Συναρτησιακός Προγραμματισμός
Η γλώσσα Haskell
(Σημειώσεις)

Χρήστος Νομικός

Ιωάννινα 2015

Περιεχόμενα

Κεφάλαιο 1

1.1 Συναρτησιακός Προγραμματισμός

Ο συναρτησιακός προγραμματισμός έχει τις ρίζες του στον λάμβδα λογισμό (lambda calculus), ο οποίος αναπτύχθηκε στη δεκαετία του 1930 από τον Alonzo Church, ως μία προσπάθεια για να δοθεί μαθηματική υπόσταση στην έννοια του υπολογισμού.

Στο συναρτησιακό προγραμματισμό, το πρόγραμμα συνίσταται από ένα πλήθος συναρτήσεων, οι οποίες ορίζονται χρησιμοποιώντας σύνθεση και αναδρομή, ξεκινώντας από ένα σύνολο πρωταρχικών συναρτήσεων (π.χ. αριθμητικών πράξεων). Ο υπολογισμός του επιθυμητού αποτελέσματος επιτυγχάνεται μέσω της αποτίμησης μίας παράστασης, η οποία μπορεί να περιέχει προκαθορισμένους τελεστές καθώς και συναρτήσεις που έχουν οριστεί στο πρόγραμμα.

Αντίθετα με τις προστακτικές γλώσσες (C, Pascal, κλπ) οι συναρτήσεις στις καθαρά συναρτησιακές γλώσσες δεν προκαλούν παρενέργειες (αλλαγές σε τιμές μεταβλητών) και η τιμή που επιστρέφουν εξαρτάται αποκλειστικά από τις τιμές των ορισμάτων. Οι καθαρές συναρτησιακές γλώσσες δεν έχουν εντολή ανάθεσης, ούτε και εντολές επανάληψης. Η επανάληψη επιτυγχάνεται με χρήση αναδρομής.

Ορισμένα χαρακτηριστικά, η ύπαρξη των οποίων διευκολύνεται από τη φύση των συναρτησιακών γλωσσών είναι :

- αναδρομή
- πολυμορφισμός
- οκνηρή αποτίμηση
- συναρτήσεις υψηλής τάξης και συναρτήσεις ως τιμές
- λίστες ως προκαθορισμένοι τύποι
- σύνθετοι τύποι ως αποτελέσματα συναρτήσεων
- σταθερές σύνθετου τύπου

1.2 Η γλώσσα Haskell

Η Γλώσσα Haskell είναι μία καθαρή συναρτησιακή γλώσσα, η οποία εφαρμόζει οκνηρή αποτίμηση. Αναπτύχθηκε τη δεκαετία του 1990, με σκοπό να αποτελέσει μια βάση για την έρευνα και την εξέλιξη των γλωσσών αυτής της κατηγορίας. Ωφείλει το όνομά της στον αμερικάνο μαθηματικό και λογικό Haskell Curry.

Στη συνέχεια θα χρησιμοποιήσουμε τη γλώσσα Haskell για να κάνουμε μία εισαγωγή στις έννοιες του συναρτησιακού προγραμματισμού. Ο σκοπός αυτών των σημειώσεων δεν είναι σε καμμία περίπτωση το να περιγραφεί πλήρως η γλώσσα Haskell. Θα περιοριστούμε κυριώς σε παραδείγματα πάνω σε αριθμούς και λίστες, ώστε να γίνει κατανοητό το πώς η αναδρομή αρκεί για την υλοποίηση επανάληψης. Επίσης θα δούμε πως ορίζονται πολυμορφικές συναρτήσεις, συναρτήσεις που παίρνουν ως ορίσματα και επιστρέφουν ως αποτελέσμα άλλες συναρτήσεις και το πώς μπορούμε να σχηματίσουμε άπειρες λίστες οι οποίες μπορουν να συμμετέχουν σε πεπερασμένους υπολογισμούς. Αντίθετα δεν θα εξετάσουμε μεταξύ άλλων το πλήρες σύστημα τύπων τις Haskell και θέματα που αφορούν είσοδο και έξοδο για δημιουργία διαδραστικών εφαρμογών.

Στη συνέχεια θα κάνουμε την υπόθεση ότι όλες οι συναρτήσεις που θα ορίσουμε περιέχονται σε ένα μοναδικό αρχείο. Η Haskell δίνει τη δυνατότητα να υποδιαιρεθεί το πρόγραμμα σε ενότητες που περιέχονται σε ξεχωριστά αρχεία, ωστόσο αυτό το χαρακτηριστικό δεν θα βοηθούσε στην καλύτερη κατανόηση των χαρακτηριστικών του συναρτησιακού προγραμματισμού.

1.3 Ο διερμηνέας hugs της Haskell

Για να εκτελεστεί ο διερμηνέας hugs της Haskell σε σύστημα Unix γράφουμε στο τερματικό `hugs` και πατάμε `<ENTER>`. Στα windows μπορούμε να ανοίξουμε τον hugs από τη λίστα προγραμμάτων ή κάνοντας διπλό κλικ στο αντίστοιχο εικονίδιο, εφόσον υπάρχει.

Ο hugs εμφανίζει την prompt

```
Prelude> _
```

Το Prelude είναι το όνομα της βασικής βιβλιοθήκης της Haskell που περιέχει ένα πλήθος από προκαθοριμένες συναρτήσεις και τελεστές μεταξύ των οποίων οι βασικοί αριθμητικοί τελεστές. Σε αυτή τη φάση μπορούμε να γράψουμε μία αριθμητική παράσταση (ή όποια άλλη παράσταση εμπλέκει μόνο προκαθορισμένες συναρτήσεις και τελεστές) και ο hugs θα την αποτιμήσει επιστρέφοντας μας το αποτέλεσμα (θα πρέπει μετά την ολοκλήρωση της παράστασης να πατήσουμε `<ENTER>`):

```
Prelude> 3+4
7
Prelude>
```

Μπορούμε να φορτώσουμε ένα αρχείο προγράμματος π.χ. το `example.hs`, το οποίο βρίσκεται στον κατάλογο από τον οποίο εκκινήσαμε τον hugs γράφοντας:

```
Prelude> :load example.hs
Main>
```

Η εντολή `:load` μπορεί πιο απλά να γραφτεί `:l`. Αν το αρχείο βρίσκεται σε διαφορετικό κατάλογο από αυτόν από τον οποίο εκτελέσαμε τον hugs, τότε θα πρέπει να γράψουμε το πλήρες μονοπάτι.

Παρατηρούμε ότι η prompt έχει αλλάξει σε `Main>`. Αυτό δηλώνει ότι το πρόγραμμα `example.hs` δεν είχε συντακτικά λάθη, και είναι πλέον φορτωμένο στη μνήμη. Έχοντας φορτώσει το πρόγραμμα μπορούμε να αποτιμήσουμε παραστάσεις που περιέχουν συναρτήσεις που ορίζονται σε αυτό, αλλά και συναρτήσεις από το Prelude, το οποίο παραμένει φορτωμένο στη μνήμη. Αν το πρόγραμμα δεν είναι συντακτικά ορθό (ή αν το αρχείο δεν υπάρχει), εμφανίζεται ένα μηνυμα λάθους και η prompt παραμένει `Prelude>`. Το `example.hs` παραμένει φορτωμένο στη μνήμη μέχρι να φορτώσουμε κάποιο άλλο αρχείο ή να γράψουμε την εντολή `:load` χωρίς να ακολουθείται από όνομα αρχείου.

Ο hugs είναι συνδεδεμένος με κάποιον editor τον οποίο μπορούμε να ανοίξουμε μέσα απο το περιβάλλον του hugs με την εντολή `:edit` ή πιο απλά `:e`. Αν η εντολή `:edit`. δεν ακολουθείται απο όνομα αρχείου τότε ανοίγει το τελευταίο αρχείο που προσπαθήσαμε να φορτώσουμε (επιτυχώς ή ανεπιτυχώς) με την εντολή `:load`.

Η εκτέλεση του `hugs` τερματίζεται με την εντολή `:quit` ή πιο απλά `:q`.

1.4 Βασικοί τύποι

Η Haskell διαθέτει ένα πλούσιο σύστημα τύπων. Στη συνέχεια θα περιγράψουμε μόνο ορισμένους βασικούς τύπους που θα χρησιμοποιήσουμε για να γίνουν κατανοητά τα κύρια χαρακτηριστικά του συναρτησιακού προγραμματισμού.

Ακέραιοι αριθμοί: η Haskell διαθέτει τον τύπο `Int` (ακέραιος με καθορισμένο μήκος) και τον τύπο `Integer` (ακέραιος με αυθαίρετα μεγάλο μήκος).

Για τους παραπάνω τύπους είναι προκαθορισμένοι μεταξύ άλλων οι τελεστές `+`, `-`, `*`, `^` (υψωση σε δύναμη) και οι συναρτήσεις `div` (ακέραια διαίρεση), `mod` (υπόλοιπο διαίρεσης), `max` (μέγιστο δύο στοιχείων), `min` (ελάχιστο δύο στοιχείων) `abs` (απόλυτη τιμή) και `negate` (αντίθετος).

```
> div 9 4
2
> mod 15 4
3
> abs (-4)
4
```

Παρατηρήσεις:

- Τα ορίσματα μίας συνάρτησης δίνονται μετά το όνομα της συνάρτησης, χωρισμένα με κενά. Ένας δυαδικός τελεστής γράφεται ανάμεσα στα ορίσματά του.

- Αν κάποιο όρισμα είναι αρνητικός αριθμος τότε πρέπει να γραφτεί μέσα σε παρένθεση.

- Μπορούμε να μετατρέψουμε μία συνάρτηση δύο μεταβλητών σε τελεστή γράφοντάς το ονομά της ανάμεσα σε δύο σύμβολα ‘ (τόνους) π.χ.

```
> 15 ‘mod‘ 4
3
```

- Αντίστροφα, μπορούμε να μετατρέψουμε έναν τελεστή σε συνάρτηση δύο μεταβλητών γράφοντάς τον μέσα σε παρενθέσεις, π.χ.

```
> (+) 3 2
5
```

Πραγματικοί αριθμοί: η Haskell διαθέτει τον τύπο `Float` (πραγματικός απλής ακρίβειας) και τον τύπο `Double` (πραγματικός διπλής ακρίβειας).

Για τους παραπάνω τύπους είναι προκαθορισμένοι μεταξύ άλλων οι τελεστές `+`, `-`, `*`, `/`, `^` (υψωση σε ακέραια δύναμη), `**` (υψωση σε πραγματική δύναμη) και οι συναρτήσεις `sqrt` (τετραγωνική ρίζα), `abs`, `negate`, `ceiling`, `floor`, `round`, `exp`, `log`, `sin`, `cos`, `tan` ...

Χαρακτήρες: Οι χαρακτήρες στη Haskell ανήκουν στον τύπο `Char`.

Οι συναρτήσεις `succ` και `pred` επιστρέφουν αντίστοιχα τον επόμενο και τον προηγούμενο χαρακτήρα με βάση την κωδικοποίηση χαρακτήρων ASCII. Η συνάρτηση `fromEnum` μετατρέπει έναν χαρακτήρα στον αντίστοιχο κωδικό ASCII, ενώ η `toEnum` κάνει την αντίστροφη μετατροπή (το `::Char` μετατρέπει το αποτέλεσμα της `toEnum`, η οποία μπορεί να επιστρέψει αποτέλεσμα διαφόρων τύπων, σε `Char`).

```
> succ 'a'
'b'
> pred 'a'
'`'
> fromEnum 'a'
97
> toEnum 100::Char
'd'
```

Αλφαριθμητικά: Τα αλφαριθμητικά στη Haskell ανήκουν στον τύπο `String` ο οποίος είναι ισοδύναμος με λίστα χαρακτήρων.

