1 Λίστες

Στη Haskell ο τύπος $[t]$ έχει πεδίο τιμών τις λίστες με στοιχεία τύπου t. Η Haskell διαθέτει ένα πλήθος τελεστών και συναρτήσεων που υλοποιούν βασικές λειτουργίες πάνω σε λίστες (οποιουδήποτε τύπου):

- Ο τελεστής `:` χρησιμοποιείται για κατασκευή λίστας. Συγκεκριμένα το `a:s` επιστρέφει τη λίστα που προκύπτει από την προσθήκη του `a` στην αρχή της λίστας `s`. Το `a` θα πρέπει να έχει ίδιο τύπο με τα στοιχεία της `s`.

  ```
  > 1 : [2,3,4]
  [1,2,3,4]
  > 'a' : "bcd"
  "abcd"
  ```

  (υπενθυμίζεται ότι ο τύπος `String` είναι ορισμένος ως `[Char]`).

- Ο τελεστής `++` συνενώνει δύο λίστες του ίδιου τύπου.

  ```
  > [1,2] ++ [3,4,5]
  [1,2,3,4,5]
  > "Has" ++ "kell"
  "Haskell"
  ```

- Η συνάρτηση `length` επιστρέφει το μήκος μίας λίστας.

  ```
  > length [1,5,4,5]
  4
  > length []
  0
  ```

- Η συνάρτηση `reverse` επιστρέφει την αντίστροφη μίας λίστας.

  ```
  > reverse [2,3,4,5]
  [5,4,3,2]
  ```

- Η συνάρτηση `null` ελέγχει αν μία λίστα είναι κενή ή όχι.

  ```
  > null [4,5,6]
  False
  > null []
  True
  ```

- Η συνάρτηση `head` επιστρέφει την κεφαλή (δηλαδή το πρώτο στοιχείο) μία μη κενής λίστας.

```
> head [1,2,3,4,5]
1
```

- Η συνάρτηση `tail` επιστρέφει την ουρά μίας μη κενής λίστας, δηλαδή τη λίστα που προκύπτει αν διαγραφεί η κεφαλή της λίστας.

```
> tail [1,2,3,4,5]
[2,3,4,5]
```

- Η συνάρτηση `last` επιστρέφει το τελευταίο στοιχείο μία μη κενής λίστας.

```
> last [1,2,3,4,5]
5
```

- Η συνάρτηση `init` επιστρέφει τη λίστα που προκύπτει αν διαγραφεί το τελευταίο στοιχείο της λίστας

```
> init [1,2,3,4,5]
[1,2,3,4]
```

Οι συναρτήσεις `head`, `tail`, `last`, `init` προκαλούν άμεσο τερματισμό του προγράμματος αν αποτιμηθούν με όρισμα την κενή λίστα και εμφανίζουν μήνυμα λάθους.

Η Haskell υποστηρίζει τις παρακάτω συντομογραφίες για λίστες ακεραίων:

- `[m .. n]` είναι η λίστα όλων των αριθμών από το `m` μέχρι και το `n` (ή η κενή λίστα αν `m>n`)

```
> [0..7]
[0,1,2,3,4,5,6,7]
> [7..0]
[]
```

- `[m,k .. n]` είναι η λίστα όλων των ακεραίων που προκύπτουν ξεκινώντας από το `m` και προχωρώντας με βήμα `k-m` μέχρι το `n`.

```
> [1,3..9]
[1,3,5,7,9]

> [1,4..12]
[1,4,7,10]

> [7,6..0]
[7,6,5,4,3,2,1,0]
```

Επίσης υποστηρίζει παρόμοιες συντομογραφίες για λίστες άλλων τύπων:

```
> ['a','d'..'z']
"adgjmpsvy"
> [0.1,0.2..1.0]
[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
```

Στη συνέχεια θα περιγράψουμε ορισμένες συναρτήσεις που υλοποιούν βασικές λειτουργίες σε λίστες.

Στον ορισμό συναρτήσεων που περιέχουν λίστες χρησιμοποιούμε το πρότυπο `(h:t)` για να δηλώσουμε τη (μη κενή) λίστα με κεφαλή το `h` και ουρά το `t` (όπου στη θέση των `h` και `t` μπορούμε να βάλουμε οποιαδήποτε ονόματα μεταβλητών).

Οι παρακάτω συναρτήσεις ορίζονται για λίστες ακεραίων. Θα δούμε αργότερα πως μπορούμε να ορίσουμε συναρτήσεις που παίρνουν ως ορίσματα λίστες οποιουδήποτε τύπου.

Για να επεξεργαστούμε τα στοιχεία μίας λίστας μέσα σε μία συνάρτηση χρησιμοποιούμε αναδρομή. Η πιό απλή περίπτωση αναδρομικής συνάρτησης που μπορεί να οριστεί πάνω σε λίστες:

- καθορίζει την τιμή του αποτελέσματος για την κενή λίστα χωρίς αναδρομή.
- καθορίζει την τιμή του αποτελέσματος για μή κενή λίστα χρησιμοποιώντας την κεφαλή της λίστας και την τιμή της ίδιας της συνάρτησης για την ουρά της λίστας.

Υπάρχουν όμως και άλλες μορφές αναδρομής, όπως θα φανεί στα επόμενα παραδείγματα.

Παράδειγμα 30: άθροισμα στοιχείων λίστας ακεραίων.

