Τμηματοποίηση και Αφηρημένοι Τύποι Δεδομένων

Γιάννης Κασσιός

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Δομή Σημερινής Διάλεξης

- Τμηματοποίηση
  - εισαγωγή / ορολογία
  - τμήματα στον Hugs
- Αφηρημένοι Τύποι Δεδομένων (ΑΤΔ)
  - εισαγωγή
  - παράδειγμα υλοποίησης και χρήσης
    - παράδειγμα αποφυγής κακής χρήσης της υλοποίησης
  - παράδειγμα αλλαγής υλοποίησης
  - παράδειγμα τύπου υψηλού επιπέδου
  - σχεδίαση ΑΤΔ
  - συμβόλαια

Τμηματοποίηση

- Τμηματοποίηση (modularization): διαχωρισμός κώδικα σε διακριτά τμήματα (modules) όσο πιο ανεξάρτητα μεταξύ τους
  - ανάπτυξη λογισμικού από ανεξάρτητες ομάδες
  - οι αλλαγές σε ένα τμήμα δεν επεκτείνονται
  - επαναχρησιμοποίηση τμήματος
- Βασικό στοιχείο: απόκρυψη πληροφορίας (information hiding)
  - απόκρυψη λεπτομερειών υλοποίησης (implementation details)
  - πρακτικά: απόκρυψη μερικών ονομάτων ενός τμήματος

Τμηματοποίηση

- Τμηματοποίηση (modularization): διαχωρισμός κώδικα σε διακριτά τμήματα (modules) όσο πιο ανεξάρτητα μεταξύ τους
  - ανάπτυξη λογισμικού από ανεξάρτητες ομάδες
  - οι αλλαγές σε ένα τμήμα δεν επεκτείνονται
  - επαναχρησιμοποίηση τμήματος
- Βασικό στοιχείο: απόκρυψη πληροφορίας (information hiding)
  - απόκρυψη λεπτομερειών υλοποίησης (implementation details)
  - πρακτικά: απόκρυψη μερικών ονομάτων ενός τμήματος

Τμηματοποίηση

- Τμηματοποίηση (modularization): διαχωρισμός κώδικα σε διακριτά τμήματα (modules) όσο πιο ανεξάρτητα μεταξύ τους
  - ανάπτυξη λογισμικού από ανεξάρτητες ομάδες
  - οι αλλαγές σε ένα τμήμα δεν επεκτείνονται
  - επαναχρησιμοποίηση τμήματος
- Βασικό στοιχείο: απόκρυψη πληροφορίας (information hiding)
  - απόκρυψη λεπτομερειών υλοποίησης (implementation details)
  - πρακτικά: απόκρυψη μερικών ονομάτων ενός τμήματος

Τμηματοποίηση

- Τμηματοποίηση (modularization): διαχωρισμός κώδικα σε διακριτά τμήματα (modules) όσο πιο ανεξάρτητα μεταξύ τους
  - ανάπτυξη λογισμικού από ανεξάρτητες ομάδες
  - οι αλλαγές σε ένα τμήμα δεν επεκτείνονται
  - επαναχρησιμοποίηση τμήματος
- Βασικό στοιχείο: απόκρυψη πληροφορίας (information hiding)
  - απόκρυψη λεπτομερειών υλοποίησης (implementation details)
  - πρακτικά: απόκρυψη μερικών ονομάτων ενός τμήματος

Τμηματοποίηση

- Τμηματοποίηση (modularization): διαχωρισμός κώδικα σε διακριτά τμήματα (modules) όσο πιο ανεξάρτητα μεταξύ τους
  - ανάπτυξη λογισμικού από ανεξάρτητες ομάδες
  - οι αλλαγές σε ένα τμήμα δεν επεκτείνονται
  - επαναχρησιμοποίηση τμήματος
- Βασικό στοιχείο: απόκρυψη πληροφορίας (information hiding)
  - απόκρυψη λεπτομερειών υλοποίησης (implementation details)
  - πρακτικά: απόκρυψη μερικών ονομάτων ενός τμήματος

Τμηματοποίηση

- Τμηματοποίηση (modularization): διαχωρισμός κώδικα σε διακριτά τμήματα (modules) όσο πιο ανεξάρτητα μεταξύ τους
  - ανάπτυξη λογισμικού από ανεξάρτητες ομάδες
  - οι αλλαγές σε ένα τμήμα δεν επεκτείνονται
  - επαναχρησιμοποίηση τμήματος
- Βασικό στοιχείο: απόκρυψη πληροφορίας (information hiding)
  - απόκρυψη λεπτομερειών υλοποίησης (implementation details)
  - πρακτικά: απόκρυψη μερικών ονομάτων ενός τμήματος

- Εξαγόμενα (exported) ονόματα: ονόματα ενός τμήματος που φαίνονται έξω από αυτό
- Διαπροσωπεία (interface): εξαγόμενα ονόματα + τύποι + προδιαγραφές
- Εισαγωγή (importation) τμήματος *S* από τμήμα *C*
  - χρήση των ονομάτων + ορισμών του *S* μέσα στο *C*
  - μόνο τα εξαγόμενα ονόματα του *C* είναι προσβάσιμα στο *S*
  - το *C* είναι πελάτης (client) ή χρήστης (user) του *S*

- Εξαγόμενα (exported) ονόματα: ονόματα ενός τμήματος που φαίνονται έξω από αυτό
- Διαπροσωπεία (interface): εξαγόμενα ονόματα + τύποι + προδιαγραφές
- Εισαγωγή (importation) τμήματος *S* από τμήμα *C*
  - χρήση των ονομάτων + ορισμών του *S* μέσα στο *C*
  - μόνο τα εξαγόμενα ονόματα του *C* είναι προσβάσιμα στο *S*
  - το *C* είναι πελάτης (client) ή χρήστης (user) του *S*

- Εξαγόμενα (exported) ονόματα: ονόματα ενός τμήματος που φαίνονται έξω από αυτό
- Διαπροσωπεία (interface): εξαγόμενα ονόματα + τύποι + προδιαγραφές
- Εισαγωγή (importation) τμήματος *S* από τμήμα *C*
  - χρήση των ονομάτων + ορισμών του *S* μέσα στο *C*
  - μόνο τα εξαγόμενα ονόματα του *C* είναι προσβάσιμα στο *S*
  - το *C* είναι πελάτης (client) ή χρήστης (user) του *S*

- Εξαγόμενα (exported) ονόματα: ονόματα ενός τμήματος που φαίνονται έξω από αυτό
- Διαπροσωπεία (interface): εξαγόμενα ονόματα + τύποι + προδιαγραφές
- Εισαγωγή (importation) τμήματος *S* από τμήμα *C*
  - χρήση των ονομάτων + ορισμών του *S* μέσα στο *C*
  - μόνο τα εξαγόμενα ονόματα του *C* είναι προσβάσιμα στο *S*
  - το *C* είναι πελάτης (client) ή χρήστης (user) του *S*