Συνεπώς όλες οι λειτουργίες που θα περιγράψουμε αργότερα για λίστες, λειτουργούν και για τον τύπο `String`.

Τιμές αλήθειας της λογικής: Ο τύπος `Bool` της Haskell έχει πεδίο τιμών {`True`, `False`}.

Οι τελεστές `&&`, `||`, `not`, υλοποιούν αντίστοιχα την σύζευξη (και), τη διάζευξη (ή) και την άρνηση (όχι).

Οι τελεστές σύγκρισης στη Haskell είναι οι `==`, `<`, `>`, `<=`, `>=` και `/=` (διάφορο), οι οποίοι δέχονται δυο ορίσματα του ίδιου διατεταγένου τύπου (π.χ. `Int`, `Integer`, `Float`, `Double`, `Char`, `String`, `Bool`) και επιστρέφουν αποτέλεσμα τύπου `Bool`:

```
> 2 /= 3
True
> (3+5) == (4+4)
True
> 'a' < 'A'
False
> "alice" <= "bob"
True
> True >= False
True
```

Σύνθετοι τύποι: Αν t_1 και t_2 είναι δύο τύποι της Haskell τότε ο (t_1, t_2) είναι επίσης τύπος με πεδίο τιμών όλα τα ζεύγη με πρώτη συνιστώσα τύπου t_1 και δεύτερη συνιστώσα τύπου t_2.

Για παράδειγμα, τα ζεύγη ακεραίων (π.χ το `(0,1)`) έχουν τύπο `(Int,Int)`, ενώ τα ζεύγη που αποτελούνται απο ένα αλφαρηθμητικό και μία λογική τιμή (π.χ το `("Chris",True)`) έχουν τύπο `(String,Bool)`.

Μπορούμε με ανάλογο τρόπο να έχουμε διάφορους τύπους τριάδων, τετράδων κλπ, σε καθέναν από τους οποίους κάθε συνιστώσα να ανήκει σε έναν συγκεκριμένο τύπο, π.χ. `(Int,Int,String,(Float,Float))`. Κάθε τύπος που ορίζεται με αυτόν τον τρόπο περιλαμβάνει πλειάδες με συγκεκριμένο μήκος.

Αν t είναι τύπος της Haskell τότε ο `[`t`]` είναι επίσης τύπος με πεδίο τιμών όλες τις λίστες με στοιχεία τύπου t.

Για παράδειγμα, οι λίστες ακεραίων έχουν τύπο `[Int]`, ενώ ο τύπος `String` είναι ισοδύναμος με `[Char]`.

Τα στοιχεία μίας λίστας δίνονται μέσα σε `[` και `]` χωρισμένα με κόμμα π.χ `[1,2,3,4]`,

`['a','b','c']`. Η κενή λίστα (οποιουδήποτε τύπου) συμβολίζεται με `[]`.

Οι πλειάδες και οι λίστες διαφέρουν σε δύο βασικά σημεία:

- Οι πλειάδες που ανήκουν στον ίδιο τύπο έχουν ίδιο πλήθος συνιστωσών, ενώ ένας τύπος λίστας περιέχει λίστες με πλήθος στοιχείων που μπορεί να είναι οποιοσδήποτε φυσικός αριθμός.
- Τα στοιχεία μίας λίστας είναι όλα του ίδιου τύπου, ενώ οι συνιστώσες μιας πλειάδας μπορεί να ανήκουν σε διαφορετικούς τύπους.

1.5 Σταθερές

Στη Haskell μπορούμε να αναθέσουμε τιμές σε συμβολικά ονόματα. Η τιμή που ανατίθεται σε ένα όνομα δεν μπορεί αλλάξει με νέα ανάθεση. Συνεπώς τα ονόματα αυτά αντιστοιχούν σε σταθερές.

```
zero :: Int
zero = 0

pi :: Float
pi = 3.14159

space :: Char
space = ' '
```

1.6 Συναρτήσεις

Ο ορισμός μίας συνάρτησης αποτελείται από μία δήλωση που καθορίζει τον τύπο της συνάρτησης (τύποι ορισμάτων - αποτελέσματος) και μία σειρά από ισότητες που καθορίζουν την τιμή της.

Παράδειγμα 1: Η συνάρτηση `sqrInt` υπολογίζει το τετράγωνο ενός ακεραίου:

```
sqrInt :: Int -> Int
sqrInt n = n * n
```

Η πρώτη γραμμή δηλώνει ότι η `sqrInt` είναι μία συνάρτηση από ακέραιους σε ακέραιους, δηλαδή παίρνει ως είσοδο έναν ακέραιο και επιστρέφει μία ακέραια τιμή. Η δεύτερη γραμμή καθορίζει ότι η τιμή της συνάρτησης για είσοδο `n` είναι το `n*n`.

Για να αποτιμήσουμε την τιμή μίας συνάρτησης, γράφουμε το όνομα της ακολουθούμενο από τις πραγματικές παραμέτρους:

```
> sqrInt 5
25
```

∎

Παράδειγμα 2: η συνάρτηση `avgInt` επιστρέφει το 'μέσο όρο' δύο ακεραίων αριθμών:

```
avgInt :: Int -> Int -> Int
avgInt a b = (a + b) `div` 2
```

∎

Οι συναρτήσεις που έχουν οριστεί μπορούν να χρησιμοποιηθούν για να ορίσουμε νέες συναρτήσεις:

Παράδειγμα 3: η συνάρτηση `avgSqrInt` επιστρέφει το 'μέσο όρο' τών τετραγώνων δύο ακεραίων αριθμών:

```
avgSqrInt :: Int -> Int -> Int
avgSqrInt a b = avgInt (sqrInt a) (sqrInt b)
```

∎

1.7 Συντακτικοί κανόνες

Τα ονόματα στη Haskell σχηματίζονται χρησιμοποιώντας χαρακτήρες Unicode που αντιστοιχούν σε κεφαλαία ή μικρά γράμματα, ψηφία και τους χαρακτήρες _ (χαρακτήρας υπογράμμισης) και ' (απλό εισαγωγικό).

Τα ονόματα των σταθερών, των συναρτήσεων και των τυπικών παραμέτρων τους, όπως επίσης των μεταβλητών τύπων που θα δούμε αργότερα πρέπει να αρχίζουν με μικρό γράμμα ή το χαρακτήρα υπογράμισης _.

Τα ονόματα των τύπων, και κατασκευαστών τύπων (όπως είναι τα `True` και `False`) πρέπει να αρχίζουν με κεφαλαίο γράμμα.

Ενα τμήμα ορισμού (δήλωση ή ισότητα) μπορεί να εκτείνεται σε περισσότερες από μία γραμμές. Οι πρόσθετες γραμμές (αν υπάρχουν) πρέπει να ξεκινούν δεξιότερα από την πρώτη γραμμή του τμήματος ορισμού.

Αν συναντηθεί γραμμή που ξεκινάει στην ίδια στήλη ή αριστερότερα από την πρώτη γραμμή ενός τμήματος ορισμού, η Haskell θεωρεί ότι η γραμμή αυτή ανήκει σε νέο τμήμα ορισμού.

Η Haskell κατά τη συντακτική ανάλυση εισάγει αυτόματα το χαρακτήρα ; ανάμεσα σε διαδοχικά τμήματα ορισμών. Επίσης οριοθετεί με τους χαρακτήρες { και } ορισμένα τμήματα του προγράμματος (όπως για παράδειγμα τους τοπικούς ορισμούς που ακολουθούν τη λέξη `where`, τους οποίους θα περιγράψουμε παρακάτω).

Για αυτό το λόγο, αν δεν τηρούμε τους κανόνες στοιχισης η Haskell ενδέχεται να μας επιστρέψει μηνύματα λάθους της μορφής `unexpected ;` ή `unexpected }`, παρότι οι συγκεκριμένοι χαρακτήρες δεν εμφανίζονται στο πρόγραμμα.

Στη Haskell μπορούμε να γράψουμε σχόλια περικλείοντάς τα στα σύμβολα {- και -}. Εναλλακτικά τα σχόλια μπορούν να ξεκινήσουν με τα σύμβολα --. Σε αυτή την περίπτωση εκτείνονται μέχρι το τέλος της γραμμής.

1.8 Παραστάσεις υπό συνθήκη

Στη Haskell υπάρχουν διάφοροι τρόποι ώστε να ορίσουμε συναρτήσεις που οι τιμές τους εξαρτώνται από κάποια συνθήκη:

- `if - then - else`: αν αληθεύει συνθήκη που ακολουθεί το `if` λαμβάνεται η τιμή που ακολουθεί το `then`, αλλιώς λαμβάνεται η τιμή που ακολουθεί το `else`.

  Παράδειγμα 4: Η παρακάτω συνάρτηση υπολογίζει το ελάχιστο δύο ακεραίων:

  ```
  minInt :: Int -> Int -> Int
  minInt m n = if m<n then m else n
  ```

  ∎

- συνθήκες φρουροί: η τιμή της συνάρτησης καθορίζεται από μία ακολουθία περιπτώσεων της μορφής: | <συνθήκη> = <τιμή>

  Οι συνθήκες εξετάζονται μία προς μία μέχρι να βρεθεί κάποια που αληθεύει, οπότε και λαμβάνεται η αντίστοιχη τιμή.

  Παράδειγμα 5: Η παρακάτω συνάρτηση υπολογίζει την απόλυτη τιμή ενός ακέραιου:

  ```
  absInt :: Int -> Int
  absInt n
    | n > 0 = n
    | otherwise = negate n
  ```

  Το `otherwise` αποτιμάται πάντα σε `True`.

  ∎

- πρότυπα: δίνεται μια σειρά ισοτήτων, που ορίζουν την τιμή της συνάρτησης, για διάφορα πρότυπα των τιμών των παραμέτρων. Τα πρότυπα εξετάζονται με τη σειρά μέχρι κάποιο να ταιριάζει με τις τιμές των πραγματικών παραμέτρων:

  Παράδειγμα 6: Η παρακάτω συνάρτηση επιστρέφει `True` αν και μόνο αν το όρισμά της είναι μηδέν:

```
isZero :: Int -> Bool
isZero 0 = True
isZero n = False
```

∎

- συνδυασμός των παραπάνω:

  Παράδειγμα 7: Η παρακάτω συνάρτηση επιστρέφει 1,0 ή -1 αν το όρισμά της είναι αντίστοιχα θετικό, μηδέν ή αρνητικό:

```
sign :: Int -> Int
sign 0 = 0
sign n
    | n>0       = 1
    | otherwise = -1
```

∎

1.9 Τοπικοί ορισμοί

Η Haskell παρέχει τη δυνατότητα να ορίσουμε τοπικά, εσωτερικά σε μία συνάρτηση, συμβολικά ονόματα ή ακόμη και ολόκληρες συναρτήσεις. Αυτό βοηθάει στην αναγνωσιμότητα του προγράμματος και σε πολλές περιπτώσεις κάνει την εκτέλεση του πιο αποδοτική. Οι τοπικές δηλώσεις ξεκινούν με τη λέξη `where`.