Αν η λίστα δεν είναι κενή, τότε το άθροισμα υπολογίζεται αθροίζοντας την κεφαλή της λίστας με το αντίστοιχο άθροισμα για την ουρά της, το οποίο υπολογίζεται με αναδρομή. Αν η λίστα είναι κενή τότε το άθροισμα έχει τιμή 0.

```
sumIntList :: [Int] -> Int
sumIntList (h:t) = h + sumIntList t
sumIntList [] = 0
```

Επειδή οποιαδήποτε λίστα ταιριάζει ακριβώς με ένα από τα πρότυπα `(h:t)` και `[]`, η σειρά των εξισώσεων στον ορισμό της `sumIntList` δεν επηρεάζει το επιστρεφόμενο αποτέλεσμα (αντίθετα από ότι συμβαίνει στους ακέραιους, όπου το 0 ταιριάζει με το γενικό πρότυπο `n`, συνεπώς η τιμή της συνάρτησης για το 0 πρέπει να ορίζεται πρίν από το ορισμό για το `n`). Επειδή κατά τον αποτίμηση της sumIntList με είσοδο μία μη κενή λίστα, η ισότητα για το

πρότυπο `(h:t)` χρησιμοποιείται πολλές φορές (όσες και το μήκος της λίστας) ενώ η ισότητα για το πρότυπο `[]` χρησιμοποιείται μόνο μία φορά, τοποθετούμε πρώτη την ισότητατα για το πρότυπο `(h:t)`. Σημειώνεται ότι θα μπορούσαμε να αντικαταστήσουμε το πρότυπο `[]` με το γενικό πρότυπο `s`, καθώς η μόνη λίστα που δεν ταιριάζει με το `(h:t)` είναι η κενή λίστα. Για λόγους σαφήνειας επιλέγουμε το πρότυπο `[]`.

Η αποτίμηση της παράστασης `sumIntList[2,7,8]` γίνεται με τον παρακάτω τρόπο:

```
  sumIntList [2,7,8]
     #    [2,7,8] <<< (h:t)
      <=> 2:[7,8] <<< (h:t)
      <=> "yes"
= 2 + (sumIntList [7,8])
     #    [7,8] <<< (h:t)
      <=> 7:[8] <<< (h:t)
      <=> "yes"
= 2 + (7 + (sumIntList [8]))
     #    [8] <<< (h:t)
      <=> 8:[] <<< (h:t)
      <=> "yes"
= 2 + (7 + (8 + (sumIntList [])))
     #    [] <<< (h:t)
      <=> "no"
     #    [] <<< []
      <=> "yes"
= 2 + (7 + (8 + 0))
= 2 + (7 + 8)
= 2 + 15
17
```

Μπορούμε αντί για πρότυπα να χρησιμοποιήσουμε τις προκαθορισμένες συναρτήσεις `null`, `head` και `tail` για να ελέγξουμε αν μία λίστα είναι κενή και να πάρουμε την κεφαλή και την ουρά της.

Θα μπορούσαμε να αντικαταστήσουμε την `sumIntList` με την παρακάτω ισοδύναμη συνάρτηση:

```
sumIntList' :: [Int] -> Int
sumIntList' s
         | null s
            = 0
         | otherwise
            = (head s) + sumIntList' (tail s)
```

Στη συνέχεια θα προτιμούμε τη χρήση προτύπων. ∎

Παράδειγμα 31: έλεγχος αν ένας αριθμός εμφανίζεται σε μία λίστα ακεραίων.

Ένας αριθμός εμφανίζεται σε μία λίστα αν ταυτίζεται με την κεφαλή της ή εμφανίζεται στην ουρά της. Η δεύτερη συνθήκη μπορεί να ελεγχθεί χρησιμοποιώντας αναδρομή. Αν η λίστα είναι κενή τότε προφανώς ο αριθμός δεν εμφανίζεται σε αυτή.

```
memberInt :: Int -> [Int] -> Bool
memberInt n (h:t) = n == h || memberInt n t
memberInt n [] = False
```

Η Haskell κάνει μερική αποτίμηση στα ορίσματα του `&&`. Αυτό σημαίνει πως αν το `n == h` έχει τιμή `True`, τότε η συνολική παράσταση αποτιμάται αυτόματα σε `True` χωρίς να γίνεται αποτίμηση του `memberInt n t`. ∎

Παράδειγμα 32: διαγραφή ενός δεδομένου αριθμού από μία λίστα ακεραίων.

Η παρακάτω συνάρτηση επιστρέφει τη λίστα που προκύπτει αν διαγραφεί η πρώτη εμφάνιση του αριθμού από τη λιστα, εφόσον ο αριθμός περιέχεται στη λίστα, αλλιώς επιστρέφει την αρχική λίστα αναλλοίωτη.

Αν ο αριθμός ταυτίζεται με την κεφαλή της λίστας, τότε επιστρέφεται η ουρά της. Αλλιώς, επιστρέφεται η λίστα με κεφαλή ίδια με αυτή της αρχικής και ουρά τη λίστα που προκύπτει αν ο αριθμός διαγραφεί από την ουρά της αρχικής. Αν η λίστα είναι κενή, τότε επιστρέφεται η κενή λίστα.

```
deleteInt :: Int -> [Int] -> [Int]
deleteInt n (h:t)
            | n == h
                = t
            | otherwise
                = h:deleteInt n t
deleteInt n [] = []
```

Η αποτίμηση της παράστασης `deleteInt 4 [7,4,5,9,1,1,0]` γίνεται με τον παρακάτω τρόπο:

```
  deleteInt 4 [7,4,5,9,1,1,0]
     #    [7,4,5,9,1,1,0] <<< (h:t)
      <=> 7:[4,5,9,1,1,0] <<< (h:t)
      <=> "yes"
     ?  4 == 7
      = False
     ?  otherwise
      = True
= 7 : (deleteInt 4 [4,5,9,1,1,0])
     #    [4,5,9,1,1,0] <<< (h:t)
```

```
      <=> 4:[5,9,1,1,0] <<< (h:t)
      <=> "yes"
     ?  4 == 4
      = True
= 7 : [5,9,1,1,0]
= [7,5,9,1,1,0]
```

∎

Ενδέχεται η μή αναδρομική ισότητα στον ορισμό μιας συνάρτησης να ορίζει την τιμή της για μή κενές λίστες.

Παράδειγμα 33: μέγιστο στοιχείο λίστας ακεραίων.

