Haskell: Βασικές Δομές και Απόκρυψη Ονομάτων

Γιάννης Κασσιός

Δομή Σημερινής Διάλεξης

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
- Λίστες (lists): σειρές ομογενών δεδομένων
  - βασικά / τύποι / συμβολοσειρές
  - διαχωρισμός
  - συναρτήσεις Haskell για λίστες
- Μοτίβα (patterns): άμεση πρόσβαση σε δεδομένα με πολύπλοκη δομή
  - μεταβλητές / σταθερές / `_`
  - μοτίβα πλειάδων / λιστών
  - δομή `case`
- Απόκρυψη ονομάτων
  - εκφράσεις `where`, `let` και λ-εκφράσεις
- Παραδείγματα
- Επισκόπηση

Δομή Σημερινής Διάλεξης

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
- Λίστες (lists): σειρές ομογενών δεδομένων
  - βασικά / τύποι / συμβολοσειρές
  - διαχωρισμός
  - συναρτήσεις Haskell για λίστες
- Μοτίβα (patterns): άμεση πρόσβαση σε δεδομένα με πολύπλοκη δομή
  - μεταβλητές / σταθερές / `_`
  - μοτίβα πλειάδων / λιστών
  - δομή `case`
- Απόκρυψη ονομάτων
  - εκφράσεις `where`, `let` και λ-εκφράσεις
- Παραδείγματα
- Επισκόπηση

Δομή Σημερινής Διάλεξης

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
- Λίστες (lists): σειρές ομογενών δεδομένων
  - βασικά / τύποι / συμβολοσειρές
  - διαχωρισμός
  - συναρτήσεις Haskell για λίστες
- Μοτίβα (patterns): άμεση πρόσβαση σε δεδομένα με πολύπλοκη δομή
  - μεταβλητές / σταθερές / _
  - μοτίβα πλειάδων / λιστών
  - δομή `case`
- Απόκρυψη ονομάτων
  - εκφράσεις `where`, `let` και λ-εκφράσεις
- Παραδείγματα
- Επισκόπηση

Δομή Σημερινής Διάλεξης

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
- Λίστες (lists): σειρές ομογενών δεδομένων
  - βασικά / τύποι / συμβολοσειρές
  - διαχωρισμός
  - συναρτήσεις Haskell για λίστες
- Μοτίβα (patterns): άμεση πρόσβαση σε δεδομένα με πολύπλοκη δομή
  - μεταβλητές / σταθερές / _
  - μοτίβα πλειάδων / λιστών
  - δομή `case`
- Απόκρυψη ονομάτων
  - εκφράσεις `where`, `let` και λ-εκφράσεις
- Παραδείγματα
- Επισκόπηση

Δομή Σημερινής Διάλεξης

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
- Λίστες (lists): σειρές ομογενών δεδομένων
  - βασικά / τύποι / συμβολοσειρές
  - διαχωρισμός
  - συναρτήσεις Haskell για λίστες
- Μοτίβα (patterns): άμεση πρόσβαση σε δεδομένα με πολύπλοκη δομή
  - μεταβλητές / σταθερές / _
  - μοτίβα πλειάδων / λιστών
  - δομή `case`
- Απόκρυψη ονομάτων
  - εκφράσεις `where`, `let` και λ-εκφράσεις
- Παραδείγματα
- Επισκόπηση

Δομή Σημερινής Διάλεξης

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
- Λίστες (lists): σειρές ομογενών δεδομένων
  - βασικά / τύποι / συμβολοσειρές
  - διαχωρισμός
  - συναρτήσεις Haskell για λίστες
- Μοτίβα (patterns): άμεση πρόσβαση σε δεδομένα με πολύπλοκη δομή
  - μεταβλητές / σταθερές / `_`
  - μοτίβα πλειάδων / λιστών
  - δομή `case`
- Απόκρυψη ονομάτων
  - εκφράσεις `where`, `let` και λ-εκφράσεις
- Παραδείγματα
- Επισκόπηση

Πλειάδες
Βασικά

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
  - σε άλλες γλώσσες: εγγραφές (records), δομές (structs), αντικείμενα (objects)
- Σύνταξη:
  (*δεδομένο0*, *δεδομένο1*, ... *δεδομένοN*)
- Παραδείγματα:

```
(1 , 3)
('a', (20 , 10 , True))
(f.g , 2.0 , 9 , '\n' , False)
```

Πλειάδες
Βασικά

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
  - σε άλλες γλώσσες: εγγραφές (records), δομές (structs), αντικείμενα (objects)
- Σύνταξη:
  (*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*)
- Παραδείγματα:

```
(1 , 3)
('a', (20 , 10 , True))
(f.g , 2.0 , 9 , '\n' , False)
```

Πλειάδες
Βασικά

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
  - σε άλλες γλώσσες: εγγραφές (records), δομές (structs), αντικείμενα (objects)
- Σύνταξη:
  (*δεδομένο0* , *δεδομένο1* , ... *δεδομένοN*)
- Παραδείγματα:

```
(1 , 3)
('a', (20 , 10 , True))
(f.g , 2.0 , 9 , '\n' , False)
```

- Πλειάδες (tuples): πεπερασμένες συλλογές ετερογενών δεδομένων συγκεκριμένου μεγέθους
  - σε άλλες γλώσσες: εγγραφές (records), δομές (structs), αντικείμενα (objects)
- Σύνταξη:
  (*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*)
- Παραδείγματα:

```
(1 , 3)
('a', (20 , 10 , True))
(f.g , 2.0 , 9 , '\n' , False)
```

Πλειάδες
Χρήση ως Εγγραφές / Αντικείμενα

- Συλλογή δεδομένων σχετικά με μία οντότητα
- Παράδειγμα: πρόγραμμα που κρατάει δεδομένα για φοιτητές
  - όνομα (`String`)
  - φύλο (`Bool`)
  - ηλικία (`Int`)

  τον τύπο `String` θα τον δούμε αργότερα
- Παραδείγματα πλειάδων που αναπαριστούν φοιτητές:

```
("Yannis" , False , 22)
("Maria" , True , 24)
```

Πλειάδες
Χρήση ως Εγγραφές / Αντικείμενα

- Συλλογή δεδομένων σχετικά με μία οντότητα
- Παράδειγμα: πρόγραμμα που κρατάει δεδομένα για φοιτητές
  - όνομα `(String)`
  - φύλο `(Bool)`
  - ηλικία `(Int)`

  τον τύπο `String` θα τον δούμε αργότερα
- Παραδείγματα πλειάδων που αναπαριστούν φοιτητές:

```
("Yannis" , False , 22)
("Maria" , True , 24)
```

Πλειάδες
Χρήση ως Εγγραφές / Αντικείμενα

- Συλλογή δεδομένων σχετικά με μία οντότητα
- Παράδειγμα: πρόγραμμα που κρατάει δεδομένα για φοιτητές
  - όνομα `(String)`
  - φύλο `(Bool)`
  - ηλικία `(Int)`

  τον τύπο `String` θα τον δούμε αργότερα
- Παραδείγματα πλειάδων που αναπαριστούν φοιτητές:

```
("Yannis" , False , 22)
("Maria" , True , 24)
```

- Συλλογή δεδομένων σχετικά με μία οντότητα
- Παράδειγμα: πρόγραμμα που κρατάει δεδομένα για φοιτητές
  - όνομα `(String)`
  - φύλο `(Bool)`
  - ηλικία `(Int)`

  τον τύπο `String` θα τον δούμε αργότερα
- Παραδείγματα πλειάδων που αναπαριστούν φοιτητές:

```
("Yannis" , False , 22)
("Maria" , True , 24)
```

- Ο τύπος της `(v0,v1,..,vn)` είναι `(t0,t1,..,tn)`
  - `t0,t1,..,tn` αντίστοιχα οι τύποι των `v0,v1,..,vn`
- Παράδειγμα:
  `('a',(20,10,True)) :: (Char,(Int,Int,Bool))`
- Ορισμοί τύπων:
  `type` *όνομα_τύπου=ορισμός*
  - Τα ονόματα τύπων αρχίζουν με κεφαλαίο
  - Δημιουργεί συνώνυμο τύπο
- Παραδείγματα ορισμών τύπων:

```
type Student = (String,Bool,Int)
type UnaryOp = Int->Int
type TF = Bool
```

- Ο τύπος της `(v0,v1,..,vn)` είναι `(t0,t1,..,tn)`
  - `t0,t1,..,tn` αντίστοιχα οι τύποι των `v0,v1,..,vn`
- Παράδειγμα:
  `('a',(20,10,True)) :: (Char,(Int,Int,Bool))`
- Ορισμοί τύπων:
  `type` *όνομα_τύπου=ορισμός*
  - Τα ονόματα τύπων αρχίζουν με κεφαλαίο
  - Δημιουργεί συνώνυμο τύπο
- Παραδείγματα ορισμών τύπων:

```
type Student = (String,Bool,Int)
type UnaryOp = Int->Int
type TF = Bool
```

- Ο τύπος της `(v0,v1,..,vn)` είναι `(t0,t1,..,tn)`
  - `t0,t1,..,tn` αντίστοιχα οι τύποι των `v0,v1,..,vn`
- Παράδειγμα:
  `('a',(20,10,True)) :: (Char,(Int,Int,Bool))`
- Ορισμοί τύπων:
  `type` *όνομα_τύπου=ορισμός*
  - Τα ονόματα τύπων αρχίζουν με κεφαλαίο
  - Δημιουργεί συνώνυμο τύπο
- Παραδείγματα ορισμών τύπων:

```
type Student = (String,Bool,Int)
type UnaryOp = Int->Int
type TF = Bool
```

- Ο τύπος της `(v0,v1,..,vn)` είναι `(t0,t1,..,tn)`
  - `t0,t1,..,tn` αντίστοιχα οι τύποι των `v0,v1,..,vn`
- Παράδειγμα:
  `('a',(20,10,True)) :: (Char,(Int,Int,Bool))`
- Ορισμοί τύπων:
  `type` *όνομα_τύπου=ορισμός*
  - Τα ονόματα τύπων αρχίζουν με κεφαλαίο
  - Δημιουργεί συνώνυμο τύπο
- Παραδείγματα ορισμών τύπων:

```
type Student = (String,Bool,Int)
type UnaryOp = Int->Int
type TF = Bool
```