Παράδειγμα 8: Έστω η συνάρτηση

$$r(x, y, z) = (x + y + z)^x + (x + y + z)^y + (x + y + z)^z.$$

Η παρακάτω υλοποίηση, υπολογίζει τό άθροισμα $x+y+z$ τρείς φορές, μία για κάθε εμφάνιση του στον τύπο υπολογισμού.

```
r' :: Int -> Int -> Int -> Int
r' x y z = (x+y+z)^x + (x+y+z)^y + (x+y+z)^z
```

Μία καλύτερη υλοποίηση προκύπτει αν ορίσουμε τοπικά το `w` να έχει την τιμή `x+y+z`. Με αυτόν τον τρόπο το άθροισμα υπολογίζεται μία μόνο φορά. Επίσης απολουστεύεται η παράσταση που δίνει το επιστρεφόμενο αποτέλεσμα.

```
r :: Int -> Int -> Int -> Int
r x y z = w^x + w^y + w^z
      where w = x + y + z
```

∎

1.10 Οκνηρή Αποτίμηση

Το επιθυμητό αποτέλεσμα στη Haskell (όπως και σε όλες τις καθαρά συναρτησιακές γλώσσες) επιτυγχάνεται μέσω της αποτίμησης μίας παράστασης. Στη συνέχεια θα δούμε πώς γίνεται η αποτίμηση μίας παράστασης στη Haskell, η οποία στηρίζεται στη στρατηγική της οκνηρής αποτίμησης (lazy evaluation).

Είναι χρήσιμο αρχικά να δούμε πώς μπορούμε να κατασκευάσουμε ένα δέντρο που να περιγράφει τη δομή μιας παράστασης.

Πρώτα θα πρέπει να αρθούν όλες οι αμφισημίες, ώστε να είναι μονοσήμαντα ορισμένη η δομή της παράστασης, δηλαδή να είναι ξεκάθαρο ποιες υποπαραστάσεις αποτελούν τα ορίσματα του κάθε τελεστή. Αυτό γίνεται με βάση του κανόνες προτεραιότητας και προσεταιρισμού, οι κυριότεροι από τους οποίους συνοψίζονται παρακάτω:

- Οι τελεστές με ένα όρισμα έχουν μεγαλύτερη προτεραιότητα από τους δυαδικούς τελεστές.
- Οι τελεστές `*`, `` `div` ``, `` `mod` `` και `/` έχουν μικρότερη προτεραιότητα από τους τελεστές `^` και `**` και μεγαλύτερη προτεραιότητα από τους τελεστές `+` και `-`.
- Οι τελεστές `+`, `-`, `*`, `` `div` ``, `` `mod` `` και `/` προσεταιρίζονται από αριστερά προς τα δεξιά.
- Οι τελεστές `^` και `**` προσεταιρίζονται από δεξιά προς τα αριστερά.
- Οι τελεστές σύγκρισης έχουν μικρότερη προτεραιότητα άπό τους δυαδικούς αριθμητικούς τελεστές και μεγαλύτεροι προτεραιότητα από τους δυαδικούς λογικούς τελεστές.
- Ο τελεστής `&&` έχει μεγαλύτερη προτεραιότητα από τον `||`
- Οι τελεστές `&&` και `||` προσεταιρίζονται από τα αριστερά προς τα δεξιά.
- Εφαρμογή συνάρτησης έχει μεγαλύτερη προτεραιότητα από όλους τους τελεστές και προσεταιρίζεται από αριστερά πρός τα δεξιά.

Με δεδομένη μία παράσταση της Haskell μπορούμε να προσθέσουμε κατάλληλα παρενθέσεις έτσι ώστε τα ορίσματα κάθε τελεστή η συνάρτησης να είναι είτε απλές παραστάσεις (σταθερές ή μεταβλητές) είτε σύνθετες παραστάσεις κλεισμένες σε παρενθέσεις.

Η παραπάνω μετατροπή μας βοηθάει στο να περιγράψουμε την παράσταση με ένα δέντρο:

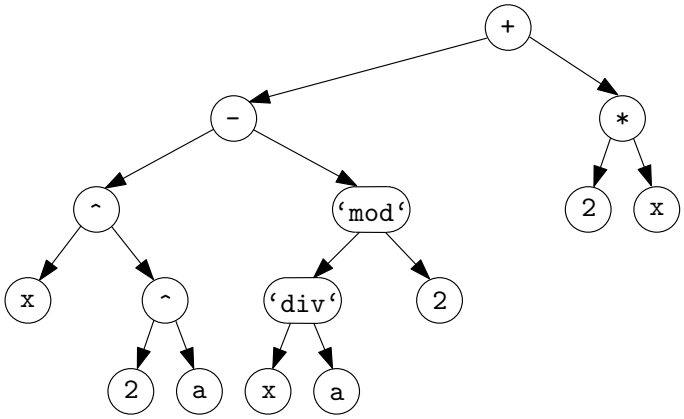
- Μία απλή παράσταση (συμβολική ή κυριολεκτική σταθερά ή μεταβλητή) παριστάνεται από ένα δέντρο αποτελούμενο από έναν κόμβο με ετικέτα τη σταθερά ή τη μεταβλητή.

- Μία σύνθετη παράσταση παριστάνεται από ένα δέντρο η ρίζα του οποίου έχει ετικέτα το όνομα της συνάρτησης ή του τελεστή που θα δώσει το τελικό αποτέλεσμα, ενώ τα παιδιά του από αριστερά προς τα δεξιά είναι οι ρίζες των δέντρων που αντιστοιχούν στις παραστάσεις που αποτελούν τα ορίσματα της συνάρτησης ή του τελεστή.

Παράδειγμα 9: Έστω η παράσταση `x^2^a-x‘div‘a‘mod‘2+2*x`

Με βάση τους κανόνες προτεραιότητας των τελεστών, η παραπάνω παράσταση γράφεται ισοδύναμα `((x^(2^a))-((x‘div‘a)‘mod‘2))+(2*x)`

Το δέντρο που αντιστοιχεί στην παραπάνω παράσταση είναι:

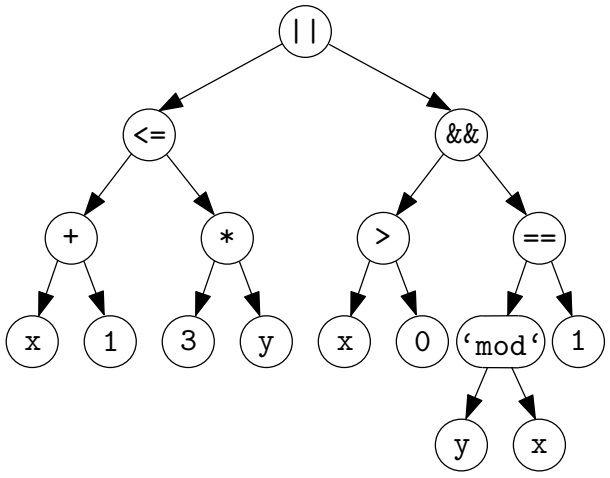


∎

Παράδειγμα 10: Έστω η παράσταση `x+1<=3*y||x>0&&y‘mod‘x==1`

Με βάση τους κανόνες προτεραιότητας των τελεστών, η παραπάνω παράσταση γράφεται ισοδύναμα `((x+1)<=(3*y))||((x>0)&&((y‘mod‘x)==1))`

Το δέντρο που αντιστοιχεί στην παραπάνω παράσταση είναι:



∎

Σύμφωνα με την τη στρατηγική της οκνηρής αποτίμησης τα ορίσματα μίας συνάρτησης (τα οποία ενδέχεται να είναι σύνθετες παραστάσεις) δεν αποτιμούνται παρά μόνο αν και όταν χρειαστούν για τον υπολογισμό του τελικού αποτελέσματος.

Αν η τιμή κάποιου ορίσματος δεν επηρεάζει το αποτέλεσμα της συνάρτησης τότε το όρισμα αυτό δεν αποτιμάται. Επίσης αν κάποιο όρισμα έχει σύνθετο τύπο και αρκεί ένα μόνο τμήμα του για να γίνει ο υπολογισμός, τότε υπολογίζεται μόνο αυτό το τμήμα.

Τα πλεονεκτήματα της οκνηρή αποτίμηση είναι αφ'ενός ότι αποφεύγονται άσκοποι υπολογισμοι και αφ'ετέρου ότι δίνεται η δυνατότητα να επεξεργαζόμαστε άπειρες δομές.

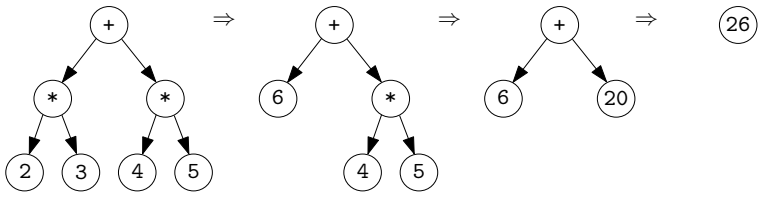
Αντίθετα με ότι συμβαίνει στις περισσότερες επιτακτικές γλώσσες, η αποτίμηση μίας παράστασης γίνεται από έξω προς τα μέσα: πρώτα εξετάζεται ο ορισμός της συνάρτησης (ή του τελεστή) που θα δώσει το τελικό αποτέλεσμα. Η συνάρτηση αυτή βρίσκεται στη ρίζα του δέντρου. Τα ορίσματα ενός τελεστή αποτιμούνται από αριστερά προς τα δεξιά.

Παράδειγμα 11: Έστω η παράσταση `2*3+4*5`. Με βάση τους κανόνες προτεραιότητας των τελεστών, η παραπάνω παράσταση γράφεται ισοδύναμα `(2*3)+(4*5)`. Η αποτίμηση της παράστασης δίνει διαδοχικά:

```
  (2*3)+(4*5)
= 6+(4*5)
= 6+20
= 26
```

Στο παράδειγμα αυτό η οκνηρή αποτίμηση δεν δημιουργεί καμία διαφορά, καθώς ο τελεστής + απαιτεί το υπολογισμό και των δύο ορισμάτων του.

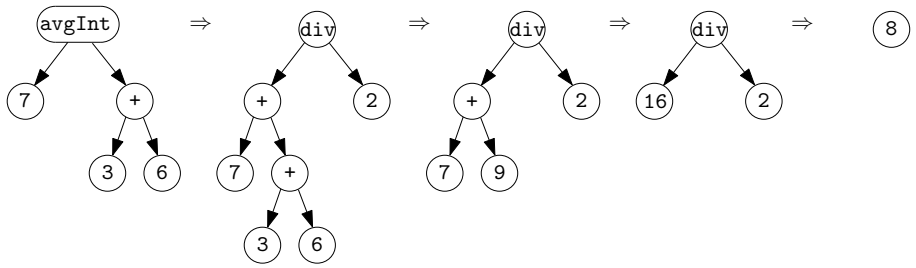
Σε μορφή δέντρων ο παραπάνω υπολογισμός παριστάνεται:



Παράδειγμα 12: Έστω η παράσταση `avgInt 7 (3+6)`. Η αποτίμηση της δίνει διαδοχικά:

```
  avgInt 7 (3+6)
= (7 + (3+6)) `div` 2
= (7+9) `div` 2
= 16 `div` 2
= 8
```

Σε μορφή δέντρων:



Παρατηρούμε ότι λόγω της οκνηρής αποτίμησης, η επεξεργασία της παράστασης ξεκίνησε από τη ρίζα του δέντρου και το όρισμα `3+5` αποτιμήθηκε αφού πρώτα έγινε η αντικατάσταση με βάση τον ορισμό της `avgInt`, από την οποία προέκυψε η αναγκαιότητα της αποτίμησή του ώστε να ολοκληρωθεί ο υπολογισμός. ∎

Αν μία τυπική παράμετρος εμφανίζεται σε περισσότερα από ένα σημεία της παράστασης που ορίζει την τιμή μιας συνάρτησης, τότε το αντίστοιχο όρισμα (πραγματική παράμετρος) αποτιμάται μία μόνο φορά.