Η παρακάτω συνάρτηση `maxIntList` προϋποθέτει ότι η λιστα είναι μη κενή (σε αντίθετη περίπτωση δεν μπορεί να οριστεί το μέγιστο στοιχείο με προφανή τρόπο). Αν η λίστα έχει ένα μονο στοιχείο, τότε αυτό είναι και το μέγιστο. Σε αντίθετη περίπτωση, το μέγιστο στοιχείο της λίστας είναι το μεγαλύτερο ανάμεσα στην κεφαλή και στο μέγιστο στοιχείο της ουράς.

```
maxIntList :: [Int] -> Int
maxIntList (h:[]) = h
maxIntList (h:t) = max h (maxIntList t)
```

∎

Παράδειγμα 34: εισαγωγή στοιχείου σε ταξινομημένη λίστα ακεραίων.

Η παρακάτω συνάρτηση `insertInt` παίρνει ως είσοδο ένα ακέραιο και μία λίστα ακεραίων, που θα πρέπει είναι ταξινομημένη σε αύξουσα τάξη, και επιστρέφει την ταξινομημένη λίστα που προκύπτει αν εισαχθεί ο ακέραιος στην κατάλληλη θέση της αρχικής λίστας.

Αν ο ακέραιος που θέλουμε να εισάγουμε είναι μικρότερος ή ίσος από την κεφαλή της δεδομένης λίστας, τότε είναι μικρότερος ή ίσος από όλα τα στοιχεία της. Σε αυτή την περίπτωση επιστρέφεται η λίστα που προκύπτει τοποθετώντας τον αριθμο στην αρχή της αρχικής λίστας. Αλλιώς, η επιστρεφόμενη λίστα έχει κεφαλή ίδια με της αρχικής λίστας, ενώ η ουρά της είναι η λίστα που προκύπτει με εισαγωγή του αριθμού στη ουρά της αρχικής λίστας. Αν η αρχική λίστα είναι κενή, επιστρέφεται μία λίστα με ένα μόνο στοιχείο (τον αριθμό που θέλουμε να εισάγουμε), η οποία είναι προφανώς ταξινομημένη.

```
insertInt :: Int -> [Int] -> [Int]
insertInt n (h:t)
            | n <= h
                = n:h:t
            | otherwise
                = h : insertInt n t
insertInt n [] = [n]
```

Αν η αρχική λίστα δεν είναι ταξινομημένη σε αύξουσα τάξη, η επιστρεφόμενη λίστα θα περιέχει όλα τα στοιχείο της αρχικής καθώς και τον νέο αριθμό, χωρίς ωστόσο να είναι ταξινομημένη. ∎

Παράδειγμα 35: εύρεση του n-οστού στοιχείου μιας λίστας ακεραίων.

Η παρακάτω συνάρτηση προϋποθέτει ότι το n αντιστοιχεί πράγματι σε κάποιο στοιχείο της λίστας (δηλαδή είναι ένας αριθμός μεταξύ του 1 και του μήκους της λίστας). Αν το n έχει τιμή 1, τότε επιστρέφεται η κεφαλή της λίστας. Σε αντίθετη περίπτωση το ζητούμενο στοιχείο είναι το $(n-1)$-οστό στοιχείο της ουράς της λίστας, το οποίο υπολογίζεται αναδρομικά. Αν το n δεν αντιστοιχεί σε στοιχείο της λίστας, τότε η αναδρομή θα οδηγήσει σε αποτίμηση της συνάρτησης με ορισμα την κενή λίστα. Σε αυτή την περίπτωση εμφανίζεται μήνυμα λάθους.

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

Η αποτίμηση της παράστασης `elemIntList 3 [1,9,6,8]` γίνεται με τον παρακάτω τρόπο:

```
  elemIntList 3 [1,9,6,8]
     #    3 <<< 1
      <=> "no"
     #    [1,9,6,8] <<< (h:t)
      <=> 1:[9,6,8] <<< (h:t)
      <=> "yes"
= elemIntList (3-1) [9,6,8]
     #    (3-1) <<< 1
      <=>  2 <<< 1
      <=> "no"
     #    [9,6,8] <<< (h:t)
      <=> 9:[6,8] <<< (h:t)
      <=> "yes"
= elemIntList (2-1) [6,8]
     #    (2-1) <<< 1
      <=>  1 <<< 1
```

```
      <=> "yes"
     #    [6,8] <<< (h:t)
      <=> 6:[8] <<< (h:t)
      <=> "yes"
= 6
```

Παράδειγμα 36: συνένωση δύο λιστών ακεραίων.

Συνένωση δύο λιστών ονομάζεται η λίστα που περιέχει όλα τα στοιχεία της πρώτης λίστας ακολουθούμενα από όλα τα στοιχεία της δεύτερης. Η κεφαλή της λίστας που προκύπτει από τη συνένωση δύο λιστών είναι η κεφαλή της πρώτης ενώ η ουρά της προκύπτει με συνένωση της ουράς της πρώτης λίστας με ολόκληρη τη δεύτερη λίστα. Αν η πρώτη λίστα είναι κενή, τότε το αποτέλεσμα της συνένωσης είναι η δεύτερη λίστα.

```
concInt :: [Int] -> [Int] -> [Int]
concInt (h:t) s = h : concInt t s
concInt [] s = s
```

Η αποτίμηση της παράστασης `concInt [2,3] [1,1,9,8]` γίνεται με τον παρακάτω τρόπο:

```
  concInt [2,3,4] [1,1,9,8]
     #    [2,3,4] <<< (h:t)
      <=> 2:[3,4] <<< (h:t)
      <=> "yes"
= 2 : (concInt [3,4] [1,1,9,8])
     #    [3,4] <<< (h:t)
      <=> 3:[4] <<< (h:t)
      <=> "yes"
= 2 : (3 : (concInt [4] [1,1,9,8]))
     #    [4] <<< (h:t)
      <=> 4:[] <<< (h:t)
      <=> "yes"
= 2 : (3 : (4: (concInt [] [1,1,9,8])))
     #    [] <<< (h:t)
      <=> "no"
     #    [] <<< []
      <=> "yes"
= 2 : (3 : (4: [1,1,9,8]))
= 2 : (3 : [4,1,1,9,8])
= 2 : [3,4,1,1,9,8]
= [2,3,4,1,1,9,8]
```

Παράδειγμα 37: σχηματισμός λίστας ζευγών από δύο δεδομένες λίστες ακεραίων.

