1 Συναρτήσεις Υψηλότερης Τάξης

Η βασική μονάδα ενός προγράμματος Haskell είναι η συνάρτηση. Ο τύπος μιας συνάρτησης καθορίζεται από τους τύπους των παραμέτρων της και τον τύπο του αποτελέσματος. Κάθε τύπος συνάρτησης, με τη σειρά του αποτελεί και έναν τύπο της Haskell, έτσι ώστε οι συναρτήσεις να μπορούν να παίρνουν ως παραμέτρους άλλες συναρτήσεις, ή ακόμα και να επιστρέφουν συναρτήσεις ως αποτέλεσμα. Επίσης υπάρχουν παραστάσεις οι τιμές των οποίων είναι μία συνάρτηση.

Μία συνάρτηση που δέχεται ως είσοδο μία συνάρτηση ή επιστρέφει ως αποτέλεσμα μία συνάρτηση, ονομάζεται συνάρτηση υψηλότερης τάξης. Ο τρόπος με τον οποίο γίνεται αυτό καθώς και η χρησιμότητά του θα φανεί στα επόμενα παραδείγματα.

Παράδειγμα 48: υπολογισμός αθροίσματος.

Στο παράδειγμα 19 έχουμε δει τη συνάρτηση `sum1`, η οποία υπολογίζει το άθροισμα $\sum_{i=1}^{n} i^i$. Αν θέλουμε να υπολογίσουμε ένα διαφορετικό άθροισμα, για παράδειγμα το $\sum_{i=1}^{n}(-i)$, μπορούμε να γράψουμε μία συνάρτηση τροποποιώντας την `sum1`. Ωστόσο, με αυτή τη στρατηγική, αν χρειάζεται να υπολογίζουμε παρόμοια αθροίσματα για ένα μεγάλο πλήθος διαφορετικών συναρτήσεων, θα πρέπει να γράψουμε για κάθε μία από αυτές μια ξεχωριστή συνάρτηση Haskell.

Η Haskell μας δίνει τη δυνατότητα να γράψουμε μία μόνο συνάρτηση για τον υπολογισμό του αθροίσματος $\sum_{i=1}^{n} f(i)$, στην οποία η συνάρτηση f θα αποτελεί παράμετρο:

```
sumF :: (Int -> Int) -> Int -> Int
sumF f 0 = 0
sumF f n = sumF f (n-1) + f n
```

Η πρώτη παράμετρος της `sumF` έχει τύπο `Int -> Int`, είναι δηλαδή μία συνάρτηση από ακεραίους σε ακεραίους.

Ο ορισμός της `sumF` μοιάζει πολύ με τον ορισμό της `sum1`, με τη διαφορά ότι υπάρχει μία παραπάνω παράμετρος (η `f`) και ότι το `n^n` έχει αντικατασταθεί από το `f n`.

Αν θέλουμε να υπολογίσουμε το $\sum_{i=1}^{10}(-i)$, γράφουμε:

```
> sumF negate 10
```

Στη θέση της `negate` μπορεί να μπεί μια οποιαδήποτε συνάρτηση έχουμε ορίσει στο ίδιο αρχείο με την `sumF`:

```
> sumF fact 6
```

Αργότερα θα δούμε πως μπορούμε να περιγράψουμε μία συνάρτηση χωρίς να της δόσουμε κάποιο όνομα (όπως αντίστοιχα μπορούμε να γράψουμε τη αριθμητική σταθερά 10). ∎

Παράδειγμα 49: υπολογισμός διπλού αθροίσματος.

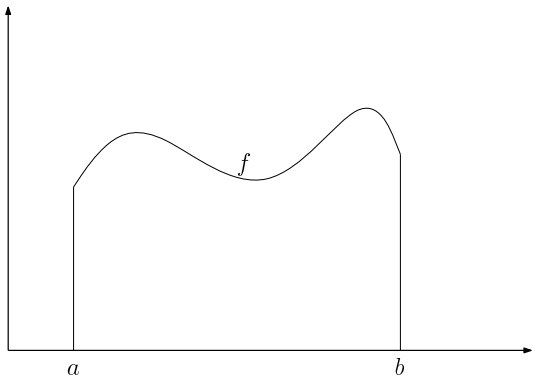
Με παρόμοιο τρόπο όπως στο προηγούμενο παράδειγμα μπορούμε να τροποποιήσουμε τη συνάρτηση `sumabcd` του παραδείγματος 25, ώστε να υπολογίζει το άθροισμα $\sum_{i=a}^{b}\sum_{j=c}^{d} f(i,j)$ για οποιαδήποτε συνάρτηση f τύπου `Int -> Int -> Int`.

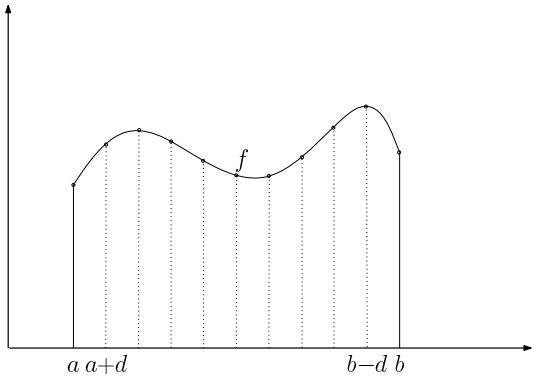
```
sumFabcd  :: (Int -> Int -> Int) ->
              Int -> Int -> Int -> Int -> Int
sumFabcd f a b c d
 = if a==b
   then
       if c==d
       then f a c
       else sumFabcd f a b c n
          + sumFabcd f a b (n + 1) d
   else sumFabcd f a m c d
      + sumFabcd f (m + 1) b c d
   where m = (a + b) `div` 2
         n = (c + d) `div` 2
```

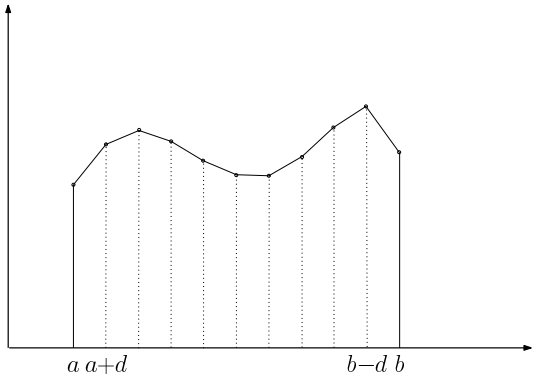
∎

Παράδειγμα 50: υπολογισμός ολοκληρώματος.

