Haskell: Βασικές Δομές Δεδομένων και Απόκρυψη Ονομάτων

Γιάννης Κασσιός

Σε αυτές τις σημειώσεις, θα εισάγουμε τις δύο βασικές δομές δεδομένων που υποστηρίζει η Haskell, τις *πλειάδες (tuples)* και τις *λίστες (lists)*. Οι πλειάδες (Ενότ. 1) είναι ο πρωταρχικός τρόπος συγκέντρωσης σε ένα στοιχείο συγκεκριμένου αριθμού δεδομένων, που δεν έχουν απαραίτητα τον ίδιο τύπο. Σε άλλες γλώσσες, το ρόλο αυτό τον έχουν οι *εγγραφές (records)*, οι *δομές (structures)* ή τα *αντικείμενα (objects)*. Οι λίστες (Ενότ. 2) είναι η δομή που συγκεντρώνει μη καθορισμένο αριθμό ομοειδών δεδομένων. Σε άλλες γλώσσες, το ρόλο αυτό τον έχουν οι *πίνακες (arrays)*.

Οι πλειάδες και οι λίστες είναι δεδομένα που αποτελούνται από άλλα απλούστερα δεδομένα (π.χ. βασικούς τύπους ή απλούστερες πλειάδες/λίστες κτλ.). Δηλαδή, σε αντίθεση με τους βασικούς τύπους, έχουν μη τετριμμένη δομή. Είναι βολικό συναρτήσεις που ορίζονται πάνω σε δομές δεδομένων να έχουν πρόσβαση κατευθείαν στη δομή τους. Αυτό επιτυγχάνεται με τα *μοτίβα (patterns)* και τη διαδικασία του *ταιριάσματος μοτίβων (pattern matching)* στις τυπικές παραμέτρους των ορισμών της Haskell. Ο ισχυρός μηχανισμός των μοτίβων της Haskell είναι το αντικείμενο της Ενότ. 3.

Στην Ενότ. 4 εισάγουμε τρεις μηχανισμούς που χρησιμοποιεί η Haskell για *απόκρυψη ονομάτων*. Οι μηχανισμοί αυτοί είναι χρήσιμοι στην τμηματοποίηση των προγραμμάτων καθώς αποφεύγουν ανούσιες συγκρούσεις ονομάτων.

Τέλος, η Ενότ. 5 περιέχει παραδείγματα χρήσης πλειάδων, λιστών, μοτίβων και απόκρυψης ονομάτων. Πολλά από τα παραδείγματα σε αυτές τις σημειώσεις είναι από το βιβλίο [Tho99].

1 Πλειάδες

1.1 Βασικά των Πλειάδων

Μία *πλειάδα (tuple)* είναι ένα αντικείμενο που περιέχει ένα πεπερασμένο αριθμό άλλων δεδομένων, όχι απαραίτητα του ιδίου τύπου. Για να φτιάξουμε μία πλειάδα, χρησιμοποιούμε την ακόλουθη σύνταξη:

(*δεδομένο0*,*δεδομένο1*,...*δεδομένοN*)

Τα δεδομένα μπορεί να έχουν βασικό τύπο, αλλά μπορεί και να είναι πιο πολύπλοκες δομές δεδομένων, πχ. συναρτήσεις, άλλες πλειάδες ή λίστες (βλ. Ενότ. 2). Για παράδειγμα, τα ακόλουθα είναι πλειάδες:

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

Στο δεύτερο παράδειγμα, έχουμε μία πλειάδα μέσα στην πλειάδα. Στο τρίτο έχουμε μία συνάρτηση (`f.g`) ως πρώτο στοιχείο της πλειάδας.

Οι πλειάδες είναι ένας βολικός τρόπος να συσχετίσουμε ετερογενή δεδομένα που έχουν να κάνουν με μία οντότητα που μας ενδιαφέρει. Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να φτιάξουμε ένα πρόγραμμα που να χειρίζεται εγγραφές φοιτητών και ότι για κάθε φοιτητή θέλουμε να συγκρατούμε το όνομά του, το φύλο του και την ηλικία του. Τότε είναι βολικό να αναφερόμαστε σε ένα φοιτητή ως μία πλειάδα τριών στοιχείων:

- Το πρώτο στοιχείο είναι μία συμβολοσειρά (βλ. Ενότ. 2.2) που περιέχει το όνομα του φοιτητή
- Το δεύτερο στοιχείο είναι μία αληθοτιμή που είναι π.χ. `True` για φοιτήτρια και `False` για φοιτητή
- Το τρίτο στοιχείο είναι ένας θετικός ακέραιος

Παράδειγμα τέτοιων πλειάδων είναι:

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

κτλ.

1.2 Τύποι Πλειάδων και Ορισμοί Τύπων

Ο τύπος μίας πλειάδας `(v0,v1,...)` γράφεται `(t0,t1,...)`, όπου `t0,t1,...` είναι οι τύποι των `v0,v1,...` αντίστοιχα. Για παράδειγμα, ο τύπος της πλειάδας `('a',(20,10,True))` είναι `(Char,(Int,Int,Bool))`. Παρατηρήστε ότι πρόκειται για μία πλειάδα δύο στοιχείων, της οποίας το πρώτο είναι ένας χαρακτήρας και το δεύτερο είναι μία πλειάδα τριών στοιχείων. Έτσι και ο τύπος της είναι μία πλειάδα δύο τύπων, ο πρώτος είναι ο τύπος `Char`, ενώ ο δεύτερος είναι ο τύπος της εσωτερικής πλειάδας που είναι `(Int,Int,Bool)`.

Βλέπουμε πως τύποι της Haskell μπορούν να *κατασκευαστούν* με βάση απλούστερους τύπους (το είδαμε αυτό και στις συναρτήσεις, όπου τύποι κατασκευάζονται με τον τελεστή `->`). Καθώς έχουμε πλέον πολύπλοκες δομές δεδομένων, είναι καλό να μπορούμε να δίνουμε ονόματα σε πολύπλοκους τύπους που εμφανίζονται πολύ συχνά στο πρόγραμμά μας. Αυτό μπορεί να γίνει μέσα σε ένα πρόγραμμα Haskell με τον *ορισμό τύπου (type definition)* που εισάγει η λέξη-κλειδί `type`:

`type` *όνομαΤύπου=ορισμός*

Τα ονόματα που δίνουμε στους τύπους, αντίθετα με αυτά που δίνουμε στις τιμές, πρέπει να αρχίζουν με *κεφαλαίο γράμμα*.

Για παράδειγμα, ο τύπος "φοιτητής" (`Student`), που συναντήσαμε στην προηγούμενη ενότητα, αντιπροσωπεύεται από μία πλειάδα τριών τύπων: του τύπου `String` (συμβολοσειρά - βλ. Ενότ 2.2), του τύπου `Bool` και του τύπου `Int`. Έτσι μπορούμε να ορίσουμε τον τύπο `Student` ως εξής:

```
type Student = (String,Bool,Int)
```

Ο τύπος `Student` από εδώ και στο εξής θα είναι *συνώνυμος* του τύπου `(String,Bool,Int)`.

Φυσικά, η δυνατότητα ορισμού τύπων δεν περιορίζεται σε πλειάδες, αλλά σε οποιονδήποτε τύπο. Μπορούμε να ορίσουμε τύπους συναρτήσεων, π.χ.

```
type UnaryOp = Int->Int
```

τύπους λιστών (βλ. Ενότ. 2) ή ακόμα και νέα ονόματα για βασικούς τύπους.

1.3 Συναρτήσεις Σχετικές με Πλειάδες

Η Haskell υποστηρίζει δύο συναρτήσεις σχετικές με πλειάδες, τις `fst` και `snd`. Η `fst` επιστρέφει το πρώτο στοιχείο μίας πλειάδας δύο στοιχείων, ενώ η `snd` το δεύτερο. Για παράδειγμα:

```
fst(1 , 'a')
```

αποτιμάται σε `1`.

Φυσικά αυτές οι συναρτήσεις δεν είναι αρκετές για να χειριστούμε πλειάδες. Εδώ υπεισέρχεται ο πιο ισχυρός μηχανισμός των *μοτίβων* που θα εξετάσουμε στην Ενότ 3.