Παράδειγμα 13: Έστω η παρακάτω συνάρτηση `f`

```
f :: Int -> Int
f n = 2^n + n
```

Η αποτίμηση της παράστασης `f (sqrInt 3)` δίνει διαδοχικά:

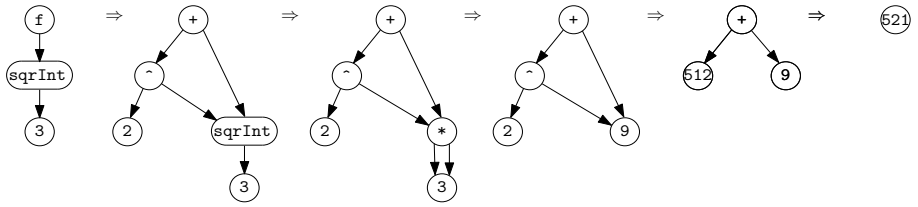
```
  f (sqrInt 3)
= (2^(sqrInt 3))+(sqrInt 3)
= (2^(3*3))+(3*3)
= (2^9)+9
= 512+9
= 521
```

Το `sqrInt 3` αποτιμάται μία μόνο φορά, παρότι εμφανίζεται σε δύο σημεία της ενδιάμεσης παράστασης.

Αυτό μπορεί να γίνει επειδή στην παραγματικότητα, οι ενδιάμεσες παραστάσεις δεν αντιστοιχούν σε δέντρα αλλά σε κατευθυνόμενα (πολυ)γραφήματα χωρίς κύκλους.

Αυτό πρακτικά σημαίνει ότι ένας κόμβος μπορεί να είναι ταυτόχρονα παιδί δύο άλλων κόμβων.

Σε μορφή γραφημάτων ο παραπάνω υπολογισμός γράφεται:



Παράδειγμα 14: Έστω η παρακάτω συνάρτηση g

```
g :: Int -> Int -> Bool
g m n = n > 0 && m `mod` n == n-1
```

Η αποτίμηση της παράστασης `g 5 0` δίνει διαδοχικά:

```
  g 5 0
= (0 > 0) && ((5 `mod` 0) == (0-1))
= False && ((5 `mod` 0) == (0-1))
= False
```

Το πρώτο όρισμα του τελεστή `&&` αποτιμάται σε `False`, συνεπώς η τιμή της παράστασης είναι σίγουρα `False` ανεξάρτητα από την τιμή του δεύτερου ορίσματος και η Haskell δεν αποτιμά καθόλου το όρισμα αυτό. Σημειώνεται ότι αν η Haskell αποτιμούσε το δεύτερο όρισμα, τότε θα επέστρεφε μήνυμα λάθους λόγω διαίρεσης με το 0. Η μερική αποτίμηση των λογικών τελεστών αποτελεί ειδική περίπτωση της οκνηρής αποτίμησης και χρησιμοποιείται ακόμη και από γλώσσες που δεν υποστηρίζουν γενικά οκνηρή αποτίμηση (όπως η C). ∎

Αν ο ορισμός μίας συνάρτησης περιέχει πρότυπα, τότε τα ορίσματα υπολογίζονται μερικώς, στο βαθμό που απαιτείται ώστε να διαπιστωθεί αν ταιριάζουν με το πρότυπο. Σε περίπτωση αποτυχίας εξετάζεται το επόμενο πρότυπο, το οποίο μπορεί να απαιτήσει αποτίμηση επιπλέον ορισμάτων.

Παράδειγμα 15: Έστω η παρακάτω συνάρτηση `h`

```
h :: Int -> Int -> Int
h m 0 = m
h 0 n = n
h m n = m*n
```

Έστω η παράσταση `h (3+4) (8 `div` 3)`

Η Haskell πρώτα θα εξετάσει αν μπορεί να εφαρμοστεί η πρώτη ισότητα. Για να το κάνει αυτό πρέπει να διαπιστώσει αν το `8 `div` 3` ταιριάζει με το πρότυπο `0`,

συνεπώς πρέπει να το αποτιμήσει. Η τιμή που προκύπτει είναι 2, οπότε η Haskell συνεχίζει εξετάζοντας αν η επόμενη ισότητα στον ορισμό της h μπορεί να εφαρμοστεί. Για να το κάνει αυτό πρέπει να διαπιστώσει αν το (3+4) ταιριάζει με το πρότυπο 0, συνεπώς πρέπει να το αποτιμήσει. Η τιμή που προκύπτει είναι 7, οπότε η Haskell συνεχίζει εξετάζοντας αν την τελευταία ισότητα στον ορισμό της h, η οποία μπορεί να εφαρμοστεί (αφού περιέχει μόνο μεταβλητές), επιστρέφοντας ως αποτέλεσμα το 14.

Συνοπτικά ο υπολογισμός φαίνεται παρακάτω. Το σύμβολο # δηλώνει την αρχή σύγκρισης με πρότυπο, η οποία απαιτείται για να συνεχιστεί υπολογισμός. Το *arg* `<<<` *pattern* δηλώνει ότι εξετάζουμε αν το *arg* ταιριάζει με το πρότυπο *pattern*, δηλαδή ότι το πρότυπο είναι αρκετά γενικό ώστε να εμπεριέχει την τιμή του ορίσματος. Το αποτέλεσμα της διερεύνησης μπορεί να είναι `"yes"` ή `"no"`. (το `<<<` δεν αποτελεί τελεστή της Haskell παρα μόνο μία συντομογραφία που χρησιμοποιούμε εδώ για να περιγράψουμε τον υπολογισμό. Το ίδιο ισχύει και για τα `"yes"`,`"no"`,`<=>` και `#`). Σύγκρίσεις με πρότυπα που είναι μεταβλητές και συνεπώς επιτυγχάνουν αμέσως παραλείπονται.

```
  h (3+4) (8 `div` 3)
    #     8 `div` 3 <<< 0
     <=>  2 <<< 0
     <=>  "no"
    #     3+4 <<< 0
     <=>  7 <<< 0
     <=>  "no"
= 7 * 2
= 14
```

Στον παραπάνω υπολογισμό, αποτιμήθηκαν και τα δύο ορίσματα της h, ωστόσο πρώτα αποτιμήθηκε το δεύτερο και μετά το πρώτο. Αυτό οφείλεται στη σειρά των ισοτήτων που ορίζουν την h και τα πρότυπα που αυτές περιέχουν. ∎

Αν ο ορισμός μίας συνάρτησης περιέχει συνθήκες φρουρούς τότε οι συνθήκες αποτιμούνται με τη σειρά, μέχρι κάποια να έχει τιμή `True`. Η τιμή της συνάρτησης επιστρέφεται από την αντίστοιχη ισότητα.

Παράδειγμα 16: Έστω η συνάρτηση:

```
s :: Int -> Int -> Int -> Int
s a b c
       | a < 0 = b
       | a > 0 = c
       | otherwise = 0
```

Για να αποτιμηθεί η παράσταση `s (2*3) (5*6) (9*4)` πρώτα αποτιμάται η συνθήκη φρουρός `a<0`. Αυτό απαιτεί αποτίμηση του ορίσματος `(2*3)`. Επειδή η συνθήκη δεν ικανοποιείται εξετάζεται η επόμενη συνθήκη φρουρός `a>0`, η οποία αληθεύει, συνεπώς η τιμή της συνάρτηση είναι η τιμή του τρίτου ορίσματός της.

Συνοπτικά ο υπολογισμός φαίνεται παρακάτω. Το σύμβολο ? δηλώνει την αρχή της αποτίμησης συνθήκης φρουρού.

```
  s (2*3) (5*6) (9*4)
      ?   (2*3) < 0
        = 6 < 0
        = False
      ?   6 > 0
        = True
= 9*4
= 36
```

Παρατηρούμε ότι στον παραπάνω υπολογισμό το όρισμα (5*6) δεν αποτιμάται γιατί δεν χρειάζεται στον υπολογισμό του αποτελέσματος. Γενικότερα στην παραπάνω συνάρτηση αποτιμάται το πολύ μία από τις παραμέτρους b,c. ∎

Αν στον ορισμό μίας συνάρτησης υπάρχουν τιμές που ορίζονται τοπικά με το **where**, τότε αυτές υπολογίζονται μόνο μία φορά αν και όταν χρειαστεί (όπως ακριβώς συμβαίνει και με τις παραμέτρους της συνάρτησης).

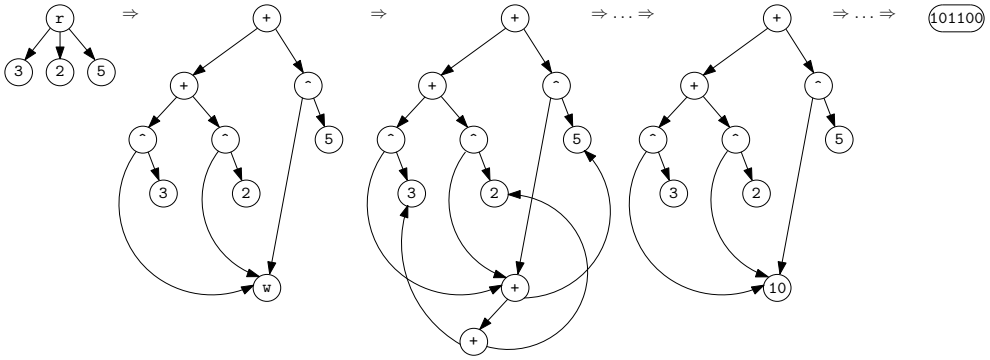
Παράδειγμα 17: Έστω η συνάρτηση:

```
r :: Int -> Int -> Int -> Int
r x y z = w^x + w^y + w^z
      where w = x + y + z
```

Η αποτίμηση του `r 3 2 5` γίνεται με τον παρακάτω τρόπο:

```
  r 3 2 5
= ((w^3) + (w^2)) + (w^5) where w = (3 + 2) + 5
    {  (3 + 2) + 5
     = 5 + 5
     = 10
    }
= ((10^3) + (10^2)) + (10^5)
= (100 + (10^2)) + (10^5)
= (1000 + 100) + (10^5)
= 1100 + (10^5)
= 1100 + 100000
= 101100
```

Σε μορφή γραφημάτων ο παραπάνω υπολογισμός γράφεται:



1.11 Αναδρομή

Στη Haskell επιτρέπεται κατά τον καθορισμό της τιμής μίας συνάρτησης να επικαλεστούμε την ίδια τη συνάρτηση. Αυτό ονομάζεται αναδρομή και η συνάρτηση αναδρομική συνάρτηση.

