Κεφάλαιο 8
Συναρτησιακός Προγραμματισμός:
Η Γλώσσα Haskell

Π. Ροντογιάννης

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Συναρτησιακές Γλώσσες Προγραμματισμού

- Γλώσσες στις οποίες ο προγραμματιστής χρησιμοποιεί σχεδόν αποκλειστικά **συναρτήσεις** για περιγραφή ενός αλγορίθμου ή επίλυση ενός προβλήματος.

Συνάρτηση

- Μία συνάρτηση δίνει μία αντιστοίχιση ανάμεσα στα στοιχεία ενός συνόλου (πεδίο ορισμού) και σε ένα άλλο σύνολο (πεδίο τιμών)

Παράδειγμα

Η συνάρτηση $f(x) = x + 1$ *είναι ιση με* $\{(0, 1), (1, 2), \cdots\}$

Παράδειγμα

Παράδειγμα

$$\{\cdots(-3, minus), (-2, minus), (-1, minus), (0, zero), (1, plus), (2, plus), (3, plus), \cdots\}$$

$$sign(x) = \begin{cases} minus, & if\ x < 0 \\ zero, & if\ x = 0 \\ plus, & if\ x > 0 \end{cases}$$

- Το x στα δύο αυτά παραδείγματα ονομάζεται **τυπική παράμετρος** της συνάρτησης.
- Η παραπάνω συνάρτηση είναι ορισμένη για όλα τα στοιχεία του συνόλου των ακεραίων και ονομάζεται **ολική (total)** συνάρτηση.

Παράδειγμα

Παράδειγμα

$$sign2(x) = \begin{cases} minus, & if\ x < 0 \\ plus, & if\ x > 0 \end{cases}$$

- Είναι **μερική (partial)** συνάρτηση.
- Μπορούμε να δούμε τις συναρτήσεις ως black boxes.

- Το 6 ονομάζεται **πραγματική (actual)** παράμετρος της *sign*.

Σχόλια

- Θα λέμε ότι η συνάρτηση **εφαρμόζεται** (applied) στην πραγματική παράμετρο.
- Στο παραπάνω παράδειγμα, η εφαρμογή αυτή γράφεται: $sign(6)$
- Μία πολύ χρήσιμη «πράξη» ανάμεσα σε συναρτήσεις είναι η **σύνθεση** (composition)

Παραδείγματα

$$max(m,n) = \begin{cases} m, & \text{αν } m > n \\ n, & \text{διαφορετικά} \end{cases}$$

$$max(a,b,c) = \begin{cases} a, & \text{αν } a \geq b \text{ και } a \geq c \\ b, & \text{αν } b \geq a \text{ και } b \geq c \\ c, & \text{αν } c \geq a \text{ και } c \geq b \end{cases}$$

$$max3(a,b,c) = max(a, max(b,c))$$

$$SM4(a,b,c,d) = sign(max(a, max3(b,c,d)))$$

Παράδειγμα Συναρτησιακού Προγράμματος (1)

Υπολογισμός **μεγίστου** δύο αριθμών

Πρόγραμμα

```
max(a,b) = if (a>b) then a else b
```

- Το πρόγραμμα μοιάζει πολύ με έναν καθαρά μαθηματικό ορισμό.
- Τα a,b ονομάζονται **τυπικές παράμετροι** (formal parameters) της συνάρτησης max. Όταν δίνονται συγκεκριμένες τιμές στις τυπικές παραμέτρους, τότε «καλείται» η συνάρτηση με αυτές τις παραμέτρους.

Παράδειγμα Συναρτησιακού Προγράμματος (2)

- Για `max(2,3)` η συνάρτηση «καλείται» με πρώτη παράμετρο 2 και δεύτερη παράμετρο 3.
- Οι παράμετροι με τις οποίες καλείται μία συνάρτηση (όπως τα 2 και 3) ονομάζονται **πραγματικές παράμετροι** (actual parameters).

Συναρτησιακός προγραμματισμός

- Δυνατότητα δημιουργίας πιο πολύπλοκων συναρτήσεων με σύνθεση απλούστερων.
- Πρόβλημα εύρεσης μέγιστου **τριών** αριθμών
  - Ορισμός νέας συνάρτησης χρησιμοποιώντας την συνάρτηση max:

Πρόγραμμα

```
max3(a,b,c) = max(a, max(b,c))
```

Διαφορά

- Βασική διαφορά συναρτησιακών και προστακτικών γλωσσών προγραμματισμού
- Γλώσσες προγραμματισμού (C και Pascal) δεν διαθέτουν (σε αντίθεση με τις συναρτησιακές γλώσσες) την ιδιότητα της **διαφάνειας αναφοράς** (referential transparency)

Διαφάνεια αναφοράς

- Κάθε έκφραση σε ένα πρόγραμμα αντιστοιχεί σε **μία και μοναδική** τιμή.
- Αν η ίδια έκφραση ξανα-υπολογισθεί τότε θα δώσει και πάλι την **ίδια τιμή** (δηλαδή ο υπολογισμός της τιμής μίας έκφρασης δεν αλλάζει ποτέ την τιμή της)

Παράδειγμα Προγράμματος

Πρόγραμμα

```
program example (output);
var flag:  boolean;
function f(n:  integer):  integer;
begin
    if flag then f:=n
    else f:= 2*n;
    flag :=not flag
end;
begin
    flag := true;
    writeln(f(1)+f(2));
    writeln(f(2)+f(1));
end
```

Παράδειγμα Προγράμματος

- Η εκτέλεση του προγράμματος δίνει **δύο τιμές** (5 και 4).
- Θα περιμέναμε και στις δύο περιπτώσεις το ίδιο αποτέλεσμα λόγω αντιμεταθετικότητας της πρόσθεσης στα μαθηματικά
- Πρόβλημα
  - Διαφορά ανάμεσα στις συναρτήσεις, που ορίζονται σε μία γλώσσα όπως η Pascal, με τις μαθηματικές συναρτήσεις.
  - Χρήση εντολών ανάθεσης (assignments) και καθολικών (global) μεταβλητών

Διαφάνεια αναφοράς

- Οι συναρτησιακές γλώσσες διαθέτουν την ιδιότητα της διαφάνειας αναφοράς.
- Προγράμματα, που γράφονται σε αυτές αντιπροσωπεύουν στην ουσία **μαθηματικές συναρτήσεις**.

Εισαγωγή στη Γλώσσα Haskell

- **Haskell**: Σύγχρονη συναρτησιακή γλώσσα, που χρησιμοποιείται για απλές μέχρι αρκετά πολύπλοκες εφαρμογές.
- Κλήση του διερμηνέα της Haskell και εμφάνιση του συμβόλου «?». Ο χρήστης δίνει μία έκφραση για υπολογισμό.

Παράδειγμα

Παράδειγμα

```
?  (2+3)*8
40
?  sum [1..10]
55
?
```

- Ο συμβολισμός [1..10] αναπαριστά λίστα ακεραίων από το 1 μέχρι το 10.
- Το sum είναι συνάρτηση του συστήματος για υπολογισμό αθροίσματος των στοιχείων λίστας.
- Ο χρήστης μπορεί να ορίσει δικές του συναρτήσεις.

