώδικα σε άλλη γλώσσα;

Γιατί δεν είναι η πιο διαδεδομένη γλώσσα;

Building εργαλεία όχι τόσο καλά:

- ▶ Φαίνεται να υπάρχει βελτίωση (από όσο λένε online)
- ▶ Δεν αφορά την διπλωματική

Δύσκολη στην εκμάθηση για αρχάριο. Ίσως παίζουν ρόλο:

- ▶ Όχι πολύ περιγραφικές λέξεις κλειδιά
- ▶ Όχι πολύ περιγραφικά ονόματα βασικών συναρτήσεων
- ▶ Γραμματική λάμδα λογισμού

## Τι θα άλλαζα για μένα;

Μπορούν να συμπτυχθούν κομμάτια που γράφω πολύ συχνά;

- ▶ Ορισμοί Τιμών
- ▶ LambdaCase extension

Μπορούν να αλλάξουν κομμάτια ώστε να μοιάζουν περισσότερο στα αντίστοιχα άλλων γλωσσών όπου είναι πιο κατανοητά;

- ▶ Τελεία για attributes/members/fields
- ▶ Εφαρμογή συνάρτησης με ορίσματα σε παρένθεση

Υπάρχει κάτι καινούργιο που θα μπορούσα να προσθέσω;

- ▶ Ορίσματα στην αρχή ή στην μέση του ονόματος
- ▶ Ανώνυμες Παράμετροι
- ▶ Τύποι Δύναμης

## Εφαρμογή Συνάρτησης με Παρενθέσεις

Haskell	lcases
<pre>f x g x y z putStrLn "Hello World!"</pre>	<pre>f(x) g(x, y, z) print("Hello World!")</pre>

Ορίσματα πριν ή στην μέση:

<pre>show x mod x y map f l</pre>	<pre>(x)to_string (x)mod(y) apply(f)to_all_in(l)</pre>
---	--

## Ανώνυμες Παράμετροι

Σε οποιαδήποτε συνάρτηση μπορούν να λείπουν οποιαδήποτε από τα ορίσματα: κάτω παύλα.

Τα υπόλοιπα είναι παράμετροι.

Νέα συνάρτηση με είσοδο τα κενά ορίσματα.

$f(\_, 1.618, 42)$	$x \Rightarrow f(x, 1.618, 42)$
$f(3.14, \_, 42)$	$x \Rightarrow f(3.14, x, 42)$
$f(\_, 1.618, \_)$	$(x, y) \Rightarrow f(x, 1.618, y)$

## Ανώνυμες Παράμετροι

```
greetings : ListOf(String)s  
  = ["hey!", "hello!", "hi!"]
```

```
length_of(_) : String => Int
```

```
apply(_)to_all_in(  
  : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
```

```
apply(length_of(_))to_all_in(  
  : ListOf(String)s => ListOf(Int)s
```

```
apply(_)to_all_in(greetings)  
  : (String => T1) => ListOf(T1)s
```

```
>>> apply(length_of(_))to_all_in(greetings)  
  = [4, 6, 3]
```

## Το ίδιο σε Haskell

```
greetings :: [String]
greetings = ["hey!", "hello!", "hi!"]

length :: String => Int

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

map length :: [String] -> [Int]

flip map :: [a] -> (a -> b) -> [b]

flip map greetings :: (String -> a) -> [a]

>>> map length greetings
= [4, 6, 3]
```



## Ανώνυμες Παράμετροι: tuples και λίστες

Αντίστοιχα μπορούμε να αφήσουμε κενά στοιχεία tuple ή λίστας.

Νέα συνάρτηση με είσοδο τα κενά στοιχεία.

```
(42, _) : T1 => Int x T1
```

```
(_, 3.14, _) : T1 x T2 => T1 x Real x T2
```

```
[42, _] : Int => ListOf(Int)s
```

```
[_, 3.14, _] : Real^2 => ListOf(Real)s
```

Αντίστοιχα σε εκφράσεις τελεστών: παρακάτω.