- Οι συναρτήσεις `fst` και `snd` επιλέγουν το πρώτο και το δεύτερο στοιχείο ενός ζεύγους αντίστοιχα
- `fst (1, 'a') = 1`
- Πολύ αδύναμη υποστήριξη: στηριζόμαστε στη χρήση μοτίβων

- Οι συναρτήσεις `fst` και `snd` επιλέγουν το πρώτο και το δεύτερο στοιχείο ενός ζεύγους αντίστοιχα
- `fst (1, 'a') = 1`
- Πολύ αδύναμη υποστήριξη: στηριζόμαστε στη χρήση μοτίβων

- Οι συναρτήσεις `fst` και `snd` επιλέγουν το πρώτο και το δεύτερο στοιχείο ενός ζεύγους αντίστοιχα
- `fst (1, 'a') = 1`
- Πολύ αδύναμη υποστήριξη: στηριζόμαστε στη χρήση μοτίβων

- Λίστες (lists): σειρές ομογενών δεδομένων μεταβλητού μεγέθους
  - σε άλλες γλώσσες: πίνακες (arrays)
  - όμως: μεταβλητό και πιθανώς άπειρο μέγεθος
- Απλή κατασκευή λίστων:
  [*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*]
- Παραδείγματα:
  `[2,4]`
  `['a']` ≠ `'a'`
  `[]` κενή λίστα
  `[2.1 , 1.5 , 6.0 , 2.1]`
  `[(1,2,True),(4,5,False)]`
  `[[],['h'],['h','i']]`
- Σειρά και αριθμός εμφάνισης στοιχείων σημαντικά:
  `[1,2,3]` ≠ `[3,2,1]`
  `[1,2,3]` ≠ `[1,1,2,3,3,3]`

Λίστες
Βασικά

- Λίστες (lists): σειρές ομογενών δεδομένων μεταβλητού μεγέθους
  - σε άλλες γλώσσες: πίνακες (arrays)
  - όμως: μεταβλητό και πιθανώς άπειρο μέγεθος
- Απλή κατασκευή λίστων:
  [*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*]
- Παραδείγματα:
  `[2,4]`
  `['a']` ≠ `'a'`
  `[]` κενή λίστα
  `[2.1 , 1.5 , 6.0 , 2.1]`
  `[(1,2,True),(4,5,False)]`
  `[[],['h'],['h','i']]`
- Σειρά και αριθμός εμφάνισης στοιχείων σημαντικά:
  `[1,2,3]` ≠ `[3,2,1]`
  `[1,2,3]` ≠ `[1,1,2,3,3,3]`

Λίστες
Βασικά

- Λίστες (lists): σειρές ομογενών δεδομένων μεταβλητού μεγέθους
  - σε άλλες γλώσσες: πίνακες (arrays)
  - όμως: μεταβλητό και πιθανώς άπειρο μέγεθος
- Απλή κατασκευή λίστων:
  [*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*]
- Παραδείγματα:
  `[2,4]`
  `['a']` ≠ `'a'`
  `[]` κενή λίστα
  `[2.1 , 1.5 , 6.0 , 2.1]`
  `[(1,2,True),(4,5,False)]`
  `[[],['h'],['h','i']]`
- Σειρά και αριθμός εμφάνισης στοιχείων σημαντικά:
  `[1,2,3]` ≠ `[3,2,1]`
  `[1,2,3]` ≠ `[1,1,2,3,3,3]`

Λίστες
Βασικά

- Λίστες (lists): σειρές ομογενών δεδομένων μεταβλητού μεγέθους
  - σε άλλες γλώσσες: πίνακες (arrays)
  - όμως: μεταβλητό και πιθανώς άπειρο μέγεθος
- Απλή κατασκευή λίστων:
  [*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*]
- Παραδείγματα:
  `[2,4]`
  `['a']` ≠ `'a'`
  `[]` κενή λίστα
  `[2.1 , 1.5 , 6.0 , 2.1]`
  `[(1,2,True),(4,5,False)]`
  `[[],['h'],['h','i']]`
- Σειρά και αριθμός εμφάνισης στοιχείων σημαντικά:
  `[1,2,3]` ≠ `[3,2,1]`
  `[1,2,3]` ≠ `[1,1,2,3,3,3]`

Λίστες
Βασικά

- Λίστες (lists): σειρές ομογενών δεδομένων μεταβλητού μεγέθους
  - σε άλλες γλώσσες: πίνακες (arrays)
  - όμως: μεταβλητό και πιθανώς άπειρο μέγεθος
- Απλή κατασκευή λίστων:
  [*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*]
- Παραδείγματα:
  `[2,4]`
  `['a']` ≠ `'a'`
  `[]` κενή λίστα
  `[2.1 , 1.5 , 6.0 , 2.1]`
  `[(1,2,True),(4,5,False)]`
  `[[],['h'],['h','i']]`
- Σειρά και αριθμός εμφάνισης στοιχείων σημαντικά:
  `[1,2,3]` ≠ `[3,2,1]`
  `[1,2,3]` ≠ `[1,1,2,3,3,3]`

Λίστες
Βασικά

- Λίστες (lists): σειρές ομογενών δεδομένων μεταβλητού μεγέθους
  - σε άλλες γλώσσες: πίνακες (arrays)
  - όμως: μεταβλητό και πιθανώς άπειρο μέγεθος
- Απλή κατασκευή λίστων:
  [*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*]
- Παραδείγματα:
  `[2,4]`
  `['a']` $\neq$ `'a'`
  `[]` κενή λίστα
  `[2.1 , 1.5 , 6.0 , 2.1]`
  `[(1,2,True),(4,5,False)]`
  `[[],['h'],['h','i']]`
- Σειρά και αριθμός εμφάνισης στοιχείων σημαντικά:
  `[1,2,3]` $\neq$ `[3,2,1]`
  `[1,2,3]` $\neq$ `[1,1,2,3,3,3]`

Λίστες
Βασικά

- Λίστες (lists): σειρές ομογενών δεδομένων μεταβλητού μεγέθους
  - σε άλλες γλώσσες: πίνακες (arrays)
  - όμως: μεταβλητό και πιθανώς άπειρο μέγεθος
- Απλή κατασκευή λίστων:
  [*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*]
- Παραδείγματα:
  `[2,4]`
  `['a']` $\neq$ `'a'`
  `[]` κενή λίστα
  `[2.1 , 1.5 , 6.0 , 2.1]`
  `[(1,2,True),(4,5,False)]`
  `[[],['h'],['h','i']]`
- Σειρά και αριθμός εμφάνισης στοιχείων σημαντικά:
  `[1,2,3]` $\neq$ `[3,2,1]`
  `[1,2,3]` $\neq$ `[1,1,2,3,3,3]`

Λίστες
Βασικά

- Λίστες (lists): σειρές ομογενών δεδομένων μεταβλητού μεγέθους
  - σε άλλες γλώσσες: πίνακες (arrays)
  - όμως: μεταβλητό και πιθανώς άπειρο μέγεθος
- Απλή κατασκευή λίστων:
  [*δεδομένο0*,*δεδομένο1*, ... *δεδομένοN*]
- Παραδείγματα:
  `[2,4]`
  `['a']` $\neq$ `'a'`
  `[]` κενή λίστα
  `[2.1 , 1.5 , 6.0 , 2.1]`
  `[(1,2,True),(4,5,False)]`
  `[[],['h'],['h','i']]`
- Σειρά και αριθμός εμφάνισης στοιχείων σημαντικά:
  `[1,2,3]` $\neq$ `[3,2,1]`
  `[1,2,3]` $\neq$ `[1,1,2,3,3,3]`

Λίστες
Ειδική Κατασκευή Λιστών για Βασικούς Τύπους

- Για λίστες αντικειμένων `Int,Float,Char`: `[b..e]`
  - λίστα από `b` έως `e` με αύξουσα σειρά
  - για `Float` το βήμα είναι `1.0`
  - Παραδείγματα:

```
[1..5]=[1,2,3,4,5]
[2..2]=[2]
[5..1]=[]
['a'..'g']=['a','b','c','d','e','f','g']
['g'..'a']=[]
[4.1 .. 7.0] = [4.1 , 5.1 , 6.1]
```

- Με βήμα: γράφουμε και το δεύτερο στοιχείο:

```
[1,3..9]=[1,3,5,7,9]
[9,6..0]=[9,6,3,0]
['g','e'..'a']=['g','e','c','a']
[2.0 , 1.9 .. 1.7] = [2.0 , 1.9 , 1.8 , 1.7]
```

Λίστες
Ειδική Κατασκευή Λιστών για Βασικούς Τύπους

- Για λίστες αντικειμένων `Int,Float,Char`: `[b..e]`
  - λίστα από `b` έως `e` με αύξουσα σειρά
  - για `Float` το βήμα είναι `1.0`
  - Παραδείγματα:

```
[1..5]=[1,2,3,4,5]
[2..2]=[2]
[5..1]=[]
['a'..'g']=['a','b','c','d','e','f','g']
['g'..'a']=[]
[4.1 .. 7.0] = [4.1 , 5.1 , 6.1]
```

- Με βήμα: γράφουμε και το δεύτερο στοιχείο:

```
[1,3..9]=[1,3,5,7,9]
[9,6..0]=[9,6,3,0]
['g','e'..'a']=['g','e','c','a']
[2.0 , 1.9 .. 1.7] = [2.0 , 1.9 , 1.8 , 1.7]
```

Λίστες
Ειδική Κατασκευή Λιστών για Βασικούς Τύπους

- Για λίστες αντικειμένων `Int,Float,Char`: `[b..e]`
  - λίστα από `b` έως `e` με αύξουσα σειρά
  - για `Float` το βήμα είναι `1.0`
  - Παραδείγματα:

```
[1..5]=[1,2,3,4,5]
[2..2]=[2]
[5..1]=[]
['a'..'g']=['a','b','c','d','e','f','g']
['g'..'a']=[]
[4.1 .. 7.0] = [4.1 , 5.1 , 6.1]
```