Τύποι της Haskell

Σύστημα τύπων

- Πολύ χρήσιμο για ανίχνευση λαθών σε εκφράσεις και ορισμούς συναρτήσεων, που γράφει ο προγραμματιστής.
- Δίνεται ένας τύπος σε κάθε έκφραση, ο οποίος χαρακτηρίζει το είδος της τιμής, που αναπαρίσταται από αυτήν (ακέραιος, αριθμός κινητής υποδιαστολής, κλπ).

Παράδειγμα

- **Έκφραση::τύπος**: Συμβολισμός ότι η έκφραση έχει τον συγκεκριμένο τύπο.
- Όταν γράφουμε
  - `42::Int`
    εννοούμε ότι το 42 είναι **ακέραιος** (`Int` είναι το όνομα, που αναπαριστά τον τύπο των ακεραίων αριθμών).
  - `fact::Int -> Int`
    εννοούμε ότι `fact` είναι μία **συνάρτηση**, που παίρνει σαν όρισμα έναν ακέραιο και επιστρέφει έναν ακέραιο.

Λογικές Τιμές

- Συμβολίζονται με `Bool`
- Λαμβάνουν δύο τιμές: `True` και `False`
- Υποστηρίζονται από τη Haskell οι συναρτήσεις επεξεργασίας λογικών τιμών:

`a && b` είναι `True`	και το `a` και το `b` είναι `True`
`a && b` είναι `False`	διαφορετικά
`a || b` είναι `True`	ένα από τα `a` και `b` είναι `True`
`a || b` είναι `False`	διαφορετικά
`not a` είναι `True`	`a` είναι `False`
`not a` είναι `False`	`a` είναι `True`

Ακέραιες Τιμές

- Συμβολίζονται με `Int`
- Περιλαμβάνουν θετικούς και αρνητικούς αριθμούς
- Υποστηρίζονται από Haskell με πολλές συναρτήσεις για την επεξεργασία τους

Συναρτήσεις Ακέραιων Τιμών

+	Πρόσθεση
-	Αφαίρεση
*	Πολλαπλασιασμός
/	Ακέραιη Διαίρεση
rem	Υπόλοιπο
odd	Επιστρέφει True (όρισμα περιττός), διαφορετικά False
even	Επιστρέφει True (όρισμα άρτιος), διαφορετικά False
abs	Επιστρέφει απόλυτη τιμή ορίσματος

Αριθμοί Κινητής Υποδιαστολής

- Συμβολίζονται με `Float`
  - Χρησιμοποιούνται σε υπολογισμούς μεγάλης ακρίβειας
  - Πράξεις με αυτούς είναι γενικά ακριβείς. Σε πολύπλοκους υπολογισμούς ενδέχεται να υπεισέλθουν σφάλματα, που αλλοιώνουν το τελικό αποτέλεσμα.
- Αναπαρίστανται με χρήση υποδιαστολής (πχ. `3.141`) ή με επιστημονικό συμβολισμό (πχ. `5.0e-2`, είναι ισοδύναμο με `0.05`)

Χαρακτήρες

- Συμβολίζονται με `Char`
  - Ατομικοί χαρακτήρες, όπως αυτοί σε ένα πληκτρολόγιο
  - Διαφορετικός τύπος από το `Int`
- Γράφονται σαν χαρακτήρας, που περικλείεται με αποστρόφους
  - `'a'`, `'0'`, `'z'`

Συναρτήσεις και Χαρακτήρες

Επιτρέπουν τη μετατροπή ενός χαρακτήρα στον αντίστοιχο ASCII κώδικα και αντίστροφα.

Παράδειγμα

```
?  ord 'a'
97

?  chr 65
'A'
```

Η ord επιστρέφει τον αντίστοιχο ASCII κώδικα ενός χαρακτήρα, ενώ η chr το χαρακτήρα, που αντιστοιχεί σε ASCII κώδικα.

Λίστες

- Αν `t` είναι ένας τύπος της Haskell, τότε `[t]` συμβολίζει τον τύπο των **λιστών**, που περιέχουν στοιχεία τύπου `t`.
- Παράδειγμα: `[Int]` είναι λίστες με ακέραιους αριθμούς

Τρόποι γραφής λίστας

- Απλούστερη λίστα είναι η κενή λίστα
  - `[]`
- Απαρίθμηση στοιχείων λίστας
  - `[1,3,10]`
- Χρήση τελεστή «:»
  - `[1,3,10]=1:[3,10]=1:(3:[10])=1:(3:(10:[]))`

Συναρτήσεις για λίστες

`length xs`	επιστρέφει το μήκος της λίστας xs
`xs ++ ys`	Επιστρέφει μία νέα λίστα, που αποτελείται από τα στοιχεία της xs ακολουθούμενα από τα στοιχεία της ys
`concat xss`	Επιστρέφει τη λίστα των στοιχείων, που υπάρχουν σε κάθε μία από τις λίστες στην xss

Παράδειγμα

Παράδειγμα

```
?  length [1,3,10]
3

?  [1,3,10]++[2,6,5,7]
[1,3,10,2,6,5,7]

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

Όλα τα στοιχεία μίας λίστας πρέπει να έχουν τον ίδιο τύπο (όλα ακέραιοι, ή όλα χαρακτήρες, κλπ.)

Συμβολοσειρές

- Συμβολίζονται με `String`
  - Συντομογραφία του τύπου `[Char]`
- Γράφονται σαν ακολουθίες, που περικλείονται από τον χαρακτήρα «"»
  - `"hello, world"`
- Είναι ουσιαστικά λίστες χαρακτήρων

Συναρτήσεις

Εφαρμογή συναρτήσεων και για γενικές λίστες

Παράδειγμα

```
?   length "Hello"
5

?   "Hello, " ++ "World"
Hello, World

?   concat ["mary","ann"]
maryann
```

N-άδες

- Αν `t1, t2, ..., tn` είναι τύποι, τότε ο τύπος της *n*-άδας συμβολίζεται με