## Ορισμοί tuple\_type και postfix functions

tuple\_type αντίστοιχα:

- ▶ structs σε C
- ▶ classes σε OOP: μόνο attributes
- ▶ records σε Haskell

Δημιουργείται αυτόματα ένα postfix function για κάθε field:

- ▶ Κατευθείαν με όρισμα: `some_person.last_name`
- ▶ Συνάρτηση Μόνη της: `_.last_name`

## Ορισμοί tuple\_type και postfix functions

```
tuple_type Name
value (first_name, last_name) : String^2

awesome_guy: Name
  = ("Leonhard", "Euler")

>>> awesome_guy.last_name
  = "Euler"

>>> _.last_name
  : Name => String

>>> apply(_.last_name)to_all_in(_)
  : ListOf(Name)s => ListOf(String)s
```

## postfix functions για tuples που έχουν τύπο γινόμενο

"\_.1st", "\_.2nd", "\_.3rd", ... για tuples που έχουν τύπο γινόμενο.

```
tuple : Real x String  
      = (1.618, "golden ratio")
```

```
origin : Real^3  
       = (0, 0, 0)
```

```
>>> tuple.2nd  
     = "golden ratio"
```

```
>>> origin.2nd  
     = 0
```

## ".change" postfix function

Συνάρτηση αλλαγής στοιχείων tuple

```
state.change{counter = counter + 1}  
point.change{z = 2.718}
```

```
tuples : ListOf(Int^2)s  
  = [(1, 2), (3, 4), (5, 6)]  
>>> apply(_.change{1st = 1st + 1})to_all_in(tuples)  
  = [(2, 2), (4, 4), (6, 6)]
```

```
name : Name  
  = ("Jacob", "Bernoulli")  
change_first_name_to(_) : String => Name  
  = name.change{first_name = _}  
>>> change_first_name_to("Daniel")  
  = ("Daniel", "Bernoulli")
```

## Ορισμοί `or_type` και `prefix functions`

`or_type` αντίστοιχα:

- ▶ C, C++ κτλ: `enum types` αλλά πιο γενικά
- ▶ Haskell: τα πάντα είναι `data`, συμπεριλαμβανομένων των `tuple types`, `or types`

Οι τιμές τους χωρίζονται σε περιπτώσεις (με ή χωρίς εσωτερικές τιμές).

Δημιουργείται αυτόματα ένα `prefix function` για κάθε περίπτωση με εσωτερική τιμή:

- ▶ Κατευθείαν με όρισμα: `the_value:1`
- ▶ Συνάρτηση μόνη της: `the_value:_`

Pattern matching με συνάρτηση `"cases"`.

## Ορισμοί or\_type και prefix functions

```
or_type Bool
values true | false
```

```
or_type Possibly(T1)
values the_value:T1 | no_value
```

```
or_type Result(T1)OnError(T2)
values result:T1 | error:T2
```

```
>>> the_value:1
      : Possibly(Int)
>>> the_value:_
      : T1 => Possibly(T1)
>>> apply(the_value:_)to_all_in(_)
      : ListOf(T1)s => ListOf(Possibly(T1))s
```

## Τελεστές: Εφαρμογής Συνάρτησης

Τελεστής	Τύπος
$\rightarrow$	$T1 \times (T1 \Rightarrow T2) \Rightarrow T2$
$\leftarrow$	$(T1 \Rightarrow T2) \times T1 \Rightarrow T2$

Αποφυγή ανοίγματος και κλεισίματος πολλών παρενθέσεων σε αλυσίδα εφαρμογών.

Το όρισμα 'μπαίνει' στην συνάρτηση.



## Τελεστές: Εφαρμογής Συνάρτησης

```
apply(_)to_all_in(_)  
  : (T1 => T2) x ListOf(T1)s => ListOf(T2)s  
  
length_of(_) : String => Int  
  
filter(_)with(_)  
  : ListOf(T1)s x (T1 => Bool) => ListOf(T1)s  
  
(_)is_odd : Int => Bool  
  
sum_ints(_) : ListOf(Int)s => Int  
  
strings : ListOf(String)s  
  
chars_in_odd_length_strings : Int  
  = apply(length_of(_))to_all_in(strings) ->  
    filter(_)with((_)is_odd) ->  
    sum_ints(_)
```

## Τελεστές: Σύνθεσης Συναρτήσεων

Τελεστής	Τύπος
$\circ>$	$(T1 \Rightarrow T2) \times (T2 \Rightarrow T3) \Rightarrow (T1 \Rightarrow T3)$
$<\circ$	$(T2 \Rightarrow T3) \times (T1 \Rightarrow T2) \Rightarrow (T1 \Rightarrow T3)$

Γιατί να γράφεις παραμέτρους όταν μπορείς να κάνεις σύνθεση;

Μοιάζουν με σύνθεση στα μαθηματικά.

Έδειχνουν από πρώτη συνάρτηση σε δεύτερη.