2 Λίστες

2.1 Βασικά των Λιστών

Μία *λίστα (list)* είναι μία σειρά δεδομένων του ιδίου τύπου. Ο αριθμός των δεδομένων αυτών δεν είναι προκαθορισμένος και μπορεί να είναι πεπερασμένος ή και άπειρος. Τα δεδομένα αυτά λέγονται *στοιχεία (elements)* της λίστας.

Η Haskell παρέχει πολλούς τρόπους κατασκευής λιστών, ο πιο βασικός από τους οποίους είναι η απλή απαρίθμηση των δεδομένων της λίστας μέσα σε αγκύλες και χωρισμένα με κόμμα:
`[`*δεδομένο0*`,`*δεδομένο1*`,`*...δεδομένοN*`]`
Για παράδειγμα, οι παρακάτω εκφράσεις αναπαριστούν λίστες:

```
[2,4]
['a']
```

```
[]
[2.1,1.5,6.0,2.1]
```

Το τρίτο παράδειγμα `[]` είναι η *κενή λίστα*, μία λίστα χωρίς στοιχεία. Το δεύτερο παράδειγμα είναι μία λίστα με ένα μόνο στοιχείο, το `'a'`. Προσέξτε ότι η λίστα με ένα στοιχείο *δεν* ισοδυναμεί με αυτό το στοιχείο (δηλ. τα `'a'` και `['a']` δεν είναι ίσα).

Παρατηρήστε ότι τα στοιχεία κάθε λίστας ανήκουν σε ένα μόνο τύπο. Αυτός ο τύπος δεν είναι κατ' ανάγκη βασικός. Θα μπορούσαμε για παράδειγμα να έχουμε λίστες πλειάδων, λιστών ή συναρτήσεων:

```
[(1,2,True),(4,5,False)]
[[],['h'],['h','i']]
```

Πρέπει επίσης να σημειώσουμε, πως η *σειρά* και ο *αριθμός εμφάνισης* των στοιχείων έχουν σημασία. Οι παρακάτω λίστες *δεν* είναι ίσες μεταξύ τους:

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

Υπάρχουν ειδικοί κατασκευαστές για μερικές κατηγορίες λιστών βασικού τύπου. Αν οι εκφράσεις `b` και `e` είναι και οι δύο τύπου `Int` ή `Float` ή `Char`, τότε η έκφραση

```
[b..e]
```

αναπαριστά τη λίστα που περιέχει τα στοιχεία από `b` μέχρι `e` στη σειρά (για `Float` το βήμα είναι `1.0`). Για παράδειγμα, οι παρακάτω ισοδυναμίες ισχύουν:

```
[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]
```

Σε περίπτωση που γράψουμε και το δεύτερο στοιχείο της λίστας, τότε αυτό χρησιμοποιείται για να καθοριστεί η απόσταση από κάθε στοιχείο στο γειτονικό του. Το βήμα αυτό μπορεί να είναι και αρνητικό. Για παράδειγμα:

```
[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]
```

2.2 Τύποι Λιστών και Συμβολοσειρές

Αν T είναι ένας τύπος, τότε ο τύπος των λιστών των οποίων τα στοιχεία είναι τύπου T είναι `[`T`]`. Για παράδειγμα, ο τύπος λιστών ακεραίων αριθμών είναι ο `[Int]`. Ο τύπος λιστών συναρτήσεων από ακέραιους σε ακέραιους είναι `[Int->Int]`.

Δισδιάστατες (και αντίστοιχα πολυδιάστατες) λίστες, μπορούν να αναπαρασταθούν ως λίστες λιστών. Για παράδειγμα, ο τύπος δισδιάστατων λιστών ακεραίων είναι `[[Int]]` και ένα παράδειγμα στοιχείου αυτού του τύπου είναι:

```
[ [1,2,3],
  [4,5,6]]
```

Η κενή λίστα `[]` είναι η μόνη λίστα που ανήκει *σε όλους τους τύπους λιστών*.

Μία λίστα χαρακτήρων λέγεται και *συμβολοσειρά*. Ο τύπος των συμβολοσειρών είναι επομένως ο `[Char]`, αλλά στη Haskell αυτός ο τύπος έχει και το συνώνυμο `String`:

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

Σταθερές τύπου `String` μπορούν να κατασκευαστούν στη Haskell, περικλείοντας το περιεχόμενό τους μέσα σε διπλά εισαγωγικά (`"`). Για παράδειγμα, οι παρακάτω ισοδυναμίες ισχύουν:

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

Παρατηρήστε πώς μπορούμε να χρησιμοποιήσουμε τους ειδικούς χαρακτήρες που εισάγονται με `\` και μέσα στις σταθερές τύπου `String`.

2.3 Διαχωρισμός

Η κατασκευή μίας λίστας με *διαχωρισμό (list comprehension)* είναι ένα από τα πιο δυνατά χαρακτηριστικά της Haskell.

Η βασική ιδέα είναι η κατασκεύη μίας λίστας μετασχηματίζοντας τα στοιχεία μίας άλλης. Ένα πολύ απλό παράδειγμα είναι το εξής: έστω ότι έχουμε μία λίστα `L` και θέλουμε να πάρουμε τη λίστα που αποτελείται από τα στοιχεία της `L` διπλασιασμένα. Για παράδειγμα, αν

```
l=[3,5,6]
```

τότε θέλουμε το αποτέλεσμα να είναι ίσο με `[6,10,12]`. Η έκφραση που κατασκευάζει αυτή τη λίστα στη Haskell είναι η εξής:

```
[2*n | n<-l]
```

Η έκφραση μπορεί να κατανοηθεί ως εξής: για κάθε αντικείμενο `n` της λίστας `l`, το αποτέλεσμά μας περιέχει την τιμή `2*n`. Η υποέκφραση `n<-l` λέγεται *γεννήρια (generator)*. Η γεννήτρια δίνει στην τοπική μεταβλητή `n` με τη σειρά όλες τις τιμές των στοιχείων που περιέχονται στην `l`. Η υποέκφραση `2*n` είναι η *έκφραση μετασχηματισμού*. Τα στοιχεία `n` της `l` μετασχηματίζονται σε `2*n` στην καινούρια λίστα.

Στο παράδειγμά μας, η `l` παίρνει την τιμή `[3,5,6]`. Αυτό σημαίνει ότι η `n` παίρνει διαδοχικά τις τιμές 3, 5 και 6 και άρα η έκφραση μετασχηματισμού `2*n` παίρνει διαδοχικά τις τιμές 6, 10 και 12. Το αποτέλεσμα είναι η λίστα όλων των τιμών που παίρνει η έκφραση μετασχηματισμού, δηλαδή η λίστα `[6,10,12]`.

Στις εκφράσεις διαχωρισμού, μπορούμε να προσθέσουμε *συνθήκες ελέγχου* που φιλτράρουν τα δεδομένα που παράγει η γεννήτρια. Οι συνθήκες ελέγχου είναι εκφράσεις τύπου `Bool`. Αν ένα από τα στοιχεία που παράγονται κάνει μία συνθήκη ελέγχου ψευδή, τότε δεν περνάει στην έκφραση μετασχηματισμού. Για παράδειγμα, μπορούμε να παράγουμε τη λίστα που περιέχει τα διπλάσια *μόνο των περιττών στοιχείων* της `l` ως εξής:

```
[2*n | n<-l , n `mod` 2 == 1]
```

Στο παράδειγμά μας, αν `l=[3,5,6]`, τότε η παραπάνω έκφραση θα αποτιμηθεί σε `[6,10]`. Το στοιχείο 6 δε θα παράγει αντίστοιχο στοιχείο στο αποτέλεσμα, γιατί θα φιλτραριστεί από την συνθήκη ελέγχου `n `mod` 2 == 1` που παίρνει τιμή `False` για `n=6`.