Η παρακάτω συνάρτηση με είσοδο δύο λίστες ακεραίων, επιστρέφει μία λίστα από ζεύγη τέτοια ώστε το i-οστό ζεύγος του αποτελέσματος να αποτελείται από τα i-οστά στοιχεία των δύο λιστών. Αν οι λίστες έχουν διαφορετικά μήκη τα επιπλέον στοιχεία της μεγαλύτερης λίστας παραλείπονται.

Αν και οι δύο λίστες είναι μη κενές, τότε η κεφαλή της επιστρεφόμενης λίστας είναι ένα ζεύγος που απαρτίζεται από τις κεφαλές των δύο αρχικών λιστών, ενώ η ουρά της προκύπτει αναδρομικά σχηματίζοντας ζεύγη από τα στοιχεία των ουρών των αρχικών λιστών. Αν κάποια από τις δύο λίστες είναι κενή, επιστρέφεται η κενή λίστα.

```
makePairsInt :: [Int] -> [Int] -> [(Int,Int)]
makePairsInt (h1:t1) (h2:t2)
     = (h1,h2) : makePairsInt t1 t2
makePairsInt r s = []
```

∎

Παράδειγμα 38: συγχώνευση ταξινομημένων λιστών ακεραίων.

Η παρακάτω συνάρτηση `mergeInt` δέχεται ως είσοδο δύο λιστες ακεραίων, οι οποίες θα πρέπει να είναι ταξινομημένες και επιστρέφει μία λίστα με τα στοιχεία και των δυο λιστών η οποία είναι επίσης ταξινομημένη. Το πλήθος των εμφανίσεων ενός στοιχείου στην τελική λίστα ισούται με το άθροισμα των εμφανίσεων του στις δύο αρχικές λίστες.

Η κεφαλή της επιστρεφόμενης λίστας είναι η μικρότερη από τις κεφαλές των δύο λιστών. Η ουρά της επιστρεφόμενης λίστας προκύπτει αναδρομικά με συγχώνευση της ουράς της λίστας που αρχικά είχε τη μικρότερη κεφαλή και ολόκληρης της άλλης λίστας.

```
mergeInt :: [Int] -> [Int] -> [Int]
mergeInt r@(h1:t1) s@(h2:t2)
     | h1 <= h2
          = h1 : mergeInt t1 s
     | otherwise
          = h2 : mergeInt r t2
mergeInt r [] = r
mergeInt [] s = s
```

Το `r@(h1:t1)` αναθέτει το όνομα `r` στη λίστα `(h1:t1)`.

Η αποτίμηση της παράστασης `mergeInt [1,4] [2,3]` γίνεται με τον παρακάτω τρόπο:

```
  mergeInt [2,3] [1,4,5]
     #    [2,3] <<< (h1:t1)
      <=> 2:[3] <<< (h1:t1)
      <=> "yes"
     #    [1,4,5] <<< (h2:t2)
      <=> 1:[4,5] <<< (h2:t2)
      <=> "yes"
     ?  2 <= 1
      = False
     ?  otherwise
      = True
= 1 : (mergeInt [2,3] [4,5])
     #    [2,3] <<< (h1:t1)
      <=> 2:[3] <<< (h1:t1)
      <=> "yes"
     #    [4,5] <<< (h2:t2)
      <=> 4:[5] <<< (h2:t2)
      <=> "yes"
     ?  2 <= 4
      = True
= 1 : (2: (mergeInt [3] [4,5]))
     #    [3] <<< (h1:t1)
      <=> 3:[] <<< (h1:t1)
      <=> "yes"
     #    [4,5] <<< (h2:t2)
      <=> 4:[5] <<< (h2:t2)
      <=> "yes"
     ?  3 <= 4
      = True
= 1 : (2: (3: (mergeInt [] [4,5])))
     #    [] <<< (h1:t1)
      <=> "no"
     #    [4,5] <<< []
      <=> "no"
     #    [] <<< []
      <=> "yes"
= 1 : (2: (3: [4,5]))
= 1 : (2: [3,4,5])
= 1 : [2,3,4,5]
= [1,2,3,4,5]
```

∎

Παράδειγμα 39: έλεγχος για το αν μια λίστα ακεραίων είναι ταξινομημένη σε αύξουσα τάξη.

Μία λίστα με τουλάχιστον δύο στοιχεία είναι ταξινομημένη σε αύξουσα τάξη αν το πρώτο στοιχείο είναι μικρότερο ή ίσο από το δεύτερο και η ουρά της είναι ταξινομημένη. Ο έλεγχος για την ουρά γίνεται χρησιμοποιώντας αναδρομή. Μιά λίστα με λιγότερα από δύο στοιχεία είναι προφανώς ταξινομημένη.

Στην Haskell το πρότυπο `(a:b:r)` περιγράφει μια λίστα με τουλάχιστον δύο στοιχεία, όπου το πρωτο συμβολίζεται `a` και το δεύτερο `b`, ενώ η λίστα που αποτελείται από τα υπόλοιπα στοιχεία συμβολίζεται `r`.

```
isSortedInt :: [Int] -> Bool
isSortedInt (a:b:r) = a <= b && isSortedInt (b:r)
isSortedInt s = True
```

Λόγω της μερικής αποτίμησης, αν το `a <= b` έχει τιμή `False`, τότε η παράσταση στο δεξί μελός της πρώτης ισότητας αποτιμάται άμεσα σε `False` χωρίς να γίνεται η αποτίμηση του `isSortedInt (b:r)`. ∎

Παράδειγμα 40: διαχωρισμός μίας λίστας ακεραίων σε δύο λίστες με σχεδόν ίσο μήκος.