## Τελεστές: Σύνθεσης Συναρτήσεων

```
split(_)to_words : String => ListOf(String)s
```

```
apply(_)to_all_in(_)  
  : (T1 => T2) x ListOf(T1)s => ListOf(T2)s
```

```
reverse_str(_) : String => String
```

```
merge_words(_) : ListOf(String)s => String
```

```
reverse_words_in(_) : String => String  
  = split(_)to_words o>  
    apply(reverse_str(_))to_all_in(_) o>  
    merge_words(_)
```

## Τελεστές: Αριθμητικοί

Τελεστής	Τύπος
<code>~</code>	<code>(@A)To_The(@B)Is(@C) --&gt; @A x @B =&gt; @C</code>
<code>*</code>	<code>(@A)And(@B)Multiply_To(@C) --&gt; @A x @B =&gt; @C</code>
<code>/</code>	<code>(@A)Divided_By(@B)Is(@C) --&gt; @A x @B =&gt; @C</code>
<code>+</code>	<code>(@A)And(@B)Add_To(@C) --&gt; @A x @B =&gt; @C</code>
<code>-</code>	<code>(@A)Minus(@B)Is(@C) --&gt; @A x @B =&gt; @C</code>

Ότι περιμένει κανείς για τους στάνταρ τύπους.

Γενικευμένοι για κόλπα python (κρατώντας την σιγουριά τύπων!).

Πιο δυνατά κόλπα από python: μπορούν να οριστούν για οποιοδήποτε συνδιασμό τύπων από τον χρήστη.

## Τελεστές: Αριθμητικοί

```
>>> 'a' + 'b'  
= "ab"
```

```
>>> 'w' + "ord"  
= "word"
```

```
>>> "Hello " + "World!"  
= "Hello World!"
```

```
>>> 5 * 'a'  
= "aaaaa"
```

```
>>> 5 * "hi"  
= "hihihihihi"
```

```
>>> "1,2,3" - ', '  
= "123"
```

## Τελεστές: Σύγκρισης και Λογικοί

Op	Τύπος
==	<code>(@A)And(@B)Can_Be_Equal --&gt; @A x @B =&gt; Bool</code>
!=	<code>(@A)And(@B)Can_Be_Unequal --&gt; @A x @B =&gt; Bool</code>
>	<code>(@A)Can_Be_Greater_Than(@B) --&gt; @A x @B =&gt; Bool</code>
<	<code>(@A)Can_Be_Less_Than(@B) --&gt; @A x @B =&gt; Bool</code>
>=	<code>(@A)Can_Be_Gr_Or_Eq_To(@B) --&gt; @A x @B =&gt; Bool</code>
<=	<code>(@A)Can_Be_Le_Or_Eq_To(@B) --&gt; @A x @B =&gt; Bool</code>
&	<code>(@A)Has_And --&gt; @A^2 =&gt; @A</code>
	<code>(@A)Has_Or --&gt; @A^2 =&gt; @A</code>

Μπορούν να συγκριθούν διαφορετικοί τύποι.

Τελεστές 'και' και 'ή' πολυμορφικοί για να χρησιμοποιούνται και για bitwise operations.

## Τελεστές: Περιβάλλοντος

Op	Τύπος
; >	$(@E)Has\_Use \rightarrow @E(T1) \times (T1 \Rightarrow @E(T2)) \Rightarrow @E(T2)$
;	$(@E)Has\_Then \rightarrow @E(T1) \times @E(T2) \Rightarrow @E(T2)$

Για συνδιασμούς δράσεων σε περιβάλλον (IO, State κτλ).

Αντιστοιχούν στους operators "bind" και "then" των Monads.

Το ';' μοιάζει αρκετά στην χρήση με imperative παρόλο που είναι functional.

## Τελεστές: Περιβάλλοντος

Op	Τύπος
<code>; &gt;</code>	$(@E)Has\_Use \rightarrow @E(T1) \times (T1 \Rightarrow @E(T2)) \Rightarrow @E(T2)$
<code>;</code>	$(@E)Has\_Then \rightarrow @E(T1) \times @E(T2) \Rightarrow @E(T2)$

```
main: (EmptyVal)FromIO
  = print("I'll repeat the line") ; get_line ;> print(_)
```

```
print(_) : String => (EmptyVal)FromIO
```

```
print("I'll repeat the line") : (EmptyVal)FromIO
```

```
get_line : (String)FromIO
```



## Ένα ωραίο παράδειγμα με διάφορα

```
important_stuff : ListOf(String)s  
  = [ "the Ultimate Question of Life"  
      , "the Universe"  
      , "Everything"  
      ]
```

```
prepend : String => String  
  = "The answer to " + _
```

```
>>> apply(prepend o> (42, _))to_all_in(important_stuff)  
  = [ (42, "The answer to the Ultimate Question of Life")  
      , (42, "The answer to the Universe")  
      , (42, "The answer to Everything")  
      ]
```

