Αφηρημένοι Τύποι Δεδομένων

Γιάννης Κασσιός

Σε αυτές τις σημειώσεις θα μιλήσουμε για την *τμηματοποίηση*, όπως υποστηρίζεται στη Haskell, και τους *αφηρημένους τύπους δεδομένων - ΑΤΔ*. Η τμηματοποίηση, και κυρίως οι ΑΤΔ που μπορούν να κατασκευαστούν μέσω αυτής, είναι σημαντικά συστατικά συστημάτων λογισμικού μέσης και μεγάλης πολυπλοκότητας.

Στην Ενότ. 1, θα δούμε τι είναι τμηματοποίηση, τη σχετική ορολογία και θα δούμε πώς αυτή υποστηρίζεται στον Hugs.

Στην Ενότ. 2, θα μιλήσουμε πιο ειδικά για τους ΑΤΔ, πώς δημιουργούνται στη Haskell και πώς υλοποιούνται και χρησιμοποιούνται. Θα δείξουμε παραδείγματα στα οποία οι ΑΤΔ συμβάλλουν στη σωστή χρήση των δεδομένων, στις τοπικές μετατροπές που, χάρη στην απόκρυψη πληροφορίας των ΑΤΔ, δεν επεκτείνονται στο υπόλοιπο πρόγραμμα και στην υψηλού επιπέδου περιγραφή των δεδομένων σε έναν ΑΤΔ. Τέλος, θα μιλήσουμε για το σχεδιασμό των ΑΤΔ και την έννοια των τυπικών συμβολαίων για την περιγραφή της διαπροσωπείας ενός ΑΤΔ.

1 Τμηματοποίηση

1.1 Η Έννοια της Τμηματοποίησης

Η *τμηματοποίηση (modularization)*, όπως συζητήσαμε και στις εισαγωγικές σημειώσεις, είναι η πιο σημαντική ιδέα που αναπτύχθηκε στην προσπάθεια για την κατασκευή περίπλοκου λογισμικού: πρόκειται για το διαχωρισμό του λογισμικού σε διακριτά *τμήματα (modules)* που είναι *όσο το δυνατό πιο ανεξάρτητα μεταξύ τους*. Υπάρχουν πολλοί λόγοι που αυτό είναι επιθυμητό στον προγραμματισμό πολύπλοκου λογισμικού:

- Η ανάπτυξη λογισμικού μπορεί να γίνει από πολλά άτομα / ομάδες που μπορούν να δουλεύουν ανεξάρτητα και να μην επηρρεάζουν η μία την άλλη (ακόμα και να μη γνωρίζουν η μία την ύπαρξη της άλλης, όπως γίνεται στην ανάπτυξη *βιβλιοθηκών (libraries)*).
- Οι αλλαγές υλοποίησης και οι διορθώσεις σφαλμάτων σε ένα τμήμα, εφόσον σέβονται τη *διαπροσωπεία (interface)* του, γίνονται χωρίς να επεκτείνονται σε άλλα τμήματα.
- Ένα τμήμα μπορεί να επαναχρησιμοποιηθεί πολλές φορές σε πολλά διαφορετικά προγράμματα, αφού αντιπροσωπεύει μία ανεξάρτητη λειτουργικότητα που μπορεί να φανεί χρήσιμη σε διάφορες καταστάσεις.

Βασικό στοιχείο στην τμηματοποίηση είναι η *απόκρυψη πληροφορίας (information hiding)* και συγκεκριμένα η απόκρυψη των *λεπτομερειών υλοποίησης (implementation details)* ενός κομματιού κώδικα. Έτσι ένα τμήμα σε μία οποιαδήποτε γλώσσα προγραμματισμού προσφέρει περιορισμένη πρόσβαση στις οντότητες που κατασκευάζονται μέσα του. Ο λόγος που τα τμήματα υποστηρίζουν την απόκρυψη πληροφορίας είναι να δωθεί μια σχετική ελευθερία στον κατασκευαστή του τμήματος να αλλάξει την υλοποίηση του τμήματος (και ό,τιδήποτε αφορά την κρυφή - μη προσβάσιμη πληροφορία), χωρίς αυτή η αλλαγή να "φαίνεται απ'έξω", δηλαδή χωρίς να επηρρεάζει τον κώδικα άλλων τμημάτων του ιδίου προγράμματος.

Ο πιο συνηθισμένος τρόπος να γίνει αυτό είναι η απόκρυψη κάποιων ονομάτων από αυτά που χρησιμοποιούνται μέσα στο τμήμα, έτσι ώστε αυτά να φαίνονται μόνο μέσα στο τμήμα. Τα υπόλοιπα ονόματα ονομάζονται *εξαγόμενα (exported)*, καθώς είναι αυτά που φαίνονται και εκτός του τμήματος. Η *διαπροσωπεία (interface)* του τμήματος είναι το σύνολο των ονομάτων που εξάγει μαζί με τις *προδιαγραφές* (τυπικές ή σε φυσική γλώσσα) για τη συμπεριφορά αυτών των ονομάτων.

Ένα τμήμα μπορεί να χρησιμοποιηθεί μέσα σε ένα άλλο τμήμα μέσω της διαδικασίας της *εισαγωγής (importation)*. Όταν ένα τμήμα M1 *εισάγει (imports)* ένα άλλο τμήμα M2, τότε μόνο τα εξαγόμενα ονόματα του M2 είναι διαθέσιμα στον προγραμματιστή του M1. Το τμήμα που πραγματοποιεί την εισαγωγή ονομάζεται και *πελάτης (client)* ή *χρήστης (user)* του εισαγόμενου τμήματος. Επειδή ένα τμήμα μπορεί να εισαχθεί από οποιοδήποτε άλλο τμήμα, ο προγραμματιστής ενός τμήματος δε μπορεί να θεωρεί ότι γνωρίζει τους πελάτες του τμήματος αυτού.

Στην Εισαγωγή μας, αλλά και αργότερα, είδαμε ότι ο συναρτησιακός προγραμματισμός εισάγει ιδιώματα τμηματοποίησης που δεν είναι συνηθισμένα στις κλασικές γλώσσες προγραμματισμού. Η τμηματοποίηση, όπως την εννοούμε εδώ, είναι χαρακτηριστικό που ξεκίνησε κυρίως σε προστακτικές γλώσσες προγραμματισμού, αλλά δεν είναι καθόλου ασύμβατη με το συναρτησιακό μοντέλο. Σε αυτήν την ενότητα θα δούμε πώς η λειτουργικότητα των τμημάτων υλοποιείται στη Haskell.

1.2 Τμήματα και Αρχεία στον Hugs

Στη Haskell, επιτρέπεται ένα αρχείο να έχει παραπάνω από ένα τμήμα. Παρ' όλα αυτά, ο Hugs δεν επιτρέπει κάτι τέτοιο και μάλιστα επιβάλλει συγκεκριμένη ονοματολογία στα ονόματα των αρχείων που αντιστοιχούν σε τμήματα προγραμμάτων, που μοιάζει πολύ με αυτήν της Java.

Ο Hugs ψάχνει για αρχεία τμημάτων σε συγκεκριμένους καταλόγους του λειτουργικού συστήματος. Το σύνολο των κατάλογων αυτών ονομάζεται *μονοπάτι αναζήτησης (search path)* και καθορίζεται όταν τρέχουμε τον Hugs με την επιλογή `-P`. Για τον καθορισμό του μονοπατιού αναζήτησης σημειώνονται τα εξής:

- Οι διαφορετικοί κατάλογοι στο μονοπάτι χωρίζονται με `:` στα Unix/Linux συστήματα και με `;` σε συστήματα Windows.

- Στο μονοπάτι θα πρέπει να υπάρχει το αρχείο `Prelude.hs` της Haskell, για το οποίο μιλάμε στην Ενότ. 1.6, αλλιώς η οδηγία αγνοείται από τον Hugs.

- Στο μονοπάτι επιτρέπεται να γράψουμε τη συντομογραφία `{Hugs}` που συμβολίζει τον κατάλογο βιβλιοθηκών που έρχονται με την εγκατάσταση της Haskell.
- Στο μονοπάτι επιτρέπεται η συντομογραφία `*` που συμβολίζει όλους τους καταλόγους που υπάρχουν στο σημείο που γράφεται. Για παράδειγμα, το μονοπάτι `./a/b/*` περιλαμβάνει όλους τους υποκαταλόγους του `./a/b/` σε ένα σύστημα Unix/Linux.
- Το αρχείο `Prelude.hs` βρίσκεται στον κατάλογο `{Hugs}/base` (για Windows: `{Hugs}\base`).
- Σε περίπτωση μη ορισμού μονοπατιού αναζήτησης, το προκαθορισμένο μονοπάτι περιέχει τα `.` (τρέχων κατάλογος) και `{Hugs}/packages/*` (Windows: `{Hugs}\packages\*`)

Επιπλέον, σημειώστε ότι μπορούμε να χρησιμοποιήσουμε την εντολή `:s` στη γραμμή εντολών του Hugs για να δούμε και να αλλάξουμε οποιαδήποτε επιλογή στη γραμμή εντολών. Έτσι, γράφοντας `:s` θα δούμε πληροφορίες για όλες τις επιλογές γραμμής εντολών, συμπεριλαμβανομένου και του μονοπατιού αναζήτησης, ενώ για να αλλάξουμε το μονοπάτι αναζήτησης μπορούμε να γράψουμε, π.χ. την εξής εντολή (πάλι για σύστημα Unix/Linux):

```
:s -P{Hugs}/packages/*:.:~/mylibs/
```

Οταν ο Hugs αναζητεί ένα τμήμα με όνομα `A`, τότε ψάχνει σε όλους τους καταλόγους του μονοπατιού αναζήτησης για το αρχείο `A.hs`.