Η παρακάτω συνάρτηση `splitInt` χωρίζει τα στοιχεία μίας δεδομένης λίστας σε δύο λιστες, τις οποίες επιστρέφει ως ζεύγος. Η πρώτη λίστα να περιέχει τα στοιχεία της αρχικής που εμφανίζονται σε θέσεις περιττής τάξης και η δεύτερη τα υπόλοιπα. Αν η αρχική λίστα έχει άρτιο μήκος οι δύο επιστρεφόμενες λίστες έχουν το ίδιο μήκος, ενώ σε αντίθετη περίπτωση η πρώτη λίστα έχει ένα στοιχείο περισσότερο από τη δεύτερη.

Αν η αρχική λίστα έχει τουλάχιστον δύο στοιχεία, τότε τα στοιχεία αυτά αποτελούν τις κεφαλές των επιστρεφόμενων λιστών, ενώ η ουρές των επιστρεφόμεων λιστών προκύπτουν με αναδρομική διάσπαση της λίστας που προκύπτει από την αρχική με διαγραφή των δύο πρώτων στοιχείων της. Αν η αρχική λίστα είναι κενή τότε το επιστρεφόμενο ζεύγος αποτελείται από δύο κενές λίστες, ενώ αν περιέχει ένα στοιχείο αυτό το επιστρεφόμενο ζεύγος αποτελείται από τη αρχική λίστα και την κενή.

```
splitInt :: [Int] -> ([Int],[Int])
splitInt (a:b:t) = (a:r,b:s)
         where (r,s) = splitInt (t)
splitInt (a:[]) = ([a],[])
splitInt [] = ([],[])
```

∎

Παράδειγμα 41: ταξινόμηση λίστας ακεραίων.

Η ταξινόμηση μιας λίστας μπορεί να γίνει με πολλούς διαφορετικούς αλγόριθμους. Στη συνέχεια περιγράφουμε δύο συναρτήσεις για ταξινόμηση λίστας που χρησιμοποιούν κάποιες συναρτήσεις που έχουμε ήδη ορίσει.

Η συνάρτηση `insSortInt` κάνει ταξινόμηση με εισαγωγή: αν η λίστα είναι μη κενή επιστρέφει τη λίστα που προκύπτει αν εισαχθεί η κεφαλή της αρχικής λίστας στη ταξινομημένη λίστα που περιέχει ακριβώς τα στοιχεία της ουράς της αρχικής λίστας, η οποία υπολογίζεται αναδρομικά.

```
insSortInt :: [Int] -> [Int]
insSortInt (f:r) = insertInt f (insSortInt r)
insSortInt [] = []
```

Η αποτίμηση της παράστασης `inssort [5,3,0]` γίνεται με τον παρακάτω τρόπο:

```
  insSortInt [5,3,0]
     #    [5,3,0] <<< (f:r)
      <=> 5:[3,0] <<< (f:r)
      <=> "yes"
= insertInt 5 (insSortInt [3,0])
     #    insSortInt [3,0] <<< (h:t)
             #    [3,0] <<< (f:r)
              <=> 3:[0] <<< (f:r)
              <=> "yes"
      <=> insertInt 3 (insSortInt [0]) <<< (h:t)
             #    (insSortInt [0] <<< (h:t)
                     #    [0] <<< (f:r)
                      <=> 0:[] <<< (f:r)
                      <=> "yes"
               <=> insertInt 0 (insSortInt []) <<< (h:t)
                     #    insSortInt [] <<< (h:t)
                             #    [] <<< (f:r)
                              <=> "no"
                             #    [] <<< []
                              <=> "yes"
                      <=> [] <<< (h:t)
                      <=> "no"
                      <=> [] <<< []
                      <=> "yes"
               <=> [0] <<< (h:t)
               <=> 0:[] <<< (h:t)
               <=> "yes"
              ?  3 <= 0
               = False
```

```
              ?  otherwise
               = True
       <=> 0 : (insertInt 3 []) <<< (h:t)
       <=> "yes"
      ?  5 <= 0
        = False
      ?  otherwise
        = True
= 0: (insertInt 5 (insertInt 3 []))
      #    insertInt 3 [] <<< (h:t)
              #    [] <<< (h:t)
               <=> "no"
              #    [] <<< []
               <=> "yes"
       <=> [3] <<< (h:t)
       <=> 3:[] <<< (h:t)
       <=> "yes"
      ?  5 <= 3
       = False
      ?  otherwise
       = True
= 0: (3 : (insertInt 5 []))
      #    [] <<< (h:t)
       <=> "no"
      #    [] <<< []
       <=> "yes"
= 0 : (3 : [5])
= 0 : [3,5]
= [0,3,5]
```

Στην παραπάνω αποτίμηση φαίνεται ο τρόπος με τον οποίο λειτουργεί η οκνηρή αποτίμηση στην περίπτωση σύνθετων τύπων, όπως είναι η λίστες. Για παράδειγμα, για την αποτίμηση της παράστασης `insertInt 5 (insSortInt [3,0])` είναι απαραίτητο να εξεταστεί αν η `insSortInt [3,0]` είναι μη κενή λίστα ώστε να ταιριάζει με το πρότυπο `(h:t)` στον ορισμό της `insertInt`. Όταν διαπιστωθεί ότι η `insSortInt [3,0]` είναι λίστα με κεφαλή το 0, η πληροφορία αυτή χρησιμοποιήται για να συνεχιστεί ο υπολογισμός, χωρίς όμως να γίνει άμμεση αποτίμηση της ουράς της. Η ουρά της αποτιμάται αργότερα, όταν διαπιστωθεί ότι είναι επίσης απαραίτητη για την εξαγωγή του τελικού αποτελέσματος.