Φυσικά, η έκφραση μετασχηματισμού μπορεί να είναι απλά η μεταβλητή της γεννήτριας. Στην προκειμένη περίπτωση έχουμε μόνο φιλτράρισμα. Το παρακάτω παράδειγμα αποτιμάται στη λίστα όλων των περιττών στοιχείων της `l`:

```
[n | n<-l , n `mod` 2 == 1]
```

Στις γεννήτριες επιτρέπονται και τα μοτίβα (βλ. Ενότ. 3). Για παράδειγμα, έστω ότι η `l` είναι μία λίστα με *ζεύγη* ακεραίων. Μπορούμε να προσθέσουμε τους ακέραιους κάθε ζεύγους και να φτιάξουμε μία καινούρια λίστα ως εξής:

```
sumL = [m+n | (m,n)<-l]
```

π.χ. αν `l=[(1,2),(10,7),(-2,44)]`, τότε `sumL = [3,17,42]`.

Τέλος, μπορούμε να χρησιμοποιήσουμε περισσότερες από μία γεννήτριες. Σε αυτήν την περίπτωση, χρησιμοποιούνται όλοι οι συνδυασμοί στοιχείων από τις λίστες των γεννητριών. Η σειρά με την οποία γίνονται αυτοί οι συνδυασμοί θα γίνει καλύτερα κατανοητή με ένα παράδειγμα. Έστω π.χ. η έκφραση διαχωρισμού:

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

και έστω `l=[1,2,3]` και `m=[4,5]`. Τότε η λίστα `n` δίνεται από

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

Η επιλογή γίνεται ως εξής: πρώτα επιλέγουμε τον πρώτο αριθμό της λίστας `l` (από την πρώτη γεννήτρια) και τον συνδυάζουμε με όλους τους αριθμούς της λίστας `m` (από τη δεύτερη γεννήτρια). Μετά επιλέγουμε το δεύτερο αριθμό της λίστας `l` και τον συνδυάζουμε με τα στοιχεία της `m` κ.ο.κ. Για περισσότερες από δύο γεννήτριες, ο κανόνας αυτός γενικεύεται με τον προφανή τρόπο.

2.4 Συναρτήσεις της Haskell για Λίστες

Η Haskell υποστηρίζει μία μεγάλη ποικιλία συναρτήσεων για λίστες. Ας δούμε μερικές από αυτές. Οι συναρτήσεις που παρουσιάζουμε είναι *πολυμορφικές* που σημαίνει ότι εφαρμόζονται σε λίστες κάθε τύπου.

- Η `(:)` προσθέτει ένα στοιχείο στην αρχή της λίστας:

```
'a':"bc" = "abc"
0:[1,2] = [0,1,2]
False:[] = [False]
```

- Η `(++)` είναι η *συνένωση (concatenation)* λιστών:

```
"Hello " ++ "World!" = "Hello World!"
[] ++ [1,4] = [1,4]
[0,1,2] ++ [2, 1, 0] = [0,1,2,2,1,0]
```

- Η `!!` επιστρέφει ένα συγκεκριμένο στοιχείο της λίστας (η αρίθμηση των στοιχείων ξεκινάει από το `0`):

```
[3,4,5]!!0 = 3
"Example"!!6 = 'e'
[n^2 | n<-[4..10]]!!2 = 36
```

- Η `concat` παίρνει μία λίστα από λίστες και τις συνενώνει σε μία:

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

- Η `length` επιστρέφει το μέγεθος μίας λίστας:

```
length"Hi!" = 3
length[] = 0
length[2] = 1
```

- Η `head` επιστρέφει το πρώτο στοιχείο μίας λίστας:

```
head l = l!!0
```

- Η `last` επιστρέφει το τελευταίο στοιχείο μίας λίστας:

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

- Η `tail` επιστρέφει όλη τη λίστα εκτός από το πρώτο στοιχείο της:

```
tail"\"this is a list\"" = "this is a list\""
```

- Η `init` επιστρέφει όλη τη λίστα εκτός από το τελευταίο στοιχείο της:

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

- Η `replicate n c` επιστρέφει μία λίστα με `n` αντίγραφα του στοιχείου `c` (το `n` είναι φυσικός αριθμός):

```
replicate 0 6.0 = []
replicate 2 True = [True , True]
replicate 7 '7' = "7777777"
```

- Η `take n` επιστρέφει μια λίστα με τα `n` πρώτα στοιχεία του ορίσματός της (το `n` είναι φυσικός αριθμός):

```
take 4 [9..50] = [9,10,11,12]
take 0 "mmm" = []
```

- Η `drop n` επιστρέφει μία λίστα απορρίπτοντας τα `n` πρώτα στοιχεία του ορίσματός της (το `n` είναι φυσικός αριθμός):

```
drop (length l) l = []
drop 3 [0,10,20,4,5,6,7] = [4..7]
```

- Η `splitAt` σπάει τη λίστα στα δύο σε μία δοθείσα θέση και επιστρέφει ένα ζεύγος από λίστες:

```
splitAt 2 "My name is Bond" = ("My" , " name is Bond")
```

- Η `reverse` αντιστρέφει τη λίστα:

```
reverse [1,2,3] = [3,2,1]
```

- Η `zip` παίρνει δύο λίστες και επιστρέφει μία λίστα ζευγών ως εξής:

  ```
  zip [2,4] [0,3,6] = [(2,0),(4,3)]
  zip "word" [1,2,3] = [('w',1) , ('o',2) , ('r',3)]
  ```

- Η `unzip` κάνει το αντίθετο, επιστρέφοντας ένα ζεύγος από λίστες:

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

- Η `map` παίρνει μία συνάρτηση και μία λίστα, και εφαρμόζει τη συνάρτηση σε κάθε στοιχείο της λίστας:

  ```
  map f l = [f x | x<-l]
  map (^2) [1..4] = [1,4,9,16]
  map reverse ["This","is","it"] = ["sihT","si","ti"]
  ```

- Η `filter` φιλτράρει μία λίστα σύμφωνα με μία συνθήκη. Η συνθήκη περνάει ως μία συνάρτηση που επιστρέφει ένα στοιχείο `Bool`:

  ```
  filter f l = [x | x<-l , f x]
  ```

  Για παράδειγμα, αν η συνάρτηση `isEven` οριστεί ως εξής:

  ```
  isEven n = n`mod`2 == 0
  ```

  τότε

  ```
  filter isEven [1..10] = [2,4,6,8,10]
  ```

- Η `zipWith` παίρνει μία συνάρτηση `f` και δύο λίστες. Αφού κάνει `zip` στις δύο λίστες, μετά εφαρμόζει τη συνάρτηση `f` σε κάθε ζεύγος του αποτελέσματος: Για παράδειγμα:

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

- Οι συναρτήσεις `foldl` και `foldr` εφαρμόζουν μία συνάρτηση συσωρευτικά στα στοιχεία μίας λίστας. Η `foldl` εφαρμόζει αριστερή προσεταιριστικότητα ενώ η `foldr` εφαρμόζει δεξιά προσεταιριστικότητα. Και οι δύο παίρνουν, εκτός από την εφαρμοζόμενη συνάρτηση, έναν αριθμό που αντιστοιχεί στο αποτέλεσμα της εφαρμογής σε κενή λίστα:

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

- Οι συναρτήσεις `takeWhile` και `dropWhile` είναι σαν τις `take` και `drop` μόνο που παίρνουν μία συνθήκη σε μορφή συνάρτησης και σταματούν να παίρνουν/απορρίπτουν στοιχεία μόλις αυτή η συνθήκη σταματήσει να ισχύει. Για παράδειγμα, θεωρείστε ότι η `isEven` έχει τον ορισμό που της δώσαμε πιο πάνω. Τότε:

```
takeWhile isEven [4,6,8,9,9] = [4,6,8]
dropWhile isEven [4,6,8,9,9] = [9,9]
```

3 Μοτίβα

Οι συναρτήσεις επιλογής, όπως οι `fst` και `snd` και η `head` κτλ., είναι πολλές φορές αρκετά άβολες στον ορισμό μίας συνάρτησης που εφαρμόζεται σε συγκεκριμένες δομές. Για παράδειγμα, σκεφτείτε την παρακάτω συνάρτηση `norm` που παίρνει σαν παράμετρο ένα ζεύγος αριθμών x και y και υπολογίζει τη συνάρτηση $\sqrt{x^2 + y^2}$:

```
norm::(Float,Float)->Float
norm p = sqrt((fst p)^2 + (snd p)^2)
```

Ο παραπάνω ορισμός είναι άκομψος. Θα προτιμούσαμε, αντί να έχουμε μία παράμετρο `p` που αναπαριστά το ζεύγος $p = (x, y)$, να χρησιμοποιούσαμε μία δομή που να δίνει κατευθείαν τα ονόματα x και y στα μέλη του ζεύγους, κάπως έτσι:

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

Αυτό θα ξεκαθάριζε τα πράγματα, γιατί θα απέφευγε τις άσκοπες εφαρμογές των συναρτήσεων επιλογής που περιπλέκουν τον ορισμό χωρίς να προσφέρουν κάτι ουσιαστικό.