Το όνομα ενός τμήματος μπορεί να διαχωρίζεται από τελείες. Όπως και στη Java, όταν συμβαίνει αυτό, τότε ο Hugs ψάχνει για το αντίστοιχο αρχείο αντικαθιστώντας τις τελείες με το χαρακτήρα αλλαγής καταλόγου (`/` για Unix/Linux και `\` για Windows). Το όνομα του αρχείου πρέπει να συμπίπτει με το όνομα του τμήματος μετά την τελευταία τελεία και η κατάληξη πρέπει να είναι `.hs`. Για παράδειγμα, αν σε σύστημα Unix ο Hugs ψάχνει το τμήμα `A.B.C`, θα ψάξει σε όλους τους κατάλογους `A/B` ξεκινώντας από τους καταλόγους του μονοπατιού αναζήτησης. Το αρχείο που θα ψάξει θα είναι το `C.hs`.

Περισσότερες πληροφορίες για τον τρόπο αναζήτησης τμημάτων στο Hugs, μπορείτε να βρείτε στον on-line οδηγό του Hugs [Hug98].

Στα παραδείγματα αυτών των σημειώσεων, για να ξεφύγουμε από όλες αυτές τις λεπτομέρειες, θα θεωρήσουμε ότι όλα τα αρχεία μας βρίσκονται στον τρέχοντα κατάλογο, και ότι το μονοπάτι αναζήτησης περιλαμβάνει τον τρέχοντα κατάλογο.

1.3 Δήλωση Τμήματος

Η δήλωση ενός τμήματος γίνεται ως εξής:
`module` *όνομα_τμήματος* `(`*υπογραφή_εξόδου*`) where`
ορισμοί

Το όνομα του τμήματος πρέπει να αρχίζει με κεφαλαίο γράμμα. Αντίθετα απ' ό,τι μας έχει συνηθίσει η Haskell, οι ορισμοί που βρίσκονται μέσα στο τμήμα μπορούν να ξεκινούν ακριβώς στη στήλη που ξεκινάει και η λέξη `module`.

Η υπογραφή εξόδου απαριθμεί όλα τα ονόματα που *εξάγονται* από το τμήμα, χωρισμένα με κόμμα. Τα ονόματα αυτά θα πρέπει να είναι ορισμένα μέσα στο τμήμα (ή σε εισαγόμενα τμήματα, βλ. Ενότ 1.5). Τα ονόματα της υπογραφής εξόδου είναι τα μόνα που είναι ορατά εκτός του τμήματος. Ονομάζονται *εξαγόμενα ονόματα*, όπως εξηγήσαμε πιο πάνω.

Για παράδειγμα, το παρακάτω τμήμα ορίζει τα ονόματα `a,b,c,T,U` εκ των οποίων εξάγει τα `a, b` και `T`.

```
module A (a, b, T) where
a = 1
b x = c + x
c = 3
type T = (String,U)
type U = String
```

Ο τύπος και η τιμή κάθε εξαγόμενου ονόματος είναι ορατά έξω από το τμήμα, άσχετα αν έχουν κατασκευαστεί με τη χρήση μη ορατών ονομάτων. Για παράδειγμα, το όνομα `b` έχει την τιμή `\x->3+x`, αν και το όνομα `c` *δε φαίνεται* έξω από το τμήμα. Ομοίως ο εξαγόμενος τύπος `T` έχει τιμή `(String,String)`.

Οι κατασκευαστές αλγεβρικών τύπων δεν εξάγονται μόνοι τους. Στην περίπτωση που έχουμε έναν εξαγόμενο αλγεβρικό τύπο, έστω `T`, τότε, μέσα στην υπογραφή εξόδου, οι κατασκευαστές του `T` που εξάγονται πρέπει να ακολουθούν τον `T` μέσα σε παρένθεση και χωρισμένοι με κόμμα. Για παράδειγμα, το παρακάτω τμήμα επιτρέπει την εξαγωγή μόνο των κατασκευαστών `C1,C2`:

```
module B (T(C1,C2)) where
data T = C1 Int | C2 Int Int | C3 String
```

Αν θέλουμε να εξάγουμε όλους τους κατασκευαστές, τότε μπορούμε να γράψουμε `T(..)` στην υπογραφή εξόδου. Σε περίπτωση που *δεν εξάγουμε* κατασκευαστές, ο τύπος μας γίνεται ένας *αφηρημένος τύπος δεδομένων - ΑΤΔ (abstract data type - ADT)*. Τους ΑΤΔ θα τους μελετήσουμε στην Ενότ. 2.

Η αναγραφή της υπογραφής εξόδου είναι προαιρετική. Σε περίπτωση που αυτή λείπει, τότε *όλα* τα ονόματα που ορίζονται μέσα στο τμήμα, (αλλά όχι αυτά που εισάγονται από άλλα τμήματα, βλ. Ενότ. 1.4 και 1.5) είναι εξαγόμενα. Για παράδειγμα, ο παρακάτω ορισμός τμήματος:

```
module C where
a = 1
type T = String
```

είναι ισοδύναμος με

```
module C (a,T) where
a = 1
type T = String
```

1.4 Εισαγωγή Τμήματος

Για να εισάγουμε ένα τμήμα S σε ένα τμήμα C, γράφουμε την οδηγία `import` μέσα στο τμήμα C:
`import` S
Το τμήμα C τώρα γίνεται *πελάτης (ή χρήστης)* του S. Αυτό σημαίνει ότι έχει πρόσβαση σε όλα τα εξαγόμενα ονόματα του S, και γνωρίζει τους τύπους τους. Τα ονόματα αυτά λέμε ότι *εισάγονται* στο τμήμα C. Για παράδειγμα, έστω το παρακάτω τμήμα:

```
module D where
import A
c = b a
```

Παρατηρήστε ότι χρησιμοποιούμε τα ονόματα `a,b` που εισήχθησαν από το `A`. Επίσης, ορίζουμε το όνομα `c`, το οποίο είναι άσχετο με το `c` του τμήματος `A`, που είναι κρυφό σε μας. Αν φορτώσουμε το `D` στον Hugs, η αποτίμηση του `a` θα δώσει `1` ενώ η αποτίμηση του `c` θα δώσει `4`:

```
  c
= b a
= (\x->3+x)1
= 4
```

Έχουμε την επιλογή να μην εισάγουμε όλα τα ονόματα του S. Τότε, μπορεί η εντολή `import` να ακολουθείται από μία υπογραφή εξόδου μέσα σε παρενθέσεις. Μόνο τα ονόματα που αναφέρονται μέσα στην υπογραφή εισάγονται στο C. Για παράδειγμα, στο παρακάτω τμήμα εισάγονται μόνο τα `b,T` από το `A`:

```
module E where
import A(b,T)
```

Υπάρχει επίσης η αντίθετη δυνατότητα, να εισάγουμε όλα τα ονόματα *εκτός* από αυτά που αναφέρουμε. Αυτό γίνεται με τη λέξη κλειδί `hiding` που ακολουθεί την οδηγία `import` και ακολουθείται από μία υπογραφή εξόδου μέσα σε παρενθέσεις. Για παράδειγμα στο παρακάτω τμήμα εισάγονται πάλι μόνο τα `b,T`:

```
module F where
import A hiding (a)
```

Η οδηγία `import` εισάγει τα ονόματα όπως ακριβώς αναφέρονται στο εισαγόμενο τμήμα. Αυτό όμως σημαίνει ότι υπάρχει πρόβλημα όταν εισάγουμε δύο τμήματα που να έχουν κοινά εξαγόμενα ονόματα ή αν τα εισαγόμενα ονόματα ενός τμήματος ορίζονται και μέσα στο τμήμα-πελάτη. Για να αποφευχθεί αυτό το πρόβλημα, ένα όνομα n που ορίζεται μέσα σε ένα τμήμα M μπορεί να αναφέρεται και ως M`.`n. Δεν επιτρέπονται διαστήματα μεταξύ του M, της τελείας και του n (για να μη μπερδέψει ο διερμηνέας την τελεία με τον τελεστή σύνθεσης συναρτήσεων). Αυτή η σύνταξη ονομάζεται *προσδιορισμένη (qualified)* σύνταξη του n.

Η οδηγία `import qualified` εκτελεί την ίδια εισαγωγή με την `import`, αλλά τώρα ένα ονόματα n του εισαγόμενου τμήματος S μπορούν να χρησιμοποιηθούν μόνο με την προσδιορισμένη σύνταξή τους. Για παράδειγμα, κοιτάξτε πώς πρέπει να αλλάξει το τμήμα `D` αν η εισαγωγή του `A` γίνει με `import qualified`:

```
module G where
import qualified A
c = A.b A.a
```

Τέλος, το όνομα ενός τμήματος S μπορεί να αλλάξει σε S' κατά την εισαγωγή χρησιμοποιώντας την οδηγία `as` με σύνταξη: S `as` S'. Για παράδειγμα

```
module H where
import qualified A as AA
c = AA.b AA.a
```

Αυτή η δυνατότητα δίνεται και με το `import` και με το `import qualified`.