```
(t1, t2, ..., tn)
```

N-άδες

- `(Int, [Int], Float)` αναπαριστά το σύνολο των **τριάδων** των οποίων το πρώτο στοιχείο είναι ακέραιος, το δεύτερο λίστα ακεραίων και το τρίτο αριθμός κινητής υποδιαστολής.
- `(Char, Bool)` αναπαριστά το σύνολο των **δυάδων** με πρώτο στοιχείο χαρακτήρα και δεύτερο λογική τιμή.

Συναρτήσεις

- Αν `t1` και `t2` είναι τύποι, τότε `t1 -> t2` είναι ο τύπος **συνάρτησης**, που
  - παίρνει ορίσματα τύπου `t1` και
  - επιστρέφει αποτελέσματα τύπου `t2`.

Συναρτήσεις

- Μία συνάρτηση `add` η οποία παίρνει σαν όρισμα της δυάδα αριθμών και επιστρέφει το άθροισμα τους, έχει τύπο: `(Int, Int) -> Int`
- Η συνάρτηση `ord` έχει τύπο: `Char -> Int`
- Η συνάρτηση `chr` έχει τύπο: `Int -> Char`

Συναρτήσεις στην Haskell

- Η Haskell υποστηρίζει ένα ευρύ φάσμα από συναρτήσεις συστήματος (πχ `product`, `abs`, `chr`, `ord`, κλπ)
- Στις περισσότερες περιπτώσεις, ο χρήστης ορίζει επιπλέον συναρτήσεις (πχ. για επίλυση ειδικού προβλήματος)

Γενική μορφή συναρτήσεων Haskell

- `f pat1 ... patn = E`
- Καθένα από τα `pat1 ... patn` αναπαριστά όρισμα συνάρτησης και ονομάζεται **πρότυπο** (pattern).
  - Μπορεί να είναι σταθερές τιμές, μεταβλητές ή με ακόμη πιο πολύπλοκη μορφή.
- E είναι **έκφραση** της γλώσσας.
- Μία συνάρτηση ορίζεται με περισσότερες από μία εξισώσεις.

Παράδειγμα 1

Η συνάρτηση `succ` ορίζεται ως:

Πρόγραμμα

```
succ n = n + 1
```

και έχει ως μοναδικό της όρισμα τη μεταβλητή `n`.

Παράδειγμα

Η συνάρτηση `not` ορίζεται με δύο εξισώσεις:

Πρόγραμμα

```
not True = False
not False = True
```

- Ορίσματα της και στις δύο περιπτώσεις είναι μία σταθερή λογική τιμή (`True` και `False`).
- Σταθερές (ακέραιοι, συμβολοσειρές, χαρακτήρες κλπ) χρησιμοποιούνται ως πρότυπα.

Παράδειγμα

Ορίζουμε τη συνάρτηση `hello` με δύο εξισώσεις:

Πρόγραμμα

```
hello "Panos" = "Hi"
hello name = "Hello, " ++ name
                ++ " nice to meet you!"
```

Στην πρώτη εξίσωση, όρισμα είναι σταθερά τύπου `String` και στη δεύτερη μεταβλητή.

Παράδειγμα

Η προηγούμενη συνάρτηση καλείται

Παράδειγμα

```
?  hello "Panos"
Hi

?  hello "John"
Hello John, nice to meet you!
```

Παράδειγμα

- Η σειρά γραφής εξισώσεων στη Haskell είναι πολύ σημαντική, διότι κατά την κλήση της συνάρτησης, το σύστημα χρησιμοποιεί τον **πρώτο ορισμό** που εφαρμόζεται.
- Αν αντιστραφεί η σειρά εντολών στο προηγούμενο πρόγραμμα, τότε έχουμε

Παράδειγμα

```
?  hello "Panos"
Hello Panos, nice to meet you!
```

Εξισώσεις Περιπτώσεων

Έχουν γενική μορφή:

Πρόγραμμα

```
f x1 ...  xn | condition1 = e1
             | condition2 = e2
             ...
             | conditionm = em
```

Η Haskell υπολογίζει τις τιμές των συνθηκών τη μία μετά την άλλη μέχρι να βρει την πρώτη αληθή, οπότε επιστρέφει την τιμή της αντίστοιχης έκφρασης.

Παράδειγμα

Η συνάρτηση `oddity :: Int -> String` εξετάζει αν το όρισμα της είναι άρτιος ή περιττός και επιστρέφει `"even"` και `"odd"` αντίστοιχα.

Πρόγραμμα

```
oddity n | even n     = "even"
         | otherwise = "odd"
```

Η συνθήκη `otherwise` είναι ισοδύναμη με τη λογική σταθερά `True`. Με τη χρήση της σε συνθήκη η αντίστοιχη έκφραση χρησιμοποιείται οπωσδήποτε αν καμία προηγούμενη συνθήκη δεν έχει ικανοποιηθεί.

Αναδρομή

- Βασική τεχνική στις συναρτησιακές γλώσσες.
- Χρήση για ορισμό συναρτήσεων, που επεξεργάζονται διάφορους τύπους δεδομένων

Πρόβλημα

- Υπολογισμός **παραγοντικού** του αριθμού n
- Χωρισμός προβλήματος σε δύο τμήματα:
  - Το n είναι μηδέν, οπότε το $n!$ είναι 1
  - Διαφορετικά
    $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n = (n-1)! \cdot n$
- Υπολογισμός του $n!$ αφού πρώτα υπολογισθεί το $(n-1)!$ και μετά πολλαπλασιασθεί με το n.

Γραφή σε Haskell

Πρόγραμμα

```
fact n = if (n == 0) then 1 else n*fact (n-1)
```

- Το παραπάνω πρόγραμμα μοιάζει να είναι κυκλικό, αλλά δεν είναι.
- Καθώς στο δεξιό μέλος της εξίσωσης η τιμή του `n` μειώνεται κατά 1, κάποια στιγμή θα γίνει ίση με μηδέν, οπότε δεν χρειάζεται να ξανακληθεί η συνάρτηση `fact`.

Υπολογισμός του fact 3

Παράδειγμα

```
fact 3 = 3 * fact 2
       = 3 * 2 * fact 1
       = 3 * 2 * 1 * fact 0
       = 3 * 2 * 1 * 1
       = 6
```

Χρήση Εξισώσεων Περιπτώσεων

Το ίδιο πρόγραμμα γράφεται με χρήση εξισώσεων περιπτώσεων:

Πρόγραμμα

```
fact n | (n==0)     = 1
       | otherwise = n * fact (n-1)
```

Η ίδια συνάρτηση με χρήση προτύπων γράφεται:

Πρόγραμμα

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

$(n+k)$ πρότυπα της Haskell

Το ίδιο πρόγραμμα γράφεται:

Πρόγραμμα

```
fact 0     = 1
fact (n+1) = (n+1)*fact n
```

- Το `n+1` συμβουλεύει τη Haskell ότι πρέπει να περιμένει ένα όρισμα μεγαλύτερο ή ίσο του ένα.
- Διαφέρει από τους προηγούμενους στο ότι αν ο χρήστης ζητήσει την τιμή του `fact(-1)` θα λάβει διαγνωστικό μήνυμα λάθους, ενώ όλοι οι άλλοι οδηγούν σε μη-τερματισμό.

Αναδρομικά Προγράμματα με Λίστες

Τεχνικές για τον ορισμό αναδρομικών προγραμμάτων με ακέραιους, χρησιμοποιούνται και σε ορισμό αναδρομικών συναρτήσεων σε **λίστες**.

Συνάρτηση length

- Η συνάρτηση υπολογίζει το μήκος μιας λίστας.
- Δύο περιπτώσεις
  - Αν η λίστα είναι κενή, τότε η συνάρτηση πρέπει να επιστρέφει την τιμή 0
  - Αν η λίστα δεν είναι κενή, τότε μπορεί να γραφεί στη μορφή `x:xs`, όπου `x` η κεφαλή της λίστας και `xs` η ουρά της. Τότε η αρχική λίστα είναι κατά ένα στοιχείο μεγαλύτερη από την `xs` και επομένως έχει μήκος `1+length(xs)`

Πρόγραμμα

Πρόγραμμα