## Εκφράσεις Συναρτήσεων

$a \Rightarrow 17 * a + 42$

$(x, y, z) \Rightarrow \text{sqrt}(x^2 + y^2 + z^2)$

Αστερίσκος για παραμέτρους που δεν έχουν σημασία

$* \Rightarrow 42$

$(x, *, z) \Rightarrow x + z$

Matching σε παραμέτρους tuples

$((x1, y1), (x2, y2)) \Rightarrow (x1 + x2, y1 + y2)$

## Εκφράσεις Συναρτήσεων "cases"

```
(_)is_forty_two: Int => Bool  
  = cases  
    42 => true  
    ... => false
```

```
gcd_of(_)and(_): Int^2 => Int  
  = (x, cases)  
    0 => x  
    y => gcd_of(y)and((x)mod(y))
```

```
traffic_lights_match(_, _): TrafficLight^2 => Bool  
  = (cases, cases)  
    (green, green) => true  
    (amber, amber) => true  
    (red, red) => true  
    ... => false
```

## Εκφράσεις Συναρτήσεων "cases"

```
apply(_)to_all_in(_)  
  : (T1 => T2) x ListOf(T1)s => ListOf(T2)s  
  = (f(_), cases)  
    [] => []  
    [head, tail = ...] =>  
      f(head) + apply(f(_))to_all_in(tail)  
  
(_)is_sorted: ListOf(Int)s => Bool  
  = cases  
    [x1, x2, xs = ...] => (x1 <= x2) & (x2 + xs)is_sorted  
    ... => true
```

## Ορισμοί Τιμών

```
foo: Int  
  = 1
```

Ομαδοποίηση ορισμών:

```
val1, val2, val3: Int, Bool, Char  
  = 2, true, 'a'
```

Λέξη κλειδί "all":

```
golden_ratio, e, pi: all Real  
  = 1.618, 2.718, 3.1415
```

## Εκφράσεις με "where"

```
sort(_): ListOf(Int)s => ListOf(Int)s
= cases
  [] => []
  [head, tail = ...] =>
    sort(less_l) + head + sort(greater_l)
  where
    less_l, greater_l: all ListOf(Int)s
    = filter(tail)with(_ <= head)
      , filter(tail)with(_ > head)
```

## Εκφράσεις με "where"

```
sum_nodes(_): TreeOf(Int)s => Int
  = tree =>
    tree.root +
    sum_list(_) <-
    apply(sum_nodes(_))to_all_in(tree.subtrees)
  where
    sum_list(_) : ListOf(Int)s => Int
      = cases
        [] => 0
        [head, tail = ...] => head + sum_list(tail)
```

# Τύποι

	lcases	Haskell
Παραμετρικές μεταβλητές τύπων	T1 T2 T3	a b c
Ad hoc μεταβλητές τύπων	@A @B @C	a b c
Τύποι εφαρμογής συνάρτησης τύπου	Possibly(Int)	Maybe Int
Τύποι γινόμενα	Int x Real x String	(Int, Double, String)
Τύποι δύναμης	Int <sup>3</sup>	(Int, Int, Int)
Τύποι συνάρτησης	Int => Int	Int -> Int



## Τύποι με προϋπόθεση

$(@A) \text{And} (@B) \text{Add\_To} (@C) \dashrightarrow @A \times @B \Rightarrow @C$

$(@T) \text{Has\_Str\_Rep} \dashrightarrow @T \Rightarrow \text{String}$

$(@E) \text{Has\_Use} \dashrightarrow @E(T1) \times (T1 \Rightarrow @E(T2)) \Rightarrow @E(T2)$

Haskell

$\text{Add } a \ b \ c \Rightarrow a \rightarrow b \rightarrow c$

$\text{Show } a \Rightarrow a \rightarrow \text{String}$

$\text{Monad } m \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b$

## Παρατσούκλια Τύπων

```
type_nickname Ints = ListOf(Int)s
```

```
type_nickname IntStrPairs = ListOf(Int x String)s
```

```
type_nickname IO = (EmptyVal)FromIO
```

```
type_nickname Res(T1)OrErr = Result(T1)OrError(String)
```

```
type στην Haskell
```

# Λογική Τύπων

Μηχανισμός ad hoc πολυμορφισμού στην `λcases`.

Αντιστοιχεί στα `type classes` (`MultiParamTypeClasses` extension).