Η δυνατότητα αυτή δίνεται στη Haskell με το μηχανισμό των *μοτίβων (patterns)* και του *ταιριάσματος μοτίβων (pattern matching)* που αυτή υποστηρίζει. Στο συγκεκριμένο παράδειγμα, το όρισμα `(x,y)` είναι ένα *μοτίβο* που επιτρέπεται να μπει ως τυπικό όρισμα. Μόλις γίνει εφαρμογή της συνάρτησης `norm` σε ένα ζεύγος, η Haskell θα ταιριάσει το πρότυπο `(x,y)` με την παράμετρο της `norm` (η οποία θα είναι προφανώς της μορφής `(x,y)`, αφού ο τύπος της είναι `(Float,Float)` και θα περάσει αυτόματα τις σωστές τιμές στις τυπικές παραμέτρους `x` και `y`.

Η Haskell υποστηρίζει τα εξής μοτίβα για τις τυπικές παραμέτρους:

- Μεταβλητές. Αυτή η περίπτωση είναι και η μόνη που χρησιμοποιούσαμε μέχρι τώρα
- Σταθερές τιμές
- Το ειδικό μοτίβο `_`
- Μοτίβα πλειάδων

- Μοτίβα λιστών
- Μοτίβα κατασκευαστών

Σε κάθε περίπτωση αποτίμησης εφαρμογής συνάρτησης, η Haskell προσπαθεί να ταιριάξει τα πραγματικά ορίσματα με το μοτίβο των τυπικών παραμέτρων. Σε περίπτωση επιτυχίας του ταιριάσματος, οι τυπικές παράμετροι παίρνουν τις κατάληλες τιμές.

Για παράδειγμα, ας υποθέσουμε ότι αποτιμούμε την έκφραση `norm(3.0,4.0)`. Αν ο ορισμός της `norm` είναι ο πρώτος που δώσαμε:

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

τότε το ταίριασμα γίνεται ως εξής: στην τυπική παράμετρο `p` παιρνάει η τιμή `(3.0,4.0)` και η αποτίμηση προχωράει:

```
  sqrt((fst (3.0,4.0))^2 + (snd(3.0,4.0))^2)
= sqrt(3.0^2 + 4.0^2)
= sqrt(25.0)
= 5.0
```

Αν τώρα αλλάξουμε τον ορισμό της `norm` χρησιμοποιώντας μοτίβο πλειάδας:

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

τότε η αποτίμηση του `norm(3.0,4.0)` θα ταιριάξει το μοτίβο `(x,y)` με το `(3.0,4.0)`. Τα μοτίβα ταιριάζουν και οι τιμές που περνάνε στις τυπικές παραμέτρους είναι `3.0` στην `x` και `4.0` στην `y`. Η αποτίμηση συνεχίζεται ως εξής:

```
  sqrt(x^2 + y^2)
= sqrt(3.0^2 + 4.0^2)
= sqrt(25.0)
= 5.0
```

Σε αυτό το παράδειγμα βλέπουμε ότι μία πραγματική παράμετρος μπορεί να ταιριάζει σε πολλά μοτίβα.

Στη Haskell επιτρέπεται ο ορισμός μίας συνάρτησης πολλές φορές. Σε αυτές τις περιπτώσεις, η αποτίμηση μιας εφαρμογής ψάχνει όλους τους ορισμούς με τη σειρά που γραφτήκαν μέχρι να βρεθεί το πρώτο μοτίβο που να ταιριάζει στις τυπικές παραμέτρους. Σε περίπτωση που δε βρεθεί κανένα μοτίβο που να ταιριάζει, τότε ο διερμηνέας αναφέρει λάθος.

Τα μοτίβα δε χρησιμοποιούνται μόνο στις τυπικές παραμέτρους των συναρτήσεων. Χρησιμοποιούνται και στο διαχωρισμό λιστών (βλ. Ενότ. 2.3), αλλά ακόμα και στους ορισμούς ονομάτων. Για παράδειγμα, ξέρουμε ότι η συνάρτηση `unzip` επιστρέφει ένα ζεύγος λιστών. Θα μπορούσαμε να χρησιμοποιήσουμε ένα μοτίβο για να δώσουμε ονόματα και στις δύο λίστες που επιστρέφει μία κλήση της `unzip` ως εξής:

```
(l0,l1) = unzip [('w',1) , ('o',2) , ('r',3)]
```

Ο παραπάνω ορισμός είναι ισοδύναμος με:

```
l0 = fst (unzip [('w',1) , ('o',2) , ('r',3)])
l1 = snd (unzip [('w',1) , ('o',2) , ('r',3)])
```

που είναι με τη σειρά του ισοδύναμος με:

```
l0 = "wor"
l1= [1,2,3]
```

Στους καθολικούς ορισμούς, αυτό δεν είναι και τόσο χρήσιμο. Μπορεί όμως να φανεί χρήσιμο στους ορισμούς μέσα σε μια δομή `where` (βλ. Ενότ. 4).

Ας δούμε τώρα τι μοτίβα τυπικών παραμέτρων επιτρέπει η Haskell. Σε ό,τι ακολουθεί θα δούμε όλα τα μοτίβα εκτός από αυτά με τους κατασκευαστές, που θα εξετάσουμε αργότερα στο μάθημα. Επίσης θα δούμε τη χρήση μοτίβων μέσα σε εκφράσεις με τη δομή `case` και τη χρήση *αντικρούσιμων μοτίβων (refutable patterns)* στις εκφράσεις διαχωρισμού.

3.1 Μεταβλητές

Μία μεταβλητή ταιριάζει με όλες τις τυπικές παραμέτρους. Αυτό το μοτίβο είναι το μόνο που χρησιμοποιούσαμε μέχρι τώρα.

3.2 Σταθερές Τιμές και το Μοτίβο _

Μία σταθερά μπορεί να χρησιμοποιηθεί ως μοτίβο. Η σταθερά ταιριάζει μόνο με τον εαυτό της. Αυτό το χαρακτηριστικό μπορεί να χρησιμοποιηθεί ώστε ένας ορισμός συνάρτησης με φρουρούς να "σπάσει" σε πολλούς μικρότερους ορισμούς.

Για παράδειγμα, ο ορισμός της `factorial`, που είδαμε σε προηγούμενες σημειώσεις να γίνεται με φρουρούς, μπορεί να σπάσει σε δύο ορισμούς ως εξής:

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

Ο πρώτος ορισμός ταιριάζει μόνο όταν η πραγματική παράμετρος είναι το `0`. Ο δεύτερος ορισμός ταιριάζει πάντα, γιατί χρησιμοποιεί ως μοτίβο μία μεταβλητή.

Η αποτίμηση της `factorial 0` θα ταιριάξει με το μοτίβο του πρώτου ορισμού, δίνοντας `1` (ο δεύτερος ορισμός επίσης ταιριάζει αλλά αγνοείται). Η αποτίμηση της `factorial 2` θα αποτύχει να ταιριάξει με το μοτίβο του πρώτου ορισμού, αλλά θα ταιριάξει με το δεύτερο, δίνοντας `2*factorial(2-1)` που τελικά θα αποτιμηθεί σε `2`.

Το ειδικό μοτίβο _ λειτουργεί όπως οι μεταβλητές: ταιριάζει με όλα τα πραγματικά ορίσματα. Αλλά, σε αντίθεση με τις μεταβλητές, δε δίνει όνομα στο όρισμα με το οποίο ταιριάζει. Έτσι χρησιμεύει σε ορισμούς στους οποίους για κάποιο λόγο το όρισμα αυτό δεν παίρνει μέρος. Για παράδειγμα, ο ορισμός της `fst` είναι:

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

Αφού το δεύτερο μέρος του ζεύγους δεν υπάρχει πουθενά στον ορισμό, το όνομά του δε χρειάζεται.