1.5 Εξαγωγή Ονομάτων Εισαγόμενων Τμημάτων

Έστω ένα τμήμα C που εισάγει ένα τμήμα S. Τα εξαγόμενα ονόματα του S αποτελούν και ονόματα του C, που μπορούν με τη σειρά τους να εξαχθούν. Για παράδειγμα, το παρακάτω τμήμα εισάγει το τμήμα `A` και επιλέγει να εξάγει τα ονόματα `a,T` από το τμήμα αυτό:

```
module I (a, T, c) where
import A
c = "Hi"
```

Εδώ εξάγεται και το όνομα `c` του τμήματος `I` με τιμή `"Hi"` (και όχι το όνομα `c` από το τμήμα `A`, το οποίο είναι κρυφό).

Σε περίπτωση που δεν δίνεται υπογραφή εξόδου στο τμήμα `C`, τότε μόνο τα ονόματα που ορίζονται στο `C` εξάγονται, ενώ αυτά των εισαγόμενων τμημάτων δεν εξάγονται. Στον παρακάτω ορισμό:

```
module J where
import A
c = "Hi"
```

μόνο το `c` εξάγεται.

Αν θέλουμε το τμήμα C να επαν-εξάγει όλα τα εξαγόμενα ονόματα του S, τότε πρέπει να συμπεριλάβουμε τη δήλωση `module` S μέσα στην υπογραφή εξόδου του C. Για παράδειγμα, το παρακάτω τμήμα `K` εξάγει τα `a,b,T` από το `A` και το `c` από το `K`:

```
module K (module A, c) where
import A
c = "Hi"
```

Βλέπουμε επίσης ότι η κεφαλίδα

```
module L where
```

είναι ισοδύναμη με την

```
module L (module L) where
```

1.6 Τμήματα `Main` και `Prelude`

Το τμήμα `Prelude` του αρχείου `Prelude.hs` περιέχει όλους τους ορισμούς των βασικών τιμών της Haskell, μερικές από τις οποίες έχουμε δει. Εισάγεται αυτόματα σε κάθε άλλο τμήμα. Μπορούμε να αλλάξουμε την εισαγωγή του χρησιμοποιώντας εντολές όπως `import hiding` και `import qualified`.

Το όνομα του τρέχοντος τμήματος όταν απουσιάζει η οδηγία `module` είναι `Main`. Στα μεταγλωττισμένα προγράμματα, θα πρέπει να υπάρχει ένα τμήμα `Main` και ένα όνομα `main` μέσα σε αυτό το τμήμα. Η αποτίμηση του ονόματος αυτού είναι και η εκτέλεση του μεταγλωττισμένου προγράμματος.

2 Αφηρημένοι Τύποι Δεδομένων

2.1 Εισαγωγή στους ΑΔΤ

Ένας *αφηρημένος τύπος δεδομένων - ΑΤΔ (abstract data type - ADT)* είναι ένας τύπος που ορίζεται μέσα σε ένα τμήμα και του οποίου η υλοποίηση είναι άγνωστη στο χρήστη. Όπως αναφέραμε πιο πάνω, για να κατασκευάσουμε έναν αφηρημένο τύπο δεδομένων, δημιουργούμε έναν αλγεβρικό τύπο και εξάγουμε το όνομα του τύπου χωρίς τους κατασκευαστές. Στη συνέχεια δημιουργούμε και εξάγουμε τις συναρτήσεις χειρισμού του τύπου αυτού που θέλουμε να είναι προσβάσιμες στο χρήστη.

Οι ΑΤΔ είναι ακόμα υψηλότερου επιπέδου από τους αλγεβρικούς. Όπως και οι αλγεβρικοί τύποι, χρησιμοποιούνται για να αντιπροσωπεύσουν αντικείμενα, είτε μαθηματικά είτε του πραγματικού κόσμου, που μας ενδιαφέρουν. Αντίθετα όμως από τους αλγεβρικούς τύπους, η εσωτερική δομή ενός ΑΤΔ είναι κρυμμένη εντελώς από το χρήστη, ο οποίος έχει πρόσβαση στις τιμές του τύπου αυτού *μόνο* μέσω των ειδικών για αυτό το σκοπό συναρτήσεων που έχει υλοποιήσει ο σχεδιαστής του.

Υπάρχουν τρεις βασικοί λόγοι για τους οποίους θέλουμε αυτόν τον περιορισμό: (α) εμποδίζουμε το χρήστη να χρησιμοποιήσει αντικείμενα του τύπου με τρόπο που δε θέλουμε, (β) επιτρέπουμε στον υλοποιητή του τύπου την ελευθερία να αλλάξει την υλοποίηση του τύπου χωρίς να ανησυχεί ότι η αλλαγή αυτή θα επιφέρει αλλαγές στον κώδικα του χρήστη και (γ) δίνουμε στο χρήστη μια υψηλού επιπέδου περιγραφή του τύπου, η οποία δεν έχει να κάνει με την υλοποίηση καθ' εαυτή, αλλά με τα αφηρημένα μαθηματικά αντικείμενα που συμβολίζει ο τύπος.

Στις επόμενες ενότητες θα επιδείξουμε τα παραπάνω τρία σημεία. Κατόπιν θα συζητήσουμε για τη σχεδίαση ΑΤΔ γενικότερα και τέλος θα αναφερθούμε στο ρόλο των τυπικών *συμβολαίων (contracts)* στον ορισμό της διαπροσωπείας ενός ΑΤΔ και ενός τμήματος γενικότερα.

2.2 Μη Πρόσβαση του Χρήστη στην Υλοποίηση

2.2.1 Υλοποίηση Στίβας

Το κλασικότερο παράδειγμα τμηματοποίησης είναι η δημιουργία *στίβας (stack)*. Η στίβα είναι μία σειριακή δομή δεδομένων, στην οποία τα δεδομένα εξέρχονται με σειρά ανάποδη από αυτή που εισέρχονται. Λέμε ότι η στίβα είναι μία LIFO (last in first out) δομή δεδομένων, γιατί τα δεδομένα που εισέρχονται τελευταία στη στίβα εξέρχονται πρώτα.

Ένα τμήμα `Stack` που υλοποιεί τη δομή της στίβας χρησιμοποιώντας μία λίστα φαίνεται παρακάτω. (Για ευκολία, θα χρησιμοποιήσουμε τη σύμβαση να γράφουμε στην υπογραφή εξόδου και τους τύπους των εξαγόμενων ονομάτων.)

```
module Stack
 ( Stack
 , emptys  -- Stack a
 , push    -- Stack a -> a -> Stack a
 , top     -- Stack a -> a
 , pop     -- Stack a -> Stack a
 , isempty -- Stack a -> Bool
 )
 where

data Stack a = S [a]

emptys = S []

push (S s) x = S (x:s)

top (S (h:_)) = h

pop (S (_:t)) = S t
```

```
isempty (S s) = s==[]
```

Η συνάρτηση που εισάγει στοιχεία στη στίβα είναι η `push` ενώ η συνάρτηση που εξάγει στοιχεία είναι η `pop`. Το στοιχείο που εξάγει η `pop` ονομάζεται το *κορυφαίο* της στίβας και, όπως βλέπουμε στην παρούσα υλοποίηση, είναι το πρώτο στοιχείο της λίστας υλοποίησης. Βλέπουμε επίσης ότι η `push` αλλάζει το τρέχον κορυφαίο στοιχείο. Τέλος, η `emptys` είναι η άδεια στίβα, η `top` επιστρέφει το κορυφαίο στοιχείο χωρίς να το βγάλει από τη στίβα και η `isempty` ελέγχει αν η στίβα είναι άδεια.

Η υλοποίηση της στίβας γίνεται εσωτερικά με μία λίστα, αλλά αυτό είναι άγνωστο στο χρήστη. Αυτό σημαίνει ότι ο χρήστης δε μπορεί να τροποποιήσει τη συμπεριφορά της στίβας παίρνοντας π.χ. ένα στοιχείο από τη μέση της, όπως θα γίνονταν στην παρακάτω συνάρτηση, αν ο κατασκευαστής S ήταν ορατός:

```
bad :: Stack a -> Stack a
bad (S s) = S (take 10 s ++ drop 11 s)
```

Μία τέτοια χρήση της στίβας δεν είναι επιθυμητή, γιατί μόνο το κορυφαίο στοιχείο μίας στίβας θα πρέπει να είναι προσπελάσιμο. Άρα η απόκρυψη της υλοποίησης της στίβας εξασφαλίζει τη LIFO συμπεριφορά της.

2.2.2 Αναλλοίωτες: Υλοποίηση Στίβας με Μετρητή

Υπάρχει ακόμα ένας λόγος για τον οποίο θέλουμε να κρατήσουμε κρυφή την υλοποίηση ενός ΑΤΔ. Είναι η διατήρηση μίας *αναλλοίωτης (invariant)* στα δεδομένα του τύπου. Μία αναλλοίωτη είναι μία σχέση των εσωτερικών δεδομένων που ισχύει σε όλα τα στοιχεία του ΑΤΔ. Η πρόσβαση του χρήστη στην υλοποίηση μπορεί να έχει ως αποτέλεσμα τη δημιουργία στοιχείων του τύπου που δεν ικανοποιούν την αναλλοίωτη και άρα έχουν μη αναμενόμενη συμπεριφορά.