Μπορούμε να υπολογίσουμε το ολοκλήρωμα $\int_a^b f(x)dx$, προσεγγίζοντάς το με ένα άθροισμα από εμβαδά τραπεζίων. Το ύψος του τραπεζίου είναι μία παράμετρος d από την οποία εξαρτάται η ποιότητα της προσέγγισης. Σχηματικά:







Η παρακάτω συνάρτηση `integral` δέχεται ως παραμέτρους μία συνάτηση `f` τύπου `Float -> Float`, τα όρια της ολοκλήρωσης `a`,`b`, και τη παράμετρο `d` και υπολογίζει προσεγγιστικά το ολοκλήρωμα $\int_{\mathrm{a}}^{\mathrm{b}} f(x)dx$. Για απλούστευση υποθέτουμε ότι `a`$\leq$`b`. Η `integeral` επιστρέφει το άθροισμα του εμβαδού ενός τραπεζίου με βάσεις μήκους `f(a)` και `f(a+d)` και της προσέγγισης του ολοκληρώματος $\int_{\mathrm{a+d}}^{\mathrm{b}} f(x)dx$ που υπολογίζεται αναδρομικά. Αν το `b-a` είναι πολύ μικρό (δηλαδή μικρότερο του `d`), τότε ολόκληρο το ολοκλήρωμα προσεγγίζεται από το εμβαδό ενός μόνο τραπεζίου. Η βοηθητική συνάρτηση `area` υπολογίζει το εμβαδό ενός τραπεζίου.

```
integral :: (Float -> Float) ->
            Float -> Float -> Float -> Float
integral f a b d
    | c < d = area (f a) (f b) c
    | otherwise = area (f a) (f a') d
                + integral f a' b d
    where c = b-a
          a' = a+d
          area :: Float -> Float -> Float -> Float
          area b1 b2 h = h*(b1 + b2)/2
```

Για να υπολογίσουμε το ολοκλήρωμα $\int_0^1 e^x dx$ (προσεγγιστικά, με ύψος τραπεζίου 0.001 - που ισοδυναμεί με χωρισμό της περιοχής σε 10000 τραπέζια)

γράφουμε:

```
> integral exp 0 1 0.001
```

∎

Στη συνέχεια θα δούμε ορισμένες συναρτήσεις υψηλής τάξης, που υλοποιούν βασικά είδη λειτουργιών πάνω σε λίστες.

Παράδειγμα 51: αλυσιδωτή εφαρμογή συνάρτησης στα στοιχεία λίστας.

Στο παράδειγμα 33 ορίσαμε τη συνάρτηση `maxIntList` για τον υπολογισμό του μέγιστου στοιχείου μίας λίστας. Το αποτέλεσμα προκύπτει αποτιμώντας τη συνάρτηση `max` με ορίσματα την κεφαλή της λίστας και το μέγιστο στοιχείο της ουράς της που υπολογίζεται χρησιμοποιώντας αναδρομή. Μπορούμε έυκολα να διαπιστώσουμε ότι το τελικό αποτέλεσμα προκύπτει με αλυσιδωτή εφαρμογή της συνάρτησης `max` σε όλα τα στοιχεία της λίστας.

Αν τροποποιήσουμε την συνάρτηση `maxIntList` μπορούμε να σχεδιάσουμε μία συνάρητηση που να υπολογίζει το έλάχιστο, το άθροισμα, το γινόμενο ή το μέγιστο κοινό διαιρέτη ανικαθιστώντας τη `max` με `min`,`+`,`*` ή `gcd`.

Εναλλακτικά, μπορούμε να εχεδιάσουμε μία συνάρτηση υψηλότερης τάξης στην οποία η συνάρτηση που εφαρμόζεται αλυσιδωτά να δίνεται ως είσοδος:

```
foldIntList :: (Int -> Int -> Int) -> [Int] -> Int
foldIntList f (h:[]) = h
foldIntList f (h:t) = f h (foldIntList f t)
```

Παραδείγματα χρήσης της `foldIntList`:

```
> foldIntList max [24,16,52]
52
> foldIntList min [24,16,52]
16
> foldIntList (+) [24,16,52]
92
> foldIntList (*) [24,16,52]
19968
> foldIntList gcd [24,16,52]
4
```

∎

Παράδειγμα 52: απεικόνιση στοιχείων λίστας ακεραίων.