```
length []     = 0
length (x:xs) = 1 + length xs
```

Υπολογισμός μήκους λίστας

Παράδειγμα

```
length [2,5,7] = 1 + length [5,7]
               = 1 + (1 + length [7])
               = 1 + (1 + (1 + length []) )
               = 1 + (1 + (1 + 0))
               = 3
```

Πρότυπο []

- Στα αναδρομικά προγράμματα με λίστες χρησιμοποιείται το πρότυπο `[]`, που αφορά την κενή λίστα και το `x:xs`, που αφορά τις μη-κενές λίστες.
- Με το πρότυπο αυτό ορίζονται δύο βασικές συναρτήσεις, που επιστρέφουν την κεφαλή και την ουρά μιας λίστας αντίστοιχα:

Πρόγραμμα

```
head (x:xs) = x
tail (x:xs) = xs
```

Παράδειγμα

Η συνάρτηση `sum`, που αθροίζει όλα τα στοιχεία λίστας, ορίζεται ως:

Πρόγραμμα

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

Ανάλογα ορίζεται η συνάρτηση `product`, που επιστρέφει το γινόμενο των στοιχείων λίστας.

Παράδειγμα

Η συνάρτηση append, που ενώνει δύο λίστες ορίζεται ως:

Πρόγραμμα

```
append []     ys = ys
append (x:xs) ys = x:(append xs ys)
```

Παράδειγμα:
`append [1,4,1] [1,5] = [1,4,1,1,5]`.

Παράδειγμα

Με χρήση της append ορίζεται η συνάρτηση reverse, που αντιστρέφει μία λίστα:

Πρόγραμμα

```
reverse []     = []
reverse (x:xs) = append (reverse xs) [x]
```

Με χρήση της reverse ορίζεται η συνάρτηση last, που επιστρέφει το τελευταίο στοιχείο μίας λίστας:

Πρόγραμμα

```
last xs = head (reverse xs)
```

Παράδειγμα:

```
reverse [1,2,3,4] = [4,3,2,1]
last [1,3,5,7] = 7
```

Παράδειγμα

Η συνάρτηση take επιστρέφει μία λίστα με τα n πρώτα στοιχεία δεδομένης λίστας (υπόθεση: η δεδομένη λίστα θα έχει πάντα τουλάχιστον n στοιχεία):

Πρόγραμμα

```
take 0 xs           = []
take (n+1) (x:xs) = x:(take n xs)
```

Παράδειγμα:

```
take 3 [1,3,4,7,8] = [1,3,4]
```

Παράδειγμα

Η συνάρτηση drop «πετάει» τα πρώτα n στοιχεία δεδομένης λίστας και επιστρέφει λίστα με τα υπόλοιπα στοιχεία (υπόθεση: η δεδομένη λίστα θα έχει πάντα τουλάχιστον n στοιχεία):

Πρόγραμμα

```
drop 0 xs           = xs
drop (n+1) (x:xs) = drop n xs
```

Παράδειγμα:

```
drop 3 [1,3,4,7,8] = [7,8]
```

Παράδειγμα

Η συνάρτηση `select` διαλέγει το n-οστό στοιχείο δεδομένης λίστας:

Πρόγραμμα

```
select 1     (x:xs) = x
select (n+1) (x:xs) = select n xs
```

Παράδειγμα:

```
select 3 [1,6,2,6,7] = 2
```

Παράδειγμα

Η συνάρτηση `member` λαμβάνει ως ορίσματα ένα στοιχείο και μία λίστα και επιστρέφει `True` ή `False` ανάλογα με το αν το στοιχείο ανήκει στη λίστα ή όχι.

Πρόγραμμα

```
member x []      = False
member x (y:xs) = if (x==y) then True
                  else (member x xs)
```

Παράδειγμα

Η συνάρτηση delete διαγράφει όλες τις εμφανίσεις ενός στοιχείου από μία λίστα:

Πρόγραμμα

```
delete x []     = []
delete x (y:xs) = if (x==y) then (delete x xs)
                  else (y:delete x xs)
```

Παράδειγμα:
delete 3 [1,3,5,3,7] = [1,5,7]

Παράδειγμα

Συναρτήσεις, που συγκρίνουν δύο λίστες και ελέγχουν αν ισχύουν δεδομένες σχέσεις μεταξύ τους.

Παράδειγμα

Η συνάρτηση `prefix` ελέγχει αν μία λίστα είναι πρόθεμα (prefix) άλλης:

Πρόγραμμα

```
prefix [] ys = True
prefix (x:xs) (y:ys) =
    if (x==y) then (prefix xs ys)
    else False
```

Στην περίπτωση για επίθεμα (suffix)

Πρόγραμμα

```
suffix xs ys = prefix (reverse xs) (reverse ys)
```

Παράδειγμα

Η συνάρτηση `less` λαμβάνει δύο ορίσματα, έναν αριθμό και μία λίστα από αριθμούς, και επιστρέφει τους αριθμούς της λίστας, που είναι μικρότεροι από το δεδομένο αριθμό:

Παράδειγμα

```
?  less 3 [1,2,1,3,4,5]
[1,2,1]
```

Παράδειγμα

Ο ορισμός της less είναι:

Πρόγραμμα

```
less x []     = []
less x (y:ys) = if (y<x) then (y:less x ys)
                else (less x ys)
```

Παράδειγμα

Η συνάρτηση greateq λαμβάνει δύο ορίσματα, έναν αριθμό και μία λίστα από αριθμούς και επιστρέφει τους αριθμούς της λίστας, που είναι μεγαλύτεροι ή ίσοι από το δεδομένο αριθμό.

Παράδειγμα

```
?  greateq 3 [1,2,1,3,3,4,5,3]
[3,3,4,5,3]
```

Παράδειγμα

- Δημιουργία συνάρτησης, που **ταξινομεί** λίστες αριθμών
- Βασική ιδέα:
  - Παίρνουμε την κεφαλή x της λίστας και βρίσκουμε όλα τα στοιχεία της υπόλοιπης λίστας, που είναι μικρότερα από την κεφαλή και μετά όλα όσα είναι μεγαλύτερα ή ίσα.
  - Τις δύο αυτές υπολίστες τις ταξινομούμε αναδρομικά και μετά τις ενώνουμε τοποθετώντας το x στη μέση.

Παράδειγμα

Αυτό γράφεται ως:

Πρόγραμμα

```
sort [] = []
sort (x:xs) = (sort (less x xs)) ++
              [x] ++
              (sort (greateq x xs))
```

Παραδείγματα

Η συνάρτηση anyorder ελέγχει αν δύο στοιχεία εμφανίζονται διαδοχικά (με οποιαδήποτε σειρά) σε μία λίστα. Ορίζουμε πρώτα τη συνάρτηση adjacent:

Πρόγραμμα