- Εξαγόμενα (exported) ονόματα: ονόματα ενός τμήματος που φαίνονται έξω από αυτό
- Διαπροσωπεία (interface): εξαγόμενα ονόματα + τύποι + προδιαγραφές
- Εισαγωγή (importation) τμήματος *S* από τμήμα *C*
  - χρήση των ονομάτων + ορισμών του *S* μέσα στο *C*
  - μόνο τα εξαγόμενα ονόματα του *C* είναι προσβάσιμα στο *S*
  - το *C* είναι πελάτης (client) ή χρήστης (user) του *S*

- Hugs: αρχείο = τμήμα
  - αντίθετα με τη Haskell
- Μονοπάτι αναζήτησης (search path): σύνολο καταλόγων που ψάχνει ο Hugs για τμήματα
  - `-P` στη γραμμή εντολών
  - αλλαγή επιλογών γραμμής εντολών μέσα στο Hugs: εντολή `:s`
- Ονόματα τμημάτων / ονόματα αρχείων:
  - όνομα τμήματος `A` → όνομα αρχείου `A.hs`
  - όνομα τμήματος `A.B.C` → όνομα αρχείου `A/B/C.hs`

- Hugs: αρχείο = τμήμα
  - αντίθετα με τη Haskell
- Μονοπάτι αναζήτησης (search path): σύνολο καταλόγων που ψάχνει ο Hugs για τμήματα
  - `-P` στη γραμμή εντολών
  - αλλαγή επιλογών γραμμής εντολών μέσα στο Hugs: εντολή `:s`
- Ονόματα τμημάτων / ονόματα αρχείων:
  - όνομα τμήματος `A` → όνομα αρχείου `A.hs`
  - όνομα τμήματος `A.B.C` → όνομα αρχείου `A/B/C.hs`

- Hugs: αρχείο = τμήμα
  - αντίθετα με τη Haskell
- Μονοπάτι αναζήτησης (search path): σύνολο καταλόγων που ψάχνει ο Hugs για τμήματα
  - `-P` στη γραμμή εντολών
  - αλλαγή επιλογών γραμμής εντολών μέσα στο Hugs: εντολή `:s`
- Ονόματα τμημάτων / ονόματα αρχείων:
  - όνομα τμήματος `A` → όνομα αρχείου `A.hs`
  - όνομα τμήματος `A.B.C` → όνομα αρχείου `A/B/C.hs`

- Hugs: αρχείο = τμήμα
  - αντίθετα με τη Haskell
- Μονοπάτι αναζήτησης (search path): σύνολο καταλόγων που ψάχνει ο Hugs για τμήματα
  - `-P` στη γραμμή εντολών
  - αλλαγή επιλογών γραμμής εντολών μέσα στο Hugs: εντολή `:s`
- Ονόματα τμημάτων / ονόματα αρχείων:
  - όνομα τμήματος `A` → όνομα αρχείου `A.hs`
  - όνομα τμήματος `A.B.C` → όνομα αρχείου `A/B/C.hs`

- Hugs: αρχείο = τμήμα
  - αντίθετα με τη Haskell
- Μονοπάτι αναζήτησης (search path): σύνολο καταλόγων που ψάχνει ο Hugs για τμήματα
  - `-P` στη γραμμή εντολών
  - αλλαγή επιλογών γραμμής εντολών μέσα στο Hugs: εντολή `:s`
- Ονόματα τμημάτων / ονόματα αρχείων:
  - όνομα τμήματος `A` → όνομα αρχείου `A.hs`
  - όνομα τμήματος `A.B.C` → όνομα αρχείου `A/B/C.hs`

- Hugs: αρχείο = τμήμα
  - αντίθετα με τη Haskell
- Μονοπάτι αναζήτησης (search path): σύνολο καταλόγων που ψάχνει ο Hugs για τμήματα
  - `-P` στη γραμμή εντολών
  - αλλαγή επιλογών γραμμής εντολών μέσα στο Hugs: εντολή `:s`
- Ονόματα τμημάτων / ονόματα αρχείων:
  - όνομα τμήματος `A` → όνομα αρχείου `A.hs`
  - όνομα τμήματος `A.B.C` → όνομα αρχείου `A/B/C.hs`

Τμηματοποίηση
Δήλωση Τμήματος

- Σύνταξη:
  `module` *όνομα_τμήματος* `(`*υπογραφή_εξόδου*`) where`
  ορισμοί
  - το όνομα τμήματος ξεκινάει με κεφαλαίο
  - οι ορισμοί ξεκινάνε στην ίδια στήλη
  - υπογραφή εξόδου: εξαγόμενα ονόματα χωρισμένα με κόμμα

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

```
a = 1
b = \x->3+x

type T = (String,String)
```

Τμηματοποίηση
Δήλωση Τμήματος

- Σύνταξη:
  `module` *όνομα_τμήματος* `(`*υπογραφή_εξόδου*`) where`
  ορισμοί
  - το όνομα τμήματος ξεκινάει με κεφαλαίο
  - οι ορισμοί ξεκινάνε στην ίδια στήλη
  - υπογραφή εξόδου: εξαγόμενα ονόματα χωρισμένα με κόμμα

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

```
a = 1
b = \x->3+x

type T = (String,String)
```

- Σύνταξη:
  `module` *όνομα_τμήματος* `(`*υπογραφή_εξόδου*`) where`
  ορισμοί
  - το όνομα τμήματος ξεκινάει με κεφαλαίο
  - οι ορισμοί ξεκινάνε στην ίδια στήλη
  - υπογραφή εξόδου: εξαγόμενα ονόματα χωρισμένα με κόμμα

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

```
a = 1
b = \x->3+x

type T = (String,String)
```

- Σύνταξη:
  `module` *όνομα_τμήματος* `(`*υπογραφή_εξόδου*`) where`
  ορισμοί
  - το όνομα τμήματος ξεκινάει με κεφαλαίο
  - οι ορισμοί ξεκινάνε στην ίδια στήλη
  - υπογραφή εξόδου: εξαγόμενα ονόματα χωρισμένα με κόμμα

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

```
a = 1
b = \x->3+x

type T = (String,String)
```

- Σύνταξη:
  `module` *όνομα_τμήματος* `(`*υπογραφή_εξόδου*`) where`
  ορισμοί
  - το όνομα τμήματος ξεκινάει με κεφαλαίο
  - οι ορισμοί ξεκινάνε στην ίδια στήλη
  - υπογραφή εξόδου: εξαγόμενα ονόματα χωρισμένα με κόμμα

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

```
a = 1
b = \x->3+x

type T = (String,String)
```

- Σύνταξη:
  `module` *όνομα_τμήματος* `(`*υπογραφή_εξόδου*`) where`
  ορισμοί
  - το όνομα τμήματος ξεκινάει με κεφαλαίο
  - οι ορισμοί ξεκινάνε στην ίδια στήλη
  - υπογραφή εξόδου: εξαγόμενα ονόματα χωρισμένα με κόμμα

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

```
a = 1
b = \x->3+x

type T = (String,String)
```

- Οι κατασκευαστές αλγεβρικών τύπων δεν εξάγονται ανεξάρτητα:

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

- Εξαγωγή όλων των κατασκευαστών: `T(..)`
- Εξαγωγή τύπου χωρίς τους κατασκευαστές: αφηρημένος τύπος δεδομένων - ΑΤΔ (abstract data type - ADT)