3.3 Μοτίβα Πλειάδων

Τα μοτίβα πλειάδων είναι αυτά που φέραμε σαν παράδειγμα στην εισαγωγή. Γράφονται ως εξής:
(*μοτίβο0* `,` *μοτίβο1* `,` *...*)
Βλέπουμε ότι μέσα στις παρενθέσεις μπαίνουν άλλα μοτίβα, που σημαίνει ότι μπορούμε να γράψουμε μοτίβα με πολύπλοκη δομή όπως `(x,(y,z))`.

Ένα μοτίβο πλειάδας ταιριάζει με μία πραγματική παράμετρο που έχει τιμή πλειάδας με ακριβώς τον ίδιο αριθμό στοιχείων που έχει και το μοτίβο. Όταν γίνει αυτό, τότε κάθε στοιχείο της πραγματικής παραμέτρου ταιριάζει αναδρομικά με το αντίστοιχο υπο-μοτίβο στο μοτίβο πλειάδας.

Στην εισαγωγή της ενότητας αυτής, είδαμε ένα απλό παράδειγμα μοτίβου πλειάδας (η συνάρτηση `norm`). Τα υπο-μοτίβα σε αυτό το παράδειγμα είναι μεταβλητές `x` και `y` και το ταίριασμα με την παράμετρο `(3.0 , 4.0)` γίνεται πολύ απλά, περνόντας το `3.0` ως τιμή του `x` και το `4.0` ως τιμή του `y`. Άλλο ένα τέτοιο απλό παράδειγμα είναι ο ορισμός της `fst` πιο πάνω. Ένα πιο σύνθετο παράδειγμα, που δείχνει πιο περίπλοκα υπο-μοτίβα είναι το εξής:

```
f(x,(y,z)) = x+y+z
```

Η αποτίμηση του `f(1,(2,3))` θα οδηγήσει σε αναδρομικό ταίριασμα του μοτίβου `(y,z)` με την πραγματική παράμετρο `(2,3)`. Τελικά το αποτέλεσμα, όπως αναμένεται, είναι `6`.

3.4 Μοτίβα Λιστών

Υποστηρίζεται το εξής μοτίβο λίστας:
μοτίβο_κεφαλής`:`*μοτίβο_ουράς*
Όλο το μοτίβο ταιριάζει με μια μη κενή λίστα. Όταν γίνει αυτό, το μοτίβο κεφαλής ταιριάζει αναδρομικά με το πρώτο στοιχείο της λίστας και το μοτίβο ουράς με τη λίστα όλων των υπόλοιπων στοιχείων.

Όταν θέλουμε να ορίσουμε μία συνάρτηση που εφαρμόζεται σε λίστες, συνήθως σπάμε τον ορισμό σε δύο περιπτώσεις: μία για την κενή λίστα και μία για μη κενή λίστα. Η κενή λίστα αναπαρίσταται από το μοτίβο-σταθερά `[]` ενώ η μη κενή από οποιοδήποτε μοτίβο περιέχει `:`. Για παράδειγμα, η συνάρτηση που αποφασίζει αν μία λίστα είναι κενή ή όχι ορίζεται ως εξής:

```
isEmpty [] = True
isEmpty (_:_) = False
```

Προσέξτε ότι το μοτίβο λίστας πρέπει να μπαίνει σε παρένθεση στον ορισμό συναρτήσεων.

Φυσικά, το μοτίβο λίστας μπορεί να χρησιμοποιηθεί για να ταιριάξει το πρώτο στοιχείο μίας λίστας με ένα όνομα:

```
head (h:_) = h
```

Αναδρομικά, μπορούμε να ταιριάξουμε όσα στοιχεία θέλουμε από την αρχή της λίστας:

```
addFirstTwo :: [Int]->Int
addFirstTwo (x:y:_) = x+y
```

Επίσης μπορούμε να ταιριάξουμε την ουρά της λίστας με ένα όνομα:

```
tail (_:t) = t
length [] = 0
length (_:t) = 1+length t
```

Τέλος, ένα παράδειγμα που ταιριάζει και τα δύο υπο-μοτίβα:

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

3.5 Η δομή `case`

Η λέξη-κλειδί `case` μας επιτρέπει να χρησιμοποιήσουμε ταίριασμα μοτίβων οπουδήποτε μέσα σε μία έκφραση. Η σύνταξη είναι η εξής:

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

Προσέξτε ότι τα μοτίβα στοιχίζονται δεξιότερα του `case` και ακολουθούν τον κανόνα στοίχισης των ορισμών.

Η έκφραση ελέγχου μετά την λέξη `case` αποτιμάται και μετά γίνεται ταίριασμα μοτίβων με τη σειρά. Όταν ένα μοτίβο ταιριάξει, η αντίστοιχη έκφραση αποτιμάται και επιστρέφεται ως τιμή της έκφρασης `case`. Σημειώστε ότι η `case` μπορεί να είναι μια υποέκφραση μίας μεγαλύτερης και ακόμα πολυπλοκότερης έκφρασης. Για παράδειγμα, η παρακάτω έκφραση επιστρέφει το πρώτο στοιχείο της λίστας `l` αυξημένο κατά `1`, εκτός αν η λίστα είναι άδεια, οπότε επιστρέφει `0`:

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

Οι περιπτώσεις της `case` μπορούν να γραφτούν και σε μία γραμμή χωρισμένες με `;`. Η παραπάνω έκφραση είναι ισοδύναμη με:

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

3.6 Αντικρούσιμα Μοτίβα

Όπως είπαμε στην Ενότ. 2.3, μπορούμε να χρησιμοποιήσουμε μοτίβα στις γεννήτριες των εκφράσεων διαχωρισμού:

```
sumL = [m+n | (m,n)<-l]
```

Σε περίπτωση που ένα στοιχείο της λίστας μίας γεννήτριας δεν ταιριάζει με το μοτίβο που χρησιμοποιούμε, τότε αυτό το στοιχείο αγνοείται και δε συμπεριλαμβάνεται στο τελικό αποτέλεσμα. Για παράδειγμα, η παρακάτω έκφραση εφαρμόζεται σε μία λίστα από λίστες `l` και απορρίπτει όλες τις κενές λίστες μέσα σε αυτή:

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

Ο μηχανισμός αυτός ονομάζεται *αντικρούσιμα μοτίβα (refutable patterns)*.

4 Τοπικά Ονόματα

Η απόκρυψη ονομάτων είναι ουσιώδης στον τμηματοποιημένο προγραμματισμό. Ο λόγος είναι ότι δε θέλουμε δύο ή περισσότερα ανεξάρτητα υλοποιημένα κομμάτια ενός προγράμματος να είναι ασύμβατα μεταξύ τους λόγω μιας επουσιώδους σύγκρουσης ονομάτων. Αυτό σημαίνει ότι τυχόν βοηθητικές τιμές που χρησιμοποιούνται σε ένα πρόγραμμα, είτε για λόγους αναγνωσιμότητας είτε για λόγους πιο αποδοτικής εκτέλεσης, θα πρέπει να αποκρύπτονται από τους χρήστες του προγράμματος και να είναι ορατές μόνο στους ορισμούς που αυτές αφορούν.

Μέχρι τώρα έχουμε δει ότι οι τυπικές παράμετροι ενός ορισμού είναι ορατές μόνο μέσα σε αυτόν τον ορισμό. Η Haskell παρέχει τρεις ακόμα μηχανισμούς απόκρυψης ονομάτων: τις προτάσεις `where` και `let` και τις λ-εκφράσεις.

4.1 Τοπικά Ονόματα σε Ορισμό με τη `where`

Η λέξη κλειδί `where` εισάγει *τοπικούς ορισμούς* που είναι ορατοί μόνο στον ορισμό στον οποίο εμφανίζεται. Η λέξη-κλειδί `where` μπαίνει ακριβώς μετά την έκφραση του ορισμού και ακολουθείται από τους τοπικούς ορισμούς. Οι τοπικοί ορισμοί ακολουθούν τους ίδιους κανόνες στοίχισης με τους κανονικούς ορισμούς. Είναι δυνατός και ο φωλιασμός δομών `where`.