Ας υποθέσουμε για παράδειγμα ότι θέλουμε να επεκτείνουμε την υλοποίηση της στίβας μας με μία συνάρτηση `count` η οποία επιστρέφει τον αριθμό των στοιχείων της στίβας. Προφανώς θα μπορούσαμε να γράψουμε την `count` πολύ γρήγορα ως εξής:

```
count (S s) = length s
```

αλλά αυτό σημαίνει ότι κάθε φορά που καλείται η `count`, ο διερμηνέας μετράει τα περιεγχόμενα της εσωτερικής λίστας, κάνοντας γραμμικό χρόνο. Θα θέλαμε ενδεχομένως μια καλύτερη υλοποίηση της `count`. Μια ιδέα είναι να κρατάμε το μήκος της εσωτερικής λίστας σαν ένα επιπλέον δεδομένο. Στο παρακάτω τμήμα `CountingStack` κάνουμε ακριβώς αυτό:

```
module CountingStack
 ( CountingStack
 , emptys  -- CountingStack a
 , push    -- CountingStack a -> a -> CountingStack a
 , top     -- CountingStack a -> a
```

```
 , pop     -- CountingStack a -> CountingStack a
 , isempty -- CountingStack a -> Bool
 , count   -- CountingStack -> Int
 )
 where

data CountingStack a = S [a] Int

emptys = S [] 0

push (S s n) x = S (x:s) (n+1)

top (S (h:_) _) = h

pop (S (_:t) n) = S t (n-1)

isempty (S _ n) = n==0

count (S _ n) = n
```

Η υλοποίηση είναι καλύτερη τώρα, καθώς η εφαρμογή της `count` παίρνει σταθερό και όχι γραμμικό χρόνο. Τώρα σε κάθε στίβα `S l n` ισχύει η αναλλοίωτη

```
n == length l
```

η οποία διατηρείται από όλες τις συναρτήσεις που εξάγονται. Αν ο χρήστης είχε πρόσβαση στον κατασκευαστή, θα μπορούσε να φτιάξει μία "στίβα" που να μην ικανοποιεί την αναλλοίωτη, όπως π.χ.

```
badstack = S [] (-1)
```

Η "στίβα" αυτή έχει κακή συμπεριφορά. Για παράδειγμα, η αποτίμηση

```
isempty (push badstack 10)
```

επιστρέφει `True`. Η απόκρυψη της υλοποίησης της στίβας αποτρέπει την καταστροφή της αναλλοίωτης των δεδομένων της στίβας από το χρήστη.

2.2.3 Χρήση Στίβας: Τεχνολόγηση Συμβολοσειρών

Είδαμε την κατασκευή ενός ΑΤΔ, αλλά δεν είδαμε ακόμα ένα παράδειγμα πελάτη. Σε αυτήν την ενότητα βλέπουμε ένα πελάτη για τη δομή στίβας που μόλις φτιάξαμε. Μία από τις πιο διαδεδομένες εφαρμογές της στίβας είναι η *τεχνολόγηση συμβολοσειρών*, που είναι πιο γνωστή με τον αγγλικό όρο *string parsing*. Στο string parsing, ελέγχουμε αν μία συμβολοσειρά ανήκει σε κάποια γραμματική.

Ας δούμε πώς μπορούμε να κάνουμε string parsing για μία απλή γραμματική. Τα τερματικά σύμβολα της γραμματικής είναι `[]()` και η γραμματική απαιτεί μία συμβολοσειρά να είναι *σταθμισμένη* δηλαδή:

- Κάθε αριστερή παρένθεση (αγκύλη) αντιστοιχεί σε μία δεξιά παρένθεση (αγκύλη).
- Δύο υποσυμβολοσειρές που είναι σταθμισμένες είναι είτε ξένες μεταξύ τους ή η μία είναι φωλιασμένη στην άλλη.

Ο τυπικός ορισμός σε μορφή BNF είναι:

```
E ::=   ''
      | '(' E ')'
      | '[' E ']'
      | E E
```

Για να κάνουμε parse αυτή τη γραμματική, θα χρησιμοποιήσουμε μία στίβα που να κρατάει όλες τις αριστερές παρενθέσεις και αγκύλες με τη σειρά που τις βλέπουμε στην υπό έλεγχο συμβολοσειρά. Όταν συναντάμε μία δεξιά παρένθεση ή αγκύλη, ελέγχουμε αν η πρώτη τιμή της στίβας περιέχει το αντίστοιχο αριστερό σύμβολο και αναλόγως σταματάμε με αρνητική απάντηση ή συνεχίζουμε το parsing, βγάζοντας την πρώτη τιμή από τη στίβα. Όταν τελειώσει η συμβολοσειρά, τότε και η στίβα πρέπει να είναι άδεια, αλλιώς υπάρχουν αριστερές παρενθέσεις και αγκύλες που δεν έχουν αντιστοιχιστεί:

```
import Stack

parse :: String -> Bool
parseaux :: String -> Stack Char -> Bool

parse str = parseaux str emptys

parseaux [] stack = isempty stack
parseaux ('(':t) stack = parseaux t (push stack '(')
parseaux ('[':t) stack = parseaux t (push stack '[')
parseaux (')':t) stack
  = not (isempty stack) && top stack == '(' && parseaux t (pop stack)
parseaux (']':t) stack
  = not (isempty stack) && top stack == '[' && parseaux t (pop stack)
```

Για να καταλάβετε πώς δουλεύει ο αλγόριθμος, μπορείτε να δοκιμάσετε να υπολογίσετε με το χέρι την `parse` εφαρμοζόμενη στις εξής έγκυρες συμβολοσειρές:

```
([]())
[(())]()
```

και τις εξής άκυρες:

```
([]()
[([])]()
[(()))()
```

Ο LIFO χαρακτήρας της στίβας είναι αυτός που εξασφαλίζει ότι το parsing γίνεται σωστά.

2.3 Αλλαγή Υλοποίησης

Σε αυτήν την ενότητα θα δείξουμε πώς οι ΑΤΔ επιτρέπουν την αλλαγή υλοποίησης χωρίς να επηρρεάζεται ο χρήστης. Ο ΑΤΔ που θα φτιάξουμε είναι η *ουρά (queue)*. Η ουρά είναι μία FIFO (first in first out) δομή δεδομένων. Αυτό σημαίνει ότι τα στοιχεία που εισέρχονται σε μία ουρά εξέρχονται από αυτήν με την ίδια σειρά με την οποία μπήκαν.

2.3.1 Πρώτη Υλοποίηση

Η πρώτη υλοποίηση της ουράς που φτιάχνουμε χρησιμοποιεί μία λίστα με τρόπο σχετικά προφανή, όπως και η στίβα που είδαμε πιο πάνω:

```
module Queue
 ( Queue
 , emptyq  -- Queue a
 , enqueue -- Queue a -> a -> Queue a
 , front   -- Queue a -> a
 , dequeue -- Queue a -> Queue a
 , isempty -- Queue a -> Bool
 )
 where

data Queue a = Q [a]

emptyq = Q []

enqueue (Q q) x = Q (q ++ [x])

front (Q (h:_)) = h

dequeue (Q (_:t)) = Q t

isempty (Q q) = q==[]
```

Η `enqueue` εισάγει ένα στοιχείο στην ουρά, βάζοντάς το στο πίσω μέρος της λίστας υλοποίησης, ενώ η `dequeue` βγάζει το πρώτο στοιχείο της λίστας υλοποίησης (και ομοίως η `front` επιστρέφει το πρώτο στοιχείο).

2.3.2 Χρήση Ουράς: Διάσχιση κατά Πλάτος

Μία από τις χρήσεις ουράς είναι ο αλγόριθμος της *διάσχισης κατά πλάτος (breadth first search)* ενός γράφου. Στον αλγόριθμο αυτό, ξεκινώντας από ένα κόμβο, επισκεπτόμαστε όλα τα στοιχεία ενός γράφου που μπορούμε να προσπελάσουμε από αυτόν τον κόμβο ως εξής:

- Πρώτα επισκεπτόμαστε τον αρχικό κόμβο.
- Μετά επισκεπτόμαστε κάθε ένα από τους γείτονες του αρχικού κόμβου.
- Για κάθε ένα γείτονα, επισκεπτόμαστε τους γείτονές του, κτλ.

Ο αλγόριθμος διατηρεί μία ουρά q στην οποία βάζουμε τους κόμβους που *πρόκειται* να επισκεφτούμε. Ο αλγόριθμος επίσης διατηρεί και ένα σύνολο `marked` από τους κόμβους που ήδη επισκεφτήκαμε. Σε κάθε βήμα:

- Βγάζουμε έναν κόμβο από την ουρά και τον επισκεπτόμαστε.
- Βάζουμε όλους τους γείτονες του κόμβου που δεν έχουμε επισκεφτεί στην ουρά.

Ο αλγόριθμος τελειώνει όταν η ουρά είναι άδεια.

Ο λόγος που χρησιμοποιείται η ουρά σε αυτήν την περίπτωση είναι για να γίνει η διάσχιση κατά πλάτος και όχι κατά βάθος. Δηλαδή, αφού επισκεφτούμε ένα κόμβο n, πρέπει να επισκεφτούμε τους γείτονες του n *πριν* προχωρήσουμε στους γείτονες των γειτόνων κτλ. Προσέξτε ότι στην παραπάνω περιγραφή, αυτό ικανοποιείται. Όταν επισκεφτόμαστε έναν κόμβο n, βάζουμε τους γείτονές του $n_0, n_1 ...$ στην ουρά. Κατόπιν, όταν επισκεφτόμαστε ένα γείτονα, πχ. n_0, βάζουμε τους δικούς του γείτονες $n_{00}, n_{01} ...$ στην ουρά. Λόγω της FIFO ιδιότητας της ουράς, θα επισκεφτούμε τον κόμβο n_1 πριν από τους n_{0x}.