- Οι κατασκευαστές αλγεβρικών τύπων δεν εξάγονται ανεξάρτητα:

```
module B (T(C1,C2)) where

data T = C1 Int | C2 Int Int | C3 String
```

- Εξαγωγή όλων των κατασκευαστών: `T(..)`
- Εξαγωγή τύπου χωρίς τους κατασκευαστές: αφηρημένος τύπος δεδομένων - ΑΤΔ (abstract data type - ADT)

- Οι κατασκευαστές αλγεβρικών τύπων δεν εξάγονται ανεξάρτητα:

```
module B (T(C1,C2)) where

data T = C1 Int | C2 Int Int | C3 String
```

- Εξαγωγή όλων των κατασκευαστών: `T(..)`
- Εξαγωγή τύπου χωρίς τους κατασκευαστές: αφηρημένος τύπος δεδομένων - ΑΤΔ (abstract data type - ADT)

- Παράλειψη υπογραφής εξόδου: όλα τα ονόματα που δηλώνονται στο τμήμα εξάγονται

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

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

- Παράλειψη υπογραφής εξόδου: όλα τα ονόματα που δηλώνονται στο τμήμα εξάγονται

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

- Εισαγωγή τμήματος *S*:
  `import` *S*
- Παράδειγμα:

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

- D πελάτης του A:

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

- Περιορισμένη εισαγωγή:

```
import A(b,T)
import A hiding (a)
```

- Εισαγωγή τμήματος *S*:
  `import S`
- Παράδειγμα:

```
module D where

import A

c = b a
```

- D πελάτης του A:
  `c = b a = (\x->3+x)1 = 4`
- Περιορισμένη εισαγωγή:
  `import A(b,T)`
  `import A hiding (a)`

- Εισαγωγή τμήματος *S*:
  `import S`
- Παράδειγμα:

```
module D where

import A

c = b a
```

- D πελάτης του A:
  `c = b a = (\x->3+x)1 = 4`
- Περιορισμένη εισαγωγή:
  `import A(b,T)`
  `import A hiding (a)`

- Εισαγωγή τμήματος *S*:
  `import S`
- Παράδειγμα:

```
module D where

import A

c = b a
```

- `D` πελάτης του `A`:
  `c = b a = (\x->3+x)1 = 4`
- Περιορισμένη εισαγωγή:
  `import A(b,T)`
  `import A hiding (a)`

- Εισαγωγή τμήματος *S*:
  `import S`
- Παράδειγμα:

```
module D where

import A

c = b a
```

- `D` πελάτης του `A`:
  `c = b a = (\x->3+x)1 = 4`
- Περιορισμένη εισαγωγή:

```
import A(b,T)

import A hiding (a)
```

- Προσδιορισμένη (qualified) σύνταξη ονόματος: $M.n$
- Προσδιορισμένη Εισαγωγή:

```
import qualified S
import S as S'
import qualified S as S'
```

```
import qualified A as AA
c = AA.b AA.a
```

- Προσδιορισμένη (qualified) σύνταξη ονόματος: $M.n$
- Προσδιορισμένη Εισαγωγή:

```
import qualified S
import S as S'
import qualified S as S'
```

```
import qualified A as AA
c = AA.b AA.a
```

- Προσδιορισμένη (qualified) σύνταξη ονόματος: $M.n$
- Προσδιορισμένη Εισαγωγή:

```
import qualified S
import S as S'
import qualified S as S'
```

```
import qualified A as AA
c = AA.b AA.a
```

- Χωρίς υπογραφή εξόδου: μόνο ονόματα που ορίζονται μέσα στο τμήμα εξάγονται:

```
module J (T,b,c) where

import A

c = "Hi"
```

εξάγει μόνο `c`

εξάγει και `b,T` από `A`

- Εξαγωγή όλων των ονομάτων ενός τμήματος:

```
module K (module A,c) where

import A

c = "Hi"
```

- Χωρίς υπογραφή εξόδου: μόνο ονόματα που ορίζονται μέσα στο τμήμα εξάγονται:

```
module J (T,b,c) where

import A

c = "Hi"
```

εξάγει μόνο `c`

εξάγει και `b,T` από `A`

- Εξαγωγή όλων των ονομάτων ενός τμήματος:

```
module K (module A,c) where

import A

c = "Hi"
```

- Χωρίς υπογραφή εξόδου: μόνο ονόματα που ορίζονται μέσα στο τμήμα εξάγονται:

```
module J (T,b,c) where

import A

c = "Hi"
```

εξάγει μόνο `c`

εξάγει και `b,T` από `A`

- Εξαγωγή όλων των ονομάτων ενός τμήματος:

```
module K (module A,c) where

import A

c = "Hi"
```

Τμηματοποίηση

`Prelude` και `Main`

- Τμήμα `Prelude`:
  - περιέχει τους ορισμούς βασικών τιμών της Haskell
  - εισάγεται αυτόματα
  - τροποποίηση εισαγωγής: με `import hiding` και `import qualified`
- Τμήμα `Main`:
  - το όνομα ενός τμήματος, όταν δεν υπάρχει δήλωση `module`
  - μεταγλώττιση: η εκτέλεση είναι η αποτίμηση της `Main.main`

Τμηματοποίηση

`Prelude` και `Main`

- Τμήμα `Prelude`:
  - περιέχει τους ορισμούς βασικών τιμών της Haskell
  - εισάγεται αυτόματα
  - τροποποίηση εισαγωγής: με `import hiding` και `import qualified`
- Τμήμα `Main`:
  - το όνομα ενός τμήματος, όταν δεν υπάρχει δήλωση `module`
  - μεταγλώττιση: η εκτέλεση είναι η αποτίμηση της `Main.main`

- Αφηρημένος Τύπος Δεδομένων - ΑΔΤ (Abstract Data Type - ADT): τύπος του οποίου η υλοποίηση είναι άγνωστη στο χρήστη
  - εξάγουμε όνομα αλγεβρικού τύπου χωρίς τους κατασκευαστές
- ΑΤΔ: πρόσβαση μόνο μέσω συναρτήσεων που έχει ορίσει ο υλοποιητής
  - ο χρήστης δε μπορεί να χρησιμοποιήσει ΑΤΔ με τρόπο "που δε θέλουμε"
  - ο υλοποιητής έχει ελευθερία αλλαγής υλοποίησης
  - υψηλότερου επιπέδου περιγραφή
- Άλλα θέματα:
  - σχεδιασμός
  - διαπροσωπεία με συμβόλαια

- Αφηρημένος Τύπος Δεδομένων - ΑΔΤ (Abstract Data Type - ADT): τύπος του οποίου η υλοποίηση είναι άγνωστη στο χρήστη
  - εξάγουμε όνομα αλγεβρικού τύπου χωρίς τους κατασκευαστές
- ΑΤΔ: πρόσβαση μόνο μέσω συναρτήσεων που έχει ορίσει ο υλοποιητής
  - ο χρήστης δε μπορεί να χρησιμοποιήσει ΑΤΔ με τρόπο "που δε θέλουμε"
  - ο υλοποιητής έχει ελευθερία αλλαγής υλοποίησης
  - υψηλότερου επιπέδου περιγραφή
- Άλλα θέματα:
  - σχεδιασμός
  - διαπροσωπεία με συμβόλαια

