Αλγεβρικοί Τύποι

Γιάννης Κασσιός

Δομή Σημερινής Διάλεξης

- Εισαγωγή στους Αλγεβρικούς Τύπους
- Τύποι Απαρίθμησης
- Κατασκευαστές με Παραμέτρους
- Εναλλακτικοί Κατασκευαστές
- Αλγεβρικοί Τύποι και Κλάσεις
- Αλγεβρικοί Τύποι και Πολυμορφισμός
- Αναδρομικοί Τύποι
- Αποδείξεις Ορθότητας: Γενική Επαγωγή

Δομή Σημερινής Διάλεξης

- Εισαγωγή στους Αλγεβρικούς Τύπους
- Τύποι Απαρίθμησης
- Κατασκευαστές με Παραμέτρους
- Εναλλακτικοί Κατασκευαστές
- Αλγεβρικοί Τύποι και Κλάσεις
- Αλγεβρικοί Τύποι και Πολυμορφισμός
- Αναδρομικοί Τύποι
- Αποδείξεις Ορθότητας: Γενική Επαγωγή

Δομή Σημερινής Διάλεξης

- Εισαγωγή στους Αλγεβρικούς Τύπους
- Τύποι Απαρίθμησης
- Κατασκευαστές με Παραμέτρους
- Εναλλακτικοί Κατασκευαστές
- Αλγεβρικοί Τύποι και Κλάσεις
- Αλγεβρικοί Τύποι και Πολυμορφισμός
- Αναδρομικοί Τύποι
- Αποδείξεις Ορθότητας: Γενική Επαγωγή

Δομή Σημερινής Διάλεξης

- Εισαγωγή στους Αλγεβρικούς Τύπους
- Τύποι Απαρίθμησης
- Κατασκευαστές με Παραμέτρους
- Εναλλακτικοί Κατασκευαστές
- Αλγεβρικοί Τύποι και Κλάσεις
- Αλγεβρικοί Τύποι και Πολυμορφισμός
- Αναδρομικοί Τύποι
- Αποδείξεις Ορθότητας: Γενική Επαγωγή

Δομή Σημερινής Διάλεξης

- Εισαγωγή στους Αλγεβρικούς Τύπους
- Τύποι Απαρίθμησης
- Κατασκευαστές με Παραμέτρους
- Εναλλακτικοί Κατασκευαστές
- Αλγεβρικοί Τύποι και Κλάσεις
- Αλγεβρικοί Τύποι και Πολυμορφισμός
- Αναδρομικοί Τύποι
- Αποδείξεις Ορθότητας: Γενική Επαγωγή

Δομή Σημερινής Διάλεξης

- Εισαγωγή στους Αλγεβρικούς Τύπους
- Τύποι Απαρίθμησης
- Κατασκευαστές με Παραμέτρους
- Εναλλακτικοί Κατασκευαστές
- Αλγεβρικοί Τύποι και Κλάσεις
- Αλγεβρικοί Τύποι και Πολυμορφισμός
- Αναδρομικοί Τύποι
- Αποδείξεις Ορθότητας: Γενική Επαγωγή

Δομή Σημερινής Διάλεξης

- Εισαγωγή στους Αλγεβρικούς Τύπους
- Τύποι Απαρίθμησης
- Κατασκευαστές με Παραμέτρους
- Εναλλακτικοί Κατασκευαστές
- Αλγεβρικοί Τύποι και Κλάσεις
- Αλγεβρικοί Τύποι και Πολυμορφισμός
- Αναδρομικοί Τύποι
- Αποδείξεις Ορθότητας: Γενική Επαγωγή

Δομή Σημερινής Διάλεξης

- Εισαγωγή στους Αλγεβρικούς Τύπους
- Τύποι Απαρίθμησης
- Κατασκευαστές με Παραμέτρους
- Εναλλακτικοί Κατασκευαστές
- Αλγεβρικοί Τύποι και Κλάσεις
- Αλγεβρικοί Τύποι και Πολυμορφισμός
- Αναδρομικοί Τύποι
- Αποδείξεις Ορθότητας: Γενική Επαγωγή

Οι τύποι μας μέχρι τώρα έχουν προβλήματα:

- Όχι απαριθμήσεις από τον χρήστη
- Όχι τύποι-αθροίσματα
  - τύποι με εναλλακτικές επιλογές π.χ. τύπος "ακέραιος ή αληθοτιμή"
- Έκθετη Δομή
- Όχι αναδρομές οριζόμενες από το χρήστη

Οι τύποι μας μέχρι τώρα έχουν προβλήματα:

- Όχι απαριθμήσεις από τον χρήστη
- Όχι τύποι-αθροίσματα
  - τύποι με εναλλακτικές επιλογές π.χ. τύπος "ακέραιος ή αληθοτιμή"
- Έκθετη Δομή
- Όχι αναδρομές οριζόμενες από το χρήστη