Παρακάτω έχουμε την υλοποίηση της διάσχισης κατά πλάτος χρησιμοποιώντας το τμήμα `Queue`:

```
import Queue

type Graph = [[Int]]

enqueueMany q = foldl enqueue q

bfs :: Graph->Int->[Int]
bfsaux :: Graph->Queue Int->[Int]->[Int]

bfs g n = bfsaux g (enqueue emptyq n) [n]

bfsaux g q marked =
  if (isempty q) then [] else bfsaux_q_non_empty g q marked
bfsaux_q_non_empty g q marked = [f] ++ bfsaux g q'' marked'
  where
```

```
q' = dequeue q
f = front q
unmarkednb = [nb | nb<-g!!f, not (nb`elem`marked)]
q'' = enqueueMany q' unmarkednb
marked' = marked ++ unmarkednb
```

Ο γράφος αναπαρίσταται με τη μορφή *λιστών γειτνίασης*. Αυτό σημαίνει ότι για κάθε κόμβο n δίνεται μία λίστα όλων των κόμβων με τους οποίους ο n γειτνιάζει. Αυτή η λίστα λέγεται *λίστα γειτνίασης* του n.

Στην περίπτωσή μας οι κόμβοι αριθμούνται από το 0 έως το `nnodes-1` (όπου `nnodes` ο αριθμός κόμβων) και ο γράφος αναπαρίσταται από μία λίστα μήκους `nnodes`, τα στοιχεία της οποίας είναι λίστες γειτνίασης όλων των κόμβων. Έτσι, ο κόμβος `n1` γειτνιάζει με τον κόμβο `n2` στον γράφο `g` αν το `n2` είναι στοιχείο της λίστας γειτνίασης `g!!n1`.

Η συνάρτηση `enqueueMany` παίρνει μία ουρά και μία λίστα με στοιχεία και βάζει όλα τα στοιχεία στην ουρά.

Η συνάρτηση `bfs` εκτελεί τη διάσχιση αρχικοποιώντας την ουρά και τη λίστα κόμβων που έχουμε επισκεφτεί ώστε να περιέχουν μόνο τον αρχικό κόμβο `n`. Η `bfs` καλεί την `bfsaux` με αυτά τα ορίσματα και η `bfsaux` πραγματοποιεί τη διάσχιση.

Η `bfsaux` ελέγχει αν η ουρά είναι άδεια. Αν είναι, τερματίζει. Αν δεν είναι, τότε:

- αφαιρεί το πρώτο στοιχείο από την ουρά:
  `q'=dequeue q` και `f=front q`
- επισκέπτεται το στοιχείο `f` (το βάζει στην αρχή της λίστας που επιστρέφει)
- παίρνει όλους τους γειτονικούς κόμβους του `f` που δεν έχουμε επισκεφτεί ακόμα[1] (στη λίστα `unmarkednb`)
- σημαδεύει τους γειτονικούς κόμβους ως επισκεφθέντες:
  `marked' = marked ++ unmarkednb`
- βάζει όλους τους γειτονικούς κόμβους στην ουρά:
  `q''=enqueueMany q' unmarkednb`
- ξανακαλεί τον εαυτό της με τις νέες παραμέτρους `q''` και `marked'`

Στο τέλος, η συνάρτηση επιστρέφει μία λίστα με τους κόμβους με τη σειρά που επισκεφτήκαμε.

[1] Η συνάρτηση `elem` της Haskell κάνει γραμμική αναζήτηση σε μία λίστα: `x`elem`l` αν και μόνον αν το `x` είναι στοιχείο της `l`.

2.3.3 Αλλαγή Υλοποίησης

Η υλοποίηση της ουράς που δείξαμε έχει ένα μειονέκτημα: ενώ η συνάρτηση `dequeue` είναι σταθερής πολυπλοκότητας (απλά παίρνει ένα στοιχείο από την κεφαλή της λίστας), η συνάρτηση `enqueue` είναι γραμμικής πολυπλοκότητας (διασχίζει ολόκληρη τη λίστα, ώστε να βάλει ένα στοιχείο στο τέλος της).

Υπάρχει τρόπος να κάνουμε την υλοποίησή μας πιο αποδοτική, επιτυγχάνοντας σταθερές πολυπλοκότητες και για τις δύο συναρτήσεις. Η ιδέα είναι να χρησιμοποιήσουμε δύο λίστες: τις `frnt` ("μπροστινή" λίστα -- την ονομάζουμε έτσι για να μην υπάρχει σύγχιση με τη συνάρτηση `front`) και `back`. Ενώ η `dequeue` αφαιρεί στοιχεία από την κεφαλή της `frnt`, η `enqueue` *προσθέτει στοιχεία στην κεφαλή της* `back`. Όταν η λίστα `frnt` αδειάσει, τότε η `dequeue` μεταφέρει (και αναποδογυρίζει) τη λίστα `back` στη λίστα `frnt` και αδειάζει τη λίστα `back`.

Ας δούμε για παράδειγμα τι συμβαίνει στην περίπτωση:

```
dequeue (enqueue (dequeue (enqueue (enqueue emptyq 4) 3)) 1)
```

Για να παρακολουθήσετε τι γίνεται, διαβάστε την έκφραση από μέσα προς τα έξω:

- Ξεκινάμε από την κενή ουρά, στην οποία έχουμε `frnt=[],back=[]`
- Εισάγουμε το στοιχείο 4. Αυτό έχει ως αποτέλεσμα: `frnt=[],back=[4]`
- Εισάγουμε το στοιχείο 3. Αυτό έχει ως αποτέλεσμα: `frnt=[],back=[3,4]`
- Κάνουμε `dequeue`. Αφού η `frnt` είναι άδεια, η `dequeue` πρώτα "αδειάζει" την ανάποδη της `back` στην `frnt`, δηλ. `frnt=[4,3],back=[]` και μετά αφαιρεί το πρώτο στοιχείο, με αποτέλεσμα `frnt=[3],back=[]`
- Εισάγουμε το στοιχείο 1. Αυτό έχει ως αποτέλεσμα: `frnt=[3],back=[1]`
- Κάνουμε `dequeue`. Αυτό έχει ως αποτέλεσμα: `frnt=[],back=[1]`

Το "άδειασμα" της `back` στην `frnt` φυσικά κοστίζει γραμμικό χρόνο, αλλά δε γίνεται σε κάθε εφαρμογή της `dequeue` (στην πρώτη μας υλοποίηση, *κάθε* εφαρμογή της `enqueue` χρειάζεται γραμμικό χρόνο). Αυτό κάνει τη νέα μας υλοποίηση καλύτερη.

Ας δούμε την καινούρια υλοποίηση σε Haskell:

```
module Queue
 ( Queue
 , emptyq  -- Queue a
 , enqueue -- Queue a -> a -> Queue a
 , front   -- Queue a -> a
 , dequeue -- Queue a -> Queue a
 , isempty -- Queue a -> Bool
```

```
 )
 where

data Queue a = Q [a] [a]

emptyq = Q [] []

enqueue (Q frnt back) x = Q frnt (x:back)

front (Q (f:_) _) = f
front (Q [] (b:bs)) = last (b:bs)

dequeue (Q (_:fs) back) = Q fs back
dequeue (Q [] (b:bs)) = dequeue (Q (reverse (b:bs)) [])

isempty (Q [] []) = True
isempty (Q _ _) = False
```

Προσέξτε τώρα το εξής: ενώ κάναμε μια μη τετριμμένη αλλαγή υλοποίησης του τμήματος `Queue`, ο πελάτης που κάνει διάσχιση κατά πλάτος δεν αντιλαμβάνεται την αλλαγή, καθώς ο δικός του κώδικας δεν πρέπει να αλλάξει. Αυτό είναι πολύ σημαντικό, αφού, όπως είδαμε, η διάσχιση κατά πλάτος είναι ένας αλγόριθμος που έχει σημαντική πολυπλοκότητα από μόνος του.

Η τμηματοποίηση, το γεγονός δηλαδή ότι δεν ανακατέψαμε την πολυπλοκότητα της υλοποίησης της ουράς με την πολυπλοκότητα των πελατών της, μας επέτρεψε να κάνουμε αυτή τη λεπτή αλλά σημαντική "εγχείριση" στην υλοποίηση της ουράς, χωρίς οι αλλαγές να επηρρεάσουν τους πελάτες. Το σημαντικό είναι φυσικά ότι, παρά τις αλλαγές, σεβαστήκαμε τη *διαπροσωπεία* της ουράς. Το τμήμα μας συμπεριφέρεται, εισάγεται και χρησιμοποιείται με τον ίδιο τρόπο όπως και πριν, όσον αφορά έναν εξωτερικό παρατηρητή.

2.4 Υψηλού Επιπέδου Περιγραφή

Η περιγραφή των δεδομένων μας σε *υψηλό επίπεδο* διαχωρίζει αυτό που συμβολίζουν τα δεδομένα αυτά (περιγραφή σε υψηλό επίπεδο) από την υλοποίησή τους (περιγραφή σε χαμηλό επίπεδο).