- Με βήμα: γράφουμε και το δεύτερο στοιχείο:

```
[1,3..9]=[1,3,5,7,9]
[9,6..0]=[9,6,3,0]
['g','e'..'a']=['g','e','c','a']
[2.0 , 1.9 .. 1.7] = [2.0 , 1.9 , 1.8 , 1.7]
```

Λίστες
Ειδική Κατασκευή Λιστών για Βασικούς Τύπους

- Για λίστες αντικειμένων `Int,Float,Char`: `[b..e]`
  - λίστα από `b` έως `e` με αύξουσα σειρά
  - για `Float` το βήμα είναι `1.0`
  - Παραδείγματα:

```
[1..5]=[1,2,3,4,5]
[2..2]=[2]
[5..1]=[]
['a'..'g']=['a','b','c','d','e','f','g']
['g'..'a']=[]
[4.1 .. 7.0] = [4.1 , 5.1 , 6.1]
```

- Με βήμα: γράφουμε και το δεύτερο στοιχείο:

```
[1,3..9]=[1,3,5,7,9]
[9,6..0]=[9,6,3,0]
['g','e'..'a']=['g','e','c','a']
[2.0 , 1.9 .. 1.7] = [2.0 , 1.9 , 1.8 , 1.7]
```

- Τύπος λιστών στοιχείων τύπου T: `[T]`
  - παραδείγματα: `[Int]`, `[Int->Int]`
- Δισδιάστατες λίστες: `[[Int]]`
- `[]` ανήκει σε όλους τους τύπους

- Τύπος λιστών στοιχείων τύπου T: `[T]`
  - παραδείγματα: `[Int]`, `[Int->Int]`
- Δισδιάστατες λίστες: `[[Int]]`
- `[]` ανήκει σε όλους τους τύπους

- Τύπος λιστών στοιχείων τύπου T: `[T]`
  - παραδείγματα: `[Int]`, `[Int->Int]`
- Δισδιάστατες λίστες: `[[Int]]`
- `[]` ανήκει σε όλους τους τύπους

- Συμβολοσειρά (string): λίστα χαρακτήρων:
  `type String = [Char]`
- Σταθερές τύπου `String`: μέσα σε διπλά εισαγωγικά:
  `"A string" = ['A',' ','s','t','r','i','n','g']`
  `"\"Hi\"\n" = ['\"' , 'H' , 'i' , '\"' , '\n']`
  `"" = []`
  `"6" = ['6']`
- Ειδικοί χαρακτήρες: χρησιμοποιούνται και στα strings

- Συμβολοσειρά (string): λίστα χαρακτήρων:

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

- Σταθερές τύπου `String`: μέσα σε διπλά εισαγωγικά:

```
"A string" = ['A',' ','s','t','r','i','n','g']
"\"Hi\"\n" = ['\"' , 'H' , 'i' , '\"' , '\n']
"" = []
"6" = ['6']
```

- Ειδικοί χαρακτήρες: χρησιμοποιούνται και στα strings

- Συμβολοσειρά (string): λίστα χαρακτήρων:

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

- Σταθερές τύπου `String`: μέσα σε διπλά εισαγωγικά:

```
"A string" = ['A',' ','s','t','r','i','n','g']
"\"Hi\"\n" = ['\"' , 'H' , 'i' , '\"' , '\n']
"" = []
"6" = ['6']
```

- Ειδικοί χαρακτήρες: χρησιμοποιούνται και στα strings

- Έκφραση διαχωρισμού λιστών (list comprehension): δημιουργία μίας λίστας με φιλτράρισμα και μετασχηματισμό των στοιχείων άλλων λιστών
  - από τα πιο εκφραστικά μέσα της Haskell
- Μετασχηματισμός. Έστω `l=[3,5,6]`. Τότε:
  `[2*n | n<-l] = [6,10,12]`
  - έκφραση μετασχηματισμού
  - γεννήτρια
  - `n` τοπικό
- +Φιλτράρισμα: `[2*n | n<-l , n `mod` 2 == 1]`
  - αποτιμάται σε `[6,10]`
  - συνθήκη ελέγχου
  - χωρίς μετασχηματισμό: έκφραση μετασχηματισμού `n`

- Έκφραση διαχωρισμού λιστών (list comprehension): δημιουργία μίας λίστας με φιλτράρισμα και μετασχηματισμό των στοιχείων άλλων λιστών
  - από τα πιο εκφραστικά μέσα της Haskell
- Μετασχηματισμός. Έστω `l=[3,5,6]`. Τότε:

  `[2*n | n<-l] = [6,10,12]`
  - έκφραση μετασχηματισμού
  - γεννήτρια
  - `n` τοπικό
- +Φιλτράρισμα: `[2*n | n<-l , n `mod` 2 == 1]`
  - αποτιμάται σε `[6,10]`
  - συνθήκη ελέγχου
  - χωρίς μετασχηματισμό: έκφραση μετασχηματισμού `n`

- Έκφραση διαχωρισμού λιστών (list comprehension): δημιουργία μίας λίστας με φιλτράρισμα και μετασχηματισμό των στοιχείων άλλων λιστών
  - από τα πιο εκφραστικά μέσα της Haskell
- Μετασχηματισμός. Έστω `l=[3,5,6]`. Τότε:

  `[2*n | n<-l] = [6,10,12]`
  - έκφραση μετασχηματισμού
  - γεννήτρια
  - `n` τοπικό
- +Φιλτράρισμα: `[2*n | n<-l , n `mod` 2 == 1]`
  - αποτιμάται σε `[6,10]`
  - συνθήκη ελέγχου
  - χωρίς μετασχηματισμό: έκφραση μετασχηματισμού `n`

- Έκφραση διαχωρισμού λιστών (list comprehension): δημιουργία μίας λίστας με φιλτράρισμα και μετασχηματισμό των στοιχείων άλλων λιστών
  - από τα πιο εκφραστικά μέσα της Haskell
- Μετασχηματισμός. Έστω `l=[3,5,6]`. Τότε:

  `[2*n | n<-l] = [6,10,12]`
  - έκφραση μετασχηματισμού
  - γεννήτρια
  - `n` τοπικό
- +Φιλτράρισμα: `[2*n | n<-l , n `mod` 2 == 1]`
  - αποτιμάται σε `[6,10]`
  - συνθήκη ελέγχου
  - χωρίς μετασχηματισμό: έκφραση μετασχηματισμού `n`

- Έκφραση διαχωρισμού λιστών (list comprehension): δημιουργία μίας λίστας με φιλτράρισμα και μετασχηματισμό των στοιχείων άλλων λιστών
  - από τα πιο εκφραστικά μέσα της Haskell
- Μετασχηματισμός. Έστω `l=[3,5,6]`. Τότε:

  `[2*n | n<-l] = [6,10,12]`
  - έκφραση μετασχηματισμού
  - γεννήτρια
  - `n` τοπικό
- +Φιλτράρισμα: `[2*n | n<-l , n `mod` 2 == 1]`
  - αποτιμάται σε `[6,10]`
  - συνθήκη ελέγχου
  - χωρίς μετασχηματισμό: έκφραση μετασχηματισμού `n`

- Έκφραση διαχωρισμού λιστών (list comprehension): δημιουργία μίας λίστας με φιλτράρισμα και μετασχηματισμό των στοιχείων άλλων λιστών
  - από τα πιο εκφραστικά μέσα της Haskell
- Μετασχηματισμός. Έστω `l=[3,5,6]`. Τότε:

  `[2*n | n<-l] = [6,10,12]`
  - έκφραση μετασχηματισμού
  - γεννήτρια
  - `n` τοπικό
- +Φιλτράρισμα: `[2*n | n<-l , n `mod` 2 == 1]`
  - αποτιμάται σε `[6,10]`
  - συνθήκη ελέγχου
  - χωρίς μετασχηματισμό: έκφραση μετασχηματισμού `n`

- Έστω `l=[1,2,3]` και `m=[4,5]`. Τότε ο ορισμός

  ```
  n = [(x,y) | x<-l, y<-m]
  ```

  δίνει

  ```
  n = [(1,4),(1,5) ,(2,4),(2,5),(3,4),(3,5)]
  ```

- Το `x` ταιριάζει με το 1 και συνδυάζεται με κάθε `y<-m`
- Μετά, το `x` ταιριάζει με το 2 και συνδυάζεται με κάθε `y<-m` κ.ο.κ.
- Προφανής γενίκευση για παραπάνω από μία γεννήτρια

- Έστω `l=[1,2,3]` και `m=[4,5]`. Τότε ο ορισμός

  ```
  n = [(x,y) | x<-l, y<-m]
  ```

  δίνει

  ```
  n = [(1,4),(1,5) ,(2,4),(2,5),(3,4),(3,5)]
  ```

- Το `x` ταιριάζει με το `1` και συνδυάζεται με κάθε `y<-m`
- Μετά, το `x` ταιριάζει με το `2` και συνδυάζεται με κάθε `y<-m` κ.ο.κ.
- Προφανής γενίκευση για παραπάνω από μία γεννήτρια

- Έστω `l=[1,2,3]` και `m=[4,5]`. Τότε ο ορισμός

  ```
  n = [(x,y) | x<-l, y<-m]
  ```

  δίνει

  ```
  n = [(1,4),(1,5) ,(2,4),(2,5),(3,4),(3,5)]
  ```

- Το `x` ταιριάζει με το `1` και συνδυάζεται με κάθε `y<-m`
- Μετά, το `x` ταιριάζει με το `2` και συνδυάζεται με κάθε `y<-m` κ.ο.κ.
- Προφανής γενίκευση για παραπάνω από μία γεννήτρια

- Έστω `l=[1,2,3]` και `m=[4,5]`. Τότε ο ορισμός

  ```
  n = [(x,y) | x<-l, y<-m]
  ```

