Εισαγωγή στη γλώσσα Haskell

Γιάννης Κασσιός

Σε αυτές τις σημειώσεις ξεκινάμε την παρουσίαση της συναρτησιακής γλώσσας Haskell. Οι σημειώσεις χωρίζονται σε τρείς ενότητες: τα βασικά χρήσης ενός διερμηνέα Haskell, θέματα σχετικά με τον ορισμό συναρτήσεων και παραδείγματα.

1 Βασική Χρήση της Haskell

1.1 Πρόσβαση στη Haskell

Για να χρησιμοποιήσουμε τη Haskell χρειαζόμαστε ένα διερμηνέα ή ένα μεταγλωττιστή της Haskell. Στο μάθημα αυτό θα χρησιμοποιούμε το διερμηνέα Hugs98. Μπορείτε να κατεβάσετε το Hugs98 για τα βασικότερα λειτουργικά συστήματα από την ιστοσελίδα
`http://haskell.org/hugs/`
Στους υπολογιστές με Linux στη σχολή, ο Hugs98 είναι διαθέσιμος ως:
`/home/appl/hugs98/bin/hugs`

Εναλλακτικά, μπορεί να χρησιμοποιηθεί και ο Glaskow Haskell Compiler (GHC) ο οποίος λειτουργεί και ως διερμηνέας και ως μεταγλωττιστής. Ο GHC είναι διαθέσιμος στη σελίδα:
`http://haskell.org/ghc/`

Ο διερμηνέας εμφανίζει μία κονσόλα. Στην κονσόλα αυτή, ο χρήστης μπορεί να γράψει εκφράσεις Haskell προς αποτίμηση ή εντολές που κατευθύνονται στο διερμηνέα.

Οι εντολές διερμηνέα αρχίζουν με τον χαρακτήρα `:`. Οι πιο σημαντικές είναι οι:

`:load`	φόρτωση προγράμματος Haskell (βλ. Ενότ. 1.4)
`:cd`	αλλαγή τρέχοντος καταλόγου στο λειτουργικό σύστημα
`:reload`	επαναφόρτωση ενός προγράμματος

Οι εκφράσεις που γράφουμε στο διερμηνέα αποτιμώνται και ο διερμηνέας εμφανίζει το αποτέλεσμα της αποτίμησης.

1.2 Αποτίμηση Εκφράσεων

Μία έκφραση της Haskell είναι ένα από τα παρακάτω:

- Μία *σταθερά*. Βλ. Ενοτ. 1.3 για μερικές σταθερές που υποστηρίζονται
- Ένα *αναγνωριστικό*. Τα αναγνωριστικά που συμβολίζουν τιμές και άρα επιτρέπεται να μπουν στις εκφράσεις ξεκινάνε με *πεζό* λατινικό γράμμα και περιέχουν λατινικά γράμματα και αριθμούς. Στη Haskell και οι συναρτήσεις είναι

τιμές, επομένως έχουν ονόματα που ξεκινάνε με πεζό γράμμα. Μία λέξη-κλειδί της Haskell δε μπορεί να χρησιμοποιηθεί σαν αναγνωριστικό

- Ένας *τελεστής* εφαρμοζόμενος σε κάποιες εκφράσεις, πχ. `2+3`. Ένας τελεστής με δύο ορίσματα λέγεται *δυαδικός*. Ένας τελεστής που παίρνει μόνο ένα όρισμα λέγεται *μοναδιαίος*. Βλ. και Ενοτ. 2.4

- Μία *συνάρτηση* εφαρμοζόμενη σε έναν αριθμό εκφράσεων. Η εφαρμογή γίνεται με απλή αντιπαράθεση, π.χ. `f x y`. Παράδειγμα εφαρμογής συνάρτησης είναι: `sqrt 4`

- Μία έκφραση μέσα σε παρένθεση

Προσέξτε ότι η έννοια "έκφραση" ορίζεται αναδρομικά. Αυτό μας επιτρέπει να κατασκευάσουμε πολύπλοκες εκφράσεις όπως `(-b+sqrt(a^2-4*a*c))/(2*a*c)`

Όταν στο διερμηνέα δίνουμε μία έκφραση, ο διερμηνέας την αποτιμά και μας επιστρέφει το αποτέλεσμα. Για παράδειγμα, η αποτίμηση της έκφρασης `1+2*3` θα δώσει `7`.

1.2.1 Προτεραιότητα και Προσεταιριστικότητα

Προσέχουμε ότι, όπως είναι αναμενόμενο, ο πολλαπλασιασμός αποτιμάται πριν από την πρόσθεση (αλλιώς η έκφραση θα αποτιμόταν σε `9`). Αυτό είναι συνέπεια των κανόνων *προτεραιότητας* των τελεστών της Haskell. Αν θέλαμε την πρόσθεση να αποτιμηθεί πρώτη, τότε θα έπρεπε να χρησιμοποιήσουμε παρενθέσεις: `(1+2)*3`.

Ένα άλλο σημείο που έχει σημασία για τους δυαδικούς τελεστές είναι η *προσεταιριστικότητά* τους. Η προσεταιριστικότητα καθορίζει αν η πολλαπλή εφαρμογή ενός δυαδικού τελεστή, π.χ. `1-1-1` θα αποτιμηθεί από τα αριστερά προς τα δεξιά ή ανάποδα.

Στους τελεστές με *αριστερή* προσεταιριστικότητα, η αποτίμηση γίνεται από τα αριστερά προς τα δεξιά. Επειδή ο τελεστής `-` έχει αριστερή προσεταιριστικότητα, η έκφραση `1-1-1` είναι ίση με `(1-1)-1` και επιστρέφει `-1`. Το αντίθετο συμβαίνει με τη *δεξιά προσεταιριστικότητα*. Αν ο `-` είχε δεξιά προσεταιριστικότητα, τότε η αποτίμηση θα επέστρεφε `1`.

Υπάρχουν και τελεστές χωρίς προσεταιριστικότητα. Σε αυτούς, η πολλαπλή εφαρμογή απαγορεύεται. Για παράδειγμα είναι συντακτικό λάθος να γράψουμε `x==y==z`.

Ο Πίνακας 1 περιέχει μερικούς τελεστές της Haskell, μαζί με την προσεταιριστικότητά τους και σε φθίνουσα σειρά προτεραιότητας.

1.3 Βασικοί Τύποι της Haskell

Κάθε έκφραση έχει έναν *τύπο*. Η Haskell υποστηρίζει τέσσερις βασικούς τύπους: `Bool` (αληθοτιμές), `Int` (ακέραιοι αριθμοί), `Float` (αριθμοί κινητής υποδιαστολής) και `Char` (χαρακτήρες).

Προτερ.	Τελεστές
9	`!`(A) `!!`(A) `//`(A) `.`(Δ)
8	`**`(Δ) `^`(Δ) `^^`(Δ)
7	`*`(A) `/`(A)
6	`+`(A) `-`(A) `:+`
5	`\\` `:`(Δ) `++`(Δ)
4	`/= < <= == > >=`
3	`&&`(Δ)
2	`||`(Δ)
1	`>>`(A) `>>=`(A) `:=`
0	`$`(Δ)

Δίπλα σε κάθε τελεστή αναγράφεται η προσεταιριστικότητά του σε παρένθεση (Α-αριστερή, Δ-δεξιά). Αν δεν αναγράφεται προσεταιριστικότητα, τότε ο τελεστής δεν έχει προσεταιριστικότητα.

Πίνακας 1: Προσεταιριστικότητα και Προτεραιότητα Τελεστών της Haskell

1.3.1 `Bool`

Ο τύπος `Bool` αναπαριστά τις *αληθοτιμές*. Υπάρχουν δύο σταθερές αυτού του τύπου, η `True` και η `False`. Οι τελεστές που υποστηρίζονται για αυτόν τον τύπο είναι οι δυαδικοί `&&` (λογική σύζευξη), `||` (λογική διάζευξη), `==` (ισότητα, λογική ισοδυναμία), `/=` (ανισότητα, αποκλειστική λογική διάζευξη (exclusive or)) και ο μοναδιαίος `not` (λογική άρνηση).

Προσέξτε ότι ο τελεστής ισότητας `==` (που υποστηρίζεται και για τους υπόλους βασικούς τύπους) γράφεται με δύο χαρακτήρες ίσον, όπως και στις γλώσσες C/C++ και Java. Ο λόγος είναι ότι ο τελεστής `=` είναι δεσμευμένος για τους *ορισμούς* (βλ. Ενότ. 1.4).