```
adjacent (x,y,[]) = False
adjacent (x,y,[z]) = False
adjacent (x,y,(a:b:zs))) =
    if (x==a) && (y==b) then True
    else adjacent (x,y, (b:zs))
```

Πρόγραμμα

```
anyorder(x,y,zs)=
    adjacent(x,y,zs) || adjacent(y,x,zs)
```

Μέθοδος κλήσης συναρτήσεων στη Haskell

- Σε πολλές γλώσσες προγραμματισμού, οι τιμές των πραγματικών παραμέτρων μιας συνάρτησης υπολογίζονται πριν περάσουν στη συνάρτηση (περνάνε με **τιμή** - call by value)
    - Πλεονέκτημα: Υλοποίηση με ιδιαίτερα εύκολο τρόπο
    - Μειονέκτημα: Συχνά, πραγματοποίηση υπολογισμών που δεν χρειάζονται (πχ. για όρισμα που δεν χρησιμοποιείται στο σώμα της συνάρτησης)

Μέθοδος κλήσης συναρτήσεων στη Haskell

- Κλήση με ζήτηση (call by need)
  - Όλες οι παράμετροι περνάνε στο σώμα της συνάρτησης χωρίς να έχει υπολογισθεί αρχικά η τιμή τους. Υπολογίζεται μόνο αν κάτι τέτοιο χρειάζεται πραγματικά.
  - Η Haskell χρησιμοποιεί την κλήση με **ζήτηση**.

Παράδειγμα

Έστω η συνάρτηση:

Πρόγραμμα

```
ignore x = "I don't need to evaluate my argument"
```

Αν κληθεί η συνάρτηση:

Παράδειγμα

```
?  ignore(1/0)
I don't need to evaluate my argument
```

H Haskell θα έπρεπε να δώσει διαγνωστικό λάθος λόγω διαίρεσης με το μηδέν. Δεν δίνει όμως, γιατί υπολογίζει την τιμή του ορίσματος συνάρτησης μόνο όταν το όρισμα χρειάζεται πραγματικά για τον υπολογισμό της τιμής της συνάρτησης.

Παράδειγμα

Έστω η συνάρτηση:

Πρόγραμμα

```
f x y = if (x<=1) then x+1 else y
```

- Η συνάρτηση καλείται ως: `f 1 (fact 30)`
- Αν το πέρασμα παραμέτρων γίνεται με τιμή, τότε πριν την έναρξη υπολογισμού της τιμής της συνάρτησης, υπολογίζονται οι τιμές των πραγματικών παραμέτρων.

Παράδειγμα

- Στην κλήση με τιμή αρχικά υπολογίζεται η τιμή του `fact 30` (δεν θα χρειαστεί στο σώμα της συνάρτησης, αφού το `x` έχει τιμή 1)
- Αν η κλήση γίνει με ζήτηση, η τιμή του `fact 30` δεν υπολογίζεται, γιατί δεν απαιτείται για τον υπολογισμό του αποτελέσματος.

Πολυμορφισμός στη Haskell

- Πολυμορφική συνάρτηση: συνάρτηση, που ο τύπος της περιέχει μία ή περισσότερες μεταβλητές τύπου.
- Αποφυγή ορισμού διαφορετικών συναρτήσεων για διαφορετικούς τύπους δεδομένων

Πολυμορφισμός στη Haskell

`sum`

- Παίρνει σαν παράμετρο μία λίστα από αριθμούς και επιστρέφει έναν αριθμό, που είναι το άθροισμα όλων των στοιχείων της λίστας
- Εφαρμόζεται μόνο σε λίστες με αριθμητικά δεδομένα

`length`

- Υπολογίζει το μήκος μιας λίστας ανεξάρτητα από το είδος των στοιχείων της λίστας
- Καλείται με παράμετρο μία λίστα ακεραίων ή με μία λίστα χαρακτήρων κ.ο.κ.

Παράδειγμα

Παράδειγμα

```
?   length([1,1,3,4,7,17])
6

?   length("Hello")
5
```

Ο τύπος της `length` είναι `[a]->Int`, όπου με `a` παριστάνεται οποιοσδήποτε τύπος. Το `a` λέγεται μεταβλητή τύπου.

Άπειρες δομές δεδομένων

- Πλεονέκτημα της κλήσης με ζήτηση η δυνατότητα επεξεργασίας **άπειρων** δομών δεδομένων
- Αδύνατη η κατασκευή ή αποθήκευση ολόκληρων τέτοιων δομών
- Δυνατή η κατασκευή όλο και μεγαλύτερων τμημάτων με χρήση του χώρου, που καταλάμβαναν προηγούμενα τμήματα, που δεν χρειάζονται πλέον στον υπολογισμό.

Παράδειγμα

Πρόγραμμα υπολογισμού λίστας φυσικών αριθμών

Πρόγραμμα

```
nats n = n:(nats (n+1))
```

- Εάν ζητηθεί η τιμή του `nats 0`, η Haskell τυπώνει τη λίστα των φυσικών αριθμών
- Όταν στοιχείο της λίστας τυπωθεί, ο χώρος μνήμης, που καταλάμβανε, μπορεί να επαναχρησιμοποιηθεί.

Παράδειγμα

Ορισμός έκφρασης υπολογισμού αθροίσματος των 10 πρώτων φυσικών αριθμών:

Παράδειγμα

```
?  sum (take 10 (nats 0))
```

Ο υπολογισμός της έκφρασης αυτής είναι δυνατός, γιατί η `take` δεν επιχειρεί τον υπολογισμό της τιμής του `nats 0` πριν την έναρξη της επεξεργασίας.

Συναρτήσεις υψηλής τάξης

- Τα ορίσματα των συναρτήσεων ήταν, μέχρι τώρα, απλοί τύποι δεδομένων (πχ. ακέραιοι, λίστες κλπ).
- Βασικό χαρακτηριστικό του συναρτησιακού προγραμματισμού είναι το «πέρασμα» συναρτήσεων ως ορίσματα σε άλλες συναρτήσεις.

Πλεονέκτημα Συναρτήσεων Υψηλής τάξης

- Κλήση συναρτήσεων υψηλής τάξης με διαφορετικές συναρτήσεις σαν ορίσματα.
- Πλεονέκτημα: Αποφυγή παρόμοιου κώδικα για παρόμοιες λειτουργίες

Συναρτήσεις υψηλής τάξης

Οι συναρτήσεις δημιουργούν μία ιεραρχία ανάλογα με τον τύπο τους

- Συναρτήσεις **μηδενικής τάξης**: (σταθερές)
  - `a = 5` (ισοδύναμο με `a() = 5`)
- Συναρτήσεις **πρώτης τάξης**: έχουν ως ορίσματα συνηθισμένους τύπους δεδομένων (π.χ. ακέραιους, float, λίστες)
  - `sq(a) = a*a, suml(xs), reverse(xs)`

Συναρτήσεις υψηλής τάξης

- Συναρτήσεις **δεύτερης τάξης**: έχουν μία τουλάχιστον παράμετρο, που να είναι συνάρτηση πρώτης τάξης και όλες τις υπόλοιπες παραμέτρους $\leq$ της πρώτης τάξης

Παράδειγμα