Η συνάρτηση `mergeSortInt` κάνει ταξινόμηση με συγχώνευση: αν η λίστα έχει τουλάχιστον δύο στοιχεία, τότε επιστρέφει τη συγχώνευση των λιστών που προκύπτουν ταξινομώντας αναδρομικά τις λίστες στις οποίες διασπάται η αρχική λιστα μέσω της `splitInt`.

```
mergeSortInt :: [Int] -> [Int]
mergeSortInt s@(a:b:t)
     = mergeInt (mergeSortInt r1) (mergeSortInt r2)
     where (r1,r2) = splitInt s
mergeSortInt s = s
```

Παρότι η `mergeSortInt` φαίνεται πιο περίπλοκη, εντούτοις είναι πιο γρήγορη από την `insSortInt`. ∎

2 Οκνηρή Αποτίμηση και Απειρες Λίστες

Λόγω της οκνηρής αποτίμησης, αν ένας υπολογισμός περιέχει μία λίστα (ή γενικότερα σύνθετο τύπο) τότε αποτιμάται μόνο το τμήμα της λίστας που είναι απαραίτητο για τον υπολογισμό.

Στα παραδείγματα αποτίμησης παραστάσεων με λίστες που έχουμε δει μέχρι τώρα, δεν υπήρχε κανένα κέρδος από τη χρήση της οκνηρής αποτίμησης. Το παρακάτω παράδειγμα δείχνει το πως η αποτίμηση ενός μόνο τμήματος μίας λίστας μπορεί να συνεπάγεται εξοικονόμηση χρόνου εκτέλεσης:

Παράδειγμα 42: Η αποτίμηση της παράστασης

```
elemIntList 2 (mergeInt [1,4,8,9,11,15,17] [2,3,5,7,12,16,18])
```

γίνεται με τον παρακάτω τρόπο:

```
  elemIntList 2 (mergeInt [1,4,8,9,11,17] [2,3,5,7,12,16])
     #    2 <<< 1
      <=> "no"
     #    mergeInt [1,4,8,9,11,17] [2,3,5,7,12,16] <<< (h:t)
          #    [1,4,8,9,11,17] <<< (h1:t1)
           <=> 1:[4,8,9,11,17] <<< (h1:t1)
           <=> "yes"
          #    [2,3,5,7,12,16] <<< (h2:t2)
           <=> 2:[3,5,7,12,16] <<< (h2:t2)
           <=> "yes"
          ?  1 <= 2
           = True
      <=> 1 : (mergeInt [4,8,9,11,17] [2,3,5,7,12,16]) <<< (h:t)
      <=> "yes"
= elemIntList (2-1) (mergeInt [4,8,9,11,17] [2,3,5,7,12,16])
     #    (2-1) <<< 1
      <=> 1 <<< 1
      <=> "yes"
     #    mergeInt [4,8,9,11,17] [2,3,5,7,12,16] <<< (h:t)
          #    [4,8,9,11,17] <<< (h1:t1)
```

```
           <=> 4:[8,9,11,17] <<< (h1:t1)
            <=> "yes"
           #    [2,3,5,7,12,16] <<< (h2:t2)
            <=> 2:[3,5,7,12,16] <<< (h2:t2)
            <=> "yes"
           ?  4 <= 2
            = False
           ?  otherwise
            = True
      <=> 2 : (mergeInt [4,8,9,11,17] [3,5,7,12,16]) <<< (h:t)
      <=> "yes"
= 2
```

Παρατηρούμε ότι γίνεται μερικός υπολογισμός της λίστας που επιστρέφει η `mergeInt`, καθώς για να βρούμε το δεύτερο στοιχείο της αρκεί να βρούμε τα δύο πρώτα, ενώ τα υπόλοιπα δεν επηρεάζουν την τιμή της παράστασης. ∎

Στη Haskell μπορούμε να εύκολα να ορίσουμε άπειρες λίστες.

Παράδειγμα 43: Η παρακάτω λίστα `inf` είναι άπειρη:

```
inf :: [Int]
inf = 0 : inf
```

Αν επιχειρήσουμε να υπολογίσουμε την τιμή της `inf`, θα ξεκινήσει ένας ατέρμονος υπολογισμός:

```
  inf
= 0 : inf
= 0 : (0 : inf)
= 0 : (0 : (0 : inf))
= 0 : (0: (0 : (0 : inf)))
= 0 : (0: (0: (0 : (0 : inf))))
...
```

∎

Παράδειγμα 44: Η παρακάτω συνάρτηση επιστρέφει πάντα μία άπειρη λίστα, που περιέχει όλους τους όρους μιας αριθμητικής προόδου:

```
makeList :: Int -> Int -> [Int]
makeList a d = a : makeList (a+d) d
```

Αν επιχειρήσουμε να υπολογίσουμε την τιμή `makeList 1 3`, θα ξεκινήσει ένας ατέρμονος υπολογισμός:

```
  makeList 1 3
= 1 : makeList (1+3) 3
= 1 : ((1+3) : (makeList ((1+3)+3) 3))
```

```
= 1 : (4 : (makeList (4+3) 3))
= 1 : (4 : ((4+3) : (makeList ((4+3)+3) 3)))
= 1 : (4 : (7 : (makeList (7+3) 3)))
...
```

∎

Αν ζητήσουμε από τον διερμηνέα της Haskell να αποτιμήσει μία παράσταση που επιστρέφει μία άπειρη λίστα, όπως η `inf` ή η `makeList 1 3`, τότε θα αρχίσει μία ατέρμονη εκτύπωση στη οθόνη. Λόγω της οκνηρής αποτίμησης δεν απαιτείται να ολοκληρωθεί ο υπολογισμός της λίστας πριν να αρχίσει η εκτύπωση και έτσι η Haskell μπορεί και εκτυπώνει ένα οσοδήποτε μεγάλο πλήθος τιμών από την αρχή της λίστας.

Λόγω της οκνηρής αποτίμησης, οι άπειρες λίστες μπορούν να χρησιμοποιηθούν σε πεπερασμένους υπολογισμούς. Η Haskell δεν επιχειρεί να σχηματίσει ολόκληρη τη άπειρη λίστα (κάτι που είναι αδύνατο), αλλά κατασκευάζει μόνο το τμήμα της που είναι χρήσιμο στον υπολογισμό. Με άλλα λόγια μία άπειρη λίστα συμπεριφέρεται ως ένα ρεύμα που παρέχει ένα απεριόριστο πλήθος στοιχείων. Ο υπολογισμός χρησιμοποιεί το πλήθος στοιχείων που απαιτείται.