  δίνει

  ```
  n = [(1,4),(1,5) ,(2,4),(2,5),(3,4),(3,5)]
  ```

- Το `x` ταιριάζει με το `1` και συνδυάζεται με κάθε `y<-m`
- Μετά, το `x` ταιριάζει με το `2` και συνδυάζεται με κάθε `y<-m` κ.ο.κ.
- Προφανής γενίκευση για παραπάνω από μία γεννήτρια

- Προσθήκη κεφαλής `(:)`

  `'a':"bc" = "abc"`

- Συνένωση `(++)`

  `[0,1,2] ++ [2, 1, 0] = [0,1,2,2,1,0]`

- Πρόσβαση σε στοιχείο `(!!)`

  `[3,4,5] !! 0 = 3`

  `[n^2 | n<-[4..10]] !! 2 = 36`

- Συνένωση πολλών λιστών `concat`

  `concat[[],[10,4],[99]] = [10,4,99]`

- Μέγεθος λίστας `length`

  `length "Hi!" = 3`

Λίστες
Συναρτήσεις της Haskell για Λίστες - II

- Κεφαλή `head` / τελευταίο στοιχείο `last`

```
head l = l!!0
last l = l!!(length l - 1)
```

- Ουρά `tail` / πρώτα στοιχεία `init`

```
tail[1,2,3] = [2,3]
init[1,2,3]=[1,2]
```

- Επανάληψη `replicate`

```
replicate 7 '7' = "7777777"
```

- Επιλογή `take` / απόρριψη `drop` αριθμού στοιχείων

```
take 4 [9..50] = [9,10,11,12]
drop 3 [0,10,20,4,5,6,7] = [4..7]
```

- Σπάσιμο λίστας `splitAt`

```
splitAt n l = (take n l , drop n l)
```

- Αναστροφή λίστας `reverse`

Λίστες
Συναρτήσεις της Haskell για Λίστες - III

- Zipping `zip`

```
zip "word" [1,2,3] = [('w',1) , ('o',2) , ('r',3)]
```

- Unzipping `unzip`

```
unzip [('w',1) , ('o',2) , ('r',3)]
   = ("wor" , [1,2,3])
```

- Εφαρμογή συνάρτησης σε κάθε στοιχείο `map`

```
map f l = [f x | x<-l]
map (^2) [1..4] = [1,4,9,16]
```

- Φιλτράρισμα λίστας `filter`

```
filter f l = [x | x<-l , f x]
filter (>0) [-1..20] = [1..20]
```

- Zipping με συνάρτηση `zipWith`

```
zipWith (+) [1,2,3] [4,5,6] = [5,7,9]
```

- Συσσωρευτικός υπολογισμός (folding) `foldl` και `foldr`

  ```
  foldl 0 (-) [1,1,1] = ((0-1)-1)-1 = -3
  foldr 0 (-) [1,1,1] = (1-(1-1))-0 = 1
  ```

- Επιλογή `takeWhile` / απόρριψη `dropWhile` με συνθήκη

  ```
  takeWhile (<9) [4,6,8,9,9] = [4,6,8]
  dropWhile (<9) [4,6,8,9,9] = [9,9]
  ```

Μοτίβα

- Μοτίβα (patterns): απόδοση ονόματος σε μέρη μίας δομής δεδομένων
- Παράδειγμα: συνάρτηση νόρμας πάνω σε ζεύγη αριθμών:

$$norm(x, y) = \sqrt{x^2 + y^2}$$

- Ορισμός Haskell:

```
norm p = sqrt ((fst p)^2 + (snd p)^2)
```

  - άκομψο! Θέλουμε να ονομάσουμε απ' ευθείας τα μέρη του `p`, όπως στο μαθηματικό ορισμό
- Ορισμός με χρήση μοτίβου:

```
norm (x,y) = sqrt (x^2 + y^2)
```

  - μοτίβο
- Εφαρμογή `norm(3.0,4.0)`
  - ταιριάζει (pattern matching) το μοτίβο `(x,y)` με `(3.0,4.0)`
  - περνάει τιμές `x`→`3.0` και `y`→`4.0`

Μοτίβα

- Μοτίβα (patterns): απόδοση ονόματος σε μέρη μίας δομής δεδομένων
- Παράδειγμα: συνάρτηση νόρμας πάνω σε ζεύγη αριθμών:

$$norm(x, y) = \sqrt{x^2 + y^2}$$

- Ορισμός Haskell:

```
norm p = sqrt ((fst p)^2 + (snd p)^2)
```

  - άκομψο! Θέλουμε να ονομάσουμε απ' ευθείας τα μέρη του `p`, όπως στο μαθηματικό ορισμό
- Ορισμός με χρήση μοτίβου:

```
norm (x,y) = sqrt (x^2 + y^2)
```

  - μοτίβο
- Εφαρμογή `norm(3.0,4.0)`
  - ταιριάζει (pattern matching) το μοτίβο `(x,y)` με `(3.0,4.0)`
  - περνάει τιμές `x`→`3.0` και `y`→`4.0`

Μοτίβα

- Μοτίβα (patterns): απόδοση ονόματος σε μέρη μίας δομής δεδομένων
- Παράδειγμα: συνάρτηση νόρμας πάνω σε ζεύγη αριθμών:

$$norm(x, y) = \sqrt{x^2 + y^2}$$

- Ορισμός Haskell:

```
norm p = sqrt ((fst p)^2 + (snd p)^2)
```

  - άκομψο! Θέλουμε να ονομάσουμε απ' ευθείας τα μέρη του `p`, όπως στο μαθηματικό ορισμό

- Ορισμός με χρήση μοτίβου:

```
norm (x,y) = sqrt (x^2 + y^2)
```

  - μοτίβο

- Εφαρμογή `norm(3.0,4.0)`
  - ταιριάζει (pattern matching) το μοτίβο `(x,y)` με `(3.0,4.0)`
  - περνάει τιμές `x`→`3.0` και `y`→`4.0`

Μοτίβα

- Μοτίβα (patterns): απόδοση ονόματος σε μέρη μίας δομής δεδομένων
- Παράδειγμα: συνάρτηση νόρμας πάνω σε ζεύγη αριθμών:

$$norm(x, y) = \sqrt{x^2 + y^2}$$

- Ορισμός Haskell:

```
norm p = sqrt ((fst p)^2 + (snd p)^2)
```

  - άκομψο! Θέλουμε να ονομάσουμε απ' ευθείας τα μέρη του `p`, όπως στο μαθηματικό ορισμό
- Ορισμός με χρήση μοτίβου:

```
norm (x,y) = sqrt (x^2 + y^2)
```

  - μοτίβο
- Εφαρμογή `norm(3.0,4.0)`
  - ταιριάζει (pattern matching) το μοτίβο `(x,y)` με `(3.0,4.0)`
  - περνάει τιμές `x`→`3.0` και `y`→`4.0`

Μοτίβα

- Μοτίβα (patterns): απόδοση ονόματος σε μέρη μίας δομής δεδομένων
- Παράδειγμα: συνάρτηση νόρμας πάνω σε ζεύγη αριθμών:

$$norm(x, y) = \sqrt{x^2 + y^2}$$

- Ορισμός Haskell:

```
norm p = sqrt ((fst p)^2 + (snd p)^2)
```

  - άκομψο! Θέλουμε να ονομάσουμε απ' ευθείας τα μέρη του `p`, όπως στο μαθηματικό ορισμό

- Ορισμός με χρήση μοτίβου:

```
norm (x,y) = sqrt (x^2 + y^2)
```

  - μοτίβο

- Εφαρμογή `norm(3.0,4.0)`
  - ταιριάζει (pattern matching) το μοτίβο `(x,y)` με `(3.0,4.0)`
  - περνάει τιμές `x`→`3.0` και `y`→`4.0`

Μοτίβα

- Μοτίβα (patterns): απόδοση ονόματος σε μέρη μίας δομής δεδομένων
- Παράδειγμα: συνάρτηση νόρμας πάνω σε ζεύγη αριθμών:

$$norm(x, y) = \sqrt{x^2 + y^2}$$

- Ορισμός Haskell:

```
norm p = sqrt ((fst p)^2 + (snd p)^2)
```

  - άκομψο! Θέλουμε να ονομάσουμε απ' ευθείας τα μέρη του `p`, όπως στο μαθηματικό ορισμό
- Ορισμός με χρήση μοτίβου:

```
norm (x,y) = sqrt (x^2 + y^2)
```

  - μοτίβο
- Εφαρμογή `norm(3.0,4.0)`
  - ταιριάζει (pattern matching) το μοτίβο `(x,y)` με `(3.0,4.0)`
  - περνάει τιμές `x`→`3.0` και `y`→`4.0`

- Χρήση σε ορισμούς:
  `(l0,l1) = unzip [('w',1) , ('o',2) , ('r',3)]`
  ισοδύναμο με:
  `l0="wor"`
  `l1=[1,2,3]`
  - χρήσιμο σε μηχανισμούς τοπικών ονομάτων
- Χρήση σε εκφράσεις διαχωρισμού:
  `[m+n | (m,n)<-l]`
  αποτιμάται σε `[3,17,42]` αν `l=[(1,2),(10,7),(-2,44)]`
- Μία συνάρτηση μπορεί να ορίζεται πολλές φορές. Το πρώτο μοτίβο που ταιριάζει χρησιμοποιείται

- Χρήση σε ορισμούς:
  `(l0,l1) = unzip [('w',1) , ('o',2) , ('r',3)]`
  ισοδύναμο με:
  `l0="wor"`
  `l1=[1,2,3]`
  - χρήσιμο σε μηχανισμούς τοπικών ονομάτων
- Χρήση σε εκφράσεις διαχωρισμού:
  `[m+n | (m,n)<-l]`
  αποτιμάται σε `[3,17,42]` αν `l=[(1,2),(10,7),(-2,44)]`
- Μία συνάρτηση μπορεί να ορίζεται πολλές φορές. Το πρώτο μοτίβο που ταιριάζει χρησιμοποιείται