Για παράδειγμα, όλες οι γλώσσες προγραμματισμού προσφέρουν κάποια υποστήριξη για ακεραίους αριθμούς γραμμένους στο δεκαδικό σύστημα. Ενδεχομένως ξέρουμε ότι η αναπαράστασή τους μέσα στη μηχανή είναι σε δυαδικό σύστημα σε μορφή *συμπληρώματος ως προς 2 (two's complement)* κτλ. Αλλά αυτό δε μας απασχολεί (εκτός ίσως αν δουλεύουμε σε γλώσσες πολύ χαμηλού επιπέδου, όπως η C) και δεν το σκεφτόμαστε όταν πάμε να υλοποιήσουμε έναν αλγόριθμο. Αντίθετα, σκεφτόμαστε τα αφηρημένα δεδομένα που αναπαρίστανται, δηλαδή τους ακεραίους αριθμούς, τα στοιχεία του συνόλου $\mathbb{Z}$.

Επίσης, στη Haskell και σε άλλες υψηλού επιπέδου γλώσσες, οι λίστες είναι δεδομένα των οποίων η υλοποίηση δε μας απασχολεί. Καταλαβαίνουμε τις λίστες τύπου A ως αντικείμενα του συνόλου A^ω και δε σκεφτόμαστε αν έχουν υλοποιηθεί εσωτερικά με δείκτες κτλ. Φυσικά, και πάλι οι γλώσσες χαμηλού επιπέδου, όπως η C, δίνουν άμεση πρόσβαση στην υλοποίηση.

Το πλεονέκτημα της αναπαράστασης σε υψηλό επίπεδο, είναι ότι δεν ασχολούμαστε με τις λεπτομέρειες αναπαράστασης αυτών των δεδομένων, πράγμα μάλλον άσχετο με τους σκοπούς του αλγόριθμου τον οποίον σχεδιάζουμε, αλλά με τα αφηρημένα αντικείμενα που αναπαριστούν τα δεδομένα μας. Οι λεπτομέρειες της υλοποίησης των δεδομένων και η πολυπλοκότητά τους δεν προστείθενται στην πολυπλοκότητα του προβλήματος υψηλότερου επιπέδου που προσπαθούμε να λύσουμε.

Οι ΑΤΔ είναι ο υπ' αριθμόν ένα μηχανισμός που πετυχαίνει αυτήν την αφαίρεση. Η υλοποίηση ενός τύπου είναι κρυμμένη από το χρήστη, ο οποίος είναι "αναγκασμένος" να σκεφτεί αφαιρετικά για αυτόν τον τύπο. Σε αυτήν την ενότητα θα κατασκευάσουμε έναν ΑΤΔ που υλοποιεί τη μαθηματική έννοια του *συνόλου (set)*, μίας συλλογής δηλαδή από στοιχεία στην οποία, αντίθετα με τις λίστες, δε μας ενδιαφέρει ούτε η σειρά, ούτε ο πληθυσμός τους. Το μόνο που μας ενδιαφέρει για ένα σύνολο είναι εάν ένα στοιχείο είναι στο σύνολο αυτό ή όχι.

Στα μαθηματικά, η συνολοθεωρία θεωρείται ότι μπορεί να χρησιμοποιηθεί ως υλοποίηση κάθε άλλης θεωρίας. Οι φυσικοί αριθμοί, οι συναρτήσεις, οι λίστες κτλ. μπορούν να αναπαρασταθούν ως σύνολα. Μία ισχυρή τάση στις γλώσσες προγραμματισμού είναι η *δηλωτική σημασιολογία*, η οποία εκφράζει τις *σημασίες* των προγραμμάτων ως μέλη ειδικών συνόλων.

Καμία από αυτές τις μαθηματικές εφαρμογές των συνόλων δε μας ενδιαφέρει εδώ. Αυτό που μας ενδιαφέρει, είναι να δημιουργήσουμε μία δομή, την οποία ο χρήστης μπορεί να κατασκευάσει εισάγοντας και διαγράφοντας στοιχεία, αλλά δε μπορεί και δεν ενδιαφέρεται να ρωτήσει για τον αριθμό εισαγωγών ενός στοιχείου ή για τη σειρά με την οποία τα στοιχεία εισήχθησαν. Το γεγονός αυτό, δίνει περισσότερη καθαρότητα στις προθέσεις του χρήστη, σε αντίθεση με την περίπτωση που αυτός χρησιμοποιούσε μία λίστα, αφού οι λίστες μπορούν να χρησιμοποιηθούν με περισσότερους τρόπους από ό,τι τα σύνολα. Το γεγονός ότι ένα σύνολο πιθανόν να έχει υλοποιηθεί εσωτερικά απλώς με μία λίστα, είναι μία λεπτομέρεια που δεν αφορά τον χρήστη.

2.4.1 Υλοποίηση του `Set`

Ας δούμε λοιπόν το τμήμα `Set` που ορίζει και εξάγει τον τύπο συνόλου `Set` και σχετικές με αυτόν πράξεις. Η υλοποίηση γίνεται για απλότητα με λίστες, αλλά θα μπορούσαν να χρησιμοποιηθούν και καλύτερες, πιο αποδοτικές δομές. Ο τύπος είναι μεν πολυμορφικός, αλλά υπάρχει ο περιορισμός τα στοιχεία που περιέχονται στα σύνολα να μπορούν να συγκριθούν με ισότητα, για να μπορούμε να ελέγξουμε αν ένα στοιχείο υπάρχει στο σύνολο ή όχι. Οι υλοποιήσεις των συναρτήσεων θα πρέπει να είναι λίγο-πολύ προφανείς:

```
module Set
 ( Set
```

```
 , emptyset --      Eq a => Set a
 , singleton --     Eq a => a -> Set a
 , el --            Eq a => a -> Set a -> Bool
 , union --         Eq a => Set a -> Set a -> Set a
 , comprehension -- Eq a => Set a -> (a->Bool) -> Set a
 , intersection --  Eq a => Set a -> Set a -> Set a
 , difference --    Eq a => Set a -> Set a -> Set a
 , fromList --      Eq a => [a] -> Set a
 , toList --        Eq a => Set a -> [a]
 )
 where

data Eq a => Set a = S [a]

emptyset :: Eq a => Set a
emptyset = S []

singleton x = S [x]

el x (S xs) = x`elem`xs

union (S xs) (S ys) = S (xs ++ [y | y<-ys, not(y`elem`xs)])

comprehension (S xs) f = S [x | x<-xs, f x]

intersection set1 set2 = comprehension set1 (`el`set2)

difference set1 set2 = comprehension set1 (not.(`el`set2))

fromList xs = S (unique xs)
  where
  unique [] = []
  unique (x:xs) = x:unique (filter (x/=) xs)

toList (S xs) = xs

instance (Eq a,Show a) => Show (Set a) where
  show (S xs) = "{" ++ showElements (xs) ++ "}"
    where
    showElements [] = ""
    showElements (x:xs) = show x ++ showRest xs
    showRest [] = ""
```

```
  showRest (x:xs) = "," ++ show x ++ showRest xs
```

Βλέπουμε ότι έχουν υλοποιηθεί συναρτήσεις για δημιουργία κενού συνόλου (empty set), δημιουργία συνόλου ενός στοιχείου (singleton), ένωση (union), τομή (intersection), διαχωρισμό (comprehension) με βάση κάποια συνάρτηση φιλτραρίσματος καθώς και αφαίρεσης (difference). Η συνάρτηση `el` ελέγχει αν ένα στοιχείο βρίσκεται στο σύνολο. Η υλοποίησή μας διατηρεί την αναλλοίωτη ότι ένα στοιχείο του συνόλου εμφανίζεται μόνο μία φορά στη λίστα.

Στην υλοποίησή μας έχουμε επίσης προσθέσει συναρτήσεις `fromList` και `toList` που μετατρέπουν ένα σύνολο σε λίστα και αντίστροφα. Αυτό γίνεται για να δώσουμε στο χρήστη τη δυνατότητα να χρησιμοποιεί την πλούσια υποστήριξη της Haskell σε λίστες με τα σύνολά μας. Ο χρήστης δηλαδή μπορεί να πάρει τα στοιχεία ενός συνόλου ως λίστα, να επέμβει σε αυτά χρησιμοποιώντας συναρτήσεις για λίστες και να τα ξαναμετατρέψει σε σύνολο. Προσέξτε ότι η `toList` δεν είναι αντίστροφη της `fromList` και ούτε θα έπρεπε: η δομή του συνόλου ξεχνάει τη σειρά και τον πληθυσμό των στοιχείων.

Στην υλοποίησή μας έχουμε επίσης προσθέσει ακόμα ένα "δωράκι" για τον πελάτη. Έχουμε κάνει τον τύπο συνόλου στιγμιότυπο της κλάσης `Show` (αρκεί τα στοιχεία του να ανήκουν επίσης σε αυτήν την κλάση). Η συνάρτησή μας `show` για τα σύνολα τυπώνει ένα σύνολο στη μορφή

```
{x,y,z...}
```

Με αυτόν τον τρόπο μπορούμε να "δούμε" ένα σύνολο κατ' ευθείαν στον Hugs, π.χ. η αποτίμηση του

```
(fromList[1,2,3])`union`(fromList[2,3,4])
```

δίνει

```
{1,2,3,4}
```

Μία σημαντική επισήμανση που κάνουμε εδώ είναι ότι: *οι δηλώσεις στιγμιοτύπων και οι σχετικές συναρτήσεις για εξαγόμενους τύπους είναι πάντα εξαγόμενες*. Παρατηρήστε ότι δε χρειάστηκε να εξάγουμε τη `show` στο παραπάνω παράδειγμα.