Για να είναι καλά ορισμένη μία αναδρομική συνάρτηση (δηλαδή για να επιστρέφει αποτέλεσμα για όλες τις τιμές που επιτρέπεται να λάβουν τα ορίσματά της) θα πρέπει:

- να υπάρχουν κάποιες τιμές των ορισμάτων της για τις οποίες το αποτέλεσμα υπολογίζεται από μία εξίσωση που δεν χρησιμοποιεί αναδρομή.
- για τις υπόλοιπες τιμές των ορισμάτων, οι αλυσιδωτές χρήσεις αναδρομικών εξισώσεων να έχουν πεπερασμένο μήκους.

Στη Hakell η αναδρομή χρησιμοποιείται για να πραγματοποιήσουμε έναν υπολογισμό που απαιτεί επανάληψη.

Στα παρακάτω παραδείγματα χρησιμοποιούμε αναδρομή για να υλοποιήσουμε συναρτήσεις, οι περισσότερες από τις οποίες ορίζονται για φυσικούς αριθμούς (μη αρνητικούς ακεραίους) ή γενικότερα για ένα υποσύνολο των ακεραίων. Για απλούστευση, κατά τον ορισμό των συναρτήσεων Haskell που υλοποιουν αυτές τις μαθηματικές συναρτήσεις, υποθέτουμε ότι οι παράμετροι τους έχουν τιμές που ικανοποιούν τους απαραίτητους περιορισμούς. Αργότερα θα δούμε πώς μπορούμε να χειριστούμε τις υπόλοιπες περιπτώσεις.

Η πιό απλή περίπτωση αναδρομικής συνάρτησης μπορεί να οριστεί πάνω σε φυσικούς αριθμούς με τον παρακάτω τρόπο:

- η τιμή της συνάρτησης για το 0 ορίζεται χωρίς αναδρομή.
- η τιμή της συνάρτησης για $n > 0$ ορίζεται χρησιμοποιώντας την τιμή της συνάρτησης για το $(n - 1)$.

Υπάρχουν όμως και άλλες μορφές αναδρομικών συναρτήσεων, ορισμένες από τις οποίες εμφανίζονται στα επόμενα παραδείγματα.

Παράδειγμα 18: παραγοντικό.

Το παραγοντικό ορίζεται για $n \geq 0$ από τον παρακάτω αναδρομικό τύπο:

$$n! = \begin{cases} 1 & \text{αν } n = 0 \\ n \cdot (n-1)! & \text{αν } n > 0 \end{cases}$$

Ο παραπάνω ορισμός μεταφράζεται άμεσα σε Haskell:

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact (n-1)
```

Η Haskell αποτιμάει το `fact 4` με τον παρακάτω τρόπο:

```
  fact 4
     #    4 <<< 0
      <=> "no"
= 4 * (fact (4-1))
     #    4-1 <<< 0
      <=> 3 <<< 0
      <=> "no"
= 4 * (3 * (fact (3-1)))
     #    3-1 <<< 0
      <=> 2 <<< 0
      <=> "no"
= 4 * (3 * (2 * (fact (2-1))))
     #    2-1 <<< 0
      <=> 1 <<< 0
      <=> "no"
= 4 * (3 * (2 * (1 * (fact (1-1)))))
     #    1-1 <<< 0
      <=> 0 <<< 0
      <=> "yes"
= 4 * (3 * (2 * (1 * 1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

∎

Παράδειγμα 19: υπολογισμός του αθροίσματος $\sum_{i=1}^{n} i^i$.

Σε μία προστακτική γλώσσα (C, Pascal) θα υπολογίζαμε το παραπάνω άθροισμα με μία εντολή for. Στη Haskell πρέπει να χρησιμοποιήσουμε αναδρομή.

Παρατηρούμε ότι το παραπάνω άθροισμα μπορεί να οριστεί χρησιμοποιώντας αναδρομή:

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

και υπολογίζεται από την παρακάτω συνάρτηση Haskell:

```
sum1 :: Int -> Int
sum1 0 = 0
sum1 n = sum1 (n-1) + n^n
```

∎

Η αναδρομική ισότητα δεν είναι απαραίτητο να επικαλείται την τιμή της συνάρτησης γαι το $(n-1)$. Επίσης ενδέχεται μία μη αναδρομική εξίσωση στον ορισμό της συνάρτησης να ορίζει το αποτέλεσμα όταν η παράμετρος έχει τιμή διαφορετική του 0.

Παράδειγμα 20: άθροισμα ψηφίων φυσικού αριθμού.

Μπορούμε να χρησιμοποιήσουμε τον τελεστή ‘mod‘ για να απομονώσουμε του τελευταίο ψηφίο του αριθμού και τον ‘div‘ για να διαγράψουμε το ψηφιο αυτό από τον αριθμό:

```
sumOfDigits :: Int -> Int
sumOfDigits 0 = 0
sumOfDigits n =
      sumOfDigits (n ‘div‘ 10) + n ‘mod‘ 10
```

Η παραπάνω συνάρτηση μπορεί να τροποποιηθεί έτσι ώστε για όλους τους μονοψήφιους αριθμούς το αποτέλεσμα να ορίζεται χωρίς αναδρομή. Στην περιπτωση αυτή η συνάρτηση γράφεται πιο κομψα χρησιμοποιώντας συνθήκες φρουρούς.

```
sumOfDigits’ :: Int -> Int
sumOfDigits’ n
     | n <= 9
          = n
     | otherwise
          = sumOfDigits’ (n ‘div‘ 10) + n ‘mod‘ 10
```

∎

Για συναρτήσεις που έχουν περισσότερες από μία παραμέτρους, υπάρχουν πολλές εναλλακτικές επιλογές για το ποια τιμή της συνάρτησης θα χρησιμοποιεί η αναδρομική ισότητα.

Παράδειγμα 21: υπολογισμός δύναμης.

Η παρακάτω συνάρτηση υπολογίζει (όχι με ιδιαίτερα αποδοτικό τρόπο) το n^k, για $k \geq 0$.

```
powerSlow :: Int -> Int -> Int
powerSlow n 0 = 1
powerSlow n k = n * powerSlow n (k-1)
```

Για την αποτίμηση του το n^k με χρήση της `powerSlow` απαιτείται ένα πλήθος βημάτων που είναι ανάλογο του k.

Μια καλύτερη υλοποίηση στηρίζεται στην παρακάτω παρατήρηση:

- αν $k = 2\lambda$ τότε $n^k = n^{2\lambda} = (n^\lambda)^2$
- αν $k = 2\lambda + 1$ τότε $n^k = n^{2\lambda+1} = (n^\lambda)^2 \cdot n$.

```
power :: Int -> Int -> Int
power n 0 = 1
power n k
    | k `mod` 2 == 0
        = p*p
    | otherwise
        = p*p*n
    where p = power n (k `div` 2)
```

Για την αποτίμηση του το n^k με χρήση της `power` απαιτείται ένα πλήθος βημάτων που είναι ανάλογο του λογάριθμου του k.

Για παράδειγμα, η αποτίμηση της παράστασης `power 2 9` γίνεται με τον παρακάτω τρόπο:

```
  power 2 9
     #    9 <<< 0
      <=> "no"
     ?  (9 `mod` 2) == 0
      = 1 == 0
      = False
     ?  otherwise
      = True
= (p*p)*2 where p = power 2 (9 `div` 2)
      { power 2 (9 `div` 2)
           #    9 `div` 2 <<< 0
            <=> 4 <<< 0
            <=> "no"
           ?  (4 `mod` 2) == 0
            = 0 == 0
            = True
       = p*p where p = power 2 (4 `div` 2)
           { power 2 (4 `div` 2)
                #    4 `div` 2 <<< 0
                 <=> 2 <<< 0
```

```
                <=> "no"
               ?  (2 `mod` 2) == 0
                = 0 == 0
                = True
          = p*p where p = power 2 (2 `div` 2)
              { power 2 (2 `div` 2)
                    #    2 `div` 2 <<< 0
                     <=> 1 <<< 0
                     <=> "no"
                    ?  (1 `mod` 2) == 0
                     = 1 == 0
                     = False
                    ?  otherwise
                     = True
               = (p*p)*2 where p = power 2 (1 `div` 2)
                    { power 2 (1 `div` 2)
                         #    1 `div` 2 <<< 0
                          <=> 0 <<< 0
                          <=> "yes"
                     = 1
                    }
               = (1*1)*2
               = 1*2
               = 2
              }
         = 2*2
         = 4
        }
     = 4*4
     = 16
    }
= (16*16)*2
= 256*2
= 512
```

∎

Παράδειγμα 22: υπολογισμός συνδυασμών.

Έστω m και n φυσικοί αριθμοί με $n \geq m$. Συμβολίζουμε με $\binom{n}{m}$ τους συνδυασμούς των n ανά m δηλαδή το πλήθος των διαφορετικών υποσυνόλων με m στοιχεία ένός συνόλου με n στοιχεία.

Είναι γνωστό ότι ισχύει ο παρακάτω αναδρομικό τύπος:

$$\binom{n}{m} = \begin{cases} 1 & m = 0,\ n \geq 0 \\ \frac{n \cdot \binom{n-1}{m-1}}{m} & \text{αν } n \geq m \geq 1 \end{cases}$$

Ο παραπάνω τύπος κωδικοποιείται άμεσα σε Haskell:

```
comb :: Int -> Int -> Int
comb n 0  = 1
comb n m  =  comb (n-1) (m-1) * n `div` m
```

∎

Παράδειγμα 23: υπολογισμός του αθροίσματος $\sum_{i=m}^{n} i^i$, για $m \leq n$.

Υπολογίζεται από την παρακάτω συνάρτηση Haskell:

```
sum2 :: Int -> Int -> Int
sum2 m n
        | m>n
          = 0
        | otherwise
          = m^m + sum2 (m+1) n
```

Ο υπολογισμός του `sum2 m n` με βάση την αναδρομική ισότητα, απαιτεί τον υπολογισμό του `sum2 (m+1) n`. Συνεπώς κατά την αποτίμη, οι τιμές των παραμέτρων της `sum2` ποτέ δεν μειώνονται. Ωστόσο, η αποτίμηση ολοκληρώνεται πάντα σε πεπερασμένο αριθμό βημάτων, καθώς η διαφορά των τιμών των παραμέτρων της `sum2` μειώνεται μετά από κάθε εφαρμογή της αναδρομικής εξίσωσης και όταν αυτή γίνει αρνητική (`m>n`) η αναδρομή σταματάει. ∎

Σε ορισμένες περιπτώσεις ενδέχεται να μην είναι τελείως προφανής ο τρόπος με τον οποίο μία συνάρτηση θα οριστεί αναδρομικά.

Παράδειγμα 24: υπολογισμός του μέγιστου κοινού διαιρέτη δύο θετικών αριθμών.

Δίνουμε πρώτα μία λιγότερη καλή λύση, η τεχνική της οποίας όμως είναι γενικότερα εφαρμόσιμη: δεδομένων δύο θετικών αριθμών m και n με $m \leq n$, γνωρίζουμε ότι ο μέγιστος κοινός διαιρέτης είναι ένας αριθμός μεταξύ 1 και m (αφού θα πρέπει να διαιρεί ακριβώς τον m).

Μπορούμε να βρούμε τον αριθμό εξετάζοντας όλους τους αριθμούς από το m μέχρι το 1, μέχρι να βρούμε κάποιον που να διαιρεί ταυτόχρονα τον m και τον n.