Απο τις συναρτήσεις για λίστες που έχουμε δεί εως τώρα:

- οι `sumIntList`, `maxIntList`, `insSortList`, `mergeSortList`, καθώς και οι προκαθορισμένες συναρτήσεις `length`, `reverse`, `last` δεν λειτουργουν με άπειρες λίστες.

- η `memberInt`, επιστρέφει αποτέλεσμα μόνο αν η απάντηση είναι `True`, αλλιώς πέφτει σε ατέρμονο υπολογισμό. Η `isSorted` επιστρέφει αποτέλεσμα μόνο αν η απάντηση είναι `False`.

- οι `deleteInt`, `insertInt`, `mergeInt`, `splitInt` και η προκαθοριεμένη συνάρτηση `tail`, με είσοδο άπειρες λίστες επιστρέφουν αποτέλεσμα που είναι επίσης άπειρη λίστα.

- οι `concInt`, και ο ισοδύναμος τελεστής `++`, με είσοδο μία ή δύο άπειρες λίστες επιστρέφουν αποτέλεσμα που είναι επίσης άπειρη λίστα. Ωστόσο αν η πρώτη λίστα είναι άπειρη, τότε η συνένωση δεν μπορεί να οριστεί, συνεπώς το αποτέλεσμα που επιστρέφεται σε αυτή την περίπτωση (το οποίο είναι ίσο με την πρώτη λίστα) δεν μπορεί να θεωρηθεί σωστό.

- η προκαθοριεμένη συνάρτηση `init` με είσοδο μία άπειρη λίστα επιστρέφει την ίδια τη λίστα. Το αποτέλεσμα αυτό επίσης δεν μπορεί να θεωρηθεί σωστό, αφού ο ορισμός του `init` απαιτεί διαγραφή του τελευταίου στοιχείου από τη λίστα, το οποίο όμως δεν ορίζεται για άπειρη λίστα.

- η `makePairsInt` με είσοδο δύο άπειρες λίστες επιστρέφει επίσης άπειρη λίστα. Αν όμως μία από τις δύο λίστες είναι πεπερασμένη, τότε επιστρέφει πεπερασμένο αποτέλεσμα.

- η `elemIntList`, καθώς και οι προκαθοριεμένες συναρτήσεις `head` και `null`, με είσοδο άπειρη λίστα επιστρέφουν πεπερασμένο αποτέλεσμα.

Παράδειγμα 45: Η αποτίμηση της παράστασης `elemIntList 3 (makeList 1 3)` απαιτεί πεπερασμένο πλήθος βημάτων, παρότι το δεύτερο όρισμα της `elemIntList` είναι μία άπειρη λίστα.

```
  elemIntList 3 (makeList 1 3)
     #    3 <<< 1
      <=> "no"0
     #    makeList 1 3 <<< (h:t)
      <=> 1 : (makeList (1+3) 3) <<< (h:t)
      <=> "yes"
= elemIntList (3-1) (makeList (1+3) 3)
     #    (3-1) <<< 1
      <=> 2 <<< 1
      <=> "no"
     #    makeList (1+3) 3 <<< (h:t)
      <=> (1+3) : (makeList ((1+3)+3) 3) <<< (h:t)
      <=> "yes"
= elemIntList (2-1) (makeList ((1+3)+3) 3)
     #    (2-1) <<< 1
      <=> 1 <<< 1
      <=> "yes"
     #    makeList ((1+3)+3) 3 <<< (h:t)
      <=> ((1+3)+3) : (makeList (((1+3)+3)+3) 3) <<< (h:t)
      <=> "yes"
= (1+3)+3
= 4+3
= 7
```

∎

Παράδειγμα 46: Η συνάρτηση `search` αναζητεί έναν ακέραιο `n` σε μία λίστα ταξινομημένη σε αύξουσα τάξη.

Αν η κεφαλή της λιστας είναι μεγαλύτερη από `n` τότε επιστρέφει `False`: όλα τα στοιχεία της λίστας είναι μεγαλύτερα του `n` αφού η λίστα είναι ταξινομημένη. Αν η κεφαλή της λιστας είναι `n` τότε επιστρέφει `True`. Τέλος, αν η κεφαλή της λιστας είναι μικρότερη από `n`, τότε επιστρέφει το αποτέλεσμα της αναζήτησης του `n` στην ουρά της λιστας, το οποίο υπολογίζεται αναδρομικά.

```
searchInt :: Int -> [Int] -> Bool
searchInt n (h:t)
      | n < h = False
      | n == h = True
      | otherwise = searchInt n t
searchInt n [] = False
```

Η `searchInt` με είσοδο μία λίστα ταξινομημένη σε αύξουσα τάξη, η οποία περιέχει άπειρο πλήθος διαφορετικών ακεραίων, επιστρέφει πάντοτε αποτέλεσμα. Για παράδειγμα η αποτίμηση της παράστασης `searchInt 13 (makeList 2 7)` γίνεται με τον παρακάτω τρόπο:

```
  searchInt 13 (makeList 2 7)
     #    makeList 2 7 <<< (h:t)
      <=> 2 : (makeList (2+7) 7) <<< (h:t)
      <=> "yes"
     ?  13 < 2
      = False
     ?  13 == 2
      = False
     ?  otherwise
      = True
= searchInt 13 (makeList (2+7) 7)
     #    makeList (2+7) 7 <<< (h:t)
      <=> (2+7) : (makeList ((2+7)+7) 7) <<< (h:t)
      <=> "yes"
     ?  13 < (2+7)
      = 13 < 9
      = False
     ?  13 == 9
      = False
     ?  otherwise
      = True
= searchInt 13 (makeList (9+7) 7)
     #    makeList (9+7) 7 <<< (h:t)
      <=> (9+7) : (makeList ((9+7)+7) 7) <<< (h:t)
      <=> "yes"
     ?  13 < (9+7)
      = 13 < 16
      = True
= False
```

∎

Παράδειγμα 47: Ενα ακόμη παράδειγμα άπειρης λίστας, είναι η λίστα που περιέχει όλους τους πρώτους αριθμούς και η οποία μπορεί να οριστεί με την βοήθεια του κόσκινου του Ερατοσθένη. Περιγράφουμε πρώτα πως λειτουργεί το κόσκινο του ερατοσθένη:

- Ξεκινάμε με μία λίστα που περιέχει όλους τους φυσικούς αριθμούς που είναι μεγαλύτεροι ή ίσοι του 2: $[2, 3, 4, 5, \ldots]$.
- Το 2 είναι πρώτος αριθμός. Όσοι αριθμοί δεξιά του 2 στη λίστα διαιρούνται με το 2 δεν είναι πρώτοι αριθμοί και τους διαγράφουμε από τη λίστα. Η νέα λίστα που προκύπτει είναι: $[2, 3, 5, 7, 9, 11, \ldots]$.