Μπορούμε να σχεδιάσουμε μία συνάρτηση η οποία με είσοδο μία λίστα ακεραίων θα επιστρέφει τη λίστα που προκύπτει αν αντικαταστήσουμε κάθε στοιχείο της λίστας με τον αντίθετό του:

```
negateIntList :: [Int] -> [Int]
negateIntList [] = []
negateIntList (h:t) = negate h : (negateIntList t)
```

Μπορούμε να τροποποιήσουμε την παραπάνω συνάρτηση σχηματίζοντας μία συνάρτηση υψηλότερης τάξη, στην οποία η απεικόνιση των στοιχείων θα γίνεται μέσω μίας οποιασδήποτε συνάρτησης (στη θέση της `negate`), η οποία θα δίνεται ως παράμετρος:

```
mapIntList :: (Int -> Int) -> [Int] -> [Int]
mapIntList f [] = []
mapIntList f (h:t) = f h : (mapIntList f t)
```

Παραδείγματα χρήσης της `mapIntList`:

```
> mapIntList fact [1..5]
[1,2,6,24,120]
> mapIntList fib [1..8]
[1,2,3,5,8,13,21,34]
> mapIntList sqrtInt [5,10,15,20,25,30]
[2,3,3,4,5,5]
```

∎

Παράδειγμα 53: επιλογή στοιχείων από λίστα ακεραίων.

Μία αρκετά συνήθης λειτουργία είναι η επιλογή τως στοιχείων μίας λίστας που ικανοποιούν κάποιο κριτήριο, η οποία μπορεί να υλοποιηθεί σε Haskell με μία συνάρτηση υψηλότερης τάξης. Το κριτήριο επιλογής περιγράφεται από μία συνάρτηση τύπου `Int -> Bool`:

```
filterIntList :: (Int -> Bool) -> [Int] -> [Int]
filterIntList f [] = []
filterIntList f (h:t)
     | f h
          = h : filterIntList f t
     | otherwise
          = filterIntList f t
```

Παράδειγμα χρήσης της `mapIntList`:

```
> filterIntList even [1..10]
[2,4,6,8,10]
```

∎

2 Συναρτήσεις ως τιμές

Στη Haksell οι συναρτήσεις αποτελούν τύπους δεδομένων. Είδαμε ότι μπορούμε να ορίσουμε συναρτήσεις που δέχονται συναρτήσεις ως ορίσματα. Θα δούμε στη συνέχεια πώς μπορούμε να γράψουμε παραστάσεις οι οποίες έχουν ως τιμή μία συνάρτηση. Αυτό επιτρέπει να ορίσουμε συναρτήσεις που επιστρέφουν ως αποτέλεσμα συναρτήσεις.

Η Haskell έχει τον τελεστή `.` ο οποίος παίρνει ως ορίσματα δύο συναρτήσεις και επιστρέφει τη σύνθεσή τους. Αν το πρώτο όρισμα του `.` είναι τύπου $\texttt{t}_3$ `->` $\texttt{t}_2$ και το δεύτερο τύπου $\texttt{t}_1$ `->` $\texttt{t}_3$, τότε το αποτέλεσμα είναι συνάρτηση τύπου $\texttt{t}_1$ `->` $\texttt{t}_2$.

Οι παραστάσεις `(f.g) x` και `f (g x)` εχουν παντοτε την ίδια τιμή. Για παράδειγμα η τιμή της παράστασης `sqrt.abs` είναι η συνάρτηση $\sqrt{|x|}$. Αν ωστόσο γράψουμε στο διερμηνέα της Haskell

```
> sqrt.abs
```

αυτός θα μας επιστρέψει ένα μήνυμα λάθους: η τιμή μίας παράστασης επιτρέπεται να είναι συνάρτηση, ωστόσο αυτή η τιμή δεν μπορεί να τυπωθεί στην οθόνη. Το ίδιο μήνυμα λάθους θα πάρουμε αν γράψουμε:

```
> sqrt
```

Μπορούμε να εκτυπώσουμε την τιμή της `sqrt.sin` για έναν συγκεριμένο αριθμό:

```
> (sqrt.abs) (-4.0)
2.0
```

Το ίδιο αποτέλεσμα θα παίρναμε αν γράφαμε

```
> sqrt (abs (-4.0))
2.0
```

Συμπεραίνουμε ότι για να υπολογίσουμε μεμονωμένες τιμές της σύνθεσης, δεν απαιτείται χρήση του τελεστή `..` Ο τελεστής `.` είναι απαραίτητος όταν θέλουμε να περάσουμε τη σύνθεση δυό συναρτήσεων ως όρισμα σε μία συνάρτηση υψηλότερης τάξης (ή να την επιστρέψουμε ως αποτέλεσμα, όπως θα δούμε σύντομα).

Παράδειγμα 54: Για να υπολογίσουμε το ολοκλήρωμα $\int_{-2}^{2} \sqrt{|x|}\ dx$ προσεγγιστικά με την `integral`, χωρίζοντας το διάστημα ολοκλήρωσης σε 4000 τραπέζια, γράφουμε:

```
> integral (sqrt.abs) (-2) 2 0.001
```

Σημειώνεται ότι χωρίς τον τελεστή συνθεσης η παραπάνω χρήση της `integral` δεν θα ήταν δυνατή. Θα έπρεπε η συνάρτηση $\sqrt{|x|}$ να έχει οριστεί μεσα στο πρόγραμμα ως ξεχωριστή συνάρτηση και να χρησιμοποιείται το όνομά της στην παραπάνω παράσταση. ∎

Η Haskell παρέχει έναν γενικότερο τρόπο με τον οποίο μπορούμε να περιγράψουμε συναρτήσεις χωρίς να απαιτείται να τους δώσουμε ένα συμβολικό όνομα. Αυτό γίνεται με χρήση του τελεστή `\` (ονομάζεται τελεστής lambda). Ο τελεστής `\` ακολουθείται από τη λίστα των παραμέτρων της συνάρτησης, το σύμβολο `->` και την παράσταση από την οποία υπολογίζεται η τιμή της συνάρτησης.

Η δυνατότητα περιγραφής συναρτήσεων χωρίς να τους αποδοθεί συμβολικό όνομα, είναι αντίστοιχη με τη δυνατότητα χρήσης κυριολεκτικών σταθερών άλλων τύπων (π.χ. `10`, `"abc"`, `False` ...).