Η αναζήτηση γίνεται με τη βοήθεια της αναδρομικής συνάρτηση `seekGCD`, η οποία με είσοδο τους θετικούς m, n, k επιστρέφει τον μεγαλύτερο αριθμό μεταξύ του 1 και του k που διαιρεί τους m και n.

```
gcdSlow :: Int -> Int -> Int
gcdSlow m n
           | m <= n
              = seekGCD m n m
           | otherwise
              = seekGCD n m n
seekGCD :: Int -> Int -> Int -> Int
seekGCD m n k
           | m `mod` k == 0 && n `mod` k == 0
              = k
           | otherwise
              = seekGCD m n (k-1)
```

Η `seekGCD` επιστρέφει αποτέλεσμα για οποιαδήποτε τριάδα θετικών τιμών m, n, k: στη χειρότερη περίπτωση μετά από $k-1$ εφαρμογές της αναδρομικής ισότητας το τρίτο όρισμα της συνάρτηση θα είναι 1 και πρώτη συνθήκη-φρουρός θα ικανοποιηθεί.

Μία καλύτερη λύση στηρίζεται στον αλγόριθμο του Ευκλείδη, η ορθότητα του οποίου εξασφαλίζεται από το παρακάτω μαθηματικό θεώρημα: αν m, n είναι θετικοί αριθμοί με $m < n$ τότε το σύνολο των κοινών διαιρετών των m και n ισούται με το σύνολο των κοινών διαιρετών των $n - m$ και m.

```
gcdEuc :: Int -> Int -> Int
gcdEuc m n
    | m==n
       = n
    | m<n
       = gcdEuc (n-m) m
    | otherwise
       = gcdEuc (m-n) n
```

Η τιμή της συνάρτησης για δύο θετικούς ακέραιους m και n διαφορετικούς μεταξύ τους ορίζεται χρησιμοποιώντας την τιμή της συνάρτησης για δύο επίσης θετικούς ακεραίους (ενδεχομένως ίσους) που έχουν άθροισμα μικρότερο του $m + n$. Επειδή το άθροισμα δύο θετικών ακεραίων δεν μπορεί να είναι μικρότερο του 2, συμπεραίνουμε ότι η αναδρομή πάντοτε σταματάει. Για παράδειγμα, η Haskell αποτιμάει το `gcdEuc 84 36` με τον παρακάτω τρόπο:

```
  gcdEuc 84 36
     ?  84 == 36
      = False
     ?  84 < 36
      = False
     ?  otherwise
      = True
= gcdEuc (84-36) 36
     ?  (84-36) == 36
      = 48 == 36
```

```
      = False
    ?  48 < 36
      = False
    ?  otherwise
      = True
= gcdEuc (48-36) 36
    ?  (48-36) == 36
      = 12 == 36
      = False
    ?  12 < 36
      = True
= gcdEuc (36-12) 12
    ?  (36-12) == 12
      = 24 == 12
      = False
    ?  24 < 12
      = False
    ?  otherwise
      = True
= gcdEuc (24-12) 12
    ?  (24-12) == 12
      = 12 == 12
      = True
= 12
```

Μπορούμε να τροποποιήσουμε την παραπάνω συνάρτηση έτσι ώστε η τιμή της να ορίζεται αναδρομικά ακόμη και όταν τα ορίσματα της έχουν ίσες τιμές. Σε αυτή την περίπτωση η αναδρομή σταματάει όταν το πρώτο όρισμα γίνει 0:

```
gcdEuc' :: Int -> Int -> Int
gcdEuc' 0 n = n
gcdEuc' m n
    | m<n
        = gcdEuc' (n-m) m
    | otherwise
        = gcdEuc' (m-n) n
```

Προσοχή: Η παράσταση `gcdEuc' 0 0` αποτιμάται σε 0, το οποίο δεν είναι σωστό αποτέλεσμα. Ωστόσο, ο σκοπός μας είναι `gcdEuc'` να δουλεύει σωστά μόνο για θετικές τιμές των παραμέτρων. Η πρώτη ισότητα είναι τεχνητή ώστε να δουλεύει σωστά η αναδρομή.

Ο μέγιστος κοινός διαιρέτης μπορεί να υπολογιστεί με πιο αποδοτικό τρόπο με βάση τις παρακάτω παρατηρήσεις.

Ας υποθέσουμε ότι $n > m$ και ότι η διαίρεση n δια m δίνει πηλίκο q και υπόλοιπο r. Συνεπώς $n = q \cdot m + r$. Ισχύει $gcd(m, n) = gcd(n, m) = gcd(q \cdot m + r, m)$. Εφαρμόζοντας το θεώρημα q φορές έχουμε:

$$
\begin{aligned}
gcd(q \cdot m + r, m) &= gcd((q-1) \cdot m + r, m) \\
&= \ldots \\
&= gcd(m + r, m) \\
&= gcd(r, m)
\end{aligned}
$$

Κάθε μία από τις παραπάνω q ισότητες αντιστοιχεί και σε μία εφαρμογή της αναδρομικής ισότητας στον ορισμό της `gcdEuc'`. Από τις παραπάνω ισότητες προκύπτει ότι $gcd(m, n) = gcd(n \bmod m, m)$. Η παρακάτω συνάρτηση `gcdFast` παρακάμπτει όλες τις ενδιάμεσες εφαρμογές της αναδρομικής ισότητας, αντικαθιστώντας απ' ευθείας το n με το $n \bmod m$:

```
gcdFast :: Int -> Int -> Int
gcdFast 0 m = m
gcdFast m n
    | m < n
        = gcdFast (n `mod` m) m
    | otherwise
        = gcdFast (m `mod` n) n
```

∎

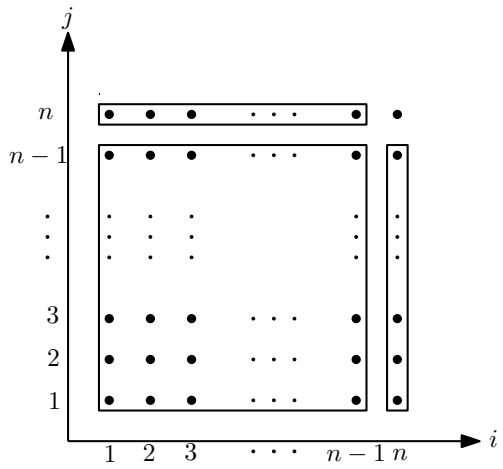
Παράδειγμα 25: υπολογισμός του διπλού αθροίσματος $\sum_{i=1}^{n} \sum_{j=1}^{n} i^j$.

Γενικά για να υπολογίσουμε ένα άθροισμα χρησιμοποιώντας αναδρομή προσπαθούμε να χωρίσουμε τους όρους που πρέπει να προστεθούν σε ξένα υποσύνολα, καποια από τα οποία θα σχηματίζουν αθροίσματα τις ίδιας μορφής με λιγότερους προσθετέους τα οποία μπορούν να υπολογιστούν με αναδρομή, ενώ τα υπόλοιπα θα σχηματίζουν απλούστερης μορφής αθροίσματα (π.χ. απλά αθροίσματα) ή θα είναι μεμονωμένοι όροι.

Παρατηρούμε ότι το παραπάνω άθροισμα ικανοποιεί την αναδρομική ισότητα:

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

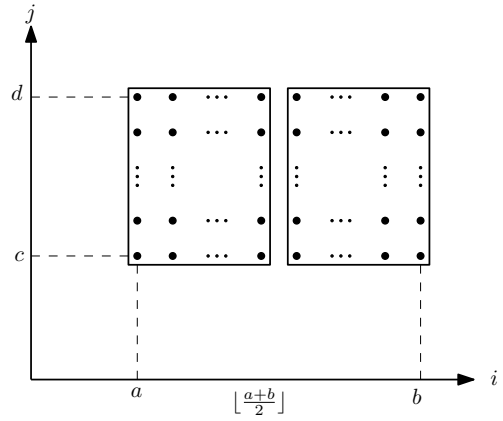
Σχηματικά αυτό παριστάνεται με τον παρακάτω τρόπο:

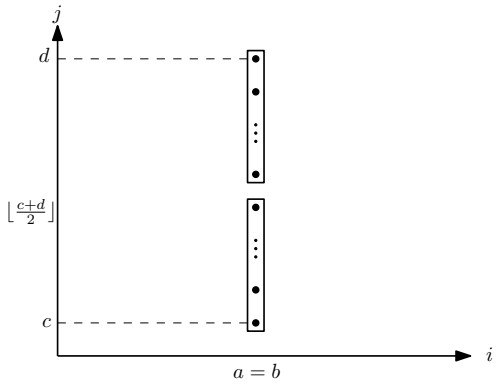


Μπορούμε γράψουμε μια συνάρτηση Haskell για υπολογισμό του αθροίσματος με βάση τον παραπάνω τύπο, γράφοντας και δύο βοηθητικές συναρτήσεις για τον υπολογισμό των απλών αθροισμάτων. Θα αναβάλουμε για αργότερα την περιγραφή μίας τέτοιας συνάρτησης.

Σε ορισμένες περιπτώσεις είναι πιο βολικό να υλοποιήσουμε μία πιο γενική συνάρτηση από αυτή που μας ενδιαφέρει. Στη συγκεκριμένη περίπτωση θα σχεδιάσουμε μία συνάρτηση για υπολογισμό του το άθροίσματος $\sum_{i=a}^{b}\sum_{j=c}^{d} i^j$ με $a \leq b$ και $c \leq d$.

Στην παρακάτω συνάρτηση `sumabcd`, αν το πλήθος των προσθετέων είναι μεγαλύτερο του ένα, το αποτέλεσμα ορίζεται αναδρομικά με διαχωρισμό των τιμών για τον έναν από του δύο δείκτες σε δύο περίπου ίσα διαστήματα. Σχηματικά:






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

Μπορούμε να αποφύγουμε τελείως τη διαίρεση, χωρίζοντας διαφορετικά τις τιμές των δεικτών:

```
sumabcd' :: Int -> Int -> Int -> Int -> Int
sumabcd' a b c d =
    if a==b then
       if c==d then a^c
               else sumabcd' a b c c
                  + sumabcd' a b (c + 1) d
            else sumabcd' a a c d
               + sumabcd' (a + 1) b c d
```

Η `sumabcd'` δεν χρησιμοποιεί διαίρεση, ωστόσο σχηματίζει μεγαλύτερες αναδρομικές αλυσίδες.

Το ζητούμενο άθροισμα $\sum_{i=1}^{n} \sum_{j=1}^{n} i^j$ μπορεί να υπολογιστεί από την παρακάτω συνάρτηση:

```
doubleSum0n :: Int -> Int
doubleSum0n n = sumabcd 1 n 1 n
```

∎

Σε ορισμένες περιπτώσεις, η απευθείας χρήση ενός αναδρομικού τύπου μπορεί να επιβαρύνει τον χρόνο υπολογισμού. Αυτό συμβαίνει όταν ο ίδιος υπολογισμός επαναλαμβάνεται πολλές φορές.