Ορισμοί ατομικών προτάσεων τύπων  $\Rightarrow$  `class declarations`

Θεωρήματα προτάσεων τύπων  $\Rightarrow$  `instance declarations`

## Ορισμοί Ατομικών Προτάσεων Τύπων

```
type_proposition (@T)Has_Str_Rep  
needed (_,_)to_string: @T => String
```

```
type_proposition (@T)Has_A_Wrapper  
needed wrap(_): T1 => @T(T1)
```

```
type_proposition (@T)Has_Internal_App  
needed apply(_)_inside(_): (T1 => T2) x @T(T1) => @T(T2)
```

```
(_,_)to_string: (@T)Has_Str_Rep --> @T => String  
wrap(_): (@T)Has_A_Wrapper --> T1 => @T(T1)  
apply(_)_inside(_)  
  : (@T)Has_Internal_App -->  
    (T1 => T2) x @T(T1) => @T(T2)
```

## Θεωρήματα Τύπων

```
type_theorem (Possibly(_))Has_A_Wrapper
proof wrap(_) = the_value:_
```

```
type_theorem (ListOf(_))sHas_A_Wrapper
proof wrap(_) = []
```

```
type_theorem (Possibly(_))Has_Internal_App
proof
  apply(_)inside(_) =
    (f(_), cases)
      no_value => no_value
      the_value:x => the_value:f(x)
```

```
type_theorem (ListOf(_))sHas_Internal_App
proof apply(_)inside(_) = apply(_)to_all_in(_)
```

## Χρήση Των Ad Hoc Πολυμορφικών Συναρτήσεων

```
a, b : all Possibly(Int)
      = wrapper(1), no_value
l1, l2, l3 : all ListOf(Int)s
            = wrapper(1), [], [1, 2, 3]
```

```
>>> a                                >>> l1
      = the_value:1                    = [1]
```

```
>>> apply(_ + 1)inside(a)            >>> apply(_ + 1)inside(b)
      = the_value:2                    = no_value
```

```
>>> apply(_ + 1)inside(l1)           >>> apply(_ + 1)inside(l2)
      = [2]                            = []
```

```
>>> apply(_ + 1)inside(l3)
      = [2, 3, 4]
```

## Ορισμοί Προτάσεων Τύπων Μετονομασίας

```
type_proposition (@T)Has_Equality  
equivalent (@T)And(@T)Can_Be_Equal
```

```
type_proposition (@A)And(@B)Are_Comparable  
equivalent  
  (@A)Can_Be_Less_Than(@B), (@A)And(@B)Can_Be_Equal,  
  (@A)Can_Be_Greater_Than(@B)
```

```
type_proposition (@T)Has_Comparison  
equivalent (@T)And(@T)Are_Comparable
```

## Θεωρήματα Συνεπαγωγής

type\_theorem

```
(@A)And(@B)Can_Be_Equal --> (@A)And(@B)Can_Be_Unequal  
proof a \= b = not(a == b)
```

type\_theorem

```
(@A)Can_Be_Greater_Than(@B) -->  
  (@A)Can_Be_Le_Or_Eq_To(@B)  
proof a <= b = not(a > b)
```

type\_proposition (@A)And(@B)Have\_Eq\_And\_Gr  
equivalent

```
(@A)And(@B)Can_Be_Equal, (@A)Can_Be_Greater_Than(@B)
```

type\_theorem

```
(@A)And(@B)Have_Eq_And_Gr -->  
  (@A)Can_Be_Gr_Or_Eq_To(@B)  
proof a >= b = a == b | a > b
```



# Υλοποίηση Parser και Μετάφραση σε Haskell

Βιβλιοθήκη Parsec:

- ▶ Parser Combinator Βιβλιοθήκη
- ▶ Context-sensitive, Infinite look-ahead γραμματικές
  - ▶ Πολύ πρακτικό για σύστημα indentation
- ▶ Μοιάζει αρκετά με BNF στην χρήση

Μετάφραση σε Haskell:

- ▶ Συντακτική ανάλυση
- ▶ Μετατροπές πάνω στο AST
- ▶ Νέο AST σε Haskell

(Compile Haskell με ghc).

Μια ματιά στον κώδικα (αν υπάρχει χρόνος).

# Συμπεράσματα

Έχει γίνει:

- ▶ Σχεδίαση της γλώσσας
  - ▶ Γραμματική με indentation σύστημα
  - ▶ Ορισμοί
- ▶ Μετάφραση σε Haskell

Μελλοντικά:

- ▶ Semantic analysis
  - ▶ Scoping σφάλματα
  - ▶ Σύστημα Τύπων ενσωματωμένο
- ▶ Interpreter/REPL
  - ▶ Evaluation
  - ▶ Type info
  - ▶ Πληροφορίες σχετικά με Ad Hoc πολυμορφισμό
    - ▶ Χρήση Prolog;
- ▶ Βιβλιοθήκες, Documentation κτλ