- Ο αριθμός που ακολουθεί το 2 στη νέα λίστα είναι το 3 που είναι πρώτος αριθμός. Όσοι αριθμοί δεξιά του 3 στη λίστα διαιρούνται με το 3 δεν είναι πρώτοι αριθμοί και τους διαγράφουμε από τη λίστα. Η νέα λίστα που προκύπτει είναι: $[2, 3, 5, 7, 11, 13, 17, 19, 23, 25 \ldots]$.

- Συνεχίζουμε αυτή τη διαδικασία, επιλέγοντας κάθε φορά ως πρώτο αριθμό τον αμέσως επόμενο αριθμό στη λίστα από αυτόν που εξετάσαμε στο προηγούμενο βήμα, και διαγράφοντας από τη λίστα του αριθμούς που βρίσκονται δεξιά του και διαιρούνται με αυτόν.

- Αν ένα αριθμός είναι πρώτος δεν διαιρείται με κανέναν προηγούμενο και άρα θα παραμείνει στη λίστα και θα αναγνωριστεί ως πρώτος αριμθός.

- Αν ένας αριθμός δεν είναι πρώτος, τότε διαιρείται με κάποιον πρώτο που είναι μικρότερός του και άρα θα διαγραφεί από τη λίστα.

- Μετά από άπειρα βήματα η λίστα θα περιέχει όλους τους πρώτους αριθμούς.

Η παραπάνω κατασκευή της λίστας των πρώτων αριθμών είναι ορθή από μαθηματική άποψη. Αν ωστόσο επιχειρήσουμε να περιγράψουμε την παραπάνω διαδικασία σε μία γλώσσα χωρίς οκνηρή αποτίμηση, τότε ο υπολογισμός θα χρειαστεί άπειρο χρόνο για να διαγράψει τα πολλαπλάσια του 2 από τη λίστα, με αποτέλεσμα να μην μπορέσει ποτέ να διαπιστώσει ότι το 3 είναι επίσης πρώτος αριθμός.

Στη Haskell αντίθετα μπορούμε να κωδικοποιήσουμε το κόσκινο του Ερατοσθένη, έτσι ώστε να ορίσουμε την άπειρη λίστα που περιέχει όλους τους πρώτους αριθμούς. Λόγω της οκνηρής αποτίμηση η διαγραφή των πολλαπλασίων του 2 (και στη συνέχεια του 3, του 5 κλπ) από τη λίστα θα καθυστερεί, με αποτέλεσμα τον σχηματισμό στοιχείων στην αρχή της λίστας.

```
primes :: [Int]
primes = sieve [2..]
sieve :: [Int] -> [Int]
sieve (h:t) = h : sieve (elim h t)
      where elim :: Int -> [Int] -> [Int]
            elim a (b:r)
                    | b `mod` a == 0
                         = elim a r
                    | otherwise
                         = b : elim a r
            elim a [] = []
sieve [] = []
```

Η αποτίμηση της παράστασης `primes` θα απαιτούσε άπειρο χρόνο, ωστόσο ένα οσοδήποτε μεγάλο αρχικό της κομμάτι μπορεί να σχηματιστεί σε πεπερασμένο χρόνο:

```
  primes
= sieve [2..]
     #    [2..] <<< (h:t)
      <=> 2 : [3..] <<< (h:t)
      <=> "yes"
= 2 : (sieve (elim 2 [3..]))
     #    elim 2 [3..] <<< (h:t)
           #    [3..] <<< (b:r)
            <=> 3 : [4..] <<< (b:r)
            <=> "yes"
            ?  (3 `mod` 2) == 0
             = 1 == 0
             = False
            ?  otherwise
             = True
      <=> 3 : (elim 2 [4..]) <<< (h:t)
      <=> "yes"
= 2 : (3 : (sieve (elim 3 (elim 2 [4..]))))
     #    elim 3 (elim 2 [4,5]) <<< (h:t)
           #    elim 2 [4..] <<< (b:r)
                 #    [4..] <<< (b:r)
                  <=> 4 : [5..] <<< (b:r)
                  <=> "yes"
                 ?  (4 `mod` 2) == 0
                  = 0 == 0
                  = True
            <=> elim 2 [5..] <<< (b:r)
                 #    [5..] <<< (b:r)
                  <=> 5 : [6..] <<< (b:r)
                  <=> "yes"
                 ?  (5 `mod` 2) == 0
                  = 1 == 0
                  = False
                 ?  otherwise
                  = True
            <=> 5 : (elim 2 [6..]) <<< (b:r)
            <=> "yes"
           ?  (5 `mod` 3) == 0
            = 2 == 0
            = False
           ?  otherwise
            = True
      <=> 5 : (elim 3 (elim 2 [6..])) <<< (h:t)
      <=> yes
= 2:(3:(5:(sieve (elim 5 (elim 3 (elim 2 [6..]))))))
...
```

Μπορούμε να χρησιμοποιήσουμε την λίστα `primes` για να βρούμε τον n-οστό πρώτο:

```
> elemIntList 7 primes
17
```

ή να ελέγξουμε άν ένας αριθμός είναι πρώτος

```
> searchInt 15 primes
False
> searchInt 23 primes
True
```