1.3.2 `Int`

Ο τύπος `Int` αναπαριστά τους *ακέραιους αριθμούς*. Σταθερές αυτού του τύπου κατασκευάζουμε αναπαραθέτοντας δεκαδικά ψηφία κατά τον αναμενόμενο τρόπο. Η χρήση του μοναδιαίου τελεστή `-` επιτρέπεται για την κατασκευή αρνητικών αριθμών. Δυαδικοί τελεστές για αυτόν τον τύπο συμπεριλαμβάνουν τους `+ - ^ *` (αντίστοιχα: πρόσθεση, αφαίρεση, ύψωση σε δύναμη, πολλαπλασιασμός). Συναρτήσεις για αυτόν τον τύπο είναι οι `div mod abs` (αντίστοιχα: ακέραια διαίρεση, υπόλοιπο ακέραιας διαίρεσης και απόλυτη τιμή). Επίσης υποστηρίζονται τελεστές σύγκρισης `== /= > < >= <=` με την αναμενόμενη σημασιολογία.

1.3.3 `Char`

Ο τύπος `Char` αναπαριστά τους *χαρακτήρες*. Σταθερές αυτού του τύπου κατασκευάζουμε περικλείοντας ένα χαρακτήρα σε μονά εισαγωγικά, πχ. `'a'`. Επίσης, υποστηρίζονται οι παρακάτω ειδικοί χαρακτήρες: `'\t' '\n' '\\' '\''`

`'\"'` (αντίστοιχα: tab, νέα γραμμή, backslash (\), μονό εισαγωγικό (') και διπλό εισαγωγικό (")). Η συνάρτηση `ord` παίρνει ένα χαρακτήρα και επιστρέφει τον κωδικό ASCII του χαρακτήρα αυτού. Η συνάρτηση `chr` είναι η αντίστροφη της `ord`.

1.3.4 `Float`

Ο τύπος `Float` αναπαριστά τους *αριθμούς κινητής υποδιαστολής* (αριθμοί που δεν είναι κατ' ανάγκη ακέραιοι). Σταθερές τέτοιου τύπου κατασκευάζουμε είτε χρησιμοποιώντας υποδιαστολή π.χ. `1.4 -2.0` κτλ. είτε χρησιμοποιώντας την *"επιστημονική" γραφή* a`e`b όπου a είναι ένας αριθμός κινητής υποδιαστολής και b είναι ένας ακέραιος. Η παραπάνω σταθερά συμβολίζει τον αριθμό $10^b a$. Για παράδειγμα, η σταθερά `2.0e-3` συμβολίζει τον αριθμό 0.002.

Οι τελεστές `+ - *` καθώς και όλοι οι τελεστές σύγκρισης υποστηρίζονται για τον τύπο `Float` και έχουν την ίδια σημασιολογία με τους αντίστοιχους του `Int`. Ο τελεστής `/` συμβολίζει κανονική (όχι ακέραια) διαίρεση. Υπάρχουν δύο τελεστές ύψωσης σε δύναμη: ο `^` (το δεύτερο όρισμα πρέπει να είναι φυσικός αριθμός) και ο `**` (το δεύτερο όρισμα είναι τύπου `Float`).

Επίσης υποστηρίζονται οι εξής συναρτήσεις: `abs` (απόλυτη τιμή), `ceiling floor round` (μετατροπή στο μεγαλύτερο, μικρότερο και πλησιέστερο ακέραιο), `cos sin tan` (συνημίτονο, ημίτονο, εφαπτομένη), `log` (λογάριθμος με βάση το e) και `sqrt` (τετραγωνική ρίζα). Η συνάρτηση `fromIntegral` μετατρέπει έναν ακέραιο σε αριθμό κινητής υποδιαστολής. Η σταθερά `pi` συμβολίζει τον αριθμό π.

Σημειώστε ότι, αντίθετα με τις περισσότερες γλώσσες προγραμματισμού, δε γίνονται αυτόματες μετατροπές μεταξύ των τύπων `Int` και `Float`. Για παράδειγμα, αυτή η έκφραση δεν επιτρέπεται

```
floor pi  +  2.5
```

γιατί ο τύπος της έκφρασης `floor pi` είναι `Int` και δε μπορεί να προστεθεί σε σταθερά τύπου `Float`. Το σωστό θα είναι να μετατρέψουμε την έκφραση χρησιμοποιώντας τη συνάρτηση `fromIntegral`:

```
fromIntegral(floor pi)  +  2.5
```

(επιστρέφει `5.5`).

1.4 Προγράμματα Haskell και Ορισμοί

Αν ο προγραμματιστής της Haskell δε μπορεί να ορίσει τα δικά του ονόματα (και άρα τις δικές του συναρτήσεις), ο διερμηνέας της Haskell δε χρησιμεύει παραπάνω από ένα κομπιουτεράκι τσέπης. Ο ορισμός των τιμών νέων ονομάτων γίνεται σε αρχεία κειμένου με κατάληξη `.hs` που λέγονται *προγράμματα Haskell (Haskell scripts)*.

Μπορούμε να φορτώσουμε τους ορισμούς ενός αρχείου χρησιμοποιώντας την εντολή `:load` του διερμηνέα, ακολουθούμενη από το όνομα του αρχείου χωρίς την επέκταση `.hs`. Για παράδειγμα, ας φτιάξουμε ένα πρόγραμμα στον τρέχοντα κατάλογο. Το όνομα αυτού του αρχείου θα είναι `firsthaskell.hs` και τα περιεγχόμενά του θα είναι:

```
one::Int
one = 1
two::Int
two = 2
```

Τώρα μπορούμε να φορτώσουμε στο διερμηνέα το πρόγραμμα με την εντολή:

```
:load firsthaskell
```

Αν το πρόγραμμα δεν είναι στον τρέχοντα κατάλογο, μπορούμε να αλλάξουμε τον τρέχοντα κατάλογο με την εντολή `:cd` π.χ.:

```
:cd "c:\Documents and Settings\superhacker\path\to\haskell\scripts"
```

Αφού φορτώσουμε το πρόγραμμα στο διερμηνέα, όλοι οι ορισμοί μας είναι σε ισχύ. Το πρόγραμμα που φορτώσαμε ορίζει δύο νέα ονόματα, τα `one` και `two`. Μπορούμε να δώσουμε στο διερμηνέα να αποτιμήσει την έκφραση `one+two`. Το αποτέλεσμα της αποτίμησης είναι `3`.

Ένα πρόγραμμα Haskell αποτελείται από *προτάσεις (sentences)* και *σχόλια (comments)*. Ένα σχόλιο ξεκινάει από δύο παύλες (`-`) και επεκτείνεται ως το τέλος της γραμμής:

```
-- this is a Haskell comment
  -- this is another Haskell comment
```

Ο διερμηνέας αγνοεί όλα τα σχόλια και χειρίζεται μόνο τις προτάσεις. Τα σχόλια χρησιμεύουν για επεξηγήσεις του κώδικα σε φυσική γλώσσα.

Μία πρόταση ξεκινάει από τον πρώτο χαρακτήρα μίας γραμμής που δεν είναι διάστημα και συμπεριλαμβάνει τη γραμμή στην οποία αρχίζει, καθώς και όλες τις γραμμές που ακολουθούν και αρχίζουν πιο μέσα από τον πρώτο χαρακτήρα. Όλα τα σχόλια αγνοούνται. Για παράδειγμα, οι επόμενες προτάσεις είναι ισοδύναμες:

```
one = 0+1
```

και

```
one --this is a comment.  it is ignored
 =
  0
   +  --yet another comment
 1
```

Μία πρόταση S τελειώνει εκεί που αρχίζει η επόμενη πρόταση, δηλαδή στην πρώτη γραμμή που ξεκινάει στην ίδια στήλη ή αριστερότερα από την πρόταση S. Πχ. τα παρακάτω περιέχουν δύο προτάσεις:

```
one = 0+1
two = 2
```

και

```
 one
  =
   0
    +1
two =
 2
```

Η χρήση του ειδικού χαρακτήρα `;` επίσης τερματίζει μία πρόταση. Για παράδειγμα, το παρακάτω κομμάτι Haskell είναι ισοδύναμο με τα δύο παραπάνω:

```
one = 0+1 ; two = 2
```

Μία πρόταση μπορεί να είναι είτε *καθορισμός τύπου (type annotation)* είτε *ορισμός (definition)* ενός αναγνωριστικού. Αν και ο καθορισμός τύπου δεν είναι απαραίτητος, συνηθίζουμε να τον δίνουμε για κάθε αναγνωριστικό που ορίζουμε.