```
result = twice(sq, 2)
twice(f,y) = f(f y)
```

Είναι δεύτερης τάξης διότι το πρώτο όρισμα είναι μία συνάρτηση πρώτης τάξης, ενώ το δεύτερο είναι μηδενικής τάξης.

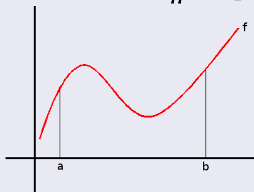
Συναρτήσεις

- Συναρτήσεις **τρίτης τάξης**: έχουν μία τουλάχιστον παράμετρο, που να είναι συνάρτηση δεύτερης τάξης και όλες οι υπόλοιπες $\leq$ της δεύτερης τάξης

Παράδειγμα

Ορισμός συνάρτησης `root f a b` *για τον υπολογισμό των* **ριζών πραγματικών συναρτήσεων** *στο διάστημα* `[a,b]`.

```
integrate(simpson,f,a,b)
...
simpson(f,a,b)
...
```



- Η `integrate` παίρνει σαν πρώτη παράμετρο μία μέθοδο ολοκλήρωσης.

Συναρτήσεις

- Γενικά μπορούν να ορισθούν συναρτήσεις οποιασδήποτε τάξης, ακόμη και συναρτήσεις άπειρης τάξης:
  `self(g) = g(g)` (όχι στη Haskell)

Παράδειγμα

Η συνάρτηση `map` παίρνει ως πρώτο της όρισμα μία συνάρτηση `f` την οποία εφαρμόζει σε κάθε στοιχείο του δεύτερου της όρισματος (που είναι μία λίστα)

Πρόγραμμα

```
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Παράδειγμα

Αν ορισθεί στο πρόγραμμα η συνάρτηση

Πρόγραμμα

```
sq a = a*a
```

Τότε,

Παράδειγμα

```
?  map sq [1,3,5]
[1,9,25]
```

H `map` εφάρμοσε την `sq` σε κάθε στοιχείο της λίστας `[1,3,5]`

Παράδειγμα

Η τιμή της έκφρασης

Παράδειγμα

```
?  map sq (nats 0)
```

είναι μία λίστα με τα τετράγωνα των φυσικών αριθμών

Παράδειγμα

Η ακόλουθη συνάρτηση ορίζει ένα «συναρτησιακό παραγοντικό»:

Πρόγραμμα

```
ffact f n = if (n<1) then 1
            else (f n)*(ffact f (n-1))
```

- Αν η `ffact` κληθεί ως `ffact sq 5`, με `sq a = a*a`, θα υπολογίσει την τιμή: `(sq 5)*(sq 4)*...*(sq 1)`.
- Αν η `ffact` κληθεί ως `ffact cb 3`, με `cb b = b*b*b`, θα υπολογίσει την τιμή: `(cb 3)*(cb 2)*(cb 1)`.

Άλλα Χαρακτηριστικά της Haskell

λ-αφαιρέσεις (ή λ-abstractions)

- Η συνάρτηση `inc x = x+1` μπορεί να γραφεί και ως: `inc = \x -> x+1`
- Η συνάρτηση `add x y = x+y` γράφεται και: `add = \x y -> x+y`
- Το πλεονέκτημα αυτού του τρόπου γραφής είναι ότι μπορούμε να έχουμε «ανώνυμες» συναρτήσεις

Παράδειγμα

Παράδειγμα

```
result = (\x->x+1) 5
result = root (\x->x*x-2) 5
```

Άλλα Χαρακτηριστικά της Haskell

Λίστες με Συνθήκες (List Comprehensions)

- Λίστες, που χρησιμοποιούν το βασικό συμβολισμό για τον ορισμό συνόλων
- Παράδειγμα: `[n*n | n <- [1..100]]`
  Είναι η λίστα, που περιέχει όλα τα τετράγωνα των αριθμών από το 1 μέχρι το 100.

Λίστες με συνθήκες

- Γενική μορφή: `[body | qualifiers]`
- Τα qualifiers μπορεί να είναι:
  - Boolean Expressions
  - `var<-exp`
- Αν είναι περισσότερα από ένα, χωρίζονται με κόμματα.

Παραδείγματα

Το καρτεσιανό γινόμενο δύο λιστών xs, ys

Παράδειγμα

```
[(x,y)| x<-xs, y<-ys]
```

Παράδειγμα: `[(x,y)| x<-[1..4], y<-[2,4..10]]`

Παραδείγματα

Πρόγραμμα

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

Πρόγραμμα

```
perms [] = [[]]
perms xs = [a:ys| a<-xs, ys<-perms(delete a xs)]
```

Παράδειγμα:

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

Παραδείγματα

Πρόγραμμα

```
factors n = [i | i<-[1..n], n 'mod' i == 0]
```

Πρόγραμμα

```
pythagorian n = [(a,b,c) |
        a<-[1..n], b<-[1..n], c<-[1..n],
        a*a + b*b == c*c]
```

Ακολουθία Fibonacci με άπειρες τιμές

Πρόγραμμα

```
fib = 1 :  1 :  [a+b |(a,b)<-zip fib (tail fib)]
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

Πως δουλεύει το παραπάνω πρόγραμμα

Παράδειγμα

```
fib = 1:1:[a+b|(a,b)<-zip (1:1...)  (1:...)]
  = 1:1:[a+b|(a,b)<-(1,1):zip (1:...)  (...)]
  = 1:1:2:[a+b|(a,b)<-zip (1:2...)  (2:...)]
  = 1:1:2:3:[a+b|(a,b)<-zip (2:3...)  (3:...)]
```

Η ακολουθία `fib` παράγεται με μία διαδικασία, που ονομάζεται «chasing the tail».

Τύποι οριζόμενοι από το χρήστη

Η Haskell επιτρέπει στο χρήστη να ορίσει νέους, πιο ισχυρούς τύπους.

Παράδειγμα

Η Haskell δεν υποστηρίζει τον τύπο δεδομένων δυαδικό δέντρο, το οποίο είναι:

- Κενό δέντρο, ή
- Κόμβος με δύο δυαδικά δέντρα σαν παιδιά

Παράδειγμα

Ορισμός του νέου τύπου δεδομένων στη Haskell:

Ορισμός

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Ο ορισμός αφορά δέντρα, που περιέχουν στοιχεία τύπου a.

Παράδειγμα

Προγράμματα, που επεξεργάζονται τον νέο τύπο δεδομένων:

Πρόγραμμα

```
elements Empty = []
elements (Node v l r) =
    (elements l) ++ [v] ++ (elements r)
```

Η παραπάνω συνάρτηση συγκεντρώνει όλα τα στοιχεία ενός δέντρου σε λίστα.

Παράδειγμα

Η ερώτηση:

Παράδειγμα