Παράδειγμα χρήσης της `where` είναι το εξής:

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

```
xx = x^2
yy = y^2
```

Παρατηρούμε ότι η λέξη `where` ξεκινάει δεξιότερα από την πρώτη γραμμή του ορισμού, που σημαίνει ότι είναι μέρος του ορισμού της `norm`. Οι ορισμοί που ακολουθούν τη `where` εισάγουν δύο τοπικά ονόματα `xx` και `yy` που χρησιμοποιούνται πιο πάνω στον ορισμό της `norm`. Προσέξτε ότι οι τοπικοί ορισμοί μπαίνουν ακριβώς κάτω ή και δεξιότερα από τη `where` και ακολουθούν τους κανόνες για τους καθολικούς ορισμούς.

4.2 Τοπικά Ονόματα σε Έκφραση με τη `let`

Η λέξη-κλειδί `let` εισάγει τοπικούς ορισμούς μέσα σε εκφράσεις. Η σύνταξη είναι:
`let` *τοπικοί ορισμοί* `in` *έκφραση*
Οι τοπικοί ορισμοί χωρίζονται με `;`. Ολόκληρη η δομή είναι μία έκφραση που μπορεί να μπει σε μεγαλύτερες εκφράσεις. Για παράδειγμα, η παρακάτω έκφραση

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

αποτιμάται σε 9.

4.3 Ανώνυμες Συναρτήσεις: οι λ-εκφράσεις

Στη Haskell όταν κατασκευάσουμε μία συνάρτηση δεν είμαστε υποχρεωμένοι να της δώσουμε και ένα όνομα. Για να φτιάξουμε μία ανώνυμη συνάρτηση χρησιμοποιούμε τη δομή που λέγεται *λ-έκφραση*. Η σύνταξη της λ-έκφρασης είναι:
`\`*τοπική μεταβλητή*`->`*έκφραση*

Η συνάρτηση που προκύπτει είναι αυτή που θα ορίζονταν από τον ορισμό:
`f` *τοπική μεταβλητή=έκφραση*
(αλλά χωρίς το όνομα `f` να ορίζεται πουθενά). Για παράδειγμα, η συνάρτηση

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

είναι ίση με την `(+1)`.

Φυσικά, η λ-έκφραση είναι μία κανονική έκφραση, που μπορεί να χρησιμοποιηθεί μέσα σε πολυπλοκότερες εκφράσεις. Για παράδειγμα, η έκφραση:

```
(\x->x+1)(17+42) * 2
```

αποτιμάται σε 120. Για να την κατανοήσουμε καλύτερα, μπορούμε να δούμε την έκφραση αυτή ως ισοδύναμη της

```
let f x = x+1 in   f(17+42) * 2
```

Να και ένα πιο περίπλοκο παράδειγμα, που συμπεριλαμβάνει σύνθεση συναρτήσεων

```
((\x->x+1).(\x->2*x)) 8
```

(αποτιμάται σε 17).

Οι κανόνες αποφυγής σύγκρουσης ονομάτων που συζητήσαμε στις προηγούμενες σημειώσεις ισχύουν και με τις λ-εκφράσεις. Για παράδειγμα, η επόμενη έκφραση

```
(\x-> \y->x+y)y
```

δε μπορεί να αποτιμηθεί σε `\y->y+y`. Η σωστή αποτίμηση συμβαίνει, αφού αλλάξουμε το όνομα της τυπικής παραμέτρου `y`:

```
(\x-> \y->x+y)y = (\x-> \z->x+z)y = \z->y+z
```

5 Παραδείγματα

5.1 Υπολογισμός Ακολουθίας Fibonacci με Πλειάδες

Στις προηγούμενες σημειώσεις είδαμε έναν μη αποδοτικό τρόπο να υπολογίσουμε την ακολουθία Fibonacci. Το πρόβλημα που είχαμε ήταν ότι για να υπολογίσουμε έναν αριθμό Fibonacci χρειαζόμαστε τους *δύο* προηγούμενους. Η αναδρομή που χρησιμοποιήσαμε καλούσε τη συνάρτηση `fib` δύο φορές, προκαλώντας (εκθετικά) πολλούς περιττούς υπολογισμούς.

Μπορούμε να λύσουμε το πρόβλημα αν η συνάρτησή μας δεν υπολογίζει έναν, αλλά *δύο* αριθμούς Fibonacci κάθε φορά. Έχοντας ένα ζεύγος διαδοχικών αριθμών Fibonacci $(f_k, f_{k\text{-}1})$, το επόμενο ζευγος υπολογίζεται πολύ απλά με αναφορά μόνο στο $(f_k, f_{k\text{-}1})$:

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

$$f_k = f_k$$

Μία συνάρτηση που θα υπολογίζει αναδρομικά ένα ζεύγος Fibonacci πρέπει να επιστρέφει δύο αριθμούς. Αυτό μπορεί να επιτευχθεί με πλειάδες (η συνάρτηση θα επιστρέφει ένα *ζεύγος* ακεραίων, δηλαδή μία πλειάδα δύο ακεραίων). Έτσι λοιπόν, ορίζουμε τη συνάρτηση `fibpair` ως εξής:

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

Το προφανές πρόβλημα είναι ότι η `fibpair (n-1)` καλείται αναδρομικά τρεις φορές, ενώ το αποτέλεσμά της χρειάζεται μόνο μία! Για να λυθεί αυτό το θέμα, θα χρησιμοποιήσουμε μία πρόταση `where` που θα κάνει την κλήση μόνο μία φορά. Ο ορισμός της `fibpair` αλλάζει ως εξής:

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

Βλέπουμε εδώ μία πολύ ουσιαστική χρήση της `where` πέραν της αναγνωσιμότητας του προγράμματος: με το να αποθηκεύουμε το αποτέλεσμα ενός υπολογισμού αποφεύγουμε να τον επαναλάβουμε. Το αποτέλεσμα είναι να κάνουμε έναν εκθετικό αλγόριθμο γραμμικό.

Η επόμενη αλλαγή μας είναι στυλιστική. Είπαμε ότι τα μοτίβα επιτρέπονται και στους ορισμούς. Θα απαλλαγούμε από τον ορισμό του `p` και την ανάγκη να χρησιμοποιήσουμε τις συναρτήσεις `fst` και `snd` χρησιμοποιώντας μοτίβο πλειάδας στον ορισμό μετά τη `where`:

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

Τώρα φυσικά, χρειαζόμαστε και τη συνάρτηση `fib` που επιστρέφει μόνο έναν αριθμό Fibonacci (ο χρήστης της συνάρτησης ζητάει μόνο έναν αριθμό και δεν ενδιαφέρεται να μάθει ότι χρησιμοποιήσαμε πλειάδες για να τον υπολογίσουμε). Έχοντας τη `fibpair`, ο ορισμός της `fib` είναι προφανής:

```
fib = fst . fibpair
```

Εναλλακτικά, αν αντιπαθούμε την `fst` μπορούμε να γράψουμε

```
fib n = f where (f,_) = fibpair n
```

αλλά ο πρώτος ορισμός είναι μικρότερος και εμείς θα προτιμήσουμε αυτόν στη συνέχεια.

Η τελευταία αλλαγή που θα κάνουμε είναι να κρύψουμε τη βοηθητική συνάρτηση `fibpair` μέσα στον ορισμό της `fib`. Ο λόγος είναι ότι θέλουμε ο χρήστης να έχει πρόσβαση μόνο στην `fib` και δεν έχει νόημα να χρησιμοποιούμε καθολικά το όνομα `fibpair`. Ο τελικός ορισμός της `fib` είναι:

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

5.2 Ταξινόμηση με Εισαγωγή (Insertion Sort)

Σε αυτό το παράδειγμα θα δείξουμε πώς υλοποιείται σε Haskell ο αλγόριθμος ταξινόμησης λιστών Insertion Sort. Στον αλγόριθμο αυτό, η ταξινόμηση βασίζεται στην έννοια της *εισαγωγής (insertion)* ενός στοιχείου σε μία ήδη ταξινομημένη λίστα. Ο αλγόριθμος εκτελείται σε `length l` βήματα, όπου `l` η λίστα προς ταξινόμηση. Στο βήμα `i` έχουμε εξασφαλίσει ότι το κομμάτι της λίστας από `length l-i` έως `length l`

έχει ταξινομηθεί και εισάγουμε το στοιχείο `length l-i-1` στο ταξινομημένο κομμάτι, εξασφαλίζοντας για το επόμενο βήμα ότι το κομμάτι από `length l-i-1` έως `length l` έχει ταξινομηθεί.