Ο καθορισμός τύπου δίνει στη Haskell τον τύπο ενός αναγνωριστικού. Γίνεται ως εξής:
αναγνωριστικό`::`*τύπος*
Για παράδειγμα:

```
one::Int
```

Ο ορισμός δίνει την τιμή ενός αναγνωριστικού. Γίνεται ως εξής: *αναγνωριστικό=έκφραση*
Για παράδειγμα:

```
one=1
```

Οι ορισμοί ονομάτων με τιμές βασικών τύπων δεν είναι ιδιαίτερα χρήσιμοι. Αυτό που έχει σημασία είναι οι ορισμοί συναρτήσεων, που θα δούμε στην επόμενη ενότητα.

2 Συναρτήσεις

Ο ορισμός συναρτήσεων είναι ο τρόπος με τον οποίο γράφουμε προγράμματα σε Haskell. Σε αυτήν την ενότητα θα δούμε τα βασικά του ορισμού συναρτήσεων.

2.1 Τύποι, Ορισμοί και Εφαρμογή Συναρτήσεων

Ο τύπος μίας συνάρτησης που παίρνει παραμέτρους τύπου $A, B, C, ...$ και επιστρέφει αποτέλεσμα τύπου R είναι A`->`B`->`C`->...->`R. Έτσι, για παράδειγμα, το παρακάτω μας λέει ότι η `add` είναι μία συνάρτηση δύο ακεραίων ορισμάτων που επιστρέφει ένα ακέραιο αποτέλεσμα:

```
add::Int->Int->Int
```

Για να ορίσουμε την `add`, μπορούμε να ονομάσουμε τα ορίσματά της. Τα ονόματα που τους δίνουμε λέγονται *τυπικοί παράμετροι (formal parameters)* του ορισμού. Το αποτέλεσμα της εφαρμογής της `add` το γράφουμε μετά τον τελεστή ορισμού `=`. Έτσι, ο παρακάτω ορισμός μας λέει ότι η `add` επιστρέφει το άθροισμα των δύο ορισμάτων της, τα οποία έχουμε ονομάσει `x` και `y`:

```
add::Int->Int->Int
add x y = x+y
```

Με τον παραπάνω ορισμό, η αποτίμηση του `add 2 3` θα γίνει ως εξής: Η διαδικασία αποτίμησης θα περάσει το 2 ως τιμή της τυπικής παραμέτρου `x` και το 3 ως τιμή της τυπικής παραμέτρου `y` και θα αποτιμήσει την έκφραση `2+3`. Το αποτέλεσμα θα είναι `5`. Σε μία εφαρμογή συνάρτησης, όπως η έκφραση `add 2 3`, τα ορίσματα που περνάμε (εδώ `2` και `3`) λέγονται *πραγματικοί παράμετροι (actual paremeters)* της εφαρμογής.

Για να εξηγούμε καλύτερα τι συμβαίνει στις αποτιμήσεις εφαρμογών συναρτήσεων, θα χρησιμοποιήσουμε τον εξής συμβολισμό: Έστω `E` και `t` εκφράσεις και `x` αναγνωριστικό. Τότε η έκφραση E^{t}_{x} θα συμβολίζει την έκφραση `E` αφού αντικαταστήσουμε το `x` με `t`. Για παράδειγμα, η παραπάνω αποτίμηση μπορεί να γραφτεί:

$\texttt{add 2 3} = \texttt{(x+y)}^{\texttt{2}\ \texttt{3}}_{\texttt{x}\ \texttt{y}} = \texttt{2+3} = \texttt{5}$

Οι τυπικές παράμετροι είναι *τοπικές* στον ορισμό της συνάρτησης και δε μπορούν να προσπελαστούν έξω από αυτόν. Αυτό σημαίνει ότι μία μεταβλητή *έξω από τον ορισμό* της οποίας το όνομα τυχαίνει να συμπίπτει με το όνομα μίας τυπικής παραμέτρου, αναφέρεται παρ'όλα αυτά σε *άλλο* αντικείμενο. Για παράδειγμα, στους παρακάτω ορισμούς

```
f x = x+1
x = 5
```

ο ακέραιος που ορίζεται στην πρόταση `x=5` δεν έχει καμία σχέση με την τυπική παράμετρο `x` της `f`.

Ο ακέραιος `x` είναι ορατός παντού και μπορεί να προσπελαστεί από άλλους ορισμούς και ερωτήματα, πχ. η αποτίμηση της έκφρασης

```
x+1
```

θα δώσει `6`.

Από την άλλη, η τυπική παράμετρος `x` της `f` χρησιμεύει μόνο για τον ορισμό της `f`. Ο μόνος τρόπος να την προσπελάσουμε είναι να εφαρμόσουμε την `f` σε κάποιο πραγματικό όρισμα. Η αποτίμηση της εφαρμογής `f 10` για παράδειγμα, οδηγεί στην αντικατάσταση της τυπικής παραμέτρου `x` στο σώμα της `f` με την πραγματική `10`, έτσι ώστε η αποτίμηση να δίνει `10+1` δηλαδή `11`:

$\texttt{f 10} = \texttt{(x+1)}^{\texttt{10}}_{\texttt{x}} = \texttt{10+1} = \texttt{11}$

Αυτή η αποτίμηση *δε θα αλλάξει* την τιμή της εξωτερικής μεταβλητής `x`, που δεν έχει σχέση με την τυπική παράμετρο `x`. Η τιμή της `x` θα παραμείνει `5`.

Δείτε ακόμα και την αποτίμηση των συναρτήσεων `f x` και `f(x+1)`. Η πρώτη αποτίμηση θα αντικαταστήσει την τυπική παράμετρο `x` με την εξωτερική μεταβλητή `x`, δίνοντας `x+1` που είναι ίσο με `5+1` που είναι ίσο με `6`:

$\texttt{f x} = \texttt{(x+1)}^{\texttt{x}}_{\texttt{x}} = \texttt{x+1} = \texttt{5+1} = \texttt{6}$

Στην έκφραση $\texttt{(x+1)}^{\texttt{x}}_{\texttt{x}}$, το πάνω `x` είναι η εξωτερική μεταβλητή και το κάτω η τυπική παράμετρος. Η δεύτερη αποτίμηση θα αντικαταστήσει την τυπική παράμετρο `x` με την έκφραση `x+1`. Σε αυτήν την έκφραση, το όνομα `x` αναφέρεται πάλι στην εξωτερική μεταβλητή και έτσι η αποτίμηση θα δώσει `7`:

$\texttt{f(x+1)} = \texttt{(x+1)}^{\texttt{x+1}}_{\texttt{x}} = \texttt{(x+1)+1} = \texttt{(5+1)+1} = \texttt{7}$

Είναι καλό να αποφεύγεται η συνωνυμία τυπικών παραμέτρων με εξωτερικά ονόματα, ώστε να μην υπάρχουν τέτοιες δυσκολίες. Εφόσον οι τυπικές παράμετροι δε φαίνονται έξω από τον ορισμό, το όνομά τους δεν έχει σημασία. Επομένως μπορούμε να τους αλλάξουμε το όνομα, χωρίς πρόβλημα. Ο ορισμός

```
f z = z+1
```

είναι ισοδύναμος με αυτόν που χρησιμοποιούσαμε μέχρι τώρα, αλλά το όνομα της τυπικής παραμέτρου δε συγχέεται με αυτό της εξωτερικής μεταβλητής x. Τώρα είναι ευκολότερο να δούμε τι συμβαίνει:

$\texttt{f x} = \texttt{(z+1)}_{\texttt{z}}^{\texttt{x}} = \texttt{x+1} = \texttt{5+1} = \texttt{6}$

$\texttt{f(x+1)} = \texttt{(z+1)}_{\texttt{z}}^{\texttt{x+1}} = \texttt{(x+1)+1} = \texttt{(5+1)+1} = \texttt{7}$

2.2 Currying

Αν περάσουμε σε μία συνάρτηση λιγότερες παραμέτρους από αυτές που περιμένει, τότε αυτή θα επιστρέψει μία νέα συνάρτηση, η οποία περιμένει και τις υπόλοιπες παραμέτρους πριν υπολογίσει το αποτέλεσμα. Στο παραπάνω παράδειγμα, η έκφραση `add 2` είναι μία συνάρτηση τύπου `Int->Int` (παίρνει έναν ακέραιο και επιστρέφει έναν ακέραιο). Η λειτουργία της `add 2` είναι να επιστρέφει το όρισμά της αυξημένο κατά 2, πχ. η αποτίμηση της `(add 2)3` επιστρέφει 5.