Παράδειγμα 26: αριθμοί Fibonacci. Οι αριθμοί Fibonacci ορίζονται από τον παρακάτω αναδρομικό τύπο:

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 \\ f_n &= f_{n-2} + f_{n-1}, \text{ για } n \geq 2 \end{aligned}$$

Ο παραπάνω αναδρομικός τύπος μεταφράζεται άμεσα σε Haskell:

```
fibSlow :: Int -> Int
fibSlow 0 = 1
fibSlow 1 = 1
fibSlow n = fibSlow (n-2) + fibSlow (n-1)
```

Η παραπάνω υλοποίηση αποτελεί παράδειγμα κακής χρήσης της αναδρομής, καθώς οι ίδιες τιμές υπολογίζονται πάρα πολλές φορές. Για να γίνει αυτό εμφανές δίνουμε την αποτίμηση του `fibSlow 5`:

```
  fibSlow 5
     #    5 <<< 0
      <=> "no"
     #    5 <<< 1
      <=> "no"
= (fibSlow (5-2)) + (fibSlow (5-1))
     #    (5-2) <<< 0
      <=> 3 <<< 0
      <=> "no"
     #    3 <<< 1
      <=> "no"
= ((fibSlow (3-2)) + (fibSlow (3-1))) + (fibSlow (5-1))
     #    (3-2) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= (1 + (fibSlow (3-1))) + (fibSlow (5-1))
     #    (3-1) <<< 0
      <=> 2 <<< 0
      <=> "no"
     #    2 <<< 1
      <=> "no"
= (1 + ((fibSlow (2-2)) + (fibSlow (2-1)))) + (fibSlow (5-1))
     #    (2-2) <<< 0
      <=> 0 <<< 0
      <=> "yes"
= (1 + (1 + (fibSlow (2-1)))) + (fibSlow (5-1))
```

```
     #    (2-1) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= (1 + (1 + 1)) + (fibSlow (5-1))
= (1 + 2) + (fibSlow (5-1))
= 3 + (fibSlow (5-1))
     #    (5-1) <<< 0
      <=> 4 <<< 0
      <=> "no"
     #    4 <<< 1
      <=> "no"
= 3 + ((fibSlow (4-2)) + (fibSlow (4-1)))
     #    (4-2) <<< 0
      <=> 2 <<< 0
      <=> "no"
     #    2 <<< 1
      <=> "no"
= 3 + (((fibSlow (2-2)) + (fibSlow (2-1))) + (fibSlow (4-1)))
     #    (2-2) <<< 0
      <=> 0 <<< 0
      <=> "yes"
= 3 + ((1 + (fibSlow (2-1))) + (fibSlow (4-1)))
     #    (2-1) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= 3 + ((1 + 1) + (fibSlow (4-1)))
= 3 + (2 + (fibSlow (4-1)))
     #    (4-1) <<< 0
      <=> 3 <<< 0
      <=> "no"
     #    3 <<< 1
      <=> "no"
= 3 + (2 + ((fibSlow (3-2)) + (fibSlow (3-1))))
     #    (3-2) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= 3 + (2 + (1 + (fibSlow (3-1))))
     #    (3-1) <<< 0
      <=> 2 <<< 0
      <=> "no"
```

```
     #    2 <<< 1
      <=> "no"
= 3 + (2 + (1 + ((fibSlow (2-2)) + (fibSlow (2-1)))))
     #    (2-2) <<< 0
      <=> 0 <<< 0
      <=> "yes"
= 3 + (2 + (1 + (1 + (fibSlow (2-1)))))
     #    (2-1) <<< 0
      <=> 1 <<< 0
      <=> "no"
     #    1 <<< 1
      <=> "yes"
= 3 + (2 + (1 + (1 + 1)))
= 3 + (2 + (1 + 2))
= 3 + (2 + 3)
= 3 + 5
= 8
```

Παρατηρούμε ότι το f_3 υπολογίζεται 2 φορές, το f_2 υπολογίζεται 3 φορές, και το το f_1 υπολογίζεται 5 φορές. Γενικότερα, για να υπολογιστεί το f_n, υπολογίζεται αναδρομικά f_i φορές το f_{n-i}, για όλες τις τιμές του i από 1 έως n. Η συνάρτηση f_n αυξάνει όμως πάρα πολύ γρήγορα σε σχέση με το n και αυτό έχει ως αποτέλεσμα η παραπάνω υλοποίηση να μην είναι αποδοτική.

Η παρακάτω υλοποίηση της συνάρτησης, υπολογίζει αποδοτικά το f_n κάνοντας διαδοχικές αθροίσεις, από "κάτω προς τα πάνω":

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fibHlp (n-1) 1 1
    where fibHlp :: Int -> Int -> Int -> Int
          fibHlp 1 a b = a + b
          fibHlp k a b = fibHlp (k-1) b (a+b)
```

Ο υπολογισμός του f_5 αποτιμώντας την παράσταση `fib 5` είναι σαφώς γρηγορότερος:

```
  fib 5
     #    5 <<< 0
      <=> "no"
     #    5 <<< 1
      <=> "no"
= fibHlp (5-1) 1 1
     #    (5-1) <<< 1
      <=> 4 <<< 1
      <=> "no"
= fibHlp (4-1) 1 (1+1)
```

```
     #    (4-1) <<< 1
      <=> 3 <<< 1
      <=> "no"
= fibHlp (3-1) (1+1) (1+(1+1))
     #    (3-1) <<< 1
      <=> 2 <<< 1
      <=> "no"
= fibHlp (2-1) (1+(1+1)) ((1+1)+(1+(1+1)))
     #    (2-1) <<< 1
      <=> 1 <<< 1
      <=> "yes"
= (1+(1+1)) + ((1+1)+(1+(1+1)))
= (1+2) + (2+(1+2))
= 3 + (2+3)
= 3 + 5
= 8
```

Στον παραπάνω υπολογισμό, η αποτίμηση του ορίσματος `1+1` γίνεται μία μόνο φορά, όταν διαπιστωθεί ότι αυτό είναι απαραίτητο για τον υπολογισμό του αποτελέσματος και είναι ορατή σε όλα τα τμήματα της παράστασης που το χρησιμοποιούν.

Παρόμοια, το όρισμα `(1+(1+1))` υπολογίζεται μία μόνο φορά, αφού ενδιάμεσα πάρει τη μορφή `(1+2)`. ∎

Στο παρακάτω παράδειγμα αυτό θα φανεί η σημασία των τοπικών ορισμών, σε περίπτωση που μία τιμή που υπολογίζεται αναδρομικά χρησιμοποιείται σε περισσότερα από ένα σημεία του ορισμού της συνάρτησης.

Παράδειγμα 27: ακέραιο μέρος της τετραγωνικής ρίζας ενός αριθμού.

Χρησιμοποιούμε την παρακάτω ιδιότητα:

- αν ο αριθμός n είναι τέλειο τετράγωνο, τότε $\lfloor\sqrt{n}\rfloor = \sqrt{n} = \lfloor\sqrt{n-1}\rfloor + 1$. Σε αυτή την περίπτωση ισχύει $n = (\lfloor\sqrt{n-1}\rfloor + 1)^2$.
- αν ο n δεν είναι τέλειο τετράγωνο, τότε $\lfloor\sqrt{n}\rfloor = \lfloor\sqrt{n-1}\rfloor$ και $n < (\lfloor\sqrt{n-1}\rfloor + 1)^2$

Η παρακάτω συνάρτηση δεν χρησιμοποιεί τοπικούς ορισμούς:

```
sqrtIntSlow :: Int -> Int
sqrtIntSlow 0 = 0
sqrtIntSlow n
     | (sqrtIntSlow (n-1) + 1)^2 == n
         = sqrtIntSlow (n-1) + 1
     | otherwise
         = sqrtIntSlow (n-1)
```

Στην παραπάνω υλοποίηση γίνεται κακή χρήση της αναδρομής: κατά τον υπολογισμό του $\lfloor\sqrt{n}\rfloor$ το $\lfloor\sqrt{(n-i)}\rfloor$ υπολογίζεται 2^i φορές, συνεπώς το συνολικό πλήθος βημάτων είναι εκθετικό ως προς το n.

Η παρακάτω υλοποίηση για να υπολογίσει το $\lfloor\sqrt{n}\rfloor$ υπολογίζει μόνο μία φορά το $\lfloor\sqrt{(n-1)}\rfloor$ και του δίνει τοπικά όνομα με χρήση του where. Με αυτό τον τρόπο το πλήθος των βημάτων γίνεται αναλογο του n.

```
sqrtInt :: Int -> Int
sqrtInt 0 = 0
sqrtInt n
     | (m + 1)^2 == n
         = m + 1
     | otherwise
         = m
     where m = sqrtInt (n-1)
```

Μία ακόμη πιο αποδοτική υλοποίηση στηρίζεται στην δυαδική αναζήτηση της σωστής τιμής στο διάστημα 0 έως n, στην οποία το πλήθος των βημάτων είναι λογαριθμικό ως προς το n.

Ορίζουμε τοπικά μία συνάρτηση `sqrtHlp` η οποία παίρνει ως είσοδο ένα διάστημα μέσα στο οποίο βρίσκεται η τετραγωνική ρίζα. Μέσα στην `sqrtHlp` ορίζουμε τοπικά το `c`, που είναι το μέσο του διαστήματος. Κάθε εφαρμογή της αναδρομικής ισότητας μειώνει το μήκος του διαστήματος στο οποίο βρίσκεται η τετραγωνική ρίζα στο μισό. Όταν το μήκος αυτό γίνει ένα, η τετραγωνική ρίζα έχει εντοπιστεί και η αναδρομή σταματάει.

```
sqrtIntFast :: Int -> Int
sqrtIntFast n = sqrtHlp 0 n
         where sqrtHlp :: Int -> Int -> Int
               sqrtHlp a b
                   | a==b
                         = a
                   | c*c > n
                         = sqrtHlp a (c-1)
                   | otherwise
                          = sqrtHlp c b
                     where c = (a+b+1) `div` 2
```

Το `sqrtIntFast 40` αποτιμάται με τον παρακάτω τρόπο:

```
  sqrtIntFast 40
= sqrtHlp 0 40
     ?  0 == 40
      = False
     ?  (c*c) > 40 where c = ((0+40)+1) `div` 2
           {  ((0+40)+1) `div` 2
```

```
            = (40+1) `div` 2
            = 41 `div` 2
            = 20
           }
      = (20*20) > 40
      = 400 > 40
      = True
= sqrtHlp 0 (20-1)
     ?  0 == (20-1)
        0 == 19
      = False
     ?  (c*c) > 40 where c = ((0+19)+1) `div` 2
           {  ((0+19)+1) `div` 2
            = (19+1) `div` 2
            = 20 `div` 2
            = 10
           }
      = (10*10) > 40
      = 100 > 40
      = True
= sqrtHlp 0 (10-1)
     ?  0 == (10-1)
        0 == 9
      = False
     ?  (c*c) > 40 where c = ((0+9)+1) `div` 2
           {  ((0+9)+1) `div` 2
            = (9+1) `div` 2
            = 10 `div` 2
            = 5
           }
      = (5*5) > 40
      = 25 > 40
      = False
     ?  otherwise
      = True
= sqrtHlp 5 9
     ?  5 == 9
      = False
     ?  (c*c) > 40 where c = ((5+9)+1) `div` 2
           {  ((5+9)+1) `div` 2
            = (14+1) `div` 2
            = 15 `div` 2
            = 7
           }
      = (7*7) > 40
      = 49 > 40
```