Για παράδειγμα, έστω ότι η λίστα προς ταξινόμηση είναι η

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

Στο πρώτο βήμα εισάγουμε το `0` στην κενή υπο-λίστα στα δεξιά του. Αυτό δεν αλλάζει τη λίστα. Στο δεύτερο βήμα, εισάγουμε το `2` στην ταξινομημένη υπο-λίστα `[0]` στα δεξιά του. Αυτό κάνει τη λίστα

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

και τώρα το ταξινομημένο κομμάτι είναι το `[0,2]`. Τώρα εισάγουμε το `1` στο ταξινομημένο κομμάτι και έχουμε

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

(το ταξινομημένο κομμάτι είναι τώρα το `[0,1,2]`). Τέλος εισάγουμε το `3` στο ταξινομημένο κομμάτι και παίρνουμε την ταξινομημένη λίστα

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

Για να υλοποιήσουμε τον αλγόριθμο αυτό σε Haskell, πρέπει πρώτα να ορίσουμε τη συνάρτηση εισαγωγής `ins`. Η συνάρτηση παίρνει ένα στοιχείο και μία λίστα και επιστρέφει μία λίστα:

```
ins :: Int->[Int]->[Int]
```

Ο ορισμός χωρίζεται σε τρείς περιπτώσεις: στην πρώτη περίπτωση η λίστα είναι κενή:

```
ins i [] = [i]
```

Στη δεύτερη περίπτωση η λίστα δεν είναι κενή και το στοιχείο προς εισαγωγή είναι μικρότερο ή ίσο από την κεφαλή της λίστας. Τότε το στοιχείο πρέπει να μπει στην αρχή της λίστας. Στην τελευταία περίπτωση, το στοιχείο είναι μεγαλύτερο από την κεφαλή και πρέπει να εισαχθεί στην ουρά της λίστας:

```
ins i (x:xs)
 | i<=x   =  i:x:xs
 | True   =  x:(ins i xs)
```

Ας δούμε τώρα τη συνάρτηση `iSort` που υλοποιεί την ταξινόμηση Insertion Sort:

```
iSort :: [Int]->[Int]
```

Η ιδέα για την υλοποίηση είναι η εξής: Σε περίπτωση που η λίστα είναι κενή, τότε έχουμε τελειώσει:

```
iSort [] = []
```

Αλλιώς, η `iSort` σε ένα όρισμα `x:xs` υλοποιείται αν πρώτα εξασφαλίσουμε ότι η λίστα μετά την κεφαλή είναι ταξινομημένη και μετά εισάγουμε την κεφαλή `x` στην ταξινομημένη λίστα. Η ταξινόμηση της ουράς γίνεται φυσικά με αναδρομική κλήση της `iSort`. Άρα:

```
iSort (x:xs) = ins x (iSort xs)
```

Ο ορισμός της `iSort` είναι εντυπωσιακά απλούστερος από αυτόν που θα είχαμε σε χαμηλότερου επιπέδου γλώσσες.

5.3 Γρήγορη Ταξινόμηση (Quick Sort)

Ένας αλγόριθμος που κατά μέσο όρο είναι πολύ αποδοτικότερος της ταξινόμησης με εισαγωγή είναι ο Quick Sort του Tony Hoare. Στην ταξινόμηση Quick Sort, επιλέγουμε ένα στοιχείο `x` της λίστας και μετά σπάμε τη λίστα σε δύο μικρότερες λίστες: τα στοιχεία που είναι μικρότερα ή ίσα του `x` και τα στοιχεία που είναι μεγαλύτερα του `x`. Ο αλγόριθμος καλείται αναδρομικά για αυτές τις δύο μικρότερες λίστες. Αφού οι δύο λίστες ταξινομηθούν, μετά επανενώνονται μαζί με το `x` σε μία ταξινομημένη λίστα.

Για παράδειγμα, ας υποθέσουμε ότι η λίστα προς ταξινόμηση είναι η

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

Διαλέγουμε το στοιχείο `2` και διαχωρίζουμε τη λίστα έτσι ώστε όλα τα στοιχεία μικρότερα ή ίσα με αυτό να πάνε στα αριστερά και όλα τα υπόλοιπα στα δεξιά του:

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

Καλούμε την Quick Sort αναδρομικά στις δύο υπο-λίστες. Η κλήση αυτή τις ταξινομεί:

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

Επανενώνουμε και έχουμε την αρχική λίστα ταξινομημένη:

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

Ας ορίσουμε τώρα τη συνάρτηση `qSort` στη Haskell:

```
qSort :: [Int]->[Int]
```

Σε περίπτωση κενής λίστας, δεν έχουμε να κάνουμε τίποτα:

```
qSort [] = []
```

Αν η λίστα δεν είναι κενή, τότε επιλέγουμε το πρώτο στοιχείο της, έστω `x`. Διαχωρίζουμε τη λίστα σε δύο υπο-λίστες ανάλογα με το πώς κατατάσονται τα στοιχεία της σε σχέση με το `x` και εφαρμόζουμε την `qSort` αναδρομικά σε αυτές. Τέλος, επανενώνουμε με `++`. Ο διαχωρισμός μπορεί να γίνει πολύ κομψά χρησιμοποιώντας εκφράσεις διαχωρισμού. Όλη αυτή η διαδικασία περιγράφεται από τον εξής ορισμό:

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

Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση `filter` αντί για τις εκφράσεις διαχωρισμού. Ο ορισμός θα είναι τότε:

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

Και στις δύο περιπτώσεις, ο ορισμός είναι επικεντρωμένος στο *τι* και όχι στο *πώς* υπολογίζουμε. Όπως και στην περίπτωση της Insertion Sort, ο ορισμός είναι εντυπωσιακά μικρός και πολύ ξεκάθαρος για τον άνθρωπο-αναγνώστη. Και πάλι βλέπουμε την μεγάλη εκφραστική δύναμη του συναρτησιακού προγραμματισμού και ιδιαίτερα της Haskell.

5.4 Γραμμική Αναζήτηση

Η γραμμική αναζήτηση σε μία λίστα είναι μία συσσωρευτική πράξη και ως τέτοια μπορεί να περιγραφεί πολύ έυκολα χρησιμοποιώντας τη συνάρτηση `foldr` (ή `foldl`). Για να δούμε *γιατί* έχουμε συσσωρευτικό υπολογισμό, σκεφτόμαστε ως εξής: ένα στοιχείο `x` βρίσκεται στη λίστα `l` αν και μόνον αν: είναι το πρώτο στοιχείο της ή είναι το δεύτερο στοιχείο της ή είναι το τρίτο στοιχείο της κτλ. Βλέπουμε λοιπόν ότι έχουμε μία συσσωρευτική πράξη λογικής διάζευξης. Επομένως, η γραμμική αναζήτηση θα ορίζεται κάπως έτσι:

```
lSearch :: Int->[Int]->Bool
lSearch x l = foldr (||) XXX YYY
```

όπου στη θέση `XXX` βάζουμε μία αρχική τιμή για τον υπολογισμό και στη θέση `YYY` βάζουμε μία λίστα.