Στην πραγματικότητα μπορούμε να δούμε την `add` σα μία συνάρτηση που παίρνει δύο ακεραίους και επιστρέφει έναν ακέραιο, αλλά μπορούμε να τη δούμε και σα μία συνάρτηση που παίρνει έναν ακέραιο και επιστρέφει μία συνάρτηση τύπου `Int->Int`. Η Haskell δεν κάνει κανένα διαχωρισμό μεταξύ αυτών των δύο.

Η τεχνική να ορίζουμε μια συνάρτηση πολλών ορισμάτων με αυτόν τον τρόπο, δηλαδή ως μία συνάρτηση που παίρνει ένα όρισμα και επιστρέφει μία άλλη συνάρτηση, ονομάζεται *currying* προς τιμή του Haskell Curry, που πρώτος τη χρησιμοποίησε. Όπως φαντάζεστε, και η γλώσσα Haskell έχει πάρει το όνομά της από τον Curry.

Στον ορισμό μίας συνάρτησης μπορούμε να δώσουμε λιγότερες τυπικές παραμέτρους απ' ό,τι τα ορίσματα που ζητάει ο τύπος της συνάρτησης. Σε αυτήν την περίπτωση, η έκφραση που ορίζει τη συνάρτηση πρέπει να αποτιμάται σε συνάρτηση. Για παράδειγμα, έστω ότι θέλουμε να ορίσουμε τη συνάρτηση `suc` που να παίρνει έναν ακέραιο και να επιστρέφει τον αμέσως μεγαλύτερο ακέραιο. Ο τύπος της `suc` μας λέει ότι η συνάρτηση περιέχει ένα ακέραιο όρισμα:

```
suc::Int->Int
```

Στον ορισμό εμείς μπορούμε να παραλείψουμε το όρισμα και να εξισώσουμε τη `suc` απευθείας με κάποια άλλη συνάρτηση. Στην περίπτωσή μας μπορούμε να χρησιμοποιήσουμε την `add` που ορίσαμε παραπάνω:

```
suc = add 1
```

ορισμός που θα ήταν ισοδύναμος με τον πιο "αναμενόμενο"

```
suc x = add 1 x
```

Με το currying χρειάζεται να προσέχουμε ακόμα περισσότερο την ονομασία των τυπικών παραμέτρων. Για παράδειγμα, έστω οι παρακάτω ορισμοί

```
add x y = x+y
y = 5
f = add y
```

Υπενθυμίζουμε ότι ο ακέραιος y δεν έχει σχέση με την τοπική τυπική παράμετρο y του ορισμού της add. Η ερώτηση είναι: ποια είναι η συνάρτηση f.

Μπορούμε να μπούμε στον πειρασμό να αντικαταστήσουμε την τυπική παράμετρο x στον ορισμό της add με y ως εξής:
$\texttt{f z = (add y)z = ((x + y)}^{\texttt{y}}_{\texttt{x}}\texttt{)}^{\texttt{z}}_{\texttt{y}}\texttt{ = (y+y)}^{\texttt{z}}_{\texttt{y}}\texttt{ = z+z=2*z}$
Η αντικατάσταση αυτή μας δίνει ότι η συνάρτηση f διπλασιάζει το όρισμά της. Αυτό όμως είναι λάθος! Το πρόβλημα είναι ότι η πραγματική παράμετρος y (της οποίας η τιμή είναι 5) εμφανίζεται στην ίδια έκφραση με την τυπική παράμετρο y, με την οποία *δεν έχει σχέση*. Αυτό το φαινόμενο λέγεται *σύγκρουση ονομάτων (name collision)*.

Για να υπολογίσουμε σωστά την f, θα πρέπει πριν εφαρμόσουμε το όρισμα y στην add να *μετονομάσουμε* την τυπική παράμετρο y, έτσι ώστε να μην υπάρχει σύγκρουση ονομάτων. Ο ορισμός της add είναι ισοδύναμος με

```
add x w = x+w
```

οπότε και η f υπολογίζεται ως εξής:
$\texttt{f z = (add y)z = ((x + w)}^{\texttt{y}}_{\texttt{x}}\texttt{)}^{\texttt{z}}_{\texttt{w}}\texttt{ = (y+w)}^{\texttt{z}}_{\texttt{w}}\texttt{ = y+z=5+z}$
δηλαδή (όπως θα περιμέναμε) η f είναι η συνάρτηση που προσθέτει 5 στο όρισμά της.

Γενικά, ο κανόνας σωστής εφαρμογής μίας συνάρτησης σε περίπτωση σύγκρουσης ονόματος είναι: **πρώτα αλλάζουμε τις τυπικές παραμέτρους έτσι ώστε να μην υπάρχει σύγκρουση και μετά κάνουμε τις αντικαταστάσεις**.

2.3 Σύνθεση Συναρτήσεων

Ένας πολύ χρήσιμος τελεστής που εφαρμόζεται σε συναρτήσεις είναι η *σύνθεση (composition)*. Στη Haskell, η σύνθεση συμβολίζεται με τελεία (.). Η σύνθεση δύο συναρτήσεων f.g επιστρέφει τη συνάρτηση που παίρνουμε αν εφαρμόσουμε στο όρισμα πρώτα την g και μετά, στο αποτέλεσμα αυτής της εφαρμογής, την f. Δηλαδή, ο ορισμός

```
h = f.g
```

είναι ισοδύναμος με τον ορισμό

```
h x = f(g x)
```

2.4 Τελεστές και Ενθεματική Σύνταξη

Οι συναρτήσεις της Haskell χρησιμοποιούνται *προθεματικά (prefix notation)*. Αυτό σημαίνει ότι όταν εφαρμόζουμε μία συνάρτηση, πρώτα γράφουμε τη συνάρτηση και μετά ακολουθούν τα ορίσματά της. Παράδειγμα προθεματικής σύνταξης είναι η παρακάτω εφαρμογή της συνάρτησης abs:

```
abs(1+x)
```

Οι τελεστές της Haskell είναι κι αυτοί συναρτήσεις, που όμως χρησιμοποιούν *ενθεματική σύνταξη (infix notation)*. Αυτό σημαίνει ότι ο τελεστής αναγράφεται όχι πριν από όλα τα ορίσματα, αλλά μεταξύ του πρώτου και του δεύτερου ορίσματος. Αυτό είναι χρήσιμο κυρίως σε συναρτήσεις με δύο ορίσματα. Παράδειγμα ενθεματικής σύνταξης είναι η εφαρμογή του τελεστή πρόσθεσης στο παράδειγμα της προηγούμενης παραγράφου.

Η διαφορά των ενθεματικών τελεστών από τις προθεματικές συναρτήσεις είναι εντελώς επιφανειακή. Ένας ενθεματικός τελεστής μπορεί να χρησιμοποιηθεί προθεματικά αν τον βάλουμε σε παρενθέσεις, πχ.

```
(+) 1 2
```

(επιστρέφει `3`). Η χρήση των παρενθέσεων κάνει τους τελεστές να χρησιμοποιούνται όπως οι κανονικές συναρτήσεις. Για παράδειγμα, στις εισαγωγικές σημειώσεις είδαμε τον ορισμό της `sum` από τη `reduce` χρησιμοποιώντας τη συνάρτηση `(+)` η οποία είναι απλά η προθεματική μορφή του τελεστή `+`:

```
sum = reduce 0 (+)
```

Θα μπορούσαμε να περάσουμε και ένα μόνο όρισμα στην `(+)`, π.χ. ο ορισμός της `suc` μπορεί να γίνει ως εξής:

```
suc = (+) 1
```

Ομοίως, μία προθεματική συνάρτηση (με δύο παραμέτρους) μπορεί να χρησιμοποιηθεί ενθεματικά αν τη βάλουμε μέσα σε αντίστροφα εισαγωγικά (`‘`). Για παράδειγμα, ο ορισμός της `add` παραπάνω μας επιτρέπει την παρακάτω ερώτηση:

```
2 ‘add‘ 3
```

στην οποία η απάντηση είναι `5`.