- Αφηρημένος Τύπος Δεδομένων - ΑΔΤ (Abstract Data Type - ADT): τύπος του οποίου η υλοποίηση είναι άγνωστη στο χρήστη
  - εξάγουμε όνομα αλγεβρικού τύπου χωρίς τους κατασκευαστές
- ΑΤΔ: πρόσβαση μόνο μέσω συναρτήσεων που έχει ορίσει ο υλοποιητής
  - ο χρήστης δε μπορεί να χρησιμοποιήσει ΑΤΔ με τρόπο "που δε θέλουμε"
  - ο υλοποιητής έχει ελευθερία αλλαγής υλοποίησης
  - υψηλότερου επιπέδου περιγραφή
- Άλλα θέματα:
  - σχεδιασμός
  - διαπροσωπεία με συμβόλαια

- Αφηρημένος Τύπος Δεδομένων - ΑΔΤ (Abstract Data Type - ADT): τύπος του οποίου η υλοποίηση είναι άγνωστη στο χρήστη
  - εξάγουμε όνομα αλγεβρικού τύπου χωρίς τους κατασκευαστές
- ΑΤΔ: πρόσβαση μόνο μέσω συναρτήσεων που έχει ορίσει ο υλοποιητής
  - ο χρήστης δε μπορεί να χρησιμοποιήσει ΑΤΔ με τρόπο "που δε θέλουμε"
  - ο υλοποιητής έχει ελευθερία αλλαγής υλοποίησης
  - υψηλότερου επιπέδου περιγραφή
- Άλλα θέματα:
  - σχεδιασμός
  - διαπροσωπεία με συμβόλαια

- Αφηρημένος Τύπος Δεδομένων - ΑΔΤ (Abstract Data Type - ADT): τύπος του οποίου η υλοποίηση είναι άγνωστη στο χρήστη
  - εξάγουμε όνομα αλγεβρικού τύπου χωρίς τους κατασκευαστές
- ΑΤΔ: πρόσβαση μόνο μέσω συναρτήσεων που έχει ορίσει ο υλοποιητής
  - ο χρήστης δε μπορεί να χρησιμοποιήσει ΑΤΔ με τρόπο "που δε θέλουμε"
  - ο υλοποιητής έχει ελευθερία αλλαγής υλοποίησης
  - υψηλότερου επιπέδου περιγραφή
- Άλλα θέματα:
  - σχεδιασμός
  - διαπροσωπεία με συμβόλαια

- Αφηρημένος Τύπος Δεδομένων - ΑΔΤ (Abstract Data Type - ADT): τύπος του οποίου η υλοποίηση είναι άγνωστη στο χρήστη
  - εξάγουμε όνομα αλγεβρικού τύπου χωρίς τους κατασκευαστές
- ΑΤΔ: πρόσβαση μόνο μέσω συναρτήσεων που έχει ορίσει ο υλοποιητής
  - ο χρήστης δε μπορεί να χρησιμοποιήσει ΑΤΔ με τρόπο "που δε θέλουμε"
  - ο υλοποιητής έχει ελευθερία αλλαγής υλοποίησης
  - υψηλότερου επιπέδου περιγραφή
- Άλλα θέματα:
  - σχεδιασμός
  - διαπροσωπεία με συμβόλαια

Αφηρημένοι Τύποι Δεδομένων
Μη Πρόσβαση στην Υλοποίηση: Στίβες

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

Αφηρημένοι Τύποι Δεδομένων
Μη Πρόσβαση στην Υλοποίηση: Στίβες

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

Αφηρημένοι Τύποι Δεδομένων
Μη Πρόσβαση στην Υλοποίηση: Στίβες

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

Αφηρημένοι Τύποι Δεδομένων
Μη Πρόσβαση στην Υλοποίηση: Στίβες

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

Αφηρημένοι Τύποι Δεδομένων
Μη Πρόσβαση στην Υλοποίηση: Στίβες

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

- Στίβα (stack): σειριακή LIFO (last in first out) δομή δεδομένων

```
module Stack(Stack,emptys,push,top,pop,isempty)
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==[]
```

- Απαγορεύεται:

```
bad (S s) = S (take 10 s ++ drop 11 s)
```

```
module CStack(Stack,emptys,push,top,pop,isempty)
where
data Stack 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 s) = s==[]
isempty (S _ n) = n==0
count (S _ n) = n
```

```
module CStack(Stack,emptys,push,top,pop,isempty)
where
data Stack 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
```

- Αναλλοίωτη (invariant) για στίβα `S l n`:

  `n == length l`
- Απαγορεύεται:

  `badstack = S [] (-1)`
- Παράδειγμα κακής συμπεριφοράς:

  `isempty (push badstack 10)`

  επιστρέφει `True`

- Αναλλοίωτη (invariant) για στίβα `S l n`:

  `n == length l`
- Απαγορεύεται:

  `badstack = S [] (-1)`
- Παράδειγμα κακής συμπεριφοράς:

  `isempty (push badstack 10)`

  επιστρέφει `True`

- Αναλλοίωτη (invariant) για στίβα `S l n`:
  `n == length l`
- Απαγορεύεται:
  `badstack = S [] (-1)`
- Παράδειγμα κακής συμπεριφοράς:
  `isempty (push badstack 10)`
  επιστρέφει `True`

- Τεχνολόγηση Συμβολοσειρών (string parsing): ανήκει μια συμβολοσειρά σε μια γραμματική;
- Γραμματική σταθμισμένων παρενθέσεων:
  - αλφάβητο: `[]()`
  - ό,τι παρένθεση ανοίγει κλείνει
  - δύο σταθμισμένες υπο-συμβολοσειρές είναι φωλιασμένες ή ξένες
- Τυπικός ορισμός (BNF):

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

- Τεχνολόγηση Συμβολοσειρών (string parsing): ανήκει μια συμβολοσειρά σε μια γραμματική;
- Γραμματική σταθμισμένων παρενθέσεων:
  - αλφάβητο: `[]()`
  - ό,τι παρένθεση ανοίγει κλείνει
  - δύο σταθμισμένες υπο-συμβολοσειρές είναι φωλιασμένες ή ξένες
- Τυπικός ορισμός (BNF):

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

- Τεχνολόγηση Συμβολοσειρών (string parsing): ανήκει μια συμβολοσειρά σε μια γραμματική;
- Γραμματική σταθμισμένων παρενθέσεων:
  - αλφάβητο: `[]()`
  - ό,τι παρένθεση ανοίγει κλείνει
  - δύο σταθμισμένες υπο-συμβολοσειρές είναι φωλιασμένες ή ξένες
- Τυπικός ορισμός (BNF):

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

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop: `[((])](() [(()))()`

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop: `[((])]() [(()))()`

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:
    `[(()]() [(())))()`

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:
    `[(()]() [(()))()`

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:
    `[(()]() [(()))()`

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` $\rightarrow$ push
  - συναντάμε `])` $\rightarrow$ ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`

  είσοδος: `([]())` στίβα:
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:

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

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`

  είσοδος: `[]())` στίβα:`(`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:

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

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`

  είσοδος: `]())` στίβα:`([`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:

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

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`

  είσοδος: `())` στίβα:`(`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:

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

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`

  είσοδος: `))` στίβα:`((`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:

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

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`

  είσοδος: `)` στίβα:`(`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:

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

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` → push
  - συναντάμε `])` → ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`
  είσοδος: στίβα: επιτυχία
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:
    `[(()]() [(())))()`

- Μέθοδος: χρήση στίβας
  - συναντάμε `[(` $\rightarrow$ push
  - συναντάμε `])` $\rightarrow$ ελέγχουμε το κορυφαίο στοιχείο και κάνουμε pop
  - τέλος συμβολοσειράς: η στίβα πρέπει να είναι άδεια
- Παράδειγμα: `([]())`
- Παραδείγματα αποτυχίας:
  - όχι κενή στίβα στο τέλος: `([]()`
  - αποτυχία σε pop:
    `[((])]() [(()))()`

```
import Stack
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)
```

- Ουρά (queue): σειρά FIFO (first in first out) δομή δεδομένων

```
module Queue
 (Queue,emptyq,enqueue,front,dequeue,isempty)
 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==[]
```

- Διάσχιση Γράφου κατά Πλάτος (breadth first traversal):
  - επισκεπτόμαστε ένα κόμβο, μετά τους γείτονές του, μετά τους γείτονες των γειτόνων του κτλ.
- Αλγόριθμος: χρήση ουράς `q` με κόμβους που πρόκειται να επισκεφτούμε και λίστας `marked` με κόμβους που επισκεφτήκαμε:
  - βγάζουμε κόμβο από την ουρά και τον επισκεφτόμαστε.
  - βάζουμε τους γείτονές του που δεν έχουμε επισκεφτεί στην ουρά
  - τελειώνουμε όταν η ουρά έχει τελειώσει

- Διάσχιση Γράφου κατά Πλάτος (breadth first traversal):
  - επισκεπτόμαστε ένα κόμβο, μετά τους γείτονές του, μετά τους γείτονες των γειτόνων του κτλ.
- Αλγόριθμος: χρήση ουράς `q` με κόμβους που πρόκειται να επισκεφτούμε και λίστας `marked` με κόμβους που επισκεφτήκαμε:
  - βγάζουμε κόμβο από την ουρά και τον επισκεφτόμαστε.
  - βάζουμε τους γείτονές του που δεν έχουμε επισκεφτεί στην ουρά
  - τελειώνουμε όταν η ουρά έχει τελειώσει

- Διάσχιση Γράφου κατά Πλάτος (breadth first traversal):
  - επισκεπτόμαστε ένα κόμβο, μετά τους γείτονές του, μετά τους γείτονες των γειτόνων του κτλ.
- Αλγόριθμος: χρήση ουράς `q` με κόμβους που πρόκειται να επισκεφτούμε και λίστας `marked` με κόμβους που επισκεφτήκαμε:
  - βγάζουμε κόμβο από την ουρά και τον επισκεφτόμαστε.
  - βάζουμε τους γείτονές του που δεν έχουμε επισκεφτεί στην ουρά
  - τελειώνουμε όταν η ουρά έχει τελειώσει

- Περιγραφή γράφου με λίστες γειτνίασης:
  `Graph = [[Int]]`
  - `n1` γείτονας με `n2` → `n1`‘`elem`‘`(g!!n2)`

```
  sampleGraph
= [ [1]
  , [2,0]
  , [4]
  , [3,4]
  , [1,2,3,4]
  ]
```

```
0->1   1->2   2->0   2->4   3->3
3->4   4->1   4->2   4->3   4->4
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

```
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
```

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` → `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` → `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	frnt	back
emptyq	[]	[]
enqueue 4	[]	[4]
enqueue 3	[]	[3,4]
dequeue	[4,3]	[]
	[3]	[]
enqueue 1	[3]	[1]
dequeue	[]	[1]

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

- Μη αποδοτική υλοποίηση ουράς: η `enqueue` παίρνει γραμμικό χρόνο
- Εναλλακτική υλοποίηση: με δύο λίστες `frnt` και `back`
  - `enqueue` γεμίζει την `back` και `dequeue` αδειάζει τη `frnt`
  - άδεια `frnt` $\rightarrow$ `dequeue` μεταφέρει τη `back` στη `frnt`

εντολή	`frnt`	`back`
`emptyq`	`[]`	`[]`
`enqueue 4`	`[]`	`[4]`
`enqueue 3`	`[]`	`[3,4]`
`dequeue`	`[4,3]`	`[]`
	`[3]`	`[]`
`enqueue 1`	`[3]`	`[1]`
`dequeue`	`[]`	`[1]`

```
module Queue
 (Queue,emptyq,enqueue,front,dequeue,isempty) 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
```

```
module Queue
 (Queue,emptyq,enqueue,front,dequeue,isempty) 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
```

```
module Queue
 (Queue,emptyq,enqueue,front,dequeue,isempty) 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
```

```
module Queue
 (Queue,emptyq,enqueue,front,dequeue,isempty) 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
```

- Υψηλού επιπέδου περιγραφή: διαχωρισμός συμβολιζόμενου αντικειμένου από υλοποίηση
  - συμβολιζόμενο αντικείμενο: $\mathbb{Z}$, υλοποίηση: 2's complement
  - συμβολιζόμενο αντικείμενο: A^{ω}, υλοποίηση: γραμμικά συνδεδεμένες λίστες
- Ασχολούμαστε με τη συμβολιζόμενη ιδέα (concept) και όχι με την υλοποίηση.
- Οι πολυπλοκότητες της υλοποίησης δεν προστείθενται στην πολυπλοκότητα του δικού μας προβλήματος.
- Σύνολα (sets): συλλογές αντικειμένων που δεν ενδιαφέρει σειρά / πληθυσμός
  - ξεκαθαρίζει την πρόθεση του χρήστη

- Υψηλού επιπέδου περιγραφή: διαχωρισμός συμβολιζόμενου αντικειμένου από υλοποίηση
  - συμβολιζόμενο αντικείμενο: $\mathbb{Z}$, υλοποίηση: 2's complement
  - συμβολιζόμενο αντικείμενο: A^{ω}, υλοποίηση: γραμμικά συνδεδεμένες λίστες
- Ασχολούμαστε με τη συμβολιζόμενη ιδέα (concept) και όχι με την υλοποίηση.
- Οι πολυπλοκότητες της υλοποίησης δεν προστείθενται στην πολυπλοκότητα του δικού μας προβλήματος.
- Σύνολα (sets): συλλογές αντικειμένων που δεν ενδιαφέρει σειρά / πληθυσμός
  - ξεκαθαρίζει την πρόθεση του χρήστη

- Υψηλού επιπέδου περιγραφή: διαχωρισμός συμβολιζόμενου αντικειμένου από υλοποίηση
  - συμβολιζόμενο αντικείμενο: $\mathbb{Z}$, υλοποίηση: 2's complement
  - συμβολιζόμενο αντικείμενο: A^{ω}, υλοποίηση: γραμμικά συνδεδεμένες λίστες
- Ασχολούμαστε με τη συμβολιζόμενη ιδέα (concept) και όχι με την υλοποίηση.
- Οι πολυπλοκότητες της υλοποίησης δεν προστείθενται στην πολυπλοκότητα του δικού μας προβλήματος.
- Σύνολα (sets): συλλογές αντικειμένων που δεν ενδιαφέρει σειρά / πληθυσμός
  - ξεκαθαρίζει την πρόθεση του χρήστη

- Υψηλού επιπέδου περιγραφή: διαχωρισμός συμβολιζόμενου αντικειμένου από υλοποίηση
  - συμβολιζόμενο αντικείμενο: $\mathbb{Z}$, υλοποίηση: 2's complement
  - συμβολιζόμενο αντικείμενο: A^{ω}, υλοποίηση: γραμμικά συνδεδεμένες λίστες
- Ασχολούμαστε με τη συμβολιζόμενη ιδέα (concept) και όχι με την υλοποίηση.
- Οι πολυπλοκότητες της υλοποίησης δεν προστείθενται στην πολυπλοκότητα του δικού μας προβλήματος.
- Σύνολα (sets): συλλογές αντικειμένων που δεν ενδιαφέρει σειρά / πληθυσμός
  - ξεκαθαρίζει την πρόθεση του χρήστη