Οι τύποι μας μέχρι τώρα έχουν προβλήματα:

- Όχι απαριθμήσεις από τον χρήστη
- Όχι τύποι-αθροίσματα
  - τύποι με εναλλακτικές επιλογές π.χ. τύπος "ακέραιος ή αληθοτιμή"
- Έκθετη Δομή
- Όχι αναδρομές οριζόμενες από το χρήστη

Οι τύποι μας μέχρι τώρα έχουν προβλήματα:

- Όχι απαριθμήσεις από τον χρήστη
- Όχι τύποι-αθροίσματα
  - τύποι με εναλλακτικές επιλογές π.χ. τύπος "ακέραιος ή αληθοτιμή"
- Έκθετη Δομή
- Όχι αναδρομές οριζόμενες από το χρήστη

Μία πλειάδα δεν ξεκαθαρίζει τι αναπαριστά η πληροφορία που περιέχει

```
type Student = (String,Int,Bool)
```

- Μία άσχετη πλειάδα `(s,i,b)` μπορεί να χρησιμοποιηθεί κατά λάθος σαν αντικείμενο `Student`
- Ένα αντικείμενο `Student` μπορεί να χρησιμοποιηθεί κατά λάθος σαν ένα άλλο αντικείμενο που τυχαίνει να έχει την ίδια δομή
- Αποκλείει εναλλακτικές αναπαραστάσεις των ιδίων δεδομένων που τυχαίνει να έχουν την ίδια δομή
  - π.χ. αναπαράσταση ενός μιγαδικού αριθμού ως $x + yj$ ή ως $\rho e^{j\theta}$:

```
Complex1 = (Float,Float)
Complex2 = (Float,Float)
```

Μία πλειάδα δεν ξεκαθαρίζει τι αναπαριστά η πληροφορία που περιέχει

```
type Student = (String,Int,Bool)
```

- Μία άσχετη πλειάδα `(s,i,b)` μπορεί να χρησιμοποιηθεί κατά λάθος σαν αντικείμενο `Student`
- Ένα αντικείμενο `Student` μπορεί να χρησιμοποιηθεί κατά λάθος σαν ένα άλλο αντικείμενο που τυχαίνει να έχει την ίδια δομή
- Αποκλείει εναλλακτικές αναπαραστάσεις των ιδίων δεδομένων που τυχαίνει να έχουν την ίδια δομή
  - π.χ. αναπαράσταση ενός μιγαδικού αριθμού ως $x + yj$ ή ως $\rho e^{j\theta}$:

```
Complex1 = (Float,Float)
Complex2 = (Float,Float)
```

Μία πλειάδα δεν ξεκαθαρίζει τι αναπαριστά η πληροφορία που περιέχει

```
type Student = (String,Int,Bool)
```

- Μία άσχετη πλειάδα `(s,i,b)` μπορεί να χρησιμοποιηθεί κατά λάθος σαν αντικείμενο `Student`
- Ένα αντικείμενο `Student` μπορεί να χρησιμοποιηθεί κατά λάθος σαν ένα άλλο αντικείμενο που τυχαίνει να έχει την ίδια δομή
- Αποκλείει εναλλακτικές αναπαραστάσεις των ιδίων δεδομένων που τυχαίνει να έχουν την ίδια δομή
  - π.χ. αναπαράσταση ενός μιγαδικού αριθμού ως $x + yj$ ή ως $\rho e^{j\theta}$:

```
Complex1 = (Float,Float)
Complex2 = (Float,Float)
```

Μία πλειάδα δεν ξεκαθαρίζει τι αναπαριστά η πληροφορία που περιέχει

```
type Student = (String,Int,Bool)
```

- Μία άσχετη πλειάδα `(s,i,b)` μπορεί να χρησιμοποιηθεί κατά λάθος σαν αντικείμενο `Student`
- Ένα αντικείμενο `Student` μπορεί να χρησιμοποιηθεί κατά λάθος σαν ένα άλλο αντικείμενο που τυχαίνει να έχει την ίδια δομή
- Αποκλείει εναλλακτικές αναπαραστάσεις των ιδίων δεδομένων που τυχαίνει να έχουν την ίδια δομή
  - π.χ. αναπαράσταση ενός μιγαδικού αριθμού ως $x + yj$ ή ως $\rho e^{j\theta}$:

    ```
    Complex1 = (Float,Float)
    Complex2 = (Float,Float)
    ```

Εισαγωγή στους Αλγεβρικούς Τύπους
Αναδρομή Οριζόμενη από το Χρήστη

- Η αναδρομή επιτρέπεται στις τιμές, αλλά μέχρι τώρα δεν την έχουμε δει στους τύπους
- Δομές της Haskell οριζόμενες αναδρομικά:
  - Φυσικοί Αριθμοί:

$$\mathbb{N} = \{0\} \cup \{x + 1 \mid x \in \mathbb{N}\}$$

  - Λίστες:

$$A^* = \{[]\} \cup \{h : t \mid h \in A, t \in A^*\}$$

- Πολλά άλλα σύνολα ορίζονται αναδρομικά (π.χ. δέντρα). Θέλουμε έναν γενικό τρόπο να ορίζουμε αναδρομικούς τύπους