Η Haskell επιτρέπει τη δημιουργία των δικών μας ενθεματικών τελεστών χρησιμοποιώντας μία σειρά συμβόλων από τα `! # $ % & * + . / < = > ? \ ^ | : - ~` και λατινικών χαρακτήρων, αρκεί να μη σχηματίζεται υπάρχων τελεστής της Haskell. Για παράδειγμα μπορούμε να δώσουμε τον εξής ορισμό:

```
x ~~ y = (x+y)/2
```

Τώρα ο ενθεματικός τελεστής `~~` επιστρέφει το μέσο όρο των ορισμάτων του.

Η προτεραιότητα και η προσεταιριστικότητα των τελεστών που ορίζει ο χρήστης δίνεται από τις οδηγίες `infix` (χωρίς προσεταιριστικότητα), `infixl` (αριστερή προσεταιριστικότητα) και `infixr` (δεξιά προσεταιριστικότητα). Κάθε μία από αυτές τις οδηγίες ακολουθείται από έναν αριθμό προτεραιότητας (αντιστοιχεί ακριβώς στους αριθμούς προτεραιότητας του Πίνακα 1) και τον τελεστή που αφορά. Για παράδειγμα, η παρακάτω οδηγία δίνει προτεραιότητα 5 στον τελεστή `~~` που δημιουργήσαμε και καθορίζει ότι αυτός δεν έχει καμία προσεταιριστικότητα:

```
infix 5 ~~
```

2.5 Ορισμός με Φρουρούς

Ένας ορισμός συνάρτησης μπορεί να χωριστεί σε περιπτώσεις, ανάλογα με την ισχύ ή όχι κάποιων συνθηκών. Σε αυτήν την περίπτωση, ο ορισμός έχει την εξής μορφή:
οριζόμενη συνάρτηση με τυπικές παραμέτρους
| *συνθήκη0 = έκφραση0*
| *συνθήκη1 = έκφραση1*
...
| *συνθήκηn = έκφρασηn*

Οι συνθήκες αποτιμώνται με τη σειρά που εμφανίζονται. Στην πρώτη συνθήκη που επιστρέφει `True`, η αντίστοιχη έκφραση αποτιμάται και επιστρέφεται ως τιμή της οριζόμενης συνάρτησης. Για παράδειγμα, η συνάρτηση `sign`, που ορίζουμε παρακάτω, επιστρέφει το πρόσημο του ορίσματός της:

```
sign::Int->Int
sign x
  | x>0  = 1
  | x==0 = 0
  | x<0  = -1
```

Η λέξη-κλειδί `otherwise` μπορεί να μπει ως συνθήκη για να καλύψει τις περιπτώσεις που δεν έχουν καλυφθεί από τις προηγούμενες συνθήκες. Ο παραπάνω ορισμός είναι ισοδύναμος με:

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

Φυσικά, στη θέση της `otherwise` μπορούμε να βάλουμε τη συνθήκη `True`. Ο ορισμός που θα προκύψει θα είναι ισοδύναμος.

Οι συνθήκες που χρησιμοποιούμε σε αυτούς τους ορισμούς ονομάζονται *φρουροί (guards)*.

Η χρήση του ορισμού με φρουρούς βοηθάει. Δεν είναι όμως απαραίτητη. Ο τελεστής `if then else` μπορεί να χρησιμοποιηθεί για να ξεχωρίσει δύο περιπτώσεις, όπως και στις προστακτικές γλώσσες. Για παράδειγμα, ο ορισμός της `sign` μπορεί να γίνει ως εξής:

```
sign::Int->Int
sign x = if x>0 then 1 else if x==0 then 0 else -1
```

Ο ορισμός με φρουρούς είναι πιο ευανάγνωστος σε περίπτωση που οι φρουροί είναι περισσότεροι από δύο.

2.6 Συναρτήσεις Ανώτερης Τάξης

Όπως είπαμε και στις εισαγωγικές σημειώσεις, οι συναρτήσεις είναι δυνατόν να περάσουν σαν ορίσματα σε άλλες συναρτήσεις. Η συνάρτηση που δέχεται ως όρισμα

άλλες συναρτήσεις λέγεται *ανώτερης τάξης*, σε αντίθεση με τις "συνηθισμένες" συναρτήσεις που λέγονται *πρώτης τάξης*.

Στον καθορισμό τύπου μίας ανώτερης τάξης συνάρτησης, χρειάζεται να προσέξουμε να βάλουμε τον τύπο της συνάρτησης-ορίσματος μέσα σε παρενθέσεις. Για παράδειγμα, μία συνάρτηση που παίρνει ως όρισμα μια συνάρτηση τύπου `Int->Int` και επιστρέφει έναν ακέραιο έχει τύπο

```
(Int->Int)->Int
```

Αν ξεχάσουμε τις παρενθέσεις, έχουμε τον τύπο `Int->Int->Int` που αντιστοιχεί σε μία συνάρτηση πρώτης τάξης που παίρνει δύο ακεραίους και επιστρέφει έναν ακέραιο. Γενικά, πρέπει να θυμόμαστε ότι ο τελεστής `->` έχει δεξιά προσεταιριστικότητα. Ο τύπος `Int->Int->Int` είναι ίσος με τον τύπο `Int->(Int->Int)`, και *όχι* με τον τύπο `(Int->Int)->Int`.

Ας δούμε ένα παράδειγμα συνάρτησης ανώτερης τάξης. Το παράδειγμα επίσης περιέχει αναδρομικό ορισμό (που συζητάμε και πιο κάτω, στην Ενότ. 2.7). Για αυτό το παράδειγμα παρατηρούμε ότι μαθηματικές πράξεις όπως η πρόσθεση, ο πολλαπλασιασμός και η ύψωση σε δύναμη φυσικών αριθμών μπορούν να οριστούν αναδρομικά με τον ίδιο τρόπο: Η πρόσθεση ορίζεται ως η επανειλημμένη εφαρμογή της συνάρτησης `suc`. Ο πολλαπλασιασμός ορίζεται ως η επανειλημμένη εφαρμογή της πρόσθεσης. Η ύψωση σε δύναμη ορίζεται ως η επανειλημμένη εφαρμογή του πολλαπλασιασμού.

Ορίζουμε πρώτα τη συνάρτηση `apply` που υλοποιεί την επανειλημμένη εφαρμογή. Θα παίρνει τρία ορίσματα: τον αριθμό επαναλήψεων, τον αριθμό που θα επιστρέφεται σε περίπτωση 0 επαναλήψεων και τη συνάρτηση που θα εφαρμόζει. Θα επιστρέφει έναν ακέραιο:

```
apply::Int->Int->(Int->Int)->Int
apply n i f
  | n==0 = i
  | n>0  = f(apply (n-1) i f)
```

Οι υπόλοιπες πράξεις μπορούν τώρα να οριστούν ως εξής:

```
suc::Int->Int
suc x = x+1
add::Int->Int->Int
add x y = apply y x suc
mult::Int->Int->Int
mult x y = apply y 0 (add x)
power::Int->Int->Int
power x y = apply y 1 (mult x)
```

2.7 Αναδρομή

Ένας ορισμός μπορεί να περιέχει αναγνωριστικά που ορίζονται στο ίδιο αρχείο. Για παράδειγμα:

```
two = one + 1
one = 1
```

Ειδικότερα όμως, ο ορισμός ενός αναγνωριστικού μπορεί να περιέχει το ίδιο το υπό ορισμό αναγνωριστικό. Για παράδειγμα,

```
b = False && b
```

Σε αυτήν την περίπτωση λέμε ότι έχουμε *αναδρομή (recursion)* και ότι ο ορισμός είναι *αναδρομικός (recursive definition)*.

Η αναδρομή δε χρησιμεύει ιδιαίτερα στις βασικές τιμές, αλλά έχει μεγάλη σημασία για τις συναρτήσεις. Οι περισσότερες χρήσιμες συναρτήσεις ορίζονται με αναδρομή. Το κλασσικό παράδειγμα είναι η συνάρτηση που υπολογίζει το παραγοντικό, όπως είδαμε και στις εισαγωγικές σημειώσεις:

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

Εδώ το παραγοντικό ενός αριθμού `factorial n` ορίζεται με χρήση του παραγοντικού του αμέσως μικρότερου αριθμού `factorial(n-1)`. Οι περισσότερες αναδρομές είναι τόσο απλές, αλλά υπάρχουν και πιο σύνθετες περιπτώσεις.