Παράδειγμα 55: Η παράσταση `\x -> 2*x*x*sin x` έχει ως τιμή τη συνάρτηση $2x^2 \sin x$. Για να υπολογίσουμε το ολοκλήρωμα $\int_0^{\pi} 2x^2 \sin x \, dx$ γράφουμε

```
> integral (\x -> 2*x*x*sin x) 0 pi 0.001
```

Η παράσταση `\x y -> x^y` έχει ως τιμή τη συνάρτηση (δύο μεταβλητών) x^y. Για να υπολογίσουμε το άθροισμα $\sum_{i=0}^{5} \sum_{j=4}^{8} i^j$ γράφουμε

```
> sumFabcd (\x y -> x^y) 0 5 4 8
```

∎

Παράδειγμα 56: υπολογισμός του διπλού αθροίσματος $\sum_{i=1}^{n} \sum_{j=1}^{n} f(i,j)$. Έχουμε δεί ότι το παραπάνω άθροισμα ικανοποιεί την παρακάτω αναδρομική σχέση:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} f(i,j) = \begin{cases} 0 & n = 0 \\ (\sum_{i=1}^{n-1} \sum_{j=1}^{n-1} f(i,j)) + f(n,n) + & \\ \sum_{i=1}^{n-1} f(i,n) + \sum_{j=1}^{n-1} f(n,j) & \text{αν } n > 0 \end{cases}$$

Μπορούμε να ορίσουμε μία αναδρομική συνάρτηση που θα υλοποιεί τον παραπάνω αναδρομικό τύπο. Η συνάρτηση μας χρησιμοποιεί τη `sumF` για να υπολογίζει τα απλά αθροίσματα και τον τελεστή `\` για να περάσει την κατάλληλη συνάρτηση ως παράμετρο στη `sumF`.

```
sumF1n1n  :: (Int -> Int -> Int) -> Int -> Int
sumF1n1n f 0 = 0
sumF1n1n f n =  sumF1n1n f (n-1)
              + sumF (\i -> f i n) (n-1)
              + sumF (\j -> f n j) (n-1)
              + f n n
```

Στη συνέχεια θα δούμε παραδείγματα συναρτήσεων υψηλότερης τάξης οι οποίες επιστρέφουν ως αποτέλεσμα μία συνάρτηση.

Παράδειγμα 57: Ο πιο απλός τρόπος για να επιστρέψει μία συνάρτηση ως αποτέλεσμα μία άλλη συνάρτηση είναι η συνάρτηση που επιστρέφεται να έχει συμβολικό όνομα. Στο παρακάτω παράδειγμα η επιστρεφόμενη συνάρτηση είναι κάποια προκαθορισμένη συνάρτηση (`negate` ή `id`).

```
hof :: Bool -> (Int -> Int)
hof False = negate
hof True = id
```

Παράδειγμα 58: λογάριθμος με δεδομένη βάση.

Η συνάρτηση `loga` παίρνει ως είσοδο έναν αριθμό `a` και επιστρέφει τη συνάρτηση $\log_a$.

```
loga :: Float -> (Float -> Float)
loga a = \x -> log x / log a
```

Μπορούμε να χρησιμοποιήσουμε την `loga` για να υπολογίσουμε λογαρίθμους με διάφορες βάσεις:

```
> (loga 2) 1024
10.0
> (loga 10) 1000
3.0
```

Επίσης μπορούμε να την χρησιμοποιήσουμε ως όρισμα σε μία συνάρτηση υψηλότερης τάξης. Για παράδειγμα για να υπολογίσουμε προσεγγιστικά το $\int_1^{10} \log_2 x dx$ γράφουμε:

```
> integral (loga 2) 1 10 0.001
```

Παράδειγμα 59: προσεγγιστική παράγωγος.

Η παρακάτω συνάρτηση `deriv` επιστρέφει την αριθμητική προσέγγιση της παραγώγου μιας συνάρτησης. Η ακρίβεια της προσέγγισης καθορίζεται από την παράμετρο `d`.

Η προσεγγιστική τιμή της παραγώγου της f στο σημείο x είναι η τιμή της παράστασης $\frac{f(x-\frac{d}{2})-f(x+\frac{d}{2})}{d}$.

```
deriv ::(Float -> Float) -> Float -> (Float -> Float)
deriv f d = \x -> (f (x+e) - f (x-e)) / d
      where e = d/2
```

Μπορούμε να υπολογίσουμε προσσεγγιστικά τιμές των παραγώγων γνωστών συναρτήσεων:

```
> (deriv cos 0.001) 0
0.0
> (deriv sin 0.001) 0
1.0
> (deriv exp 0.001) 1
2.71821
> (deriv log 0.001) 4
0.2501011
> (deriv (\x -> x^3) 0.001) 2
11.99961
```

∎

Παράδειγμα 60: κατασκευή πολυωνύμου.

Η συνάρτηση `poly` παίρνει ως είσοδο μία λίστα πραγματικών αριθμών $[a_0, a_1, a_2, \ldots, a_k]$ και επιστρέφει το πολυώνυμο

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots a_k x^k$$

Παρατηρούμε ότι το πολυώνυμο που αντιστοιχεί στην ουρά της λίστας είναι το

$$q(x) = a_1 + a_2 x + \ldots a_k x^{k-1}$$

Συνεπώς ισχύει η σχέση $p(x) = a_0 + x \cdot q(x)$, στην οποία βασίζεται η συνάρτηση `poly`.

```
poly :: [Float] -> (Float -> Float)
poly [] = \x -> 0
poly (h:t) = \x -> h + x * ((poly t) x)
```

Μπορούμε να υπολογίσουμε την τιμή $p(3)$, όπου $p(x) = 2x^3 + x + 3$:

```
> (poly [3,1,0,2]) 3
60.0
```