- Η αναδρομή επιτρέπεται στις τιμές, αλλά μέχρι τώρα δεν την έχουμε δει στους τύπους
- Δομές της Haskell οριζόμενες αναδρομικά:
  - Φυσικοί Αριθμοί:

$$\mathbb{N} = \{0\} \cup \{x + 1 \mid x \in \mathbb{N}\}$$

  - Λίστες:

$$A^* = \{[]\} \cup \{h : t \mid h \in A,\ t \in A^*\}$$

- Πολλά άλλα σύνολα ορίζονται αναδρομικά (π.χ. δέντρα). Θέλουμε έναν γενικό τρόπο να ορίζουμε αναδρομικούς τύπους

- Η αναδρομή επιτρέπεται στις τιμές, αλλά μέχρι τώρα δεν την έχουμε δει στους τύπους
- Δομές της Haskell οριζόμενες αναδρομικά:
  - Φυσικοί Αριθμοί:

$$\mathbb{N} = \{0\} \cup \{x + 1 \mid x \in \mathbb{N}\}$$

  - Λίστες:

$$A^\star = \{[]\} \cup \{h : t \mid h \in A,\ t \in A^\star\}$$

- Πολλά άλλα σύνολα ορίζονται αναδρομικά (π.χ. δέντρα). Θέλουμε έναν γενικό τρόπο να ορίζουμε αναδρομικούς τύπους

- Η αναδρομή επιτρέπεται στις τιμές, αλλά μέχρι τώρα δεν την έχουμε δει στους τύπους
- Δομές της Haskell οριζόμενες αναδρομικά:
  - Φυσικοί Αριθμοί:

$$\mathbb{N} = \{0\} \cup \{x + 1 \mid x \in \mathbb{N}\}$$

  - Λίστες:

$$A^\star = \{[]\} \cup \{h : t \mid h \in A,\ t \in A^\star\}$$

- Πολλά άλλα σύνολα ορίζονται αναδρομικά (π.χ. δέντρα). Θέλουμε έναν γενικό τρόπο να ορίζουμε αναδρομικούς τύπους

- Η αναδρομή επιτρέπεται στις τιμές, αλλά μέχρι τώρα δεν την έχουμε δει στους τύπους
- Δομές της Haskell οριζόμενες αναδρομικά:
  - Φυσικοί Αριθμοί:

$$\mathbb{N} = \{0\} \cup \{x + 1 \mid x \in \mathbb{N}\}$$

  - Λίστες:

$$A^\star = \{[]\} \cup \{h : t \mid h \in A,\ t \in A^\star\}$$

- Πολλά άλλα σύνολα ορίζονται αναδρομικά (π.χ. δέντρα). Θέλουμε έναν γενικό τρόπο να ορίζουμε αναδρομικούς τύπους

`data` *όνομα_τύπου* = *καθορισμός_τύπου*

ξεκινάει με κεφαλαίο

θα δούμε στη συνέχεια

unexpected end of slide :p

`data` *όνομα_τύπου* `=` *καθορισμός_τύπου*

ξεκινάει με κεφαλαίο

θα δούμε στη συνέχεια

unexpected end of slide :p

`data` *όνομα_τύπου* `=` *καθορισμός_τύπου*

ξεκινάει με κεφαλαίο

θα δούμε στη συνέχεια

unexpected end of slide :p

Τύποι Απαρίθμησης

- Τύπος Απαρίθμησης (enumeration type): τύπος που απαρτίζεται από διακριτές τιμές που αναφέρονται μία μία
- Ορίζεται ως εξής:
  `data` *όνομα_τύπου* `=` *τιμή0* `|` *τιμή1* `|` ...
- Οι τιμές ξεκινάνε με κεφαλαίο και ονομάζονται κατασκευαστές (constructors) του τύπου
- Υπάρχον παράδειγμα:
```
data Bool = True | False
```
- Παράδειγμα ορισμένο από το χρήστη:
```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun

happy Sat = True

happy Sun = True

happy _ = False
```

Τύποι Απαρίθμησης

- Τύπος Απαρίθμησης (enumeration type): τύπος που απαρτίζεται από διακριτές τιμές που αναφέρονται μία μία
- Ορίζεται ως εξής:
  `data` *όνομα_τύπου* `=` *τιμή0* `|` *τιμή1* `|` ...
- Οι τιμές ξεκινάνε με κεφαλαίο και ονομάζονται κατασκευαστές (constructors) του τύπου
- Υπάρχον παράδειγμα:

```
data Bool = True | False
```

- Παράδειγμα ορισμένο από το χρήστη:

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

happy Sat = True

happy Sun = True

happy _ = False
```

Τύποι Απαρίθμησης

- Τύπος Απαρίθμησης (enumeration type): τύπος που απαρτίζεται από διακριτές τιμές που αναφέρονται μία μία
- Ορίζεται ως εξής:
  `data` *όνομα_τύπου* `=` *τιμή0* `|` *τιμή1* `|` ...
- Οι τιμές ξεκινάνε με κεφαλαίο και ονομάζονται κατασκευαστές (constructors) του τύπου
- Υπάρχον παράδειγμα:

```
data Bool = True | False
```

- Παράδειγμα ορισμένο από το χρήστη:

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

happy Sat = True

happy Sun = True

happy _ = False
```

Τύποι Απαρίθμησης

- Τύπος Απαρίθμησης (enumeration type): τύπος που απαρτίζεται από διακριτές τιμές που αναφέρονται μία μία
- Ορίζεται ως εξής:
  `data` *όνομα_τύπου* `=` *τιμή0* `|` *τιμή1* `|` ...
- Οι τιμές ξεκινάνε με κεφαλαίο και ονομάζονται κατασκευαστές (constructors) του τύπου
- Υπάρχον παράδειγμα:

```
data Bool = True | False
```

- Παράδειγμα ορισμένο από το χρήστη:

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

happy Sat = True

happy Sun = True

happy _ = False
```

Τύποι Απαρίθμησης

- Τύπος Απαρίθμησης (enumeration type): τύπος που απαρτίζεται από διακριτές τιμές που αναφέρονται μία μία
- Ορίζεται ως εξής:
  `data` *όνομα_τύπου* `=` *τιμή0* `|` *τιμή1* `|` ...
- Οι τιμές ξεκινάνε με κεφαλαίο και ονομάζονται κατασκευαστές (constructors) του τύπου
- Υπάρχον παράδειγμα:

```
data Bool = True | False
```

- Παράδειγμα ορισμένο από το χρήστη:

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

happy Sat = True
happy Sun = True
happy _ = False
```

Τύποι Απαρίθμησης

- Τύπος Απαρίθμησης (enumeration type): τύπος που απαρτίζεται από διακριτές τιμές που αναφέρονται μία μία
- Ορίζεται ως εξής:
  `data` *όνομα_τύπου* `=` *τιμή0* `|` *τιμή1* `|` ...
- Οι τιμές ξεκινάνε με κεφαλαίο και ονομάζονται κατασκευαστές (constructors) του τύπου
- Υπάρχον παράδειγμα:

```
data Bool = True | False
```

- Παράδειγμα ορισμένο από το χρήστη:

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
happy Sat = True
happy Sun = True
happy _ = False
```

Παράμετροι στους Κατασκευαστές - Ι

- Οι κατασκευαστές μπορούν να παίρνουν παραμέτρους:
  `data` *όνομα_τύπου*
  `=` *κατασκευαστής τύπος0 τύπος1 ...*
- Κατασκευαστής με 0 παραμέτρους (π.χ. `False`,`True` κτλ.): μηδενιαίος (nullary), με μία μοναδιαίος (unary), μετά δυαδικός (binary), τριαδικός (ternary) κτλ.
- Παράδειγμα:

```
data Student = St String Int Bool
introduce (St name age sex)
  =  "Meet " ++ name ++ ". "
    ++ (if sex then "She" else "He")
    ++ " is " ++ show (age) ++ " years old."
```

- Ερώτηση: `introduce (St "Georgia" 25 True)`
  Απάντηση:

```
"Meet Georgia. She is 25 years old."
```

Παράμετροι στους Κατασκευαστές - Ι

- Οι κατασκευαστές μπορούν να παίρνουν παραμέτρους:
  `data` *όνομα_τύπου*
  `=` *κατασκευαστής τύπος0 τύπος1 ...*
- Κατασκευαστής με 0 παραμέτρους (π.χ. `False`,`True` κτλ.): μηδενιαίος (nullary), με μία μοναδιαίος (unary), μετά δυαδικός (binary), τριαδικός (ternary) κτλ.
- Παράδειγμα:

```
data Student = St String Int Bool
introduce (St name age sex)
  =   "Meet " ++ name ++ ". "
    ++ (if sex then "She" else "He")
    ++ " is " ++ show (age) ++ " years old."
```

- Ερώτηση: `introduce (St "Georgia" 25 True)`
  Απάντηση:

```
"Meet Georgia. She is 25 years old."
```

Παράμετροι στους Κατασκευαστές - Ι

- Οι κατασκευαστές μπορούν να παίρνουν παραμέτρους:
  `data` *όνομα_τύπου*
  `=` *κατασκευαστής τύπος0 τύπος1 ...*
- Κατασκευαστής με 0 παραμέτρους (π.χ. `False,True` κτλ.): μηδενιαίος (nullary), με μία μοναδιαίος (unary), μετά δυαδικός (binary), τριαδικός (ternary) κτλ.
- Παράδειγμα:

```
data Student = St String Int Bool
introduce (St name age sex)
  =   "Meet " ++ name ++ ". "
    ++ (if sex then "She" else "He")
    ++ " is " ++ show (age) ++ " years old."
```

- Ερώτηση: `introduce (St "Georgia" 25 True)`
  Απάντηση:

```
"Meet Georgia. She is 25 years old."
```

Παράμετροι στους Κατασκευαστές - Ι

- Οι κατασκευαστές μπορούν να παίρνουν παραμέτρους:
  `data` *όνομα_τύπου*
  `=` *κατασκευαστής τύπος0 τύπος1 ...*
- Κατασκευαστής με 0 παραμέτρους (π.χ. `False,True` κτλ.): μηδενιαίος (nullary), με μία μοναδιαίος (unary), μετά δυαδικός (binary), τριαδικός (ternary) κτλ.
- Παράδειγμα:

```
data Student = St String Int Bool

introduce (St name age sex)
  =   "Meet " ++ name ++ ".  "
    ++ (if sex then "She" else "He")
    ++ " is " ++ show (age) ++ " years old."
```

- Ερώτηση: `introduce (St "Georgia" 25 True)`
  Απάντηση:

```
"Meet Georgia.  She is 25 years old."
```

Παράμετροι στους Κατασκευαστές - Ι

- Οι κατασκευαστές μπορούν να παίρνουν παραμέτρους:
  `data` *όνομα_τύπου*
  `=` *κατασκευαστής τύπος0 τύπος1 ...*
- Κατασκευαστής με 0 παραμέτρους (π.χ. `False,True` κτλ.): μηδενιαίος (nullary), με μία μοναδιαίος (unary), μετά δυαδικός (binary), τριαδικός (ternary) κτλ.
- Παράδειγμα:

```
data Student = St String Int Bool

introduce (St name age sex)
  =   "Meet " ++ name ++ ". "
   ++ (if sex then "She" else "He")
   ++ " is " ++ show (age) ++ " years old."
```

- Ερώτηση: `introduce (St "Georgia" 25 True)`
  Απάντηση:
  `"Meet Georgia. She is 25 years old."`

Παράμετροι στους Κατασκευαστές - Ι

- Οι κατασκευαστές μπορούν να παίρνουν παραμέτρους:
  `data` *όνομα_τύπου*
  `=` *κατασκευαστής τύπος0 τύπος1 ...*
- Κατασκευαστής με 0 παραμέτρους (π.χ. `False`,`True` κτλ.): μηδενιαίος (nullary), με μία μοναδιαίος (unary), μετά δυαδικός (binary), τριαδικός (ternary) κτλ.
- Παράδειγμα:

```
data Student = St String Int Bool

introduce (St name age sex)
  =   "Meet " ++ name ++ ". "
   ++ (if sex then "She" else "He")
   ++ " is " ++ show (age) ++ " years old."
```

- Ερώτηση: `introduce (St "Georgia" 25 True)`
  Απάντηση:
  `"Meet Georgia. She is 25 years old."`

Παράμετροι στους Κατασκευαστές - II

- Σημεία άξια προσοχής:
  - Κατασκευή αντικειμένου με χρήση κατασκευαστή
  - Ταίριασμα μοτίβων με χρήση κατασκευαστή
  - Επίλυση του προβλήματος έκθετης δομής
- Κρύβουμε δομή ακόμη περισσότερο:

```
data Gender = Male | Female

data Student = St String Int Gender
```

Παράμετροι στους Κατασκευαστές - II

- Σημεία άξια προσοχής:
  - Κατασκευή αντικειμένου με χρήση κατασκευαστή
  - Ταίριασμα μοτίβων με χρήση κατασκευαστή
  - Επίλυση του προβλήματος έκθετης δομής
- Κρύβουμε δομή ακόμη περισσότερο:

```
data Gender = Male | Female

data Student = St String Int Gender
```

Παράμετροι στους Κατασκευαστές - II

- Σημεία άξια προσοχής:
  - Κατασκευή αντικειμένου με χρήση κατασκευαστή
  - Ταίριασμα μοτίβων με χρήση κατασκευαστή
  - Επίλυση του προβλήματος έκθετης δομής
- Κρύβουμε δομή ακόμη περισσότερο:

```
data Gender = Male | Female

data Student = St String Int Gender
```

Παράμετροι στους Κατασκευαστές - II

- Σημεία άξια προσοχής:
  - Κατασκευή αντικειμένου με χρήση κατασκευαστή
  - Ταίριασμα μοτίβων με χρήση κατασκευαστή
  - Επίλυση του προβλήματος έκθετης δομής
- Κρύβουμε δομή ακόμη περισσότερο:

```
data Gender = Male | Female

data Student = St String Int Gender
```

Παράμετροι στους Κατασκευαστές - II

- Σημεία άξια προσοχής:
  - Κατασκευή αντικειμένου με χρήση κατασκευαστή
  - Ταίριασμα μοτίβων με χρήση κατασκευαστή
  - Επίλυση του προβλήματος έκθετης δομής
- Κρύβουμε δομή ακόμη περισσότερο:

```
data Gender = Male | Female

data Student = St String Int Gender
```

Εναλλακτικοί Κατασκευαστές

`data` *όνομα_τύπου*

= *κατ0 τύπος00 τύπος01 ...* | *κατ1 ...* | *...*

Παράδειγμα:

```
data Shape = Rectangle Float Float | Circle Float
area :: Shape -> Float
area (Rectangle a b) = a*b
area (Circle r) = pi*r^2
```

Εναλλακτική Υλοποίηση:

```
data Complex
  = CompXY Float Float | CompRhTh Float Float
norm :: Complex -> Float
norm (CompXY x y) = sqrt(x^2 + y^2)
norm (CompRhTh rho _) = rho
```

Εναλλακτικοί Κατασκευαστές

`data` *όνομα_τύπου*

`=` *κατ0 τύποϛ00 τύποϛ01 ...* | *κατ1 ...* | *...*

Παράδειγμα:

```
data Shape = Rectangle Float Float | Circle Float
area :: Shape -> Float
area (Rectangle a b) = a*b
area (Circle r) = pi*r^2
```

Εναλλακτική Υλοποίηση:

```
data Complex
  = CompXY Float Float | CompRhTh Float Float
norm :: Complex -> Float
norm (CompXY x y) = sqrt(x^2 + y^2)
norm (CompRhTh rho _) = rho
```

Εναλλακτικοί Κατασκευαστές

data *όνομα_τύπου*
  = *κατ0 τύποι00 τύποι01 ...* | *κατ1 ...* | *...*

Παράδειγμα:

```
data Shape = Rectangle Float Float | Circle Float
area :: Shape -> Float
area (Rectangle a b) = a*b
area (Circle r) = pi*r^2
```

Εναλλακτική Υλοποίηση:

```
data Complex
  = CompXY Float Float | CompRhTh Float Float
norm :: Complex -> Float
norm (CompXY x y) = sqrt(x^2 + y^2)
norm (CompRhTh rho _) = rho
```

- "Αναμενόμενη"

```
instance Eq Shape where
  Rectangle a b == Rectangle c d = a==c && b==d
  Circle r == Circle q = r==q
  _ == _ = False
```

- Εναλλακτικά (`Rectangle 2 3 == Rectangle 3 2`):

```
instance Eq Shape where
  Rectangle a b == Rectangle c d
     = (a==c && b==d) || (a==d && b==c)
  Circle r == Circle q = r==q
  _ == _ = False
```

- Εναλλακτικά (`Rectangle 4 pi == Circle 2`):

```
instance Eq Shape where s == t = area s == area t
```

- "Αναμενόμενη"

```
instance Eq Shape where
  Rectangle a b == Rectangle c d = a==c && b==d
  Circle r == Circle q = r==q
  _ == _ = False
```

- Εναλλακτικά (`Rectangle 2 3 == Rectangle 3 2`):

```
instance Eq Shape where
  Rectangle a b == Rectangle c d
     = (a==c && b==d) || (a==d && b==c)
  Circle r == Circle q = r==q
  _ == _ = False
```

- Εναλλακτικά (`Rectangle 4 pi == Circle 2`):

```
instance Eq Shape where s == t = area s == area t
```

- "Αναμενόμενη"

```
instance Eq Shape where
  Rectangle a b == Rectangle c d = a==c && b==d
  Circle r == Circle q = r==q
  _ == _ = False
```

- Εναλλακτικά (`Rectangle 2 3 == Rectangle 3 2`):

```
instance Eq Shape where
  Rectangle a b == Rectangle c d
     = (a==c && b==d) || (a==d && b==c)
  Circle r == Circle q = r==q
  _ == _ = False
```

- Εναλλακτικά (`Rectangle 4 pi == Circle 2`):

```
instance Eq Shape where s == t = area s == area t
```

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False

instance Eq Weekday where
  Mon == Mon = True
  Tue == Tue = True
  Wed == Wed = True
  Thu == Thu = True
  Fri == Fri = True
  Sat == Sat = True
  Sun == Sun = True
  _ == _ = False
```

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False

instance Eq Weekday where
  Mon == Mon = True
  Tue == Tue = True
  Wed == Wed = True
  Thu == Thu = True
  Fri == Fri = True
  Sat == Sat = True
  Sun == Sun = True
  _ == _ = False
```

- `data` *ορισμός_τύπου* `deriving` *κλάσεις_Haskell*
  (είτε μία κλάση, είτε πολλές μέσα σε παρένθεση χωρισμένες με κόμμα)
- Κάνει τον τύπο στιγμιότυπο των κλάσεων, δίνοντας την πιο "αναμενόμενη" τιμή στις υπερφορτωμένες συναρτήσεις
- Δήλωση `Weekday` με ισότητα:

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving Eq
```