Στην πιο γενική περίπτωση, έχουμε την *αμοιβαία αναδρομή (mutual recursion)*. Στην αμοιβαία αναδρομή, μία ομάδα περισσότερων από ένα αναγνωριστικών ορίζονται ταυτόχρονα με χρήση αναγνωριστικών από την ίδια ομάδα. Και πάλι, η αμοιβαία αναδρομή δεν είναι χρήσιμη στις βασικές τιμές, αλλά είναι στις συναρτήσεις. Το παρακάτω είναι ένα απλοϊκό παράδειγμα αμοιβαίου αναδρομικού ορισμού των συναρτήσεων `e` και `o`:

```
e::Int->Bool
o::Int->Bool
e x = if x == 0 then True else o (abs x - 1)
o x = if x == 0 then False else e (abs x - 1)
```

Η `e` επιστρέφει `True` αν και μόνον αν το όρισμά της είναι άρτιος αριθμός. Η `o` επιστρέφει το αντίθετο.

3 Παραδείγματα

3.1 Υπολογισμός Τετραγώνου με Αναδρομή

Εδώ βλέπουμε ένα απλό παράδειγμα χρήσης *πρωταρχικής αναδρομής (primitive recursion)*. Η πρωταρχική αναδρομή είναι αναδρομή κατά την οποία ο υπολογισμός της εφαρμογής μίας συνάρτησης `f n` εξαρτάται αναδρομικά μόνο από μία (απλούστερη) εφαρμογή της `f`, π.χ. της `f(n-1)`.

Μας ζητείται να γράψουμε μία συνάρτηση που υπολογίζει το τετράγωνο ενός φυσικού αριθμού, χρησιμοποιώντας μόνο πρόσθεση και αφαίρεση. Το δύσκολο

σε αυτό το πρόβλημα είναι να βρούμε τον αναδρομικό τύπο. Γι' αυτόν το λόγο καταφεύγουμε στα μαθηματικά. Αφού θέλουμε να λύσουμε το πρόβλημά μας με αναδρομή, θα πρέπει να βρούμε ένα τύπο που να συνδέει το `n^2` με το `(n-1)^2`. Για να το κάνουμε αυτό, μπορούμε να σκεφτούμε ως εξής:

```
(n-1)^2 = n^2 - 2*n + 1
```

άρα (λύνουμε ως προς `n^2` και εκφράζουμε το `2*n` μόνο με πρόσθεση):

```
n^2 = (n-1)^2 + n + n - 1
```

Η αναδρομική λύση που παίρνουμε βάση του παραπάνω τύπου είναι:

```
sqr n
 | n==0    = 0
 | True    = sqr(n-1) + n + n - 1
```

3.2 Υπολογισμός Αριθμού Συνδυασμών με Αναδρομή

Εδώ έχουμε ένα πιο σύνθετο παράδειγμα αναδρομής. Μας ζητείται να βρούμε τον αριθμό των συνδυασμών που μπορούμε να έχουμε αν επιλέξουμε k αντικείμενα από n αντικείμενα. Ο αριθμός αυτός είναι

$$\text{comb}\ n\ k\ =\ \frac{n!}{k!(n\text{-}k)!}$$

Η αφελής λύση αυτού του προβλήματος προβλέπει τον υπολογισμό τριών παραγοντικών και μία διαίρεση. Αυτή η λύση είναι μη πρακτική, γιατί τα παραγοντικά είναι τεράστιοι αριθμοί και έτσι ακόμα και για σχετικά μικρές τιμές του n θα υπάρξει υπερχείλιση. Χρειάζεται να βρούμε μία καλύτερη αναδρομική λύση. Σκεφτόμαστε με βάση την αναδρομή του παραγοντικού:

$$\frac{n!}{k!(n\text{-}k)!} = \frac{(n\text{-}1)!n}{(k\text{-}1)!k((n\text{-}1)\text{-}(k\text{-}1))!} = \frac{n}{k}\text{comb}(n\text{-}1)(k\text{-}1)$$

Επομένως, η αναδρομική λύση, που αποφεύγει τον υπολογισμό μεγάλων αριθμών και δουλεύει σε χρόνο ανάλογο του k είναι:

```
comb :: Int->Int->Int
comb n k
 | k == 0    = 1
 | True      = n * comb (n-1) (k-1) `div` k
```

3.3 Υπολογισμός Αριθμών Fibonacci με Αναδρομή

Ας δούμε ένα παράδειγμα που δε λύνεται με πρωταρχική αναδρομή. Οι *αριθμοί Fibonacci* ορίζονται από την ακολουθία:

$$f_0 = f_1 = 1$$

$$n > 1 \Rightarrow f_n = f_{n\text{-}1} + f_{n\text{-}2}$$

Εδώ χρειαζόμαστε δύο απλούστερες αναδρομικές εφαρμογές της υπό ορισμό συνάρτησης:

```
fib n
 | n <= 1   = 1
 | True     = fib(n-1)+fib(n-2)
```

Δυστυχώς αυτή είναι μία πολύ κακή λύση του προβλήματος που τρέχει σε εκθετικό χρόνο. Το πρόβλημα είναι ότι οι δύο υπολογισμοί `fib(n-1)` και `fib(n-2)` λειτουργούν ανεξάρτητα, και υπολογίζουν πολλές φορές τα ίδια αποτελέσματα. Θα επανέρθουμε δριμύτεροι σε αυτό το πρόβλημα, όταν θα έχουμε τα εργαλεία να το λύσουμε καλύτερα.

3.4 Υπολογισμός Δύναμης με Αμοιβαία Αναδρομή και Μετασχηματισμούς

Στο παράδειγμα αυτό, θα δούμε πώς χρησιμοποιούμε αμοιβαία αναδρομή καθώς και πώς μετασχηματίζουμε ένα πρόγραμμα. Θα δούμε επίσης πως η αμοιβαία αναδρομή είναι αρκετή για να εκφράσει πολύ περίπλοκες καταστάσεις *ροής ελέγχου (control flow)*. Σε προστακτικές γλώσσες, τέτοιου είδους ροή είναι δύσκολο να ελεγχθεί και γι' αυτό αποθαρρύνεται από το μοντέλου του *δομημένου προγραμματισμού (structured programming)*. Το παράδειγμα προέρχεται από το [Heh04], όπου εφαρμόζεται σε προστακτική γλώσσα με χρήση τυπικών μεθόδων. Εδώ είναι προσαρμοσμένο στη Haskell.

Ας υποθέσουμε ότι μας δίνουν το παρακάτω πρόβλημα: Φτιάξτε μία συνάρτηση `power` που να υψώνει ένα θετικό ακέραιο σε έναν άλλο, χωρίς να χρησιμοποιεί την πράξη `^`. Η αρχική μας προσέγγιση είναι πολύ απλή (βλ. και Ενότ. 2.6):

```
power :: Int->Int->Int

power x y
 | y == 0    = 1
 | True      = x * power x (y-1)
```

Η λύση δεν είναι η καλύτερη δυνατή. Μπορούμε να μετασχηματίσουμε το πρόγραμμα ως εξής: σε περίπτωση που το `y` είναι άρτιος, μπορούμε να το διαιρέσουμε με το 2, επιτυγχάνοντας λογαριθμικό χρόνο εκτέλεσης. Σε περίπτωση που το `y` είναι περιττός, αυτή η βελτίωση δε μπορεί να γίνει. Μετασχηματίζουμε λοιπόν το πρόγραμμα, εισάγοντας δύο νέες συναρτήσεις: `power_even_non_zero_y` για την περίπτωση που το `y` είναι άρτιος και μεγαλύτερο του μηδενός και `power_odd_y` για την περίπτωση που το `y` είναι περιττός:

```
power :: Int->Int->Int
power_even_non_zero_y :: Int->Int->Int  --
power_odd_y :: Int->Int->Int            --
```

```
power x y
 | y == 0             = 1
 | y `mod` 2 == 0     = power_even_non_zero_y x y   --
 | True               = power_odd_y x y             --

power_even_non_zero_y x y = power (x*x) (y `div` 2) --

power_odd_y x y = x * power x (y-1)                 --
```

(στον κώδικα σημειώνουμε με σχόλια `--` ό,τι είναι καινούριο). Βλέπουμε ότι έχουμε ήδη αμοιβαία αναδρομικούς ορισμούς μεταξύ των τριών συναρτήσεων που ορίσαμε.