```
?  elements (Node 5
    (Node 7 Empty Empty)
    (Node 8 Empty Empty))
```

Θα επιστρέψει την απάντηση: `[7,5,8]`

Ο τύπος της συνάρτησης `elements` είναι `Tree a->[a]`

Ορισμός

Ορισμός

```
data Datatype a1...aN = constr1 | ...  | constrM
```

Όπου:

- `Datatype` το όνομα του νέου τύπου δεδομένων στην Haskell
- `a1,...,aN` διαφορετικές μεταβλητές τύπου, που αντιπροσωπεύουν ορίσματα του τύπου αυτού και
- `constr1,...,constrM` δείχνουν πως κατασκευάστηκε ο νέος τύπος. Καθένα έχει μορφή `Name b1...bR` (`b1,...,bR` μεταβλητές από τις `a1,...,aN`)

Παράδειγμα

Τύπος δεδομένων Day, που περιγράφει τις ημέρες της εβδομάδας

Ορισμός

```
data Day = Sun|Mon|Tue|Wed|Thu|Fri|Sat
```

Τώρα μπορούν να γραφούν συναρτήσεις, που τον χρησιμοποιούν:

Πρόγραμμα

```
schedule Sun = "relax"
schedule Sat = "go shopping"
schedule x   = "back to work"
```

Ένα μεγαλύτερο παράδειγμα

treesort: Παίρνει σαν input μία μη διατεταγμένη λίστα, τη μετατρέπει σε δυαδικό δέντρο και μετά μετατρέπει το δέντρο σε λίστα

Πρόγραμμα

```
treesort xs = flatten (maketree xs)

maketree []     = Empty
maketree (x:xs) = insert x (maketree xs)

insert x Empty = Node x Empty Empty
insert x (Node a l r) =
    if (x<=a) then (Node a (insert x l) r)
    else (Node a l (insert x r )
```

Συναρτήσεις

Η `flatten` είναι σαν την `elements` που ορίσαμε πιο πριν

Πρόγραμμα

```
flatten Empty = []
flatten (Node v l r) = (flatten l)
                       ++ [v]
                       ++ (flatten r)
```

Εξαγωγή τύπων

- Αν μία γλώσσα υποστηρίζει αυτόματη εξαγωγή και έλεγχο τύπων, τότε ο προγραμματιστής διευκολύνεται και πολλά από τα λάθη εντοπίζονται «at compile-time»
    - Ένας τύπος, που περιέχει μία τουλάχιστον μεταβλητή τύπου ονομάζεται **πολυμορφικός**
    - Ένας τύπος, που δεν περιέχει μεταβλητές τύπου ονομάζεται **μονομορφικός**

Εξαγωγή τύπων

- Όταν προσπαθούμε να εξάγουμε τύπους μιας συνάρτησης ενός προγράμματος ψάχνουμε στην ουσία να βρούμε τον **πιο γενικό τύπο** της συνάρτησης (με την έννοια ότι κάθε μονομορφικός τύπος, που μπορεί να πάρει η συνάρτηση αυτή είναι ειδική περίπτωση του πιο γενικού αυτού τύπου)

Παράδειγμα

Δίνεται η συνάρτηση:

Παράδειγμα

```
map(f,m) = if null(m) then []
        else f(head(m)):map(f, tail(m))
```

Τύποι για τις βασικές συναρτήσεις για list-processing:

$$
\begin{aligned}
\sigma_{null} &= [\tau_1] \rightarrow Bool \\
\sigma_{[\,]} &= [\tau_2] \\
\sigma_{head} &= [\tau_3] \rightarrow \tau_3 \\
\sigma_{tail} &= [\tau_4] \rightarrow [t_4] \\
\sigma_{:} &= (\tau_5, [t_5]) \rightarrow [t_5]
\end{aligned}
$$

Μπορούμε τώρα να γράψουμε εξισώσεις, που εξασφαλίζουν συνέπεια τύπων στον ορισμό της παραπάνω συνάρτησης

Συνέχεια

- Γνωρίζουμε τώρα ότι οι δύο κλάδοι ενός `if` πρέπει να έχουν τον ίδιο τύπο

$$\sigma_{[\,]} = [\tau_2] = [t_5] \Rightarrow t_2 = t_5$$

- Η `null` και η `head` έχουν την ίδια παράμετρο. Το ίδιο και η `tail`.

$$[t_1] = [t_3] \Rightarrow t_1 = t_3 \text{ και } [t_4] = [t_3] \Rightarrow t_1 = t_3 = t_4$$

$$\text{Και συνεπώς, } \sigma_m = [t_1]$$

Συνέχεια

- Η παράμετρος της `f` έχει τον ίδιο τύπο με αυτόν που επιστρέφει η `head`. Το αποτέλεσμα της `f` έχει τον ίδιο τύπο με το πρώτο όρισμα της «`:`». Επομένως:

$$\sigma_f = \tau_3 \rightarrow \tau_4 \Rightarrow \text{(από τα παραπάνω) } \sigma_f = t_1 \rightarrow t_2$$

- Τέλος, ο τύπος της `map` είναι:

$$\sigma_{map} = ((t_1 \rightarrow t_2, [t_1]) \rightarrow [t_2]$$

Αλγόριθμος Hindley-Milner

Η εξαγωγή τύπων για γλώσσες, που υποστηρίζουν τον λεγόμενο παραμετρικό πολυμορφισμό γίνεται με τον **αλγόριθμο των Hindley-Milner**, ο οποίος στην ουσία:

- λύνει ένα σύνολο από εξισώσεις της παραπάνω μορφής
- και επιστρέφει τον πιο γενικό τύπο για κάθε συνάρτηση του προγράμματος.

Κλάσεις τύπων και υπερφόρτωση

- Ο πολυμορφισμός για τον οποίο έχουμε ήδη μιλήσει ονομάζεται συνήθως **παραμετρικός πολυμορφισμός** (parametric polymorphism).
- Υπάρχει και ένα άλλο είδος πολυμορφισμού, που ονομάζεται **πολυμορφισμός κατά περίπτωση** (ad hoc polymorphism) ή **υπερφόρτωση** (overloading)

Παράδειγμα

- Οι αριθμοί `1`,`2` κλπ χρησιμοποιούνται για να αναπαραστήσουν κανονικούς ακέραιους (`Int`), αλλά και ακεραίους οποιασδήποτε ακρίβειας (`integer`).
- Αριθμητικοί τελεστές, όπως το + δουλεύουν σε διαφορετικά είδη αριθμών
- Ο τελεστής ισότητας «==» δουλεύει για αριθμούς, αλλά και για άλλους τύπους της Haskell (όχι όμως για όλους τους τύπους)

Παράδειγμα

- Είναι σημαντικό να τονίσουμε ότι στην περίπτωση της υπερφόρτωσης ένας τελεστής μπορεί να εφαρμόζεται σε διαφορετικούς τύπους δεδομένων και η συμπεριφορά του να είναι διαφορετική για κάθε τύπο.
- Αυτό έρχεται σε αντίθεση με την έννοια του παραμετρικού πολυμορφισμού στον οποίο ο τύπος των στοιχείων δεν παίζει ρόλο (πχ `length`)

Τελεστής ισότητας

- Η συμπεριφορά του τελεστή αυτού δεν μπορεί να υλοποιηθεί αποτελεσματικά για όλους τους τύπους (πχ η ισότητα δύο συναρτήσεων) δεν μπορεί να εξακριβωθεί στη γενική περίπτωση).
- Το ακόλουθο παράδειγμα δείχνει την αναγκαιότητα της υπερφόρτωσης

Παράδειγμα

Έστω η συνάρτηση

Πρόγραμμα