- Δήλωση `Shape` με την "αναμενόμενη" ισότητα:

```
data Shape = Rectangle Float Float | Circle Float
  deriving Eq
```

Αλγεβρικοί Τύποι και Κλάσεις

Δήλωση `deriving`

- `data` *ορισμός_τύπου* `deriving` *κλάσεις_Haskell*
  (είτε μία κλάση, είτε πολλές μέσα σε παρένθεση χωρισμένες με κόμμα)
- Κάνει τον τύπο στιγμιότυπο των κλάσεων, δίνοντας την πιο "αναμενόμενη" τιμή στις υπερφορτωμένες συναρτήσεις
- Δήλωση `Weekday` με ισότητα:

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving Eq
```

- Δήλωση `Shape` με την "αναμενόμενη" ισότητα:

```
data Shape = Rectangle Float Float | Circle Float
  deriving Eq
```

- `data` *ορισμός_τύπου* `deriving` *κλάσεις_Haskell*
  (είτε μία κλάση, είτε πολλές μέσα σε παρένθεση χωρισμένες με κόμμα)
- Κάνει τον τύπο στιγμιότυπο των κλάσεων, δίνοντας την πιο "αναμενόμενη" τιμή στις υπερφορτωμένες συναρτήσεις
- Δήλωση `Weekday` με ισότητα:

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving Eq
```

- Δήλωση `Shape` με την "αναμενόμενη" ισότητα:

```
data Shape = Rectangle Float Float | Circle Float
  deriving Eq
```

- `data` *ορισμός_τύπου* `deriving` *κλάσεις_Haskell*
  (είτε μία κλάση, είτε πολλές μέσα σε παρένθεση χωρισμένες με κόμμα)
- Κάνει τον τύπο στιγμιότυπο των κλάσεων, δίνοντας την πιο "αναμενόμενη" τιμή στις υπερφορτωμένες συναρτήσεις
- Δήλωση `Weekday` με ισότητα:

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving Eq
```

- Δήλωση `Shape` με την "αναμενόμενη" ισότητα:

```
data Shape = Rectangle Float Float | Circle Float
  deriving Eq
```

Αλγεβρικοί Τύποι και Κλάσεις

deriving: Άλλες Κλάσεις

- deriving Ord: η "αναμενόμενη" κατάταξη

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq,Ord)
--δίνει Mon<Tue, Fri>=Wed κτλ.
```

- deriving Show: η "αναμενόμενη" υλοποίηση της show

```
data Rectangle Float Float | Circle Float
  deriving Show
--η αποτίμηση του Circle 2 δίνει Circle 2.0
```

- deriving Read: η αναμενόμενη αντίστροφη της show
- deriving Enum: επιτρέπει τη χρήση τελεστών [x..y]

```
data Weekday ... deriving (Eq,Ord,Enum)
--επιτρέπει [Mon .. Fri]
```

Αλγεβρικοί Τύποι και Κλάσεις

deriving: Άλλες Κλάσεις

- `deriving Ord`: η "αναμενόμενη" κατάταξη

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq,Ord)
--δίνει Mon<Tue, Fri>=Wed κτλ.
```

- `deriving Show`: η "αναμενόμενη" υλοποίηση της `show`

```
data Rectangle Float Float | Circle Float
  deriving Show
--η αποτίμηση του Circle 2 δίνει Circle 2.0
```

- `deriving Read`: η αναμενόμενη αντίστροφη της `show`
- `deriving Enum`: επιτρέπει τη χρήση τελεστών `[x..y]`

```
data Weekday ... deriving (Eq,Ord,Enum)
--επιτρέπει [Mon .. Fri]
```

Αλγεβρικοί Τύποι και Κλάσεις

deriving: Άλλες Κλάσεις

- `deriving Ord`: η "αναμενόμενη" κατάταξη

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq,Ord)
--δίνει Mon<Tue, Fri>=Wed κτλ.
```

- `deriving Show`: η "αναμενόμενη" υλοποίηση της `show`

```
data Rectangle Float Float | Circle Float
  deriving Show
--η αποτίμηση του Circle 2 δίνει Circle 2.0
```

- `deriving Read`: η αναμενόμενη αντίστροφη της `show`
- `deriving Enum`: επιτρέπει τη χρήση τελεστών `[x..y]`

```
data Weekday ... deriving (Eq,Ord,Enum)
--επιτρέπει [Mon .. Fri]
```

- `deriving Ord`: η "αναμενόμενη" κατάταξη

```
data Weekday
  = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq,Ord)
--δίνει Mon<Tue, Fri>=Wed κτλ.
```