την προσέγγιση της τιμής της παραγώγου του $p(x)$ για $x = 3$ (η ακριβής τιμή είναι 55.0):

```
>(deriv (poly [3,1,0,2])) 0.001 3
54.99649
```

και το ολοκλήρωμα $\int_0^3 x^2\ dx$ προσεγγιστικά (η ακριβής τιμή είναι 9.0)

```
> integral (poly [0,0,1]) 0 3 0.001
9.000336
```

∎

Είδαμε μέχρι τώρα δύο βασικούς τελεστές που επιστρέφουν ως τιμή μία συνάρτηση, τον . και τον \. Στη συνέχεια θα δούμε πως μπορούμε να σχηματίσουμε παραστάσεις που επιστρέφουν ως τιμή μία συνάρτηση, κάνοντας μερική εφαρμογή συναρτήσεων.

Όταν δηλώνουμε στη Haskell μία συνάρτηση `f :: t1->t2-> ... ->tk->t` θεωρούμε ότι η `f` παίρνει k ορίσματα που έχουν τύπους `t1`, `t2`, ..., `tk` και επιστρέφει τιμή τύπου `t`. Η Haskell όμως αντιλαμβάνεται την παραπάνω δήλωση με διαφορετικό τρόπο. Ο τελεστης `->` προσεταιρίζεται από δεξιά προς τα αριστερά. Συνεπώς ο παραπάνω ορισμός μπορεί να γραφεί ισοδύναμα: `f :: t1->(t2-> ... ->(tk->t) ... )`. Αυτό σημαίνει ότι η Haskell θεωρεί την `f` ως συνάρτηση με ένα όρισμα τύπου `t1` η οποία επιστρέφει μία συνάρτηση τύπου `t2-> ... ->tk->t`.

Οταν η Haskell συναντήσει μία παράσταση της μορφής `f x1 x2 ... xk` τότε την αποτιμά σε k βήματα με τον παρακάτω τρόπο:

- Αποτιμά το `f x1`. Το αποτέλεσμα της αποτίμησης είναι μία συνάρτηση τύπου `t2-> ... ->tk->t`.
- Στη συνέχεια αποτιμά την παραπάνω συνάρτηση με όρισμα `x2`. Το αποτέλεσμα είναι μία συνάρτηση τύπου `t3-> ... ->tk->t`.
- Στο i-οστό βήμα αποτιμά μία συνάρτηση τύπου `ti-> ... ->tk->t` (που έχει προκύψει από το προηγούμενο βήμα) με όρισμα `xi`. Το αποτέλεσμα είναι μία συνάρτηση τύπου `ti+1-> ... ->tk->t`.
- Τέλος στο k-οστό βήμα αποτιμά μία συνάρτηση τύπου `tk->t` με όρισμα `xk`. Το αποτέλεσμα είναι μία τιμή τύπου `t`.

Το συνολικό αποτέλεσμα της παραπάνω αποτίμησης, είναι ισοδύναμο με τον υπολογισμό μίας συνάρτησης με k ορίσματα, συνεπώς ταυτίζεται με αυτό που θέλαμε να ορίσουμε.

Το κέρδος από το ότι η Haskell θεωρεί ότι όλες οι συναρτήσεις έχουν ένα όρισμα είναι ότι μπορούμε να αποτιμήσουμε την παραπάνω συνάρτηση `f` με λιγότερα από k ορίσματα, παίρνοντας ως αποτέλεσμα μία συνάρτηση.

Παράδειγμα 61: μπορούμε να περάσουμε ως ορίσματα στις **sumF**, **map**, κλπ, συναρτήσεις που προκύπτουν απο μερική αποτίμηση άλλων συναρτήσεων:

```
> sumF (comb 11) 5
1023
> mapIntList (power 2) [3..10]
[8,16,32,64,128,256,512,1024]
```

Μπορούμε επίσης να σχηματίσουμε συνάρτήσεις μίας μεταβλητής, σταθεροποιώντας το ένα από τα δύο ορίσματα ενός τελεστή. Στη Haskell μία τέτοια συνάρτηση παριστάνεται γράφοντας τον τελεστή με το σταθερό του όρισμα μέσα σε παρένθεση.

Παράδειγμα 62: Το `(2^)` παριστάνει τη συνάρτηση $f(n) = 2^n$, ενώ το `(^2)` παριστάνει τη συνάρτηση $f(n) = n^2$. Η ερμηνεία των `(2+)`, `(*2)`, `(`mod` 2)`, `(`div` 10)`, `(0:)` είναι προφανής. Υπενθυμίζεται ότι οι συναρτήσεις με δύο ορίσματα μπορούν να μετατραπούν σε τελεστές αν κλείσουμε το ονομά τους μέσα σε δύο σύμβολα `.

```
> mapIntList (2^) [3..10]
[8,16,32,64,128,256,512,1024]
> mapIntList (^2) [3..10]
[9,16,25,36,49,64,81,100]
> mapIntList (`power` 2) [3..10]
[9,16,25,36,49,64,81,100]
```

3 Πολυμορφισμός

Μέχρι στιγμής έχουμε δει συναρτήσεις, στις οποίες κάθε παράμετρος ανήκει σε ένα συγκεκριμένο τύπο. Πόλλες από τις συναρτήσεις που έχουμε ορίσει όμως θα μπορούσαν να οριστούν ακριβώς με το ίδιο τρόπο και για άλλους τύπους δεδομένων. Η Haskell μας δίνει τη δυνατότητα να ορίσουμε πολυμορφικές συναρτήσεις, δηλαδή συναρτήσεις οι οποίες μπορούν να δέχονται ορίσματα οποιουδήποτε τύπου. Για να ορίσουμε πολυμορφικές συναρτήσεις χρησιμοποιούμε μεταβλητές τύπου κατά τη δήλωση τύπου της συνάρτησης.