Η αρχική τιμή πρέπει να είναι `False` γιατί ξεκινάμε μην έχοντας βρει το στοιχείο. (Γενικά στους συσσωρευτικούς υπολογισμούς, η αρχική τιμή είναι συνήθως το *ουδέτερο στοιχείο* της πράξης `f` που συσσωρεύεται, δηλαδή το στοιχείο `e` εκείνο για το οποίο `f e x = f x e = x`. Στη διάζευξη, αυτό το στοιχείο είναι το `False`). Επομένως, ο ορισμός μας γίνεται:

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

Στη θέση `YYY` χρειαζόμαστε μία λίστα τύπου `[Bool]`. Η `l` δε μας κάνει γιατί είναι τύπου `[Int]`: λογική διάζευξη μεταξύ ακεραίων δεν έχει κανένα νόημα. Για να βρούμε την έκφραση `YYY`, θυμόμαστε ότι αυτό που εξετάζουμε κάθε φορά είναι εάν το τρέχον στοιχείο είναι ίσο με το `x`. Επομένως, η λίστα `YYY` είναι μία λίστα που περιέχει `True` ακριβώς σε εκείνα τα σημεία στα οποία το τρέχον στοιχείο είναι ίσο με το `x`. Για να πάρουμε μία τέτοια λίστα, πρέπει να εφαρμόσουμε την πράξη `(x==)` διαδοχικά σε όλα τα στοιχεία της `l`. Αυτό είναι η δουλειά της `map`. Ο τελικός ορισμός είναι:

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

Αυτός ο ορισμός μπορεί να μας τρομάζει λίγο. Φαίνεται σαν το πρόγραμμά μας να κάνει έναν εντελώς άχρηστο υπολογισμό: δημιουργεί μία ολόκληρη λίστα `map (x==) l` και μετά κάνει ένα συσσωρευτικό υπολογισμό πάνω της. Το πρόγραμμα δείχνει να καταναλώνει χρόνο `2*length l`, ενώ θα περιμέναμε να καταναλώνει χρόνο `length l` στη χειρότερη περίπτωση. Στην πραγματικότητα δεν είναι έτσι. Λόγω της οκνηρής αποτίμησης της Haskell, η λίστα `map (x==) l` *δεν* κατασκευάζεται. Μόνο τα στοιχεία της εκείνα που χρειάζονται κατασκευάζονται, όπως θα κάναμε σε μία γραμμική αναζήτηση σε μία προστακτική γλώσσα. Επίσης, η αποτίμηση της διάζευξης σταματάει όταν βρεθεί το πρώτο `True`, όπως ακριβώς θα θέλαμε, και πάλι λόγω οκνηρής αποτίμησης (η έκφραση `True || whatever` αποτιμάται αυτομάτως σε `True` χωρίς να αποτιμάται η έκφραση `whatever`).

Για του λόγου το αληθές, δοκιμάστε να φτιάξετε την άπειρη λίστα όλων των φυσικών αριθμών:

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

Χωρίς οκνηρή αποτίμηση, η παρακάτω αποτίμηση δε θα τερμάτιζε ποτέ:

```
lSearch 4 nat
```

Κι όμως, στη Haskell η αποτίμηση τερματίζει με αποτέλεσμα `True`. Αυτό μας δείχνει ότι η λίστα `map (x==) l` *δεν* κατασκευάζεται ολόκληρη, καθώς και ότι η αποτίμηση της `||` σταματάει όταν βρεθεί το πρώτο στοιχείο `True` στη λίστα `map (x==) l`. Φυσικά, η αποτίμηση `lSearch (-1) nat` δεν τερματίζει.

5.5 Δυαδική Αναζήτηση

Στη *δυαδική αναζήτηση (binary search)* θεωρούμε ότι η λίστα της εισόδου είναι πεπερασμένη και ταξινομημένη. Η διαδικασία είναι να ελέγξουμε το μεσαίο στοιχείο και αναλόγως με το αν είναι μεγαλύτερο, ίσο ή μικρότερο του στοιχείου που αναζητάμε να κινηθούμε αριστερά ή δεξιά του, καλώντας αναδρομικά την ίδια διαδικασία. Έτσι επιτυγχάνουμε λογαριθμικό χρόνο.

Για να πάρουμε τη λίστα με τα στοιχεία που είναι αριστερότερα του μεσαίου, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση `take`. Ομοίως χρησιμοποιούμε τη `drop` για τα υπόλοιπα στοιχεία. Η συνάρτηση `bSearch` που γράφουμε έχει δύο αρχικούς ορισμούς (ένα για κενή λίστα και ένα για λίστα ενός στοιχείου), γιατί η συνεχής διαίρεση του μήκους της λίστας με το δύο μπορεί να καταλήγει είτε στο `0` είτε στο `1`.

Ο ορισμός της `bSearch` είναι:

```
bSearch :: Int->[Int]->Bool
bSearch x [] = 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
```

6 Επισκόπηση

Στις σημειώσεις αυτές είδαμε τους δύο βασικούς τρόπους δόμησης δεδομένων που παρέχει η Haskell: τις πλειάδες και τις λίστες, το μηχανισμό των μοτίβων και τους μηχανισμούς απόκρυψης ονομάτων.

Είδαμε τα εξής:

- Οι *πλειάδες (tuples)* είναι συλλογές συγκεκριμένου αριθμού ετερογενών δεδομένων, αντίστοιχες περίπου με τις *εγγραφές* σε άλλες γλώσσες προγραμματισμού.
- Μία πλειάδα μπορεί να περιέχει δεδομένα οποιουδήποτε τύπου, ακόμα και άλλες πλειάδες.
- Η λέξη-κλειδί `type` χρησιμοποιείται για τον ορισμό *συνωνόματων τύπων*.
- Οι *λίστες (lists)* είναι συλλογές όχι προκαθορισμένου αριθμού δεδομένων του ιδίου τύπου, αντίστοιχες περίπου με τους *πίνακες* σε άλλες γλώσσες προγραμματισμού.
- Η σειρά και ο αριθμός εμφάνισης των στοιχείων μέσα σε μια λίστα είναι σημαντικά.
- Οι *συμβολοσειρές (strings)* είναι λίστες χαρακτήρων.
- Μία από τις πιο δυνατές εκφραστικά δομές της Haskell είναι ο *διαχωρισμός λιστών (list comprehension)* με τον οποίο μπορούμε να δημιουργήσουμε μία καινούρια λίστα φιλτράροντας και μετασχηματίζοντας στοιχεία από άλλες λίστες.
- Η Haskell υποστηρίζει μια μεγάλη ποικιλία συναρτήσεων για λίστες, μερικές από τις οποίες παρουσιάζονται στην Ενότ. 2.4. Μεταξύ αυτών των συναρτήσεων είναι και οι συναρτήσεις ανώτερης τάξης `map`,`filter`,`zipWith`,`foldl`,`foldr`,`takeWhile` και `dropWhile`.
- Τα *μοτίβα (patterns)* και ο μηχανισμός του *ταιριάσματος μοτίβων (pattern matching)* χρησιμοποιούνται στη Haskell για να ταιριάζουν ονόματα κατευθείαν σε υποσύνολα μη τετριμμένων δομών δεδομένων.
- Τα μοτίβα χρησιμοποιούνται κυρίως στις τυπικές παραμέτρους των ορισμών, αλλά μπορούν να χρησιμοποιηθούν και σε αυτούσιους ορισμούς καθώς και σε εκφράσεις διαχωρισμού.

- Τα μοτίβα που υποστηρίζει η Haskell είναι: μεταβλητές, σταθερές, το μοτίβο `_`, μοτίβα πλειάδων και μοτίβα λιστών.
- Η λέξη κλειδί `case` εισάγει ταίριασμα μοτίβου μέσα σε μία έκφραση.
- Τα μοτίβα σε εκφράσεις διαχωρισμού είναι *αντικρούσιμα (refutable)* που σημαίνει ότι φιλτράρουν τα στοιχεία στα οποία δεν ταιριάζουν.
- Η Haskell υποστηρίζει τρεις μηχανισμούς απόκρυψης ονομάτων, τις εκφράσεις `where` και `let` καθώς και τις λ-εκφράσεις. Η απόκρυψη ονομάτων βοηθάει στην τμηματοποίηση του κώδικα, αλλά και στην αποδοτικότητα των υπολογισμών, καθώς επιτρέπουν την αποθήκευση τιμών που χρειάζονται πολλές φορές.
- Οι λ-εκφράσεις εισάγουν ανώνυμες συναρτήσεις οπουδήποτε σε μία έκφραση. Ο κανόνας αποφυγής συγκρούσεων ονομάτων πρέπει να γίνεται σεβαστός και για τις λ-εκφράσεις.
- Στην Ενότ. 5 είδαμε πέντε παραδείγματα στα οποία χρησιμοποιήσαμε όσα μάθαμε σε αυτήν την ενότητα.

Αναφορές

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison Wesley, 1999.