```
module Set
 ( Set,emptyset,singleton,el,union
 , comprehension,intersection,difference
 , fromList,toList)
 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)])
```

```
module Set
 ( Set,emptyset,singleton,el,union
 , comprehension,intersection,difference
 , fromList,toList)
 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)])
```

```
module Set
 ( Set,emptyset,singleton,el,union
 , comprehension,intersection,difference
 , fromList,toList)
 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)])
```

```
module Set
 ( Set,emptyset,singleton,el,union
 , comprehension,intersection,difference
 , fromList,toList)
 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)])
```

```
module Set
 ( Set,emptyset,singleton,el,union
 , comprehension,intersection,difference
 , fromList,toList)
 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)])
```

```
module Set
 ( Set,emptyset,singleton,el,union
 , comprehension,intersection,difference
 , fromList,toList)
 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
```

```
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
```

```
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
```

```
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
```

```
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
```

- Δηλώσεις `instance` και υπερφορτωμένες συναρτήσεις εξάγονται αυτόματα

```
import Queue
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)
```

- Κατασκευαστές (constructors): κατασκευή τιμών.
  - πρέπει να καλύπτονται όλες οι δυνατές τιμές
  - `emptys, push, emptyq, enqueue, emptyset, singleton, union, fromList`
- Παρατηρητές (observers): εξαγωγή πληροφορίας.
  - πρέπει (μαζί με καταστροφείς) να καλύπτονται όλες οι επιθυμητές πληροφορίες
  - `top, isempty, count, front, isempty, el, toList`
- Μετατροπείς (modifiers): αλλαγή μίας τιμής σε μία άλλη.
  - `pop, dequeue, comprehension`
- Καταστροφείς (destructors): αλλαγή σε απλούστερη τιμή.
  - συνδυάζονται με παρατηρητές για πρόσβαση σε πληροφορία
  - `pop, dequeue`
- Συνδυαστές (combinators): πράξεις σε πολλές τιμές
  - `union, intersection, difference`

- Κατασκευαστές (constructors): κατασκευή τιμών.
  - πρέπει να καλύπτονται όλες οι δυνατές τιμές
  - `emptys, push, emptyq, enqueue, emptyset, singleton, union, fromList`
- Παρατηρητές (observers): εξαγωγή πληροφορίας.
  - πρέπει (μαζί με καταστροφείς) να καλύπτονται όλες οι επιθυμητές πληροφορίες
  - `top, isempty, count, front, isempty, el, toList`
- Μετατροπείς (modifiers): αλλαγή μίας τιμής σε μία άλλη.
  - `pop, dequeue, comprehension`
- Καταστροφείς (destructors): αλλαγή σε απλούστερη τιμή.
  - συνδυάζονται με παρατηρητές για πρόσβαση σε πληροφορία
  - `pop, dequeue`
- Συνδυαστές (combinators): πράξεις σε πολλές τιμές
  - `union, intersection, difference`

- Κατασκευαστές (constructors): κατασκευή τιμών.
  - πρέπει να καλύπτονται όλες οι δυνατές τιμές
  - `emptys, push, emptyq, enqueue, emptyset, singleton, union, fromList`
- Παρατηρητές (observers): εξαγωγή πληροφορίας.
  - πρέπει (μαζί με καταστροφείς) να καλύπτονται όλες οι επιθυμητές πληροφορίες
  - `top, isempty, count, front, isempty, el, toList`
- Μετατροπείς (modifiers): αλλαγή μίας τιμής σε μία άλλη.
  - `pop, dequeue, comprehension`
- Καταστροφείς (destructors): αλλαγή σε απλούστερη τιμή.
  - συνδυάζονται με παρατηρητές για πρόσβαση σε πληροφορία
  - `pop, dequeue`
- Συνδυαστές (combinators): πράξεις σε πολλές τιμές
  - `union, intersection, difference`

- Κατασκευαστές (constructors): κατασκευή τιμών.
  - πρέπει να καλύπτονται όλες οι δυνατές τιμές
  - `emptys, push, emptyq, enqueue, emptyset, singleton, union, fromList`
- Παρατηρητές (observers): εξαγωγή πληροφορίας.
  - πρέπει (μαζί με καταστροφείς) να καλύπτονται όλες οι επιθυμητές πληροφορίες
  - `top, isempty, count, front, isempty, el, toList`
- Μετατροπείς (modifiers): αλλαγή μίας τιμής σε μία άλλη.
  - `pop, dequeue, comprehension`
- Καταστροφείς (destructors): αλλαγή σε απλούστερη τιμή.
  - συνδυάζονται με παρατηρητές για πρόσβαση σε πληροφορία
  - `pop, dequeue`
- Συνδυαστές (combinators): πράξεις σε πολλές τιμές
  - `union, intersection, difference`

- Κατασκευαστές (constructors): κατασκευή τιμών.
  - πρέπει να καλύπτονται όλες οι δυνατές τιμές
  - `emptys, push, emptyq, enqueue, emptyset, singleton, union, fromList`
- Παρατηρητές (observers): εξαγωγή πληροφορίας.
  - πρέπει (μαζί με καταστροφείς) να καλύπτονται όλες οι επιθυμητές πληροφορίες
  - `top, isempty, count, front, isempty, el, toList`
- Μετατροπείς (modifiers): αλλαγή μίας τιμής σε μία άλλη.
  - `pop, dequeue, comprehension`
- Καταστροφείς (destructors): αλλαγή σε απλούστερη τιμή.
  - συνδυάζονται με παρατηρητές για πρόσβαση σε πληροφορία
  - `pop, dequeue`
- Συνδυαστές (combinators): πράξεις σε πολλές τιμές
  - `union, intersection, difference`

- Συνηθισμένη περιγραφή διαπροσωπείας: υπογραφή τύπων (εξαγόμενα ονόματα + τύποι)
  - εξασφαλίζει απουσία λαθών τύπου στον πελάτη
  - δεν εξασφαλίζει σωστή συμπεριφορά στον πελάτη
- Παράδειγμα: πελάτης στίβας
  - `value = top(pop(push (push emptys 3) 4))`
  - σωστή συμπεριφορά: `value = 3`
  - κακή υλοποίηση: `top _ = 42` (παραβιάζει τη σωστή συμπεριφορά)

- Συνηθισμένη περιγραφή διαπροσωπείας: υπογραφή τύπων (εξαγόμενα ονόματα + τύποι)
  - εξασφαλίζει απουσία λαθών τύπου στον πελάτη