Παράδειγμα 63: ταυτοτική συνάρτηση.

Ονομάζουμε ταυτοτική τη συνάρτηση idn, όπου $idn(x) = x$ για όλα τα x. Η ταυτοτική συνάρτηση ορίζεται με τον ίδιο τρόπο ανεξάρτητα από το ποιο είναι το πεδίο ορισμού της.

Στη Haskell μπορουμε να ορίσουμε την `idn` ως πολυμορφική συνάρτηση:

```
idn :: u -> u
idn x = x
```

Το `u` είναι μία μεταβλητή που παριστάνει έναν οποιονδήποτε τύπο της Haskell. Το `u -> u` δηλωνει οτι η `idn` παίρνει ως είσοδο μία παράμετρο οποιουδήποτε τύπου και επιστρέφει ένα αποτέλεσμα του ίδιου τύπου.

```
> idn 1
1
> idn '1'
'1'
> idn []
[]
```

∎

Παράδειγμα 64: σχηματισμός ζεύγους.

Η παρακάτω συνάρτηση `makepair` παίρνει ως είσοδο δύο τιμές, κάθε μία από τις οποίες ανήκει σε οποιουδήποτε τύπο, και επιστρέφει το ζεύγος που σχηματίζουν:

```
makepair :: v -> u-> (v,u)
makepair a b = (a,b)

> makepair 2 5
(2,5)
> makepair 'a' 'z'
('a','z')
> makepair (1,4) "flower"
((1,4),"flower")
```

∎

Πολλές από τις συναρτήσεις που έχουμε ορίσει για λίστες ακεραίων, μπορούν να μετατραπούν εύκολα σε πολυμορφικές, καθώς ο τρόπος με τον οποίο ορίζεται το επιστρεφόμενο αποτέλεσμα δεν λαμβάνει υπόψη τον τύπο των στοιχείων της λίστας.

Παράδειγμα 65: συνένωση δύο λιστών.

Η συνάρτηση `concInt` συνενώνει δύο λίστες ακεραίων, παραθέτωντας τη μία μετά την άλλη. Η λειτουργία της συνένωσης, μπορεί να εφαρμοστεί για λίστες οποιουδηποτε τύπου. Αν επιχειρήσουμε ωστόσο να χρησιμοποιήσουμε την συνάρτηση αυτή με όρισμα δύο λίστες άλλου τύπου (π.χ. `[Char]`), η Haksell θα μας επιστρέψει ένα μήνυμα λάθους. Το λάθος οφείλεται στο ότι έχουμε δηλώσει ότι η `concInt` δέχεται ως είσοδο δύο λίστα ακεραίων. Μπρούμε να αλλάξουμε τη δήλωση τύπου της συνάρτηση, ώστε αυτή να δέχεται ως ορίσματα λίστες οποιουδήποτε τύπου, δηλαδή να γίνει πολυμορφική:

```
conc :: [u] -> [u] -> [u]
conc (h:t) s = h : conc t s
conc [] s = s
```

Παρατηρούμε ότι ο μόνος περιορισμός που υπάρχει είναι οι δύο λίστες να είναι του ίδιου τύπου, ώστε το αποτέλεσμα να είναι λίστα με στοιχεία ενός μόνο τύπου (αλλιώς δεν θα ήταν αποδεκτη από τη Haskell).

Η `conc` με ορίσματα δύο λίστες ακεραίων είναι ισοδύναμη με την `concInt`, η οποία συνεπώς μπορεί να καταργηθεί. ∎

Παράδειγμα 66: εύρεση του n-οστού στοιχείου μιας λίστας.

Η παρακάτω συνάρτηση είναι η πολυμορφική εκδοχή της `elemIntList`:

```
elemList :: Int -> [u] -> u
elemList 1 (h:t) = h
elemList n (h:t) = elemList (n-1) t
elemList n [] = error "elemList: index out of range"
```

Τονίζεται ότι η πρώτη παράμετρος εξακολουθεί να είναι τύπου `Int`, αφού αυτή καθορίζει τη θέση του στοιχείου στη λίστα.

Η `elemList` με όρισμα λίστα ακεραίων είναι ισοδύναμη με την `elemIntList`, η οποία συνεπώς μπορεί να καταργηθεί. ∎

Αν προσπαθήσουμε να επεκτείνουμε με τον ίδιο τρόπο (δηλαδή αντικαθιστώντας το `Int` με `u` στη δήλωση τύπου) τις συναρτήσεις `member` και `delete`, η Haskell θα μας επιστρέψει ένα μήνυμα λάθους. Αυτό οφείλεται στο ότι οι `member` και `delete` δεν χρησιμοποιούν μόνο τη δομή της λίστας για να καθορίσουν το αποτέλεσμα, αλλά κάνουν και έλεγχο για ισότητα. Δύο παραστάσεις του ίδιου τύπου μπορούν να συγκριθούν για ισότητα σχεδόν για όλους τους τύπους, με κύρια εξαίρεση τους τύπους συναρτήσεων (οι οποίες αποτελούν τύπους δεδομένων στη Haskell).

Η Haskell μας δίνει τη δυνατότητα να ομαδοποιήσουμε ένα σύνολο τύπων σε μία κλαση, για τα μέλη της οποίας θα πρέπει υποχρεωτικά να ορίζεται ένα σύνολο συναρτήσεων και τελεστών που καθορίζονται κατά τον ορισμό της κλάσης. Στη συνέχεια θα χρησιμοποιήσουμε μόνο ορισμένες προκαθορισμένες κλάσεις της Haskell και θα δούμε πώς μπορούμε να ορίσουμε πολυμορφικές συναρτήσεις που ορίζονται μόνο για τους τύπους που ανήκουν σε μία συγκεκριμένη κλάση.