```
      = True
= sqrtHlp 5 (7-1)
     ?  5 == (7-1)
        5 == 6
      = False
     ?  (c*c) > 40 where c = ((5+6)+1) `div` 2
           {  ((5+6)+1) `div` 2
            = (11+1) `div` 2
            = 12 `div` 2
            = 6
           }
      = (6*6) > 40
      = 36 > 40
      = False
     ?  otherwise
      = True
= sqrtHlp 6 6
     ?  6 == 6
      = True
= 6
```

1.12 Χειρισμός μη αποδεκτών είσόδων

Οι ορισμοί των συναρτήσεων που είδαμε μέχρι τώρα, υποθέτουν ότι οι τιμές των ορισμάτων τους ικανοποιούν κάποιες συνθήκες: Συγκεκριμένα:

- οι ορισμοί των `fact`, `sum1`, `sumOfDigits`, `fib`, `sqrtInt` υποθέτουν ότι τα ορίσματα τους είναι θετικοί αριθμοί ή μηδέν.
- οι ορισμοί των συναρτήσεων για υπολογισμό του μέγιστου κοινού διαιρέτη υποθέτουν ότι τα ορίσματα τους είναι θετικοί αριθμοί.
- ο ορισμός της `comb` υποθέτει επιπλέον ότι $n \geq m$.
- αντίστοιχες ανισότητες πρέπει να ικανοποιούν και οι τιμές των παραμέτρων των `sum2` και `sumabcd`, οι οποίες όμως επιστρέφουν σωστό αποτέλεσμα και για αρνητικές τιμές των ορισμάτων τους.

Ενδέχεται το πρόγραμμα που χρησιμοποιεί τις συναρτήσεις αυτές να εξασφαλίζει ότι πάντοτε αποτιμούνται με ορίσματα που οι τιμές τους ικανοποιούν τους απαραίτητους περιορισμούς. Αν αυτό δε συμβαίνει τότε θα πρέπει να επεκτείνουμε τις συναρτήσεις μας ώστε να πραγματοποιούν τους απαραίτητους ελέγχους έτσι ώστε το πρόγραμμα να έχει πάντα προβλέψιμη συμπεριφορά.

Τα παρακάτω παραδείγματα δίνουν μερικές ιδέες για το πώς μπορούμε να χειριστούμε τις προβληματικές εισόδους.

Παράδειγμα 28: παραγοντικό.

Αν ζητήσουμε από το διρμηνέα της Haskell να αποτιμήσει τη συνάρτηση `fact` με αρνητικό όρισμα, τότε θα προκληθεί υπερχείλιση στοίβας. Μπορούμε τροποποιήσουμε την `fact` έτσι ώστε όταν το όρισμα της έχει αρνητική τιμή, να επιστρέφει ένα μήνυμα λάθους:

```
fact' :: Int -> Int
fact' n
     | n<0
        = error "fact': negative argument"
     | n=0
        = 1
     | otherwise
        = n * fact' (n-1)
```

Η συνάρτηση `error` προκαλεί άμεση διακοπή της εκτέλεσης του προγράμματος και εμφανίζει ενα μήνυμα λάθους.

Το μειονέκτημα της παραπάνω υλοποίησης είναι ότι ο έλεγχος για αρνητικό όρισμα γίνεται $(n + 1)$ φορές κατά τον υπολογισμό του $n!$ για $n \geq 0$. Μπορούμε να το αποφύγουμε τους πολλαπλούς ελέγχους σχεδιάζοντας μια συνάρτηση που θα κάνει τον απαραίτητο έλεγχο μία μόνο φορά και θα επιστρέφει αποτέλεσμα χρησιμοποιώντας την `fact` αν το όρισμα της έχει μη αρνητική τιμή. Με αυτό τον τρόπο διαχωρίζουμε τον έλεγχο της εισόδου από την αναδρομή και μπορούμε να επιλέξουμε ποια συνάρτηση θα καλέσουμε ανάλογα με το αν θέλουμε να γίνει έλεγχος της εισόδου ή όχι.

```
factGen :: Int -> Int
factGen n
     | n<0
        = error "factGen: negative argument"
     | otherwise
        = fact n
```

Αν δεν θέλουμε να τερματίστει η αποτίμηση με μήνυμα λάθους, μπορούμε να επιλέξουμε μία τιμή η οποία δεν ανήκει στο πεδίο τιμών της συνάρτησης (π.χ. `0`) και να την επιστρέφουμε, κωδικοποιώντας έτσι το λάθος:

```
factGen' :: Int -> Int
factGen' n
     | n<0
        = 0
     | otherwise
        = fact n
```

Μπορούμε να επιστρέφουμε πιο άμεσα την ένδειξη για λάθος τροποποιώντας τον τύπο του αποτελέσματος της συνάρτησης:

```
factGen'' :: Int -> (Bool,Int)
factGen'' n
     | n<0
        = (False,0)
     | otherwise
        = (True,fact n)
```

∎

Παράδειγμα 29: υπολογισμός συνδυασμών.

Το $\binom{n}{m}$ μπορεί να οριστεί ακόμη και όταν $m > n$: επειδή σε αυτή την περίπτωση ένα σύνολο με n στοιχεία δεν μπορεί να έχει υποσύνολο με m στοιχεία ισχύει $\binom{n}{m} = 0$. Αντίθετα, αν κάποιο από τα n και m είναι αρνητικός αριθμός, επειδή αυτά παριστάνουν πληθάριθμους συνόλων, το $\binom{n}{m}$ δεν μπορεί να οριστεί με προφανή τρόπο.

Με βάση τις παρατηρήσεις αυτές, μπορούμε να γράψουμε την παρακάτω γενίκευση της comb:

```
combGen :: Int -> Int -> Int
combGen n m
      | n<0 || m<0
         = error "combGen: negative argument"
      | n<m
         = 0
      | otherwise
         = comb n m
```

Θα μπορούσαμε να "κρύψουμε" την comb στο εσωτερικό της combGen:

```
combGen' :: Int -> Int -> Int
combGen' n m
     | n<0 || m<0
        = error "combGen': negative argument"
     | n<m
        = 0
     | otherwise
        = comb n m
     where comb :: Int -> Int -> Int
           comb n 0  = 1
           comb n m  =  comb (n-1) (m-1) * n `div` m
```

∎

1.13 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 ισότητα 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 [7,4,5,4,1]` γίνεται με τον παρακάτω τρόπο:

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

∎

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 [2,3,4] [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 [2,3] [1,4,5]` 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. Σε αντίθετη περίπτωση (δηλαδή αν η αρχική λίστα έχει το πολύ ένα στοιχείο) τότε το επιστρεφόμενο ζεύγος αποτελείται από την αρχική και την κενή λίστα.

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

∎

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 `insSortInt [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`. ∎

1.14 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 αποτιμήσουμε την `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 προκαθορισμένες 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"
     #    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 μικρότερη από `n`, τότε επιστρέφεται το αποτέλεσμα της αναζήτησης του `n` στην ουρά της λιστας, το οποίο υπολογίζεται αναδρομικά. Σε αντίθετη περίπτωση επιστρέφεται `True` αν και μόνο αν η κεφαλή της λιστας είναι `n` (αν η κεφαλή της λιστας είναι μεγαλύτερη από `n` τότε επειδή η λίστα είναι ταξινομημένη, όλα τα στοιχεία της είναι μεγαλύτερα του `n` και το σωστό αποτέλεσμα είναι `False`).

```
searchInt :: Int -> [Int] -> Bool
searchInt n (h:t)
      | n > h = searchInt n t
      | otherwise = n == h
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
      = 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
      = 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
      = False
     ? otherwise
      = True
= 13 == 16
= 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 αριθμός.
- 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..]) <<< (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
```

∎

1.15 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`). ∎

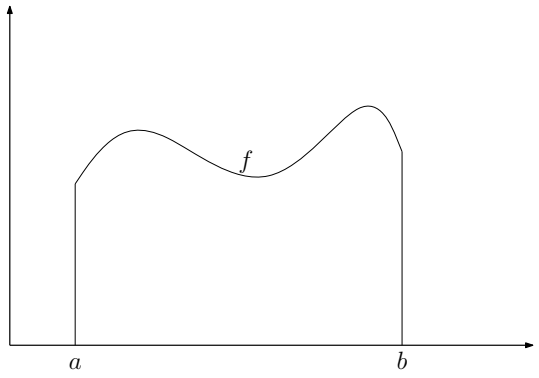
Παράδειγμα 49: υπολογισμός διπλού αθροίσματος.

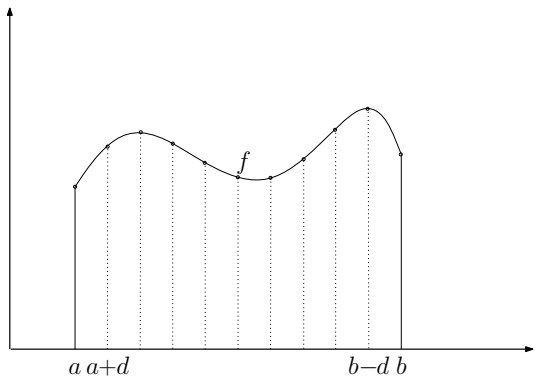
Με παρόμοιο τρόπο όπως στο προηγούμενο παράδειγμα μπορούμε να τροποποιήσουμε τη συνάρτηση `sumabcd` του παραδείγματος 25, ώστε να υπολογίζει το άθροισμα $\sum_{i=a}^{b}\sum_{j=c}^{d} f(i,j)$ για οποιαδήποτε συνάρτηση f τύπου `Int -> Int -> Int`.

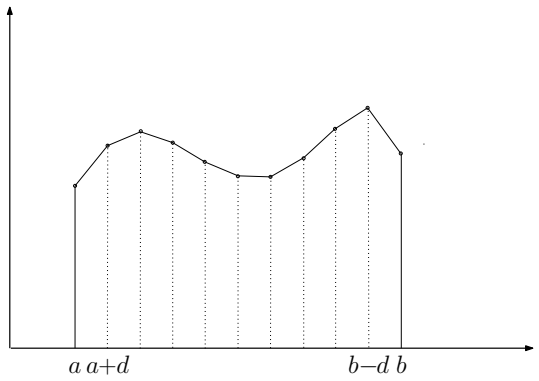
```
sumFabcd  :: (Int -> Int -> Int) ->
              Int -> Int -> Int -> Int -> Int
sumFabcd f a b c d
 = if a==b
   then
       if c==d
       then f a c
       else sumFabcd f a b c n
          + sumFabcd f a b (n + 1) d
   else sumFabcd f a m c d
      + sumFabcd f (m + 1) b c d
   where m = (a + 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 0.0001 - που ισοδυναμεί με χωρισμό της περιοχής σε 10000 τραπέζια) γράφουμε:

```
> integral exp 0 1 0.0001
```

∎

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 ελάχιστο, 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 τάξης, 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]
```

∎

1.16 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 $\texttt{t}_i\texttt{->} \ \ldots \ \texttt{->t}_k\texttt{->t}$ (που έχει προκύψει από το προηγούμενο βήμα) με όρισμα $\texttt{x}_i$. Το αποτέλεσμα είναι μία συνάρτηση τύπου $\texttt{t}_{i+1}\texttt{->} \ \ldots \ \texttt{->t}_k\texttt{->t}$.

- Τέλος στο k-οστό βήμα αποτιμά μία συνάρτηση τύπου $\texttt{t}_k\texttt{->t}$ με όρισμα $\texttt{x}_k$. 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]
```

∎

1.17 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 προκαθορισμένενες 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.

1.18 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]]
```

1.19 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)
```