Αυτό ήταν μία σημαντική βελτίωση. Μπορούμε να κάνουμε ακόμα μερικές βελτιώσεις. Κατ' αρχήν, παρατηρήστε ότι αν ξέρουμε ότι ο `y` είναι άρτιος και μεγαλύτερος του μηδενός, τότε ξέρουμε και ότι ο `y `div` 2` είναι μεγαλύτερος του μηδενός. Μπορούμε να γλυτώσουμε λοιπόν από τον έλεγχο που επιβάλλει η `power` στην αρχή της και να προχωρήσουμε κατευθείαν στον υπολογισμό που προβλέπεται για `y>0`. Για να το κάνουμε αυτό, εισάγουμε μία νέα συνάρτηση `power_non_zero_y` που θα χρησιμοποιείται όταν ξέρουμε ότι η δεύτερη παράμετρος είναι μεγαλύτερη του μηδενός:

```
power :: Int->Int->Int
power_even_non_zero_y :: Int->Int->Int
power_odd_y :: Int->Int->Int
power_non_zero_y :: Int->Int->Int         --

power x y
 | y == 0            = 1
 | True              = power_non_zero_y x y                    --

power_non_zero_y x y                                           --
 | y `mod` 2 == 0    = power_even_non_zero_y x y               --
 | True              = power_odd_y x y                         --

power_even_non_zero_y x y = power_non_zero_y (x*x) (y `div` 2) --

power_odd_y x y = x * power x (y-1)
```

Συνεχίζοντας, παρατηρούμε ότι όταν ο `y` είναι περιττός, τότε ο `y-1` είναι άρτιος. Μπορούμε, όπως και πριν, να μην πετάξουμε αυτήν την πληροφορία. Μπορούμε να τη χρησιμοποιήσουμε για να γλυτώσουμε τον έλεγχο που επιβάλλει η `power_non_zero_y`. Γι' αυτό το λόγο, δημιουργούμε μία καινούρια συνάρτηση `power_even_y` η οποία εφαρμόζεται όταν ξέρουμε ότι το `y` είναι άρτιος, αλλά δεν ξέρουμε αν είναι ίσο με το μηδέν. Ο κώδικας γίνεται τώρα:

```
power :: Int->Int->Int
power_even_non_zero_y :: Int->Int->Int
power_odd_y :: Int->Int->Int
```

```
power_non_zero_y :: Int->Int->Int
power_even_y :: Int->Int-> Int               --

power x y
 | y == 0           = 1
 | True             = power_non_zero_y x y

power_non_zero_y x y
 | y `mod` 2 == 0    = power_even_non_zero_y x y
 | True             = power_odd_y x y

power_even_non_zero_y x y = power_non_zero_y (x*x) (y `div` 2)

power_odd_y x y = x * power_even_y x (y-1)                    --

power_even_y x y                                              --
 | y == 0           = 1                                      --
 | True             = power_even_non_zero_y x y              --
```

Τέλος, θα κάνουμε την εύλογη υπόθεση ότι είναι πιο πιθανό να μας ζητηθεί η περίπτωση y>0 από την περίπτωση y==0. Επομένως, συμφέρει περισσότερο να εξετάζουμε στην αρχή αν το y είναι άρτιος ή περιττός (δύο ισοπίθανα ενδεχόμενα), απ' ό,τι αν το y είναι μηδέν ή όχι. Αυτή η αλλαγή είναι σχετικά εύκολη:

```
power :: Int->Int->Int
power_even_non_zero_y :: Int->Int->Int
power_odd_y :: Int->Int->Int
power_non_zero_y :: Int->Int->Int
power_even_y :: Int->Int-> Int

power x y                                                     --
 | y `mod` 2 == 0    = power_even_y x y                       --
 | True             = power_odd_y x y                        --

power_non_zero_y x y
 | y `mod` 2 == 0    = power_even_non_zero_y x y
 | True             = power_odd_y x y

power_even_non_zero_y x y = power_non_zero_y (x*x) (y `div` 2)

power_odd_y x y = x * power_even_y x (y-1)

power_even_y x y
 | y == 0           = 1
 | True             = power_even_non_zero_y x y
```

Σε αυτό το παράδειγμα είδαμε το μετασχηματισμό ενός συναρτησιακού προγράμματος και τη χρήση αμοιβαίας αναδρομής.

Ξεκινήσαμε από ένα απλό και ευκολονόητο πρόγραμμα και με μικρά βήματα, που όμως διατηρούν την ορθότητα του προγράμματος, καταλήξαμε σε ένα πιο περίπλοκο αλλά πιο αποδοτικό, που είναι ισοδύναμο με το πρώτο. Ο συναρτησιακός χαρακτήρας της Haskell διευκόλυνε τους μετασχηματισμούς. Σε μια προστακτική γλώσσα προγραμματισμού, το τελικό πρόγραμμα είναι χαώδες και περιλαμβάνει στοιχεία ανεπίτρεπτα για το δομημένο προγραμματισμό, όπως είσοδος στο μέσο ενός βρόγχου κτλ. Οι μετασχηματισμοί αυτοί πολύ εύκολα θα οδηγούσαν σε λάθος. Στο παράδειγμά μας, οι μετασχηματισμοί ήταν σχετικά απλοί και ευνόητοι και το τελικό πρόγραμμα όχι και τόσο χαώδες.

Σχεδόν από την αρχή του παραδείγματος, χρησιμοποιήσαμε αμοιβαία αναδρομή. Η αμοιβαία αναδρομή είναι ο γενικότερος μηχανισμός ροής ελέγχου και μπορεί από μόνη της να αναπαραστήσει *κάθε άλλη* ροή, συμπεριλαμβανομένων αυτών που υποστηρίζει ο δομημένος προγραμματισμός (βρόγχοι `while` κτλ.), αλλά και εντελώς άναρχων ροών, όπως αυτές που δημιουργούνται χρησιμοποιώντας εντολές `goto` σε προστακτικές γλώσσες χαμηλού επιπέδου.

3.5 Υπολογισμός Αθροίσματος Συνάρτησης

Στο τελευταίο μας παράδειγμα, θα φτιάξουμε μία συνάρτηση ανώτερης τάξης που να παίρνει έναν ακέραιο n και μία συνάρτηση f από ακέραιο σε ακέραιο και να υπολογίζει το άθροισμα $\sum_{i=1}^{n} f\ i$. Μετά θα τη χρησιμοποιήσουμε για να υπολογίσουμε διάφορα αθροίσματα, επιδεικνύοντας παράλληλα και τη χρήση της σύνθεσης συναρτήσεων και το currying.

Θα ονομάσουμε τη συνάρτησή μας `sumf`. Η `sumf` παίρνει μία συνάρτηση `f` τύπου `Int->Int` και έναν ακέραιο `n` και επιστρέφει έναν ακέραιο. Σε περίπτωση που το `n` είναι `0` τότε και η συνάρτηση επιστρέφει `0`. Αλλιώς καλείται αναδρομικά με όρισμα `n-1` ως εξής:

```
sumf :: (Int->Int)->Int->Int
sumf f n
 | n==0      = 0
 | True      = f n + sumf f (n-1)
```

Η `sumf` μπορεί να χρησιμοποιηθεί τώρα για να υπολογίσει διάφορα αθροίσματα. Για παράδειγμα, μπορούμε να υπολογίσουμε το άθροισμα: $1 + 2 + ... + 7$ ως εξής:

```
sumf id 7
```

(η συνάρτηση `id` της Haskell απλά επιστρέφει το όρισμά της, δηλ. `id x = x`).

Γενικότερα, μπορούμε να χρησιμοποιήσουμε currying για να ορίσουμε τη συνάρτηση `sumupto` η οποία παίρνει έναν ακέραιο n και υπολογίζει το $\sum_{i=1}^{n} i$, ως εξής:

```
sumupto :: Int->Int
sumupto = sumf id
```

Αυτό ισχύει και για όλα τα υπόλοιπα παραδείγματα, αλλά δε θα το ξανααναφέρουμε.

Ας χρησιμοποιήσουμε τη `sumf` για να υπολογίσουμε κάτι πιο περίπλοκο: το άθροισμα των τετραγώνων των αριθμών μέχρι το n, δηλαδή: $\sum_{i=1}^{n} i^2$. Σε αυτήν την περίπτωση, η `f` πρέπει να είναι η συνάρτηση που υψώνει στο τετράγωνο. Θα μπορούσαμε να ορίσουμε μία τέτοια συνάρτηση εύκολα. Αλλά η Haskell μας επιτρέπει να κάνουμε currying στους τελεστές της δίνοντάς τους μόνο μερικά από τα ορίσματά τους. Έτσι η συνάρτηση τετραγωνισμού ήδη υπάρχει: είναι η `(^2)`. Το άθροισμα των τετραγώνων επομένως είναι η συνάρτηση `sumf (^2)`. Για παράδειγμα, μπορούμε να πάρουμε το άθροισμα των τετραγώνων των αριθμών μέχρι το 5 ως εξής:

```
sumf (^2) 5
```

(η αποτίμηση δίνει 55). Η `sumf` εδώ εφαρμόζει τη συνάρτηση `(^2)` διαδοχικά στα `1,2,3,4,5`. Η εφαρμογή της `(^2)` σε ένα όρισμα `n` δίνει `n^2`. Κι έτσι όλα δουλεύουν όπως τα θέλουμε.