- Χρήση σε ορισμούς:
  `(l0,l1) = unzip [('w',1) , ('o',2) , ('r',3)]`
  ισοδύναμο με:
  `l0="wor"`
  `l1=[1,2,3]`
  - χρήσιμο σε μηχανισμούς τοπικών ονομάτων
- Χρήση σε εκφράσεις διαχωρισμού:
  `[m+n | (m,n)<-l]`
  αποτιμάται σε `[3,17,42]` αν `l=[(1,2),(10,7),(-2,44)]`
- Μία συνάρτηση μπορεί να ορίζεται πολλές φορές. Το πρώτο μοτίβο που ταιριάζει χρησιμοποιείται

- Χρήση σε ορισμούς:
  `(l0,l1) = unzip [('w',1) , ('o',2) , ('r',3)]`
  ισοδύναμο με:
  `l0="wor"`
  `l1=[1,2,3]`
  - χρήσιμο σε μηχανισμούς τοπικών ονομάτων
- Χρήση σε εκφράσεις διαχωρισμού:
  `[m+n | (m,n)<-l]`
  αποτιμάται σε `[3,17,42]` αν `l=[(1,2),(10,7),(-2,44)]`
- Μία συνάρτηση μπορεί να ορίζεται πολλές φορές. Το πρώτο μοτίβο που ταιριάζει χρησιμοποιείται

- Μεταβλητή: ταιριάζει με τα πάντα
- Σταθερά: ταιριάζει με τον εαυτό της

```
factorial 0 = 1
```

```
factorial n = n*factorial(n-1)
```

- Μοτίβο _: ταιριάζει με τα πάντα, αλλά δε δίνει όνομα στο όρισμά του

```
fst (x,_) = x
```

- Μεταβλητή: ταιριάζει με τα πάντα
- Σταθερά: ταιριάζει με τον εαυτό της

```
factorial 0 = 1
factorial n = n*factorial(n-1)
```

- Μοτίβο _: ταιριάζει με τα πάντα, αλλά δε δίνει όνομα στο όρισμά του

```
fst (x,_) = x
```

- Μεταβλητή: ταιριάζει με τα πάντα
- Σταθερά: ταιριάζει με τον εαυτό της

```
factorial 0 = 1
factorial n = n*factorial(n-1)
```

- Μοτίβο _: ταιριάζει με τα πάντα, αλλά δε δίνει όνομα στο όρισμά του

```
fst (x,_) = x
```

Μοτίβα
Μοτίβα Πλειάδας και Λίστας

- Μοτίβο Πλειάδας `(p0,p1,...)`: ταιριάζει με πλειάδα ίσου μεγέθους
- Εφαρμόζει αναδρομικά:

  `f(x,(y,_)) = ...`

  `f((1,2),(3,4))` περνάει: `x=(1,2)` και `y=3`
- Μοτίβο Λίστας `h:t` ταιριάζει με μη κενή λίστα
- Ταιριάζει την κεφαλή με το `h` και την ουρά με το `t`
- Παραδείγματα:

```
isEmpty [] = True  ;  isEmpty (_:_) = False
head (h:_) = h
addFirstTwo (x:y:_) = x+y
tail (_:t) = t
length [] = 0  ;  length (_:t) = 1+length t
sum [] = 0  ;  sum (x:xs) = x + sum xs
```

Μοτίβα
Μοτίβα Πλειάδας και Λίστας

- Μοτίβο Πλειάδας `(p0,p1,...)`: ταιριάζει με πλειάδα ίσου μεγέθους
- Εφαρμόζει αναδρομικά:
  `f(x,(y,_)) = ...`
  `f((1,2),(3,4))` περνάει: `x=(1,2)` και `y=3`
- Μοτίβο Λίστας `h:t` ταιριάζει με μη κενή λίστα
- Ταιριάζει την κεφαλή με το `h` και την ουρά με το `t`
- Παραδείγματα:

```
isEmpty [] = True  ;  isEmpty (_:_) = False
head (h:_) = h
addFirstTwo (x:y:_) = x+y
tail (_:t) = t
length [] = 0  ;  length (_:t) = 1+length t
sum [] = 0  ;  sum (x:xs) = x + sum xs
```

Μοτίβα
Μοτίβα Πλειάδας και Λίστας

- Μοτίβο Πλειάδας `(p0,p1,...)`: ταιριάζει με πλειάδα ίσου μεγέθους
- Εφαρμόζει αναδρομικά:
  `f(x,(y,_)) = ...`
  `f((1,2),(3,4))` περνάει: `x=(1,2)` και `y=3`
- Μοτίβο Λίστας `h:t` ταιριάζει με μη κενή λίστα
- Ταιριάζει την κεφαλή με το `h` και την ουρά με το `t`
- Παραδείγματα:

```
isEmpty [] = True  ;  isEmpty (_:_) = False
head (h:_) = h
addFirstTwo (x:y:_) = x+y
tail (_:t) = t
length [] = 0  ;  length (_:t) = 1+length t
sum [] = 0  ;  sum (x:xs) = x + sum xs
```

Μοτίβα
Μοτίβα Πλειάδας και Λίστας

- Μοτίβο Πλειάδας `(p0,p1,...)`: ταιριάζει με πλειάδα ίσου μεγέθους
- Εφαρμόζει αναδρομικά:
  `f(x,(y,_)) = ...`
  `f((1,2),(3,4))` περνάει: `x=(1,2)` και `y=3`
- Μοτίβο Λίστας `h:t` ταιριάζει με μη κενή λίστα
- Ταιριάζει την κεφαλή με το `h` και την ουρά με το `t`
- Παραδείγματα:

```
isEmpty [] = True  ;  isEmpty (_:_) = False
head (h:_) = h
addFirstTwo (x:y:_) = x+y
tail (_:t) = t
length [] = 0  ;  length (_:t) = 1+length t
sum [] = 0  ;  sum (x:xs) = x + sum xs
```

Μοτίβα
Μοτίβα Πλειάδας και Λίστας

- Μοτίβο Πλειάδας `(p0,p1,...)`: ταιριάζει με πλειάδα ίσου μεγέθους
- Εφαρμόζει αναδρομικά:
  `f(x,(y,_)) = ...`
  `f((1,2),(3,4))` περνάει: `x=(1,2)` και `y=3`
- Μοτίβο Λίστας `h:t` ταιριάζει με μη κενή λίστα
- Ταιριάζει την κεφαλή με το `h` και την ουρά με το `t`
- Παραδείγματα:

```
isEmpty [] = True  ;  isEmpty (_:_) = False
head (h:_) = h
addFirstTwo (x:y:_) = x+y
tail (_:t) = t
length [] = 0  ;  length (_:t) = 1+length t
sum [] = 0  ;  sum (x:xs) = x + sum xs
```

Μοτίβα

Η Έκφραση case

- Η case εισάγει ταίριασμα μοτίβων σε μία έκφραση
- Σύνταξη:

```
case έκφραση of
  μοτίβο0 -> έκφραση0
  μοτίβο1 -> έκφραση1
  ...
  μοτίβοN -> έκφρασηN
```

- Κανόνας off side ή διαχωρισμός με ;
- Παράδειγμα

```
( case l of
    [] -> -1
    (x:_) -> x
) + 1
```

ή `( case l of [] -> -1 ; (x:_) -> x ) + 1`

- Επιστρέφει την κεφαλή του `l` αυξημένη κατά 1 ή 0 αν η `l`

Μοτίβα

Η Έκφραση `case`

- Η `case` εισάγει ταίριασμα μοτίβων σε μία έκφραση
- Σύνταξη:

  `case` *έκφραση* `of`

  μοτίβο0 -> έκφραση0

  μοτίβο1 -> έκφραση1

  ...

  μοτίβοN -> έκφρασηN
- Κανόνας off side ή διαχωρισμός με `;`
- Παράδειγμα

```
( case l of
    [] -> -1
    (x:_) -> x
) + 1
```

ή `( case l of [] -> -1 ; (x:_) -> x ) + 1`

- Επιστρέφει την κεφαλή του `l` αυξημένη κατά 1 ή 0 αν η `l`

Μοτίβα

Η Έκφραση `case`

- Η `case` εισάγει ταίριασμα μοτίβων σε μία έκφραση
- Σύνταξη:

  `case` *έκφραση* `of`

  μοτίβο0 -> έκφραση0

  μοτίβο1 -> έκφραση1

  ...

  μοτίβοN -> έκφρασηN
- Κανόνας off side ή διαχωρισμός με `;`
- Παράδειγμα

```
( case l of
      [] -> -1
      (x:_) -> x
) + 1
```

ή `( case l of [] -> -1 ; (x:_) -> x ) + 1`

- Επιστρέφει την κεφαλή του `l` αυξημένη κατά 1 ή 0 αν η `l`

Μοτίβα
Η Έκφραση case

- Η `case` εισάγει ταίριασμα μοτίβων σε μία έκφραση
- Σύνταξη:

  `case` *έκφραση* `of`

  μοτίβο0 -> έκφραση0

  μοτίβο1 -> έκφραση1

  ...

  μοτίβοN -> έκφρασηN
- Κανόνας off side ή διαχωρισμός με `;`
- Παράδειγμα

```
( case l of
    [] -> -1
    (x:_) -> x
) + 1
```

ή `( case l of [] -> -1 ; (x:_) -> x ) + 1`

- Επιστρέφει την κεφαλή του `l` αυξημένη κατά 1 ή 0 αν η `l`

Μοτίβα

Η Έκφραση `case`

- Η `case` εισάγει ταίριασμα μοτίβων σε μία έκφραση
- Σύνταξη:

  `case` *έκφραση* `of`

  μοτίβο0 `->` *έκφραση0*

  μοτίβο1 `->` *έκφραση1*

  ...