Παράδειγμα 67: διαγραφή της πρώτης εμφάνισης ενός δεδομένου στοιχείου από μία λίστα.

Μπορούμε να υλοποιήσουμε την παραπάνω διαδικασία με μία πολυμορφική συνάρτηση, η οποία θα λειτουργεί για όλους τους τύπους τα στοιχεία των οποίων

μπορούν να συγκριθούν για ισότητα μέ χρήση του τελεστή ==. Η κλάση που περιέχει αυτούς τους τύπους είναι η Eq.

Η παρακάτω συνάρτηση είναι η πολυμορφική εκδοχή της deleteInt:

```
delete :: Eq u => u -> [u] -> [u]
delete n (h:t)
            | n == h
                = t
            | otherwise
                = h:delete n t
delete n [] = []
```

Το (Eq u) => δηλώνει ακριβώς το ότι η συνάρτηση ορίζεται μόνο για τύπους που ανήκουν στην κλάση Eq. ∎

Παράδειγμα 68: ταξινόμηση λίστας με εισαγωγή.

Για να μπορεί να ταξινομηθεί μία λίστα θα πρέπει τα στοιχεία της να ανήκουν σε έναν τύπο οι τιμές του οποίου είναι διατεταγμένες. Οι τύποι της Haskell με την παραπάνω ιδιότητα ανήκουν στην κλάση Ord.

Η παρακάτω συνάρτηση είναι η πολυμορφική εκδοχή της insSortInt, η οποία χρησιμοποιεί την πολυμορφική εκδοχή της insertInt:

```
insSort :: Ord u => [u] -> [u]
insSort (h:t) = insert h (insSort t)
insSort [] = []

insert :: Ord u => u -> [u] -> [u]
insert n (h:t)
            | n <= h
                = n:h:t
            | otherwise
                = h : insert n t
insert n [] = [n]
```

∎

Παράδειγμα 69: υπολογισμός τετραγώνου.

Μπορούμε να ορίσουμε τη συνάρτηση square η οποία θα υπολογιζει το τετράγωνο ενός αριθμού, για οποιονδήποτε αριθμητικό τύπο της Haksell. Οι αριθμητικοί τύποι της Haskell ανήκουν στην κλάση Num.

```
square :: Num u => u -> u
square x = x*x
```

Οι συναρτήσεις `null`, `head`, `tail`, `last`, `init`, `length`, `reverse` είναι ορισμένες στη Haskell ως πολυμορφικές συναρτήσεις και δέχονται ως ορίσματα λίστες οποιουδήποτε τύπου.

4 Πολυμορφικές Συναρτήσεις Υψηλότερης Τάξης

Μπορούμε να συνδυάσουμε τις συναρτήσεις υψηλότερης τάξης με τον πολυμορφισμό. Για παράδειγμα μπορούμε να τροποποιήσουμε τις συναρτήσεις `mapIntList`, `filterIntList` και `foldIntList`, ώστε να δέχονται ως ορίσματα λίστες και συναρτήσεις οποιουδήποτε τύπου.

Παράδειγμα 70: απεικόνιση στοιχείων λίστας.

Στο παράδειγμα 52 ορίσαμε τη συνάρτηση `mapIntList` η οποία απεικονίζει τα στοιχεία μίας λίστας ακεραίων με βάση μία συνάρτηση απεικόνισης, σχηματίζοντας μία λίστα ακεραίων ίδιου μήκους. Μπορούμε να ορίσουμε μία πολυμορφική εκδοχή της `mapIntList`, στην οποία η δεδομένη λίστα θα ανήκει σε οποιονδήποτε τύπο. Επίσης η συνάρτηση απεικόνισης ενδέχεται να έχει πεδίο τιμών διαφορετικό από το πεδίο ορισμού της.

```
mapList :: (u -> v) -> [u] -> [v]
mapList f [] = []
mapList f (h:t) = f h : (mapList f t)
```

Παραδείγματα χρήσης της `mapList`:

```
> mapList sqrtInt [5,10,15,20,25,30]
[2,3,3,4,5,5]
> mapList head [[1,2,3],[4,5,6],[7,8,9]]
[1,4,7]
> mapList (sqrt.sin) [0,pi/8..pi/2]
[0.0,0.6186139,0.8408962,0.9611863,1.0]
> mapList (\x -> 2*x-7) [10..15]
[13,15,17,19,21,23]

> mapList (conc "I am ") ["Mary","John","Chris"]
["I am Mary","I am John","I am Chris"]
> mapList ("("++) (mapList (++")") ["a","b","c"])
["(a)","(b)","(c)"]
> mapList (poly [2,0,3]) [0..4]
[2.0,5.0,14.0,29.0,50.0]
```

Παράδειγμα 71: επιλογή στοιχείων από λίστα.

Η πολυμορφική εκδοχή της `filterIntList`, χρησιμοποιείται για επιλογή στοιχείων από λιστα οποιουδήποτε τύπου:

```
filterList :: (u -> Bool) -> [u] -> [u]
filterList f [] = []
filterList f (h:t)
     | f h
          = h : filterList f t
     | otherwise
          = filterList f t
```

Παραδείγματα χρήσης της `filterList`:

```
> filterList even [1..10]
[2,4,6,8,10]
> filterList (not.null) [[1,2,3],[],[4],[],[5,8]]
[[1,2,3],[4],[5,8]]
> filterList (\x -> x>='A'&&x<='Z') "Compact Disc"
"CD"
```

∎

Παράδειγμα 72: αλυσιδωτή εφαρμογή συνάρτησης στα στοιχεία λίστας.