Ας δοκιμάσουμε τώρα να υπολογίσουμε το άθροισμα όλων των λογαρίθμων (με βάση το 10) των αριθμών μέχρι το 10, δηλ. $\sum_{i=1}^{10} \log i$:

```
sumf log 10
```

Η έκφραση αυτή δεν αποτιμάται λόγω λάθους στους τύπους. Αυτό είναι αναμενόμενο, επειδή η `sumf` περιμένει μία συνάρτηση `Int->Int` και αυτός δεν είναι ο τύπος της `log`.

Αντί να περιμένουμε μέχρι το μάθημα του πολυμορφισμού, ας συμβιβαστούμε μετατρέποντας τους λογαρίθμους σε ακέραιους αριθμούς, δηλ. $\sum_{i=1}^{10} \lfloor \log i \rfloor$. Για να το κάνουμε αυτό, θα πρέπει να χρησιμοποιήσουμε τη συνάρτηση `floor` που στρογγυλοποιεί έναν αριθμό κινητής υποδιαστολής στον αμέσως μικρότερο ακέραιο. Η συνάρτηση `floor` θα εφαρμοστεί στο αποτέλεσμα της `log`. Πρόκειται για περίπτωση σύνθεσης συναρτήσεων. Εφαρμόζουμε λοιπόν τη `sumf` στη συνάρτηση `floor.log` ως εξής:

```
sumf (floor.log) 10
```

Δυστυχώς, ούτε αυτό δουλεύει.

Το πρόβλημα είναι ότι η συνάρτηση `log` είναι τύπου `Float->Float`. Επομένως, εφόσον η `floor` είναι τύπου `Float->Int`, η `floor.log` είναι τύπου `Float->Int`. Δηλαδή, το πρόβλημα είναι ότι η `floor.log` περιμένει έναν αριθμό κινητής υποδιαστολής ενώ η `sumf` θέλει μία συνάρτηση που να παίρνει ακέραια παράμετρο. Όπως είπαμε, η Haskell δεν κάνει αυτόματες μετατροπές τύπων. Επομένως, πρέπει να εφαρμόσουμε τη συνάρτηση μετατροπής `fromIntegral` πριν δώσουμε το όρισμα στην `floor.log`. Πρόκειται για μία ακόμα σύνθεση συναρτήσεων, αυτή τη φορά από τα δεξιά: `(floor.log).fromIntegral`.

Η σύνθεση συναρτήσεων είναι *προσεταιριστική* πράξη. Αυτό σημαίνει ότι πάντα `(f.g).h = f.(g.h)`. Γι'αυτό το λόγο, μπορούμε να απαλείψουμε τελείως τις παρενθέσεις, και να γράψουμε `f.g.h`. Στην περίπτωσή μας, η συνάρτηση γίνεται `floor.log.fromIntegral`. Εφαρμόζουμε αυτή τη συνάρτηση στη `sumf`:

```
sumf (floor.log.fromIntegral) 10
```

και επιτέλους δουλεύει (το αποτέλεσμα είναι 11).

Ας δούμε πώς θα υπολογίσουμε ένα άθροισμα, πχ. το $\sum_{i=1}^{10} i^2$ εξαιρώντας π.χ. τους άρτιους i. Αυτό στα μαθηματικά θα το γράφαμε κάπως έτσι:

$$\sum_{i=1,\ i\ \textit{περιττός}}^{10} i^2$$

Αυτό που πρέπει να κάνουμε είναι να γράψουμε μία συνάρτηση που να "φιλτράρει" τους άρτιους αριθμούς επιστρέφοντας 0 στη θέση τους:

```
filterEven :: Int->Int
filterEven n
 | n `mod` 2 == 0   = 0
 | True             = n
```

Τώρα μπορούμε να χρησιμοποιήσουμε τη `filterEven` πριν περάσουμε το όρισμα στην `(^2)` ως εξής:

```
sumf ((^2).filterEven) 10
```

(επιστρέφει 165).

Τέλος, ας δούμε τον υπολογισμό ενός διπλού αθροίσματος:

$$\sum_{i=1}^{10}\sum_{j=1}^{i}(i+j)^2$$

Για να γράψουμε αυτό το άθροισμα στη Haskell, καλό είναι να ξεκινήσουμε από τα μέσα προς τα έξω, δηλαδή, από απλούστερες σε πιο σύνθετες συναρτήσεις. Η ενδότερη συνάρτηση παίρνει δύο παραμέτρους `i` και `j` και υπολογίζει το `(i+j)^2`:

```
inner i j = (i+j)^2
```

Προχωράμε τώρα πιο έξω. Θέλουμε να εκφράσουμε σε Haskell τη μαθηματική έκφραση $\sum_{j=1}^{i}(i+j)^2$. Προσέξτε ότι αυτή είναι μία συνάρτηση που παίρνει μία μεταβλητή, την `i` (η μεταβλητή `j` είναι εσωτερική στην άθροιση). Μπορούμε να χρησιμοποιήσουμε τη `sumf` για να εκφράσουμε την εξωτερική συνάρτηση:

```
outer i = sumf (inner i) i
```

Προσέξτε ότι η συνάρτηση που εφαρμόζουμε στην `sumf` είναι η `inner i`, δηλαδή η συνάρτηση που παίρνει το `j` και υπολογίζει το `(i+j)^2` και το όριο είναι το `i`. Αυτά τα βρίσκουμε κοιτώντας την έκφραση που θέλουμε να υπολογίσουμε.

Τώρα μπορούμε να κάνουμε μια άθροιση αυτής της έκφρασης για `i` από `1` έως `10` χρησιμοποιώντας πάλι την `sumf`. Ρωτάμε τη Haskell:

```
sumf outer 10
```

και η απάντηση που παίρνουμε είναι `7645`.

4 Επισκόπηση

Σε αυτήν την ενότητα είδαμε τα βασικά της χρήσης ενός διερμηνέα της Haskell όπως ο Hugs, την έννοια της έκφρασης, του τελεστή και της συνάρτησης, τους βασικούς τύπους και τη δομή ενός προγράμματος Haskell. Μετά επικεντρώσαμε την προσοχή μας ειδικότερα στους ορισμούς συναρτήσεων. Είδαμε τους τύπους συναρτήσεων, την εφαρμογή συναρτήσεων και τον κανόνα αποφυγής συγκρούσεως ονομάτων, την ιδέα του currying, τη σύνθεση συναρτήσεων, τη σχέση τελεστών-συναρτήσεων, τη χρήση φρουρών, τη χρήση συναρτήσεων ανώτερης τάξης και τους αναδρομικούς ορισμούς. Τέλος είδαμε πέντε παραδείγματα ορισμών συναρτήσεων που επέδειξαν ιδιαίτερα την πρωταρχική αναδρομή, την εκφραστικότητα της αμοιβαίας αναδρομής ως προς τη ροή ελέγχου, το μετασχηματισμό συναρτησιακών προγραμμάτων και τη χρησιμότητα των εννοιών της συνάρτησης ανώτερης τάξης, του currying και της σύνθεσης συναρτήσεων.

Στη συνέχεια θα περάσουμε σε πιο προχωρημένες έννοιες του προγραμματισμού σε Haskell, όπως για παράδειγμα το πολύ εξελιγμένο πολυμορφικό σύστημα τύπων που υποστηρίζει. Στις σημειώσεις αυτές, θα καλύψουμε σημαντικό μέρος της Haskell. Για τους ενδιαφερόμενους για περισσότερη εμβάθυνση, προτείνεται ως βοήθημα το βιβλίο [Tho99]. Τονίζεται πάντως ότι η ύλη του μαθήματος είναι οι *σημειώσεις* που διατείθενται στο μάθημα και *όχι* αυτό το βιβλίο.

Αναφορές

[Heh04] E. C. R. Hehner. *A Practical Theory of Programming*. Current edition, 2004. On-line: `http://www.cs.toronto.edu/~hehner/aPToP/` .

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