  μοτίβοN `->` *έκφρασηN*
- Κανόνας off side ή διαχωρισμός με `;`
- Παράδειγμα

```
( case l of
      [] -> -1
      (x:_) -> x
) + 1
```

  ή `( case l of [] -> -1 ; (x:_) -> x ) + 1`
- Επιστρέφει την κεφαλή του `l` αυξημένη κατά `1` ή `0` αν η `l`

- Αντικρούσιμα μοτίβα (refutable patterns) στις εκφράσεις διαχωρισμού φιλτράρουν τα στοιχεία που δεν ταιριάζουν
- Παράδειγμα:

  `[(x:xs) | (x:xs)<-l]`

  επιστρέφει τη λίστα όλων των μη κενών λιστών μέσα στην `l`

- Αντικρούσιμα μοτίβα (refutable patterns) στις εκφράσεις διαχωρισμού φιλτράρουν τα στοιχεία που δεν ταιριάζουν
- Παράδειγμα:
  `[(x:xs) | (x:xs)<-l]`
  επιστρέφει τη λίστα όλων των μη κενών λιστών μέσα στην `l`

Τοπικά Ονόματα

- Ονόματα ορατά μέσα σε ένα ορισμό: `where`
- Ονόματα ορατά μέσα σε μία έκφραση: `let`
- Ανώνυμες συναρτήσεις: λ-εκφράσεις

Τοπικά Ονόματα

where

```
norm (x,y) = sqrt (xx+yy)
  where
  xx = x^2
  yy = y^2
```

- Κανόνας off-side
- Πιθανότητα φωλιάσματος
- Αποδοτικότητα με `where`: υπολογισμός μίας υπο-έκφρασης μόνο μία φορά

Τοπικά Ονόματα

let

```
(let x=1 ; y = 2 in x+y)*3
```

- Τοπικοί ορισμοί των `x,y` ορατοί μόνο στην έκφραση `x+y`
  - αποτιμάται σε `9`
- Οι ορισμοί διαχωρίζονται με `;`

Τοπικά Ονόματα

λ-εκφράσεις

- Χρήση συνάρτησης επί τόπου, χωρίς να της δώσουμε όνομα
- Σύνταξη:
  `\`*τυπική_παράμετρος*`->`*σώμα_συνάρτησης*
- Παράδειγμα:
  `\x->x+1` είναι ίση με την `(+1)`
  - αποτίμηση της:
    `(\x->x+1)(17+42) * 2`
    δίνει `120`
- Σύγκρουση ονομάτων:
  `(\x-> \y->x+y)y = \y->y+y` ΛΑΘΟΣ!
  `(\x-> \y->x+y)y = (\x-> \z->x+z)y = \z->y+z`

Τοπικά Ονόματα

λ-εκφράσεις

- Χρήση συνάρτησης επί τόπου, χωρίς να της δώσουμε όνομα
- Σύνταξη:
  \τυπική_παράμετρος->σώμα_συνάρτησης
- Παράδειγμα:
  `\x->x+1` είναι ίση με την `(+1)`
  - αποτίμηση της:
    `(\x->x+1)(17+42) * 2`
    δίνει `120`
- Σύγκρουση ονομάτων:
  `(\x-> \y->x+y)y = \y->y+y` ΛΑΘΟΣ!
  `(\x-> \y->x+y)y = (\x-> \z->x+z)y = \z->y+z`

- Χρήση συνάρτησης επί τόπου, χωρίς να της δώσουμε όνομα
- Σύνταξη:
  `\`*τυπική_παράμετρος*`->`*σώμα_συνάρτησης*
- Παράδειγμα:
  `\x->x+1` είναι ίση με την `(+1)`
  - αποτίμηση της:
    ```
    (\x->x+1)(17+42) * 2
    ```
    δίνει `120`
- Σύγκρουση ονομάτων:
  ```
  (\x-> \y->x+y)y = \y->y+y   ΛΑΘΟΣ!
  (\x-> \y->x+y)y = (\x-> \z->x+z)y = \z->y+z
  ```

- Χρήση συνάρτησης επί τόπου, χωρίς να της δώσουμε όνομα
- Σύνταξη:
  `\`*τυπική_παράμετρος*`->`*σώμα_συνάρτησης*
- Παράδειγμα:
  `\x->x+1` είναι ίση με την `(+1)`
  - αποτίμηση της:
    `(\x->x+1)(17+42) * 2`
    δίνει `120`
- Σύγκρουση ονομάτων:
  `(\x-> \y->x+y)y = \y->y+y` ΛΑΘΟΣ!
  `(\x-> \y->x+y)y = (\x-> \z->x+z)y = \z->y+z`

- Χρήση συνάρτησης επί τόπου, χωρίς να της δώσουμε όνομα
- Σύνταξη:
  `\`*τυπική_παράμετρος*`->`*σώμα_συνάρτησης*
- Παράδειγμα:
  `\x->x+1` είναι ίση με την `(+1)`
  - αποτίμηση της:
    ```
    (\x->x+1)(17+42) * 2
    ```
    δίνει `120`
- Σύγκρουση ονομάτων:
  `(\x-> \y->x+y)y = \y->y+y` ΛΑΘΟΣ!
  `(\x-> \y->x+y)y = (\x-> \z->x+z)y = \z->y+z`

- Χρήση συνάρτησης επί τόπου, χωρίς να της δώσουμε όνομα
- Σύνταξη:
  `\`*τυπική_παράμετρος*`->`*σώμα_συνάρτησης*
- Παράδειγμα:
  `\x->x+1` είναι ίση με την `(+1)`
  - αποτίμηση της:
    ```
    (\x->x+1)(17+42) * 2
    ```
    δίνει `120`
- Σύγκρουση ονομάτων:
  ```
  (\x-> \y->x+y)y = \y->y+y
  ```
  ΛΑΘΟΣ!
  ```
  (\x-> \y->x+y)y = (\x-> \z->x+z)y = \z->y+z
  ```

Παραδείγματα

Αριθμοί Fibonacci με Πλειάδες και `where`-I

- Για να υπολογίσουμε έναν αριθμό Fibonacci χρειαζόμαστε δύο προηγούμενες τιμές της συνάρτησης `fib`
- Αν έχουμε ένα ζεύγος διαδοχικών αριθμών $(f_k, f_{k\text{-}1})$ το επόμενο ζεύγος υπολογίζεται πολύ εύκολα:

$$(f_{k+1}, f_k) = (f_k + f_{k\text{-}1}, f_k)$$

- Συνάρτηση `fibpair` :

```
fibpair 0 = (1,0)
fibpair n =
  (fst (fibpair (n-1)) + snd (fibpair (n-1))
  , fst (fibpair (n-1)) )
```

- Πολύ κακή υλοποίηση!

Παραδείγματα

Αριθμοί Fibonacci με Πλειάδες και `where`-I

- Για να υπολογίσουμε έναν αριθμό Fibonacci χρειαζόμαστε δύο προηγούμενες τιμές της συνάρτησης `fib`
- Αν έχουμε ένα ζεύγος διαδοχικών αριθμών $(f_k, f_{k\text{-}1})$ το επόμενο ζεύγος υπολογίζεται πολύ εύκολα:

$$(f_{k+1}, f_k) = (f_k + f_{k\text{-}1}, f_k)$$

- Συνάρτηση `fibpair` :

```
fibpair 0 = (1,0)
fibpair n =
  (fst (fibpair (n-1)) + snd (fibpair (n-1))
  , fst (fibpair (n-1)) )
```

- Πολύ κακή υλοποίηση!

Παραδείγματα

Αριθμοί Fibonacci με Πλειάδες και `where`-I

- Για να υπολογίσουμε έναν αριθμό Fibonacci χρειαζόμαστε δύο προηγούμενες τιμές της συνάρτησης `fib`
- Αν έχουμε ένα ζεύγος διαδοχικών αριθμών $(f_k, f_{k\text{-}1})$ το επόμενο ζεύγος υπολογίζεται πολύ εύκολα:

$$(f_{k+1}, f_k) = (f_k + f_{k\text{-}1}, f_k)$$

- Συνάρτηση `fibpair` :

```
fibpair 0 = (1,0)
fibpair n =
  (fst (fibpair (n-1)) + snd (fibpair (n-1))
  , fst (fibpair (n-1)) )
```

- Πολύ κακή υλοποίηση!

Παραδείγματα

Αριθμοί Fibonacci με Πλειάδες και `where`-I

- Για να υπολογίσουμε έναν αριθμό Fibonacci χρειαζόμαστε δύο προηγούμενες τιμές της συνάρτησης `fib`
- Αν έχουμε ένα ζεύγος διαδοχικών αριθμών $(f_k, f_{k\text{-}1})$ το επόμενο ζεύγος υπολογίζεται πολύ εύκολα:

$$(f_{k+1}, f_k) = (f_k + f_{k\text{-}1}, f_k)$$

- Συνάρτηση `fibpair` :

```
fibpair 0 = (1,0)
fibpair n =
  (fst (fibpair (n-1)) + snd (fibpair (n-1))
  , fst (fibpair (n-1)) )
```

- Πολύ κακή υλοποίηση!