```
member x []      = False
member x (y:ys) = if (x==y) then True
                  else (member x ys)
```

Αν ο τύπος της `member` είναι $a \to [a] \to Bool$ τότε αυτό θα σήμαινε ότι ο τελεστής ισότητας έχει τύπο $a \to a \to Bool$ (που δεν ισχύει γιατί το «==» δεν είναι ορισμένο για όλους τους τύπους).

Παράδειγμα

- Ο τύπος της `member` δεν είναι ο παραπάνω, αλλά κάποιος, που αντανακλά το γεγονός ότι η ισότητα είναι ορισμένη μόνο για μερικούς τύπους.
- Είναι επιθυμητό, όταν κληθεί η `member` με πρώτη παράμετρο της οποίας ο τύπος δεν επιτρέπει έλεγχο ισότητας, ο compiler της Haskell να μας ενημερώνει ότι υπάρχει πρόβλημα.

Κλάσεις τύπων

- Τα παραπάνω θέματα αντιμετωπίζονται με τις **κλάσεις τύπων** (type classes).
  - Επιτρέπουν τη δήλωση τύπων ως **στιγμιότυπα** (instances) μιας κλάσης και ορίζουν υπερφορτωμένες λειτουργίες, που σχετίζονται με την κλάση.

Παράδειγμα

Ορισμός κλάσης τύπων με τελεστή ισότητας

Ορισμός

```
class Eq a where
(==) ::  a -> a -> Bool
```

όπου Eq: όνομα της ορισμένης κλάσης
και «==»: τελεστής, που υποστηρίζεται από την κλάση.

Η παραπάνω δήλωση διαβάζεται ως: «Ένας τύπος a είναι στιγμιότυπο της Eq αν υπάρχει ένας (υπερφορτωμένος) τελεστής «==» ο οποίος να ορίζεται για τον τύπο αυτόν».

Παράδειγμα

- Όταν ένας τύπος a ανήκει στην κλάση Eq θα γράφουμε Eq a (το οποίο διαισθητικά σημαίνει ότι για τον τύπο αυτό είναι ορισμένος ένας τελεστής ισότητας). Επομένως ο τύπος της member είναι:

```
member ::  Eq a => a -> [a] -> Bool
```

- Το παραπάνω διαβάζεται: «Για κάθε τύπο a, που ανήκει στην κλάση Eq η member έχει τύπο a -> [a] -> Bool».

Δηλώσεις στιγμιότυπων

```
instance Eq Integer where
x == y = x 'integerEq' y
```

Ο ορισμός του == ονομάζεται **μέθοδος**. Η συνάρτηση integerEq είναι συνάρτηση της Haskell, για να συγκρίνει δύο ακεραίους για ισότητα. Ομοίως:

```
instance (Eq a) => Eq (Tree a ) where
Leaf a == Leaf b = a == b
(Branch l1 r1)==(Branch l2 r2)=(l1==l2)&&(r1==r2)
_ == _ = False
```

Δηλώσεις στιγμιότυπων

- Για σύγκριση δέντρων θα πρέπει τα στοιχεία, που βρίσκονται στους κόμβους του δέντρου να μπορούν να συγκριθούν μεταξύ τους (να ανήκουν στην κλάση `Eq`)
- Αυτό είναι και το νόημα της δήλωσης στην πρώτη γραμμή της παραπάνω μεθόδου `Eq a => Eq (Tree a)`
- Οι χαρακτήρες «`-`» στην τελευταία γραμμή σημαίνουν «οτιδήποτε άλλο» (και οδηγούν στην τιμή `False`).

Παράδειγμα

- Οι κλάσεις μπορούν να επεκταθούν.
- Μπορεί να ορισθεί η κλάση `Ord` η οποία κληρονομεί τους τελεστές της Eq (δηλαδή τον τελεστή == στο παράδειγμα μας) και η οποία κλάση χρησιμοποιεί κάποιους επιπλέον τελεστές:

Ορισμός

```
class (Eq a) (ord a) where
(<), (<=), (>=), (>) ::  a -> a -> Bool
max, min ::  a -> a -> Bool
```

Παράδειγμα

- Θα λέμε ότι η `Eq` είναι υπερκλάση της `Ord` (ή αντίστοιχα ότι η `Ord` είναι υπερκλάση της `Eq`).
- Θα πρέπει να ορισθούν οι μέθοδοι της υποκλάσης, αλλά μπορούν να χρησιμοποιηθούν ελεύθερα οι μέθοδοι της υπερκλάσης
- Παράδειγμα: Κοίτα το standard prelude για τον ορισμό του «>»
- Παράδειγμα: Πες ότι η `sort` έχει τώρα τύπο:
  `sort :: (Ord a) => [a] -> [a]`

Συναρτησιακά προγράμματα και επαγωγικές αποδείξεις

- Συχνά χρησιμοποιούμενη αποδεικτική διαδικασία για συναρτησιακά προγράμματα είναι η «**δομική επαγωγή**»
- Βασική ιδέα
  - Αποδεικνύουμε κάτι για τις πολύ απλές περιπτώσεις ενός σύνθετου τύπου (παράδειγμα κενή list, empty tree,κλπ)
  - Υποθέτουμε ότι το ζητούμενο ισχύει για αντικείμενα δεδομένης πολυπλοκότητας και αποδεικνύουμε ότι ισχύει και για μεγαλύτερη πολυπλοκότητα

Παράδειγμα

Πρόγραμμα

```
append [] ys = ys
append x:xs ys = x:(append xs ys)
```

- Θέλουμε να δείξουμε ότι:

```
append (append xs ys) zs =
    = append xs (append ys zs)
```

Θεώρημα

Απόδειξη:

- Η περίπτωση βάσης είναι για `xs = []`

```
append (append [] ys) zs = append ys zs
append [] (append ys zs) = append ys zs
```

- Υποθέτουμε ότι:

```
append (append xs ys) =
     append xs (append ys zs)
```

- Θα δείξουμε ότι:

```
append (append (x:xs) ys) zs =
     = append (x:(append xs ys)) zs =
     = x:(append (append xs ys) zs) =
     = x:(append xs(append ys zs)) =
     = append (x:xs)(append ys zs)
```

Άσκηση

Δίνεται ο ακόλουθος ορισμός της συνάρτησης add:

Πρόγραμμα

```
add n 0   = n
add n m+1 = 1 + add n m
```

Να δειχθεί ότι η πρόσθεση όπως ορίζεται παραπάνω είναι προσεταιριστική και αντιμεταθετική