  - δεν εξασφαλίζει σωστή συμπεριφορά στον πελάτη
- Παράδειγμα: πελάτης στίβας
  - `value = top(pop(push (push emptys 3) 4))`
  - σωστή συμπεριφορά: `value = 3`
  - κακή υλοποίηση: `top _ = 42` (παραβιάζει τη σωστή συμπεριφορά)

- Συνηθισμένη περιγραφή διαπροσωπείας: υπογραφή τύπων (εξαγόμενα ονόματα + τύποι)
  - εξασφαλίζει απουσία λαθών τύπου στον πελάτη
  - δεν εξασφαλίζει σωστή συμπεριφορά στον πελάτη
- Παράδειγμα: πελάτης στίβας
  - `value = top(pop(push (push emptys 3) 4))`
  - σωστή συμπεριφορά: `value = 3`
  - κακή υλοποίηση: `top _ = 42` (παραβιάζει τη σωστή συμπεριφορά)

- Συνηθισμένη περιγραφή διαπροσωπείας: υπογραφή τύπων (εξαγόμενα ονόματα + τύποι)
  - εξασφαλίζει απουσία λαθών τύπου στον πελάτη
  - δεν εξασφαλίζει σωστή συμπεριφορά στον πελάτη
- Παράδειγμα: πελάτης στίβας
  - `value = top(pop(push (push emptys 3) 4))`
  - σωστή συμπεριφορά: `value = 3`
  - κακή υλοποίηση: `top _ = 42` (παραβιάζει τη σωστή συμπεριφορά)

- Συνηθισμένη περιγραφή διαπροσωπείας: υπογραφή τύπων (εξαγόμενα ονόματα + τύποι)
  - εξασφαλίζει απουσία λαθών τύπου στον πελάτη
  - δεν εξασφαλίζει σωστή συμπεριφορά στον πελάτη
- Παράδειγμα: πελάτης στίβας
  - `value = top(pop(push (push emptys 3) 4))`
  - σωστή συμπεριφορά: `value = 3`
  - κακή υλοποίηση: `top _ = 42` (παραβιάζει τη σωστή συμπεριφορά)

- Συνηθισμένη περιγραφή διαπροσωπείας: υπογραφή τύπων (εξαγόμενα ονόματα + τύποι)
  - εξασφαλίζει απουσία λαθών τύπου στον πελάτη
  - δεν εξασφαλίζει σωστή συμπεριφορά στον πελάτη
- Παράδειγμα: πελάτης στίβας
  - `value = top(pop(push (push emptys 3) 4))`
  - σωστή συμπεριφορά: `value = 3`
  - κακή υλοποίηση: `top _ = 42` (παραβιάζει τη σωστή συμπεριφορά)

- Τυπικό συμβόλαιο (formal contract): μαθηματική έκφραση τύπου `Bool` που περιγράφει την επιθυμητή συμπεριφορά ενός τμήματος.
  - υλοποιητής: αποδεικνύει ότι η υλοποίηση ικανοποιεί το συμβόλαιο
    - ικανοποίηση = λογική συνεπαγωγή
  - πελάτης: χρησιμοποιεί το συμβόλαιο στις δικές του αποδείξεις
  - αλλαγή υλοποίησης: με βάση το συμβόλαιο

- Τυπικό συμβόλαιο (formal contract): μαθηματική έκφραση τύπου `Bool` που περιγράφει την επιθυμητή συμπεριφορά ενός τμήματος.
  - υλοποιητής: αποδεικνύει ότι η υλοποίηση ικανοποιεί το συμβόλαιο
    - ικανοποίηση = λογική συνεπαγωγή
  - πελάτης: χρησιμοποιεί το συμβόλαιο στις δικές του αποδείξεις
  - αλλαγή υλοποίησης: με βάση το συμβόλαιο

- Τυπικό συμβόλαιο (formal contract): μαθηματική έκφραση τύπου `Bool` που περιγράφει την επιθυμητή συμπεριφορά ενός τμήματος.
  - υλοποιητής: αποδεικνύει ότι η υλοποίηση ικανοποιεί το συμβόλαιο
    - ικανοποίηση = λογική συνεπαγωγή
  - πελάτης: χρησιμοποιεί το συμβόλαιο στις δικές του αποδείξεις
  - αλλαγή υλοποίησης: με βάση το συμβόλαιο

- Τυπικό συμβόλαιο (formal contract): μαθηματική έκφραση τύπου `Bool` που περιγράφει την επιθυμητή συμπεριφορά ενός τμήματος.
  - υλοποιητής: αποδεικνύει ότι η υλοποίηση ικανοποιεί το συμβόλαιο
    - ικανοποίηση = λογική συνεπαγωγή
  - πελάτης: χρησιμοποιεί το συμβόλαιο στις δικές του αποδείξεις
  - αλλαγή υλοποίησης: με βάση το συμβόλαιο

- Τυπικό συμβόλαιο στίβας:
  `top(push s x)=x && pop(push s x)=s`
  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:
  `top(push(S l)x) = top(S(x:l)) = x`
  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:
  `top(push s x)=x && pop(push s x)=s`
  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:
  `top(push(S l)x) = top(S(x:l)) = x`
  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:
  ```
  value
  = top(pop(push (push emptys 3) 4))
  = top(push emptys 3)
  = 3
  ```

- Τυπικό συμβόλαιο στίβας:

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

  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:

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

  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:
  `top(push s x)=x && pop(push s x)=s`
  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:
  `top(push(S l)x) = top(S(x:l)) = x`
  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:
  `top(push s x)=x && pop(push s x)=s`
  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:
  `top(push(S l)x) = top(S(x:l)) = x`
  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:
  `top(push s x)=x && pop(push s x)=s`
  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:
  `top(push(S l)x) = top(S(x:l)) = x`
  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:
  `top(push s x)=x && pop(push s x)=s`
  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:
  `top(push(S l)x) = top(S(x:l)) = x`
  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:

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

  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:

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

  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:

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

  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:

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

  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:

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

  - αποκλείει υλοποιήσεις όπως `top _ = 42`

- Απόδειξη ικανοποίησης:

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

- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:

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

  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:

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

  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:

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

  - αποκλείει υλοποιήσεις όπως `top _ = 42`
- Απόδειξη ικανοποίησης:

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

  `pop(push(S l)x) = pop(S(x:l)) = S l`
- Απόδειξη πελάτη:

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

- Τυπικό συμβόλαιο στίβας:

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

  - αποκλείει υλοποιήσεις όπως `top _ = 42`

- Απόδειξη ικανοποίησης:

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

- Απόδειξη πελάτη:

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

- Ρόλος νομικού συμβολαίου: ποιος έχει άδικο
- Ρόλος συμβολαίου στον προγραμματισμό: πού είναι το λάθος
  - υλοποιητής δε μπορεί να αποδείξει ικανοποίηση $\Rightarrow$ λάθος στο τμήμα
  - πελάτης δε μπορεί να αποδείξει χρησιμοποιώντας το συμβόλαιο $\Rightarrow$ λάθος στον πελάτη

- Ρόλος νομικού συμβολαίου: ποιος έχει άδικο
- Ρόλος συμβολαίου στον προγραμματισμό: πού είναι το λάθος
  - υλοποιητής δε μπορεί να αποδείξει ικανοποίηση $\Rightarrow$ λάθος στο τμήμα
  - πελάτης δε μπορεί να αποδείξει χρησιμοποιώντας το συμβόλαιο $\Rightarrow$ λάθος στον πελάτη

- Τμηματοποίηση
  - διαχωρισμός λογισμικού σε ανεξάρτητα τμήματα
  - απόκρυψη πληροφορίας μέσω απόκρυψη ονόματος
- Όροι
  - εξαγόμενο όνομα
  - διαπροσωπεία
  - εισαγωγή τμήματος
- Τμήματα, εισαγωγή, εξαγωγή στον Hugs

- Τμηματοποίηση
  - διαχωρισμός λογισμικού σε ανεξάρτητα τμήματα
  - απόκρυψη πληροφορίας μέσω απόκρυψη ονόματος
- Όροι
  - εξαγόμενο όνομα
  - διαπροσωπεία
  - εισαγωγή τμήματος
- Τμήματα, εισαγωγή, εξαγωγή στον Hugs

- Τμηματοποίηση
  - διαχωρισμός λογισμικού σε ανεξάρτητα τμήματα
  - απόκρυψη πληροφορίας μέσω απόκρυψη ονόματος
- Όροι
  - εξαγόμενο όνομα
  - διαπροσωπεία
  - εισαγωγή τμήματος
- Τμήματα, εισαγωγή, εξαγωγή στον Hugs

Επισκόπηση
Αφηρημένοι Τύποι Δεδομένων - Ι

- ΑΤΔ = αλγ. τύπος που εξάγεται χωρίς τους κατασκευαστές του
  - πρόσβαση μόνο μέσω συναρτήσεων υλοποιητή
  - προστατεύει την υλοποίηση από κακή χρήση (`Stack`, `CStack`)
    - αναλλοίωτες
  - δίνει δικαίωμα αλλαγής υλοποίησης (`Queue`)
  - υψηλότερο επίπεδο περιγραφής (`Set`)
    - η υπερφόρτωση εξάγεται αυτόματα
- Κατηγορίες συναρτήσεων: κατασκευαστές, παρατηρητές, τροποποιητές, καταστροφείς, συνδυαστές
  - όλες οι τιμές πρέπει να κατασκευάζονται
  - όλες οι απαραίτητες πληροφορίες πρέπει να εξάγονται
  - οι υπόλοιπες συναρτήσεις είναι βοηθητικές

Επισκόπηση
Αφηρημένοι Τύποι Δεδομένων - Ι

- ΑΤΔ = αλγ. τύπος που εξάγεται χωρίς τους κατασκευαστές του
  - πρόσβαση μόνο μέσω συναρτήσεων υλοποιητή
  - προστατεύει την υλοποίηση από κακή χρήση (`Stack`, `CStack`)
    - αναλλοίωτες
  - δίνει δικαίωμα αλλαγής υλοποίησης (`Queue`)
  - υψηλότερο επίπεδο περιγραφής (`Set`)
    - η υπερφόρτωση εξάγεται αυτόματα
- Κατηγορίες συναρτήσεων: κατασκευαστές, παρατηρητές, τροποποιητές, καταστροφείς, συνδυαστές
  - όλες οι τιμές πρέπει να κατασκευάζονται
  - όλες οι απαραίτητες πληροφορίες πρέπει να εξάγονται
  - οι υπόλοιπες συναρτήσεις είναι βοηθητικές

Επισκόπηση
Αφηρημένοι Τύποι Δεδομένων - Ι

- ΑΤΔ = αλγ. τύπος που εξάγεται χωρίς τους κατασκευαστές του
  - πρόσβαση μόνο μέσω συναρτήσεων υλοποιητή
  - προστατεύει την υλοποίηση από κακή χρήση (`Stack`, `CStack`)
    - αναλλοίωτες
  - δίνει δικαίωμα αλλαγής υλοποίησης (`Queue`)
  - υψηλότερο επίπεδο περιγραφής (`Set`)
    - η υπερφόρτωση εξάγεται αυτόματα
- Κατηγορίες συναρτήσεων: κατασκευαστές, παρατηρητές, τροποποιητές, καταστροφείς, συνδυαστές
  - όλες οι τιμές πρέπει να κατασκευάζονται
  - όλες οι απαραίτητες πληροφορίες πρέπει να εξάγονται
  - οι υπόλοιπες συναρτήσεις είναι βοηθητικές

Επισκόπηση
Αφηρημένοι Τύποι Δεδομένων - Ι

- ΑΤΔ = αλγ. τύπος που εξάγεται χωρίς τους κατασκευαστές του
  - πρόσβαση μόνο μέσω συναρτήσεων υλοποιητή
  - προστατεύει την υλοποίηση από κακή χρήση (`Stack`, `CStack`)
    - αναλλοίωτες
  - δίνει δικαίωμα αλλαγής υλοποίησης (`Queue`)
  - υψηλότερο επίπεδο περιγραφής (`Set`)
    - η υπερφόρτωση εξάγεται αυτόματα
- Κατηγορίες συναρτήσεων: κατασκευαστές, παρατηρητές, τροποποιητές, καταστροφείς, συνδυαστές
  - όλες οι τιμές πρέπει να κατασκευάζονται
  - όλες οι απαραίτητες πληροφορίες πρέπει να εξάγονται
  - οι υπόλοιπες συναρτήσεις είναι βοηθητικές

Επισκόπηση
Αφηρημένοι Τύποι Δεδομένων - Ι

- ΑΤΔ = αλγ. τύπος που εξάγεται χωρίς τους κατασκευαστές του
  - πρόσβαση μόνο μέσω συναρτήσεων υλοποιητή
  - προστατεύει την υλοποίηση από κακή χρήση (`Stack`, `CStack`)
    - αναλλοίωτες
  - δίνει δικαίωμα αλλαγής υλοποίησης (`Queue`)
  - υψηλότερο επίπεδο περιγραφής (`Set`)
    - η υπερφόρτωση εξάγεται αυτόματα
- Κατηγορίες συναρτήσεων: κατασκευαστές, παρατηρητές, τροποποιητές, καταστροφείς, συνδυαστές
  - όλες οι τιμές πρέπει να κατασκευάζονται
  - όλες οι απαραίτητες πληροφορίες πρέπει να εξάγονται
  - οι υπόλοιπες συναρτήσεις είναι βοηθητικές

- Τυπικό συμβόλαιο: μέρος της διαπροσωπείας που εκφράζει τυπικά την επιθυμητή συμπεριφορά
  - έκφραση τύπου αληθοτιμής
  - ο υλοποιητής αποδεικνύει ικανοποίηση (λογική συνεπαγωγή)
  - ο χρήστης είναι ελεύθερος να χρησιμοποιήσει
  - αλλαγή υλοποίησης με βάση το συμβόλαιο