Παραδείγματα

Αριθμοί Fibonacci με Πλειάδες και `where`-ll

- Διόρθωση με `where`:

```
fibpair 0 = (1,0)
fibpair n = (fst p + snd p , fst p)
        where p = fibpair (n-1)
```

- Χρήση μοτίβου πλειάδας για αποφυγή `p,fst,snd`:

```
fibpair 0 = (1,0)
fibpair n = (f1 + f2 , f1)
        where (f1,f2) = fibpair (n-1)
```

- Ορισμός της `fib`:

```
fib = fst.fibpair
```

- Τελικά:

```
fib = fst.fibpair
  where fibpair 0 = (1,0)
        fibpair n = (f1 + f2 , f1)
                where (f1,f2) = fibpair (n-1)
```

Παραδείγματα

Αριθμοί Fibonacci με Πλειάδες και `where`-ll

- Διόρθωση με `where`:

```
fibpair 0 = (1,0)
fibpair n = (fst p + snd p , fst p)
        where p = fibpair (n-1)
```

- Χρήση μοτίβου πλειάδας για αποφυγή `p,fst,snd`:

```
fibpair 0 = (1,0)
fibpair n = (f1 + f2 , f1)
        where (f1,f2) = fibpair (n-1)
```

- Ορισμός της `fib`:

```
fib = fst.fibpair
```

- Τελικά:

```
fib = fst.fibpair
  where fibpair 0 = (1,0)
        fibpair n = (f1 + f2 , f1)
        where (f1,f2) = fibpair (n-1)
```

Παραδείγματα

Αριθμοί Fibonacci με Πλειάδες και `where`-ll

- Διόρθωση με `where`:

```
fibpair 0 = (1,0)
fibpair n = (fst p + snd p , fst p)
        where p = fibpair (n-1)
```

- Χρήση μοτίβου πλειάδας για αποφυγή `p,fst,snd`:

```
fibpair 0 = (1,0)
fibpair n = (f1 + f2 , f1)
        where (f1,f2) = fibpair (n-1)
```

- Ορισμός της `fib`:

```
fib = fst.fibpair
```

- Τελικά:

```
fib = fst.fibpair
  where fibpair 0 = (1,0)
        fibpair n = (f1 + f2 , f1)
                where (f1,f2) = fibpair (n-1)
```

Παραδείγματα

Αριθμοί Fibonacci με Πλειάδες και `where`-ll

- Διόρθωση με `where`:

```
fibpair 0 = (1,0)
fibpair n = (fst p + snd p , fst p)
        where p = fibpair (n-1)
```

- Χρήση μοτίβου πλειάδας για αποφυγή `p,fst,snd`:

```
fibpair 0 = (1,0)
fibpair n = (f1 + f2 , f1)
        where (f1,f2) = fibpair (n-1)
```

- Ορισμός της `fib`:

```
fib = fst.fibpair
```

- Τελικά:

```
fib = fst.fibpair
  where fibpair 0 = (1,0)
        fibpair n = (f1 + f2 , f1)
            where (f1,f2) = fibpair (n-1)
```

Παραδείγματα
Insertion Sort - I

- Insert: παίρνουμε έναν αριθμό και μία ταξινομημένη λίστα και εισάγουμε τον αριθμό στη σωστή θέση
- Insertion Sort: κάνουμε Insert διαδοχικά σε κάθε στοιχείο μίας λίστας από το τέλος προς την αρχή, μέχρι η λίστα να ταξινομηθεί

```
[3,1,2,0]
[3,1,2,0]
[3,1,0,2]
[3,0,1,2]
[0,1,2,3]
```

Παραδείγματα
Insertion Sort - I

- Insert: παίρνουμε έναν αριθμό και μία ταξινομημένη λίστα και εισάγουμε τον αριθμό στη σωστή θέση
- Insertion Sort: κάνουμε Insert διαδοχικά σε κάθε στοιχείο μίας λίστας από το τέλος προς την αρχή, μέχρι η λίστα να ταξινομηθεί

```
[3,1,2,0]
[3,1,2,0]
[3,1,0,2]
[3,0,1,2]
[0,1,2,3]
```

Παραδείγματα
Insertion Sort - I

- Insert: παίρνουμε έναν αριθμό και μία ταξινομημένη λίστα και εισάγουμε τον αριθμό στη σωστή θέση
- Insertion Sort: κάνουμε Insert διαδοχικά σε κάθε στοιχείο μίας λίστας από το τέλος προς την αρχή, μέχρι η λίστα να ταξινομηθεί

```
[3,1,2,0]
[3,1,2,0]
[3,1,0,2]
[3,0,1,2]
[0,1,2,3]
```

Παραδείγματα
Insertion Sort - I

- Insert: παίρνουμε έναν αριθμό και μία ταξινομημένη λίστα και εισάγουμε τον αριθμό στη σωστή θέση
- Insertion Sort: κάνουμε Insert διαδοχικά σε κάθε στοιχείο μίας λίστας από το τέλος προς την αρχή, μέχρι η λίστα να ταξινομηθεί

```
[3,1,2,0]
[3,1,2,0]
[3,1,0,2]
[3,0,1,2]
[0,1,2,3]
```

Παραδείγματα
Insertion Sort - I

- Insert: παίρνουμε έναν αριθμό και μία ταξινομημένη λίστα και εισάγουμε τον αριθμό στη σωστή θέση
- Insertion Sort: κάνουμε Insert διαδοχικά σε κάθε στοιχείο μίας λίστας από το τέλος προς την αρχή, μέχρι η λίστα να ταξινομηθεί

```
[3,1,2,0]
[3,1,2,0]
[3,1,0,2]
[3,0,1,2]
[0,1,2,3]
```

- Insert: παίρνουμε έναν αριθμό και μία ταξινομημένη λίστα και εισάγουμε τον αριθμό στη σωστή θέση
- Insertion Sort: κάνουμε Insert διαδοχικά σε κάθε στοιχείο μίας λίστας από το τέλος προς την αρχή, μέχρι η λίστα να ταξινομηθεί

```
[3,1,2,0]
[3,1,2,0]
[3,1,0,2]
[3,0,1,2]
[0,1,2,3]
```

- Insert: παίρνουμε έναν αριθμό και μία ταξινομημένη λίστα και εισάγουμε τον αριθμό στη σωστή θέση
- Insertion Sort: κάνουμε Insert διαδοχικά σε κάθε στοιχείο μίας λίστας από το τέλος προς την αρχή, μέχρι η λίστα να ταξινομηθεί

```
[3,1,2,0]
[3,1,2,0]
[3,1,0,2]
[3,0,1,2]
[0,1,2,3]
```

Παραδείγματα
Insertion Sort - II

- Insert στη Haskell:

```
ins :: Int->[Int]->[Int]
ins i [] = [i]
ins i (x:xs)
  | i<=x = i:x:xs
  | True = x:(ins i xs)
```

- Insertion Sort στη Haskell:

```
iSort :: [Int]->[Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

Παραδείγματα
Insertion Sort - II

- Insert στη Haskell:

```
ins :: Int->[Int]->[Int]
ins i [] = [i]
ins i (x:xs)
  | i<=x = i:x:xs
  | True = x:(ins i xs)
```

- Insertion Sort στη Haskell:

```
iSort :: [Int]->[Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

Παραδείγματα
Insertion Sort - II

- Insert στη Haskell:

```
ins :: Int->[Int]->[Int]
ins i [] = [i]
ins i (x:xs)
  | i<=x = i:x:xs
  | True = x:(ins i xs)
```

- Insertion Sort στη Haskell:

```
iSort :: [Int]->[Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

Παραδείγματα
Insertion Sort - II

- Insert στη Haskell:

```
ins :: Int->[Int]->[Int]
ins i [] = [i]
ins i (x:xs)
 | i<=x = i:x:xs
 | True = x:(ins i xs)
```

- Insertion Sort στη Haskell:

```
iSort :: [Int]->[Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

Παραδείγματα
Insertion Sort - II

- Insert στη Haskell:

```
ins :: Int->[Int]->[Int]
ins i [] = [i]
ins i (x:xs)
 | i<=x = i:x:xs
 | True = x:(ins i xs)
```

- Insertion Sort στη Haskell:

```
iSort :: [Int]->[Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

Παραδείγματα
Insertion Sort - II

- Insert στη Haskell:

```
ins :: Int->[Int]->[Int]
ins i [] = [i]
ins i (x:xs)
 | i<=x = i:x:xs
 | True = x:(ins i xs)
```

- Insertion Sort στη Haskell:

```
iSort :: [Int]->[Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

Παραδείγματα
Insertion Sort - II

- Insert στη Haskell:

```
ins :: Int->[Int]->[Int]
ins i [] = [i]
ins i (x:xs)
 | i<=x = i:x:xs
 | True = x:(ins i xs)
```

- Insertion Sort στη Haskell:

```
iSort :: [Int]->[Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

- Insert στη Haskell:

```
ins :: Int->[Int]->[Int]
ins i [] = [i]
ins i (x:xs)
 | i<=x = i:x:xs
 | True = x:(ins i xs)
```

- Insertion Sort στη Haskell:

```
iSort :: [Int]->[Int]
iSort [] = []
iSort (x:xs) =  ins x (iSort xs)
```

Παραδείγματα
Quick Sort - I

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3] (αναδρομική κλήση)
[0,1,2,3]
```

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3] (αναδρομική κλήση)
[0,1,2,3]
```

Παραδείγματα
Quick Sort - I

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3] (αναδρομική κλήση)
[0,1,2,3]
```

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3] (αναδρομική κλήση)
[0,1,2,3]
```

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3] (αναδρομική κλήση)
[0,1,2,3]
```

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3] (αναδρομική κλήση)
[0,1,2,3]
```

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3] (αναδρομική κλήση)
[0,1,2,3]
```

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3]  (αναδρομική κλήση)
[0,1,2,3]
```

- Quick Sort (Tony Hoare):
  - διαλέγουμε ένα στοιχείο `x` της λίστας
  - σπάμε τη λίστα σε δύο λίστες: `<=x` και `>x`
  - εφαρμόζουμε αναδρομικά την Quick Sort στις λίστες
  - επανενώνουμε

```
[2,1,3,0]
[1,0] ++ 2 ++ [3]
[0,1] ++ 2 ++ [3]  (αναδρομική κλήση)
[0,1,2,3]
```

- Υλοποίηση στη Haskell:

```
qSort:[Int]->[Int]
qSort [] = []
qSort (x:xs) =
    qSort[y | y<-xs, y<=x] ++ [x]
 ++ qSort[y | y<-xs, y>x]
```

- Εναλλακτική υλοποίηση με `filter`:

```
qSort (x:xs) =
    qSort(filter (<=x) xs) ++ [x]
 ++ qSort(filter (>x) xs)
```

- Υλοποίηση στη Haskell:

```
qSort:[Int]->[Int]
qSort [] = []
qSort (x:xs) =
    qSort[y | y<-xs, y<=x] ++ [x]
 ++ qSort[y | y<-xs, y>x]
```

- Εναλλακτική υλοποίηση με `filter`:

```
qSort (x:xs) =
    qSort(filter (<=x) xs) ++ [x]
 ++ qSort(filter (>x) xs)
```

- Υλοποίηση στη Haskell:

```
qSort:[Int]->[Int]
qSort [] = []
qSort (x:xs) =
    qSort[y | y<-xs, y<=x] ++ [x]
 ++ qSort[y | y<-xs, y>x]
```

- Εναλλακτική υλοποίηση με `filter`:

```
qSort (x:xs) =
    qSort(filter (<=x) xs) ++ [x]
 ++ qSort(filter (>x) xs)
```

- Υλοποίηση στη Haskell:

```
qSort:[Int]->[Int]
qSort [] = []
qSort (x:xs) =
    qSort[y | y<-xs, y<=x] ++ [x]
 ++ qSort[y | y<-xs, y>x]
```

- Εναλλακτική υλοποίηση με `filter`:

```
qSort (x:xs) =
    qSort(filter (<=x) xs) ++ [x]
 ++ qSort(filter (>x) xs)
```

Παραδείγματα
Γραμμική Αναζήτηση

- Πρόβλημα: δίνεται λίστα `l` και στοιχείο `x`. Υπάρχει το `x` στην `l`;

  ```
  lSearch :: Int->[Int]->Bool
  ```

- Λύση με γραμμική αναζήτηση και συσσωρευτικό υπολογισμό (`foldr`)
- Πρόκειται για συσσωρευτικό υπολογισμό διάζευξης
- Εφαρμόζουμε στη λίστα που δημιουργείται από τη σύγκριση όλων των στοιχείων της `l` με `x`

  ```
  lSearch x l =
    foldr False (||) (map (x==) l)
  ```

- Οκνηρή αποτίμηση $\rightarrow$ όντως γραμμική αναζήτηση!