Η ιδέα της αλυσιδωτής εφαρμογής ενός τελεστή σε όλα τα στοιχεία μίας λίστας μπορεί να επεκταθεί για λίστες οποιουδήποτε τύπου:

```
foldList :: (u -> u -> u) -> [u] -> u
foldList f (h:[]) = h
foldList f (h:t) = f h (foldList f t)
```

Παραδείγματα χρήσης της `foldList`:

```
> foldList (+) [2.3,3.4,10.8]
16.5
> foldList (||) [False,False,True,True]
True
> foldList (++) ["Sword","fish","trombones"]
"Swordfishtrombones"
> foldList (\x y -> x ++ " " ++ y) ["Sword","fish","trombones"]
"Sword fish trombones"
> foldList max ["Greece","Germany","England","France"]
"Greece"
```

∎

Η Haskell έχει προκαθορισμένες τις συναρτήσεις `map`, `filter`, οι οποίες έχουν την ίδια λειτουργία με τις `mapList`, `filterList` αντίστοιχα.

Επίσης η Haskell διαθέτει ένα σύντομο τρόπο για να σχηματίζουμε λίστες που προκύπτουν με απεικόνιση η/και επιλογή στοιχείων μιας δεδομένης λίστας.

Το `[f x | x <- s]` διαβάζεται ως "η λίστα των τιμών `f x`, όπου το `x` ανήκει στη λίστα `s`". Το `x <- s` ονομάζεται γεννήτρια, γιατί δημιουργεί τα στοιχεία από τα οποία σχηματίζεται το αποτέλεσμα.

Η γεννήτρια μπορεί να ακολουθείται από μία ή περισσότερες λογικές συνθήκες που επιλέγουν κάποια από τα στοιχεία που παράγει η γεννήτρια. Για παράδειγμα το `[x | x <- s, x>0]` διαβάζεται ως "η λίστα των στοιχείων της `s`, που είναι θετικοί αριθμοί".

Θα μπορούσαμε να ορίσουμε εναλλακτικά τις `mapList` και `filterList` χρησιμοποιώντας τις παραπάνω συντομογραφίες:

```
mapList' :: (u -> v) -> [u] -> [v]
mapList' f s = [f x | x <- s]

filterList' :: (u -> Bool) -> [u] -> [u]
filterList' f s = [x | x <- s, f x]
```

Μπορούμε να κάνουμε ταυτόχρονα επιλογή και απεικόνιση:

```
> [sqrt x | x <- [1.0,-1.0,2.0,-2.0], x>=0]
[1.0,1.4142135623731]
> [head x | x <- [[1,2,3],[],[4,5]], not (null x)]
[1,4]
```

Μπορούμε να έχουμε περισσότερες από μία γεννήτριες:

```
> [(x,y) | x <- [1..3], y <- ['a'..'c']]
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c'),
(3,'a'),(3,'b'),(3,'c')]
> [[x..y] | x<- [1..4], y <- [1..4], y-x>1]
[[1,2,3],[1,2,3,4],[2,3,4]]
```

5 Λίστες με στοιχεία λίστες

Τα στοιχεία μίας λίστας ενδέχεται να είναι επίσης λίστες. Χρησιμοποιώντας λίστες αυτής της μορφής μπορούμε να παραστήσουμε πίνακες. Κάθε εσωτερική λίστα αντιστοιχεί σε μία γραμμή του πίνακα.

Είναι προφανές ότι υπάρχουν λίστες της παραπάνω μορφής που δεν παριστάνουν πίνακες. Για να αποτελεί μία λίστα από λίστες αναπαράσταση ενός πίνακα

θα πρέπει να είναι μή κενή και τα στοιχεία της να είναι μή κενές λίστες που όλες έχουν το ίδιο πλήθος στοιχείων.

Παράδειγμα 73: ανάστροφος πίνακα.

Η παρακάτω συνάρτηση δέχεται ως είσοδο μία λίστα από λίστες, η οποία θα πρέπει να παριστάνει έναν πίνακα, και επιστρέφει τον ανάστροφό του.

Αν ο πίνακας έχει ένα μόνο στοιχείο, τότε επιστρέφεται ως έχει. Αν αποτελείται από μία μόνο γραμμή μήκους $n > 1$, τότε επιστρέφεται η λίστα που περιέχει τις n λίστες μήκους 1 που προκύπτουν με διάσπαση της μοναδικής γραμμής του αρχικού πίνακα, έτσι ώστε ο επιστρεφόμενος πίνακας να έχει n γραμμές και 1 στήλη.

Αν ο πίνακας έχει περισσότερες από μία γραμμές, τότε επιστρέφεται το αποτέλεσμα της βοηθητικής συνάρτησης `transposeHlp` με ορίσματα την λίστα που παριστάνει την πρώτη γραμμή του πίνακα και τη λίστα από λίστες που παριστάνει το ανάστροφο του πίνακα που προκύπτει από τον αρχικό αν διαγραφεί η πρώτη του γραμμή.

Η `transposeHlp` με ορίσματα μία λίστα και μία λίστα από λίστες, επιστρέφει μία λίστα από λίστες που προκύπτει τοποθετώντας κάθε στοιχείο της πρώτης λίστας στην αρχή της αντίστοιχης λίστας-στοιχείου της δεύτερης λίστας.

```
transposeIntMatrix :: [[Int]] -> [[Int]]
transposeIntMatrix ((h:[]):[]) = [[h]]
transposeIntMatrix ((h:t):[]) =
        [h] : transposeIntMatrix [t]
transposeIntMatrix (r:m) =
        transposeHlp r (transposeIntMatrix m)
        where transposeHlp :: [Int]->[[Int]]->[[Int]]
              transposeHlp [] [] = []
              transposeHlp (h:t) (r:m)
                   = (h:r):(transposeHlp t m)
```

∎