2.4.2 Χρήση του `Set`

Ως παράδειγμα για τη χρήση του `Set` θα αλλάξουμε τον κώδικα διάσχισης κατά πλάτος της Ενότ. 2.3.2, ώστε να χρησιμοποιεί ένα σύνολο `marked` αντί για λίστα. Η αλλαγή αυτή καταδεικνύει στον αναγνώστη του προγράμματος ότι στη συλλογή `marked` δε μας ενδιαφέρει η σειρά και το πλήθος των στοιχείων, αλλά μόνο το αν ένα στοιχείο υπάρχει ή όχι μέσα στη συλλογή. Ο νέος κώδικας έχει ως εξής:

```
import Queue
import Set

type Graph = [[Int]]
```

```
enqueueMany q = foldl enqueue q

bfs :: Graph->Int->[Int]
bfsaux :: Graph->Queue Int->Set Int->[Int]

bfs g n = bfsaux g (enqueue emptyq n) (singleton n)

bfsaux g q marked =
  if (isempty q) then [] else bfsaux_q_non_empty g q marked
bfsaux_q_non_empty g q marked = [f] ++ bfsaux g q'' marked'
  where
  q' = dequeue q
  f = front q
  unmarkednb = [nb | nb<-g!!f, not (nb`el`marked)]
  q'' = enqueueMany q' unmarkednb
  marked' = marked `union` (fromList unmarkednb)
```

2.5 Σχεδίαση ΑΤΔ

Σε αυτήν την ενότητα θα αναφερθούμε σε γενικά θέματα σχεδίασης ενός ΑΤΔ. Το βασικό θέμα σχεδίασης είναι τι τιμές είναι χρήσιμες να συμπεριληφθούν στη διαπροσωπεία του τύπου. Χωρίζουμε τις τιμές σε μερικές κατηγορίες, ανάλογα με το ρόλο που παίζουν:

- Οι *κατασκευαστές (constructors)* είναι οι τιμές εκείνες των οποίων ο ρόλος είναι να κατασκευάζουν αντικείμενα του ΑΤΔ. Κατασκευαστές είναι οι `emptys`, `push` των στιβών, οι `emptyq`, `enqueue` από το τμήμα `Queue` και οι `emptyset`, `singleton`, `union` και `fromList` από το τμήμα `Set`. Ο κύριος στόχος όταν σχεδιάζουμε τους κατασκευαστές ενός ΑΤΔ είναι να μπορούμε με αυτούς να κατασκευάσουμε *όλες* τις τιμές του τύπου αυτού που μας ενδιαφέρουν.

  Παρατηρήστε ότι μέσα στους κατασκευαστές θα πρέπει να υπάρχει τουλάχιστον μία συνάρτηση που να κατασκευάζει μία τιμή του ΑΤΔ χωρίς να παίρνει μία άλλη τιμή του ΑΤΔ. Αλλιώς ο ΑΤΔ δε μπορεί να χρησιμοποιηθεί καθόλου, επειδή δε μπορεί να κατασκευαστεί καμία τιμή του. Τέτοιοι κατασκευαστές είναι οι `emptys`, `emptyq`, `emptyset`, `singleton` και `fromList`.

- Οι *παρατηρητές (observers)* είναι οι τιμές εκείνες των οποίων ο ρόλος είναι να εξάγουν ένα κομμάτι πληροφορίας από τον ΑΤΔ. Παρατηρητές είναι οι `top`, `isempty`, `count` από τις στίβες, οι `front`, `isempty` από το τμήμα `Queue` και οι `el`, `toList` από το τμήμα `Set`. Ο κύριος στόχος όταν σχεδιάζουμε παρατηρητές είναι να μπορούμε να δίνουμε πρόσβαση σε όλες τις πληροφορίες που μας ενδιαφέρει να εξάγουμε από τον τύπο. Η εξαγωγή όλων των δεδομένων μπορεί

να γίνει με συνδυασμό παρατηρητών και καταστροφέων (βλ. πιο κάτω). Για παράδειγμα, στις στίβες χρειαζόμαστε και την pop μαζί με την top για να έχουμε πρόσβαση σε όλα τα στοιχεία που περιέχονται στη λίστα. Ο ρόλος πολλών παρατηρητών είναι να διαχωρίσουν ιδιαίτερες περιπτώσεις του τύπου, όπως πχ. η κενή στίβα ή ουρά.

- Οι *μετατροπείς (modifiers)* είναι οι συναρτήσεις εκείνες που παίρνουν ένα αντικείμενο του ΑΤΔ, πιθανώς κάποια άλλα δεδομένα, και φτιάχνουν ένα καινούριο αντικείμενο του ΑΤΔ. Μετατροπείς είναι οι pop από τις στίβες, dequeue από το τμήμα Queue και comprehension από το τμήμα Set. Οι κατασκευαστές είναι διαφορετικοί από τους μετατροπείς όσον αφορά το ρόλο τους στον τύπο. Οι κατασκευαστές εξασφαλίζουν ότι όλες οι τιμές του τύπου είναι κατασκευάσιμες (π.χ. μπορούμε να φτιάξουμε όλες τις στίβες χρησιμοποιώντας μόνο emptys και push). Ο ρόλος των μετατροπέων είναι βοηθητικός: διευκολύνουν τη γρήγορη κατασκευή μίας τιμής ξεκινώντας από μία άλλη (όπως, π.χ. η συνάρτηση comprehension). Οι *καταστροφείς* είναι μια σημαντική ειδική περίπτωση των μετατροπέων.

- Οι *καταστροφείς (destructors)* είναι μετατροπείς που πηγαίνουν "ανάποδα" στη λογική της κατασκευής, παίρνοντας μία τιμή του ΑΔΤ και παράγοντας μία καινούρια κατά μία έννοια απλούστερη τιμή. Η pop των στιβών και η dequeue του τμήματος Queue είναι καταστροφείς. Σε πολλούς τύπους οι καταστροφείς χρησιμοποιούνται σε συνδυασμό με τους παρατηρητές για να δωθεί πρόσβαση σε όλα τα δεδομένα μίας τιμής του τύπου. Έτσι η pop μαζί με την top και την isempty δίνουν πρόσβαση σε όλα τα δεδομένα μίας στίβας, ενώ η dequeue με τις front,isempty δίνουν πρόσβαση σε όλα τα δεδομένα μίας ουράς.

- Οι *συνδυαστές (combinators)* είναι συναρτήσεις που εφαρμόζονται σε περισσότερες από μία τιμές του ΑΤΔ. Συνδυαστές είναι οι union, intersection, difference του τμήματος Set.

Όταν σχεδιάζουμε ένα ΑΤΔ, τυπικά σκεπτόμαστε με βάση τις παραπάνω κατηγορίες, φροντίζοντας για κάθε κατηγορία να υποστηρίζεται ικανοποιητικός αριθμός συναρτήσεων. Οι κατασκευαστές, οι παρατηρητές και οι καταστροφείς είναι απολύτως απαραίτητοι. Οι κατασκευαστές θα πρέπει να μπορούν να κατασκευάσουν όλες τις επιθυμητές τιμές. Οι παρατηρητές και οι καταστροφείς θα πρέπει να μπορούν να παράγουν όλες τις επιθυμητές πληροφορίες που θέλουμε να είναι προσβάσιμες στο χρήστη. Οι άλλες κατηγορίες εξαγόμενων τιμών μπορούν να θεωρηθούν βοηθητικές, αλλά η παρουσία τους συνεισφέρει πολλές φορές ουσιαστικά στο πόσο εύχρηστος είναι ένας ΑΤΔ.

2.6 Τυπικά Συμβόλαια

Σε αυτήν την ενότητα θα κάνουμε μία μικρή αναφορά στο ρόλο που παίζουν τα *τυπικά συμβόλαια (formal contracts)* στον ορισμό της διαπροσωπείας ενός τμήματος. Το θέμα

αυτό είναι πολύ μεγάλο και εν πολλοίς άσχετο με το συναρτησιακό προγραμματισμό, οπότε δε θα το συζητήσουμε αναλυτικά.

Ας υποθέσουμε ότι ένας πελάτης C χρησιμοποιεί ένα τμήμα S. Η συνηθισμένη περιγραφή της διαπροσωπείας του S είναι μόνο μία *υπογραφή τύπων (type signature)*, δηλαδή, η λίστα των εξαγόμενων ονομάτων του S μαζί με τους τύπους τους. Αυτό εξασφαλίζει στον πελάτη C ότι, αν χρησιμοποιήσει σωστά τα εξαγόμενα ονόματα, δε θα έχει σφάλματα τύπων. Αυτό όμως δεν είναι αρκετό για να μας εξασφαλίσει ότι η συμπεριφορά του C θα είναι ορθή, καθώς δεν είναι αρκετά περιοριστικό για την υλοποίηση του S.