  ```
  nat = [0]++[x+1 | x<-nat]
  ```

  αποτιμήστε τα `lSearch 4 nat` και `lSearch (-1) nat`

Παραδείγματα
Γραμμική Αναζήτηση

- Πρόβλημα: δίνεται λίστα `l` και στοιχείο `x`. Υπάρχει το `x` στην `l`;

  ```
  lSearch :: Int->[Int]->Bool
  ```

- Λύση με γραμμική αναζήτηση και συσσωρευτικό υπολογισμό (`foldr`)
- Πρόκειται για συσσωρευτικό υπολογισμό διάζευξης
- Εφαρμόζουμε στη λίστα που δημιουργείται από τη σύγκριση όλων των στοιχείων της `l` με `x`

  ```
  lSearch x l =
    foldr False (||) (map (x==) l)
  ```

- Οκνηρή αποτίμηση $\rightarrow$ όντως γραμμική αναζήτηση!

  ```
  nat = [0]++[x+1 | x<-nat]
  ```

  αποτιμήστε τα `lSearch 4 nat` και `lSearch (-1) nat`

- Πρόβλημα: δίνεται λίστα `l` και στοιχείο `x`. Υπάρχει το `x` στην `l`;

  ```
  lSearch :: Int->[Int]->Bool
  ```

- Λύση με γραμμική αναζήτηση και συσσωρευτικό υπολογισμό (`foldr`)
- Πρόκειται για συσσωρευτικό υπολογισμό διάζευξης
- Εφαρμόζουμε στη λίστα που δημιουργείται από τη σύγκριση όλων των στοιχείων της `l` με `x`

  ```
  lSearch x l =
    foldr False (||) (map (x==) l)
  ```

- Οκνηρή αποτίμηση → όντως γραμμική αναζήτηση!

  ```
  nat = [0]++[x+1 | x<-nat]
  ```

  αποτιμήστε τα `lSearch 4 nat` και `lSearch (-1) nat`

Παραδείγματα
Γραμμική Αναζήτηση

- Πρόβλημα: δίνεται λίστα `l` και στοιχείο `x`. Υπάρχει το `x` στην `l`;

```
lSearch :: Int->[Int]->Bool
```

- Λύση με γραμμική αναζήτηση και συσσωρευτικό υπολογισμό (`foldr`)
- Πρόκειται για συσσωρευτικό υπολογισμό διάζευξης
- Εφαρμόζουμε στη λίστα που δημιουργείται από τη σύγκριση όλων των στοιχείων της `l` με `x`

```
lSearch x l =
  foldr False (||) (map (x==) l)
```

- Οκνηρή αποτίμηση → όντως γραμμική αναζήτηση!

```
nat = [0]++[x+1 | x<-nat]
```

αποτιμήστε τα `lSearch 4 nat` και `lSearch (-1) nat`

- Πρόβλημα: δίνεται λίστα `l` και στοιχείο `x`. Υπάρχει το `x` στην `l`;

  ```
  lSearch :: Int->[Int]->Bool
  ```

- Λύση με γραμμική αναζήτηση και συσσωρευτικό υπολογισμό (`foldr`)
- Πρόκειται για συσσωρευτικό υπολογισμό διάζευξης
- Εφαρμόζουμε στη λίστα που δημιουργείται από τη σύγκριση όλων των στοιχείων της `l` με `x`

  ```
  lSearch x l =
    foldr False (||) (map (x==) l)
  ```

- Οκνηρή αποτίμηση $\rightarrow$ όντως γραμμική αναζήτηση!

  ```
  nat = [0]++[x+1 | x<-nat]
  ```

  αποτιμήστε τα `lSearch 4 nat` και `lSearch (-1) nat`

Παραδείγματα
Δυαδική Αναζήτηση

- Λίστα εισόδου: πεπερασμένη και ταξινομημένη
- Ελέγχουμε το μεσαίο στοιχείο και συνεχίζουμε στο σωστό μισό της λίστας
  - δύο αρχικοί ορισμοί
- Παίρνουμε τα πρώτα `m` στοιχεία με τη `take`
- Και τα υπόλοιπα με την `drop`

```
bSearch :: Int->[Int]->Bool
bSearch [] = False
bSearch x (y:[]) = x==y
bSearch x l
   | l!!m>x = bSearch x (take m l)
   | True   = bSearch x (drop m l)
   where m = length l `div` 2
```

Παραδείγματα
Δυαδική Αναζήτηση

- Λίστα εισόδου: πεπερασμένη και ταξινομημένη
- Ελέγχουμε το μεσαίο στοιχείο και συνεχίζουμε στο σωστό μισό της λίστας
  - δύο αρχικοί ορισμοί
- Παίρνουμε τα πρώτα `m` στοιχεία με τη `take`
- Και τα υπόλοιπα με την `drop`

```
bSearch :: Int->[Int]->Bool
bSearch [] = False
bSearch x (y:[]) = x==y
bSearch x l
   | l!!m>x = bSearch x (take m l)
   | True   = bSearch x (drop m l)
   where m = length l `div` 2
```

Παραδείγματα
Δυαδική Αναζήτηση

- Λίστα εισόδου: πεπερασμένη και ταξινομημένη
- Ελέγχουμε το μεσαίο στοιχείο και συνεχίζουμε στο σωστό μισό της λίστας
  - δύο αρχικοί ορισμοί
- Παίρνουμε τα πρώτα `m` στοιχεία με τη `take`
- Και τα υπόλοιπα με την `drop`

```
bSearch :: Int->[Int]->Bool
bSearch [] = False
bSearch x (y:[]) = x==y
bSearch x l
    | l!!m>x = bSearch x (take m l)
    | True   = bSearch x (drop m l)
    where m = length l `div` 2
```

Παραδείγματα
Δυαδική Αναζήτηση

- Λίστα εισόδου: πεπερασμένη και ταξινομημένη
- Ελέγχουμε το μεσαίο στοιχείο και συνεχίζουμε στο σωστό μισό της λίστας
  - δύο αρχικοί ορισμοί
- Παίρνουμε τα πρώτα `m` στοιχεία με τη `take`
- Και τα υπόλοιπα με την `drop`

```
bSearch :: Int->[Int]->Bool
bSearch [] = False
bSearch x (y:[]) = x==y
bSearch x l
   | l!!m>x = bSearch x (take m l)
   | True   = bSearch x (drop m l)
   where m = length l `div` 2
```

Παραδείγματα
Δυαδική Αναζήτηση

- Λίστα εισόδου: πεπερασμένη και ταξινομημένη
- Ελέγχουμε το μεσαίο στοιχείο και συνεχίζουμε στο σωστό μισό της λίστας
  - δύο αρχικοί ορισμοί
- Παίρνουμε τα πρώτα `m` στοιχεία με τη `take`
- Και τα υπόλοιπα με την `drop`

```
bSearch :: Int->[Int]->Bool
bSearch [] = False
bSearch x (y:[]) = x==y
bSearch x l
   | l!!m>x = bSearch x (take m l)
   | True   = bSearch x (drop m l)
   where m = length l `div` 2
```

Επισκόπηση - I

- Πλειάδες: συλλογές συγκεκριμένου αριθμού ετερογενών δεδομένων
- Συνωνόματοι τύποι: λέξη-κλειδί `type`
- Λίστες: συλλογές όχι προκαθορισμένου αριθμού δεδομένων του ιδίου τύπου
  - σειρά και αριθμός εμφάνισης σημαντικά
- Συμβολοσειρές: λίστες χαρακτήρων
- Διαχωρισμός: δημιουργία λίστας με μετασχηματισμό και φιλτράρισμα των στοιχείων άλλων λιστών
- Συναρτήσεις της Haskell για λίστες

Επισκόπηση - II

- Μοτίβα και ταίριασμα μοτίβων: ταιριάζουν ονόματα κατευθείαν σε υποσύνολα μη τετριμμένων δομών δεδομένων
  - μπαίνουν σε τυπικές παραμέτρους, ορισμούς, εκφράσεις διαχωρισμού, δομές `case`
  - τα μοτίβα σε εκφράσεις διαχωρισμού είναι αντικρούσιμα
  - μοτίβα Haskell: μεταβλητές, σταθερές, το μοτίβο `_`, μοτίβα πλειάδων, μοτίβα λιστών
- Απόκρυψη ονομάτων: βοηθάει στην τμηματοποίηση του κώδικα και στην αποδοτικότητα των υπολογισμών
  - μηχανισμοί: `where`, `let` και λ-εκφράσεις
  - προσοχή στον κανόνα αποφυγής συγκρούσεων ονομάτων
- Παραδείγματα