Ας πάρουμε ένα πολύ απλό παράδειγμα χρήσης στίβας:

```
import Stack

value = top(pop(push (push emptys 3) 4))
```

Παρατηρήστε ότι η ορθή συμπεριφορά αυτού του κώδικα είναι να δώσει την τιμή `3` στη μεταβλητή `value`. (Φυσικά οι στίβες χρησιμοποιούνται σε πολύ πιο χρήσιμα προγράμματα από αυτό. Αυτό το παράδειγμα είναι επίτηδες απλουστευτικό). Η ορθή συμπεριφορά όμως δε μπορεί να εξασφαλιστεί από τη διαπροσωπεία που μόλις περιγράψαμε. Ο υλοποιητής στίβας είναι ελεύθερος να αυτοσχεδιάσει, με ορισμούς όπως, π.χ.

```
top _ = 42
```

που, αν και σέβονται τη διαπροσωπεία τύπων, δε σέβονται τη συμπεριφορά που πρέπει να έχει μία στίβα και μπορούν έτσι να προκαλέσουν σφάλμα στον πελάτη. Στο παρόν παράδειγμα, ο πελάτης περιμένει την τιμή `3` και παίρνει την τιμή `42`.

Οι υπογραφές τύπων δεν αρκούν για να περιγράψουν την πλήρη διαπροσωπεία ενός τμήματος. Χρειάζεται ένας τρόπος περιγραφής της συμπεριφοράς των τιμών που ορίζονται μέσα στο τμήμα. Αυτός ακριβώς είναι ο ρόλος του *τυπικού συμβολαίου*. Το τυπικό συμβόλαιο είναι μία μαθηματική έκφραση τύπου αληθοτιμής στην οποία εμφανίζονται τα εξαγόμενα ονόματα ενός τμήματος. Η έκφραση αυτή περιγράφει την *επιθυμητή συμπεριφορά του τμήματος*. Ο υλοποιητής του τμήματος οφείλει να *αποδείξει ότι οι ορισμοί του συνεπάγονται το συμβόλαιο*. Ο πελάτης μπορεί να *υποθέσει ότι το συμβόλαιο ισχύει* ώστε να προχωρήσει στις δικές του αποδείξεις ορθότητας.

Στο παράδειγμά μας, ένα συμβόλαιο που μπορεί να περιγράψει τη συμπεριφορά της στίβας είναι το παρακάτω (υπονοείται ότι αυτό ισχύει για κάθε `s::Stack a, x::a`):

```
top(push s x)=x  &&  pop(push s x)=s
```

Με αυτό το συμβόλαιο, ο υλοποιητής δε μπορεί να κάνει αυθαίρετες υλοποιήσεις, π.χ. η `top` δε μπορεί να είναι μία σταθερή συνάρτηση. Ο υλοποιητής οφείλει να αποδείξει ότι η υλοποίησή του *ικανοποιεί*, με άλλα λόγια *συνεπάγεται λογικά* το συμβόλαιο.

Για παράδειγμα, η υλοποίηση στίβας της Ενότ. 2.2.1 ικανοποιεί το συμβόλαιο:

```
top(push(S l)x) = top(S(x:l)) = x
```

και

```
pop(push(S l)x) = pop(S(x:l)) = S l
```

και άρα μπορεί να χρησιμοποιηθεί.

Ο πελάτης τώρα μπορεί να χρησιμοποιήσει το συμβόλαιο στις δικές του αποδείξεις, πχ. μπορεί να αποδείξει ότι `value = 3` ως εξής:

```
  value
= top(pop(push (push emptys 3) 4))
= top(push emptys 3)
= 3
```

Ο υλοποιητής είναι ελεύθερος να αλλάξει την υλοποίηση του τμήματος, αλλά θα πρέπει να αποδείξει ότι το συμβόλαιο συνεχίζει να ικανοποιείται. Εφόσον οι αποδείξεις ορθότητας του πελάτη έχουν στηριχθεί μόνο στο συμβόλαιο και όχι σε λεπτομέρειες υλοποίησης (που ο πελάτης δεν ξέρει), ο υλοποιητής είναι σίγουρος ότι η αλλαγή υλοποίησης δε θα επηρρεάσει τον πελάτη.

Ο ρόλος ενός νομικού συμβολαίου είναι να ξεκαθαρίζει, σε περίπτωση που κάτι πάει στραβά σε μία συμφωνία, ποιος έχει δίκιο και ποιος έχει άδικο. Στην περίπτωση του προγραμματισμού, το συμβόλαιο ξεκαθαρίζει, σε περίπτωση που το πρόγραμμα δε δουλεύει όπως περιμένουμε, σε ποιο τμήμα βρίσκεται το λάθος. Αν ο υλοποιητής ενός τμήματος δε μπορεί να αποδείξει ότι η υλοποίησή του ικανοποιεί το συμβόλαιο του τμήματος, τότε το λάθος βρίσκεται μέσα στο τμήμα. Από την άλλη, αν ο πελάτης δε μπορεί, χρησιμοποιώντας το συμβόλαιο, να αποδείξει την ορθότητα του δικού του κομματιού, τότε το λάθος βρίσκεται στον πελάτη.

3 Επισκόπηση

Στις σημειώσεις αυτές είδαμε τα εξής:

- Η *τμηματοποίηση* είναι ο διαχωρισμός του λογισμικού σε τμήματα όσο το δυνατό πιο ανεξάρτητα μεταξύ τους.
- Η τμηματοποίηση είναι θεμελειώδης στην ανάπτυξη λογισμικού σχετικά μεγάλης πολυπλοκότητας.
- Η ανεξαρτησία των τμημάτων επιτυγχάνεται μέσω της *απόκρυψης πληροφορίας* και συγκεκριμένα μέσω της απόκρυψης των *λεπτομεριών υλοποίησης*. Η απόκρυψη πληροφορίας επιτυγχάνεται μέσω της απόκρυψης μερικών *ονομάτων* ενός τμήματος από άλλα.
- Τα ονόματα ενός τμήματος που φαίνονται έξω από το τμήμα ονομάζονται *εξαγόμενα*.
- Η *διαπροσωπεία* ενός τμήματος περιλαμβάνει τα εξαγόμενα ονόματα μαζί με τους τύπους τους και μια περιγραφή για τη συμπεριφορά τους.
- Ένα τμήμα C *εισάγει* ένα άλλο τμήμα S για να έχει πρόσβαση στα εξαγόμενα ονόματα του S. Το τμήμα C λέγεται *πελάτης* ή *χρήστης* του S.

- Ο Hugs υποστηρίζει τμηματοποίηση πολύ όμοια με αυτήν της Java. Λεπτομέρειες για τη χρήση των τμημάτων στον Hugs υπάρχουν στις σημειώσεις αυτές, αλλά και στην online βοήθεια για τον Hugs [Hug98].
- Ένας *αφηρημένος τύπος δεδομένων - ΑΤΔ* στη Haskell είναι ένας αλγεβρικός τύπος που εξάγεται χωρίς τους κατασκευαστές του.
- Η πρόσβαση σε έναν ΑΤΔ γίνεται μόνο μέσω συναρτήσεων που έχει ορίσει ο υλοποιητής του για αυτόν το σκοπό
- Ο περιορισμός πρόσβασης στους ΑΤΔ χρησιμεύει στο να μη χρησιμοποιούνται οι τιμές του με τρόπο που θέλει να απαγορεύσει ο υλοποιητής. Μία σημαντική υπο-περίπτωση αυτού του σημείου είναι η διατήρηση των *αναλλοίωτων* των εσωτερικών δεδομένων ενός ΑΤΔ. Μία αναλλοίωτη είναι μία συνθήκη για τα εσωτερικά δεδομένα, η οποία θα πρέπει να ισχύει για κάθε τιμή του ΑΤΔ.
- Ο περιορισμός πρόσβασης δίνει επίσης τη δυνατότητα στον υλοποιητή να αλλάξει την υλοποίησή του χωρίς να επηρρεάσει το υπόλοιπο πρόγραμμα, αρκεί η νέα υλοποίηση να σέβεται τη διαπροσωπεία του ΑΤΔ.
- Οι ΑΤΔ μας βοηθάνε να αναπαραστήσουμε αφηρημένα μαθηματικά δεδομένα, επειδή διαχωρίζουν τις συμβολιζόμενες οντότητες από την υλοποίησή τους.
- Στο σχεδιασμό ενός ΑΤΔ ξεχωρίζουμε τις εξής κατηγορίες τιμών: κατασκευαστές, παρατηρητές, μετατροπείς, καταστροφείς και συνδυαστές.
- Οι κατασκευαστές θα πρέπει να μπορούν να κατασκευάσουν από μόνοι τους οποιαδήποτε επιθυμητή τιμή του ΑΤΔ.
- Οι παρατηρητές, σε συνδυασμό με τους καταστροφείς θα πρέπει να μπορούν να εξάγουν οποιαδήποτε πληροφορία θέλουμε από τον ΑΤΔ.
- Οι υπόλοιπες συναρτήσεις μπορούν να χαρακτηριστούν βοηθητικές στη χρήση του ΑΤΔ.
- Ένα *τυπικό συμβόλαιο* είναι μία συνθήκη που περιγράφει με μαθηματικό τρόπο τη συμπεριφορά ενός ΑΤΔ.
- Ο υλοποιητής έχει υποχρέωση να αποδείξη ικανοποίηση του συμβολαίου, δηλαδή ότι οι ορισμοί του συνεπάγονται το συμβόλαιο.
- Ο χρήστης είναι ελεύθερος να χρησιμοποιήσει το συμβόλαιο, αλλά όχι την υλοποίηση για τις δικές του αποδείξεις.
- Μετατροπές υλοποίησης γίνονται μόνο με τρόπο που να διατηρεί την ικανοποίηση του συμβολαίου.

Αναφορές

[Hug98] The hugs 98 user's guide, 1998. Online: `http://cvs.haskell.org/Hugs/pages/users_guide/index.html` .