- `deriving Show`: η "αναμενόμενη" υλοποίηση της `show`

```
data Rectangle Float Float | Circle Float
  deriving Show
--η αποτίμηση του Circle 2 δίνει Circle 2.0
```

- `deriving Read`: η αναμενόμενη αντίστροφη της `show`
- `deriving Enum`: επιτρέπει τη χρήση τελεστών `[x..y]`

```
data Weekday ... deriving (Eq,Ord,Enum)
--επιτρέπει [Mon .. Fri]
```

- `data` *όνομα_τύπου μεταβλητές_τύπων* = ...
- `data SomeType a b = SomeConstructor a b` δημιουργεί τον τύπο `SomeType a b` για κάθε `a,b`
- Παράδειγμα πολυμορφικού αλγεβρικού τύπου: λίστες (`[a]`)

- `data` *όνομα_τύπου μεταβλητές_τύπων* `=` ...
- `data SomeType a b = SomeConstructor a b` δημιουργεί τον τύπο `SomeType a b` για κάθε `a,b`
- Παράδειγμα πολυμορφικού αλγεβρικού τύπου: λίστες (`[a]`)

- `data` *όνομα_τύπου μεταβλητές_τύπων* `=` ...
- `data SomeType a b = SomeConstructor a b` δημιουργεί τον τύπο `SomeType a b` για κάθε `a,b`
- Παράδειγμα πολυμορφικού αλγεβρικού τύπου: λίστες (`[a]`)

- Τύπος `Maybe`:

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Read, Show)
```

  - `Just x`: ο υπολογισμός επιστρέφει `x`
  - `Nothing`: ο υπολογισμός έπεσε σε λάθος
- Διαίρεση με πιθανότητα λάθους:

```
mydiv :: Int -> Int -> Maybe Int
mydiv x 0 = Nothing
mydiv x y = Just (x`div`y)
```

- `Maybe` ανάλογο του `throws`
- `Nothing` ανάλογο του `throw`

- Τύπος `Maybe`:

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Read, Show)
```

  - `Just x`: ο υπολογισμός επιστρέφει `x`
  - `Nothing`: ο υπολογισμός έπεσε σε λάθος
- Διαίρεση με πιθανότητα λάθους:

```
mydiv :: Int -> Int -> Maybe Int
mydiv x 0 = Nothing
mydiv x y = Just (x`div`y)
```

- `Maybe` ανάλογο του `throws`
- `Nothing` ανάλογο του `throw`

- Τύπος `Maybe`:

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Read, Show)
```

  - `Just x`: ο υπολογισμός επιστρέφει `x`
  - `Nothing`: ο υπολογισμός έπεσε σε λάθος

- Διαίρεση με πιθανότητα λάθους:

```
mydiv :: Int -> Int -> Maybe Int
mydiv x 0 = Nothing
mydiv x y = Just (x`div`y)
```

- `Maybe` ανάλογο του `throws`
- `Nothing` ανάλογο του `throw`

- Τύπος `Maybe`:

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Read, Show)
```

  - `Just x`: ο υπολογισμός επιστρέφει `x`
  - `Nothing`: ο υπολογισμός έπεσε σε λάθος
- Διαίρεση με πιθανότητα λάθους:

```
mydiv :: Int -> Int -> Maybe Int
mydiv x 0 = Nothing
mydiv x y = Just (x`div`y)
```

- `Maybe` ανάλογο του `throws`
- `Nothing` ανάλογο του `throw`

- Τύπος `Maybe`:

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Read, Show)
```

  - `Just x`: ο υπολογισμός επιστρέφει `x`
  - `Nothing`: ο υπολογισμός έπεσε σε λάθος
- Διαίρεση με πιθανότητα λάθους:

```
mydiv :: Int -> Int -> Maybe Int
mydiv x 0 = Nothing
mydiv x y = Just (x`div`y)
```

- `Maybe` ανάλογο του `throws`
- `Nothing` ανάλογο του `throw`

- Τύπος `Maybe`:

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Read, Show)
```

  - `Just x`: ο υπολογισμός επιστρέφει `x`
  - `Nothing`: ο υπολογισμός έπεσε σε λάθος
- Διαίρεση με πιθανότητα λάθους:

```
mydiv :: Int -> Int -> Maybe Int
mydiv x 0 = Nothing
mydiv x y = Just (x`div`y)
```

- `Maybe` ανάλογο του `throws`
- `Nothing` ανάλογο του `throw`

- Διοχέτευση λάθους από συνάρτηση με `Maybe`:

```
myfunc :: Int -> Int -> Maybe Int
myfunc x y =
  if division == Nothing then Nothing
  else Just (z + 1)
  where division = x`mydiv`y
        Just z = division
```

- Ορίζουμε συνάρτηση `liftMaybe`:

```
liftMaybe :: (a->b) -> Maybe a -> Maybe b
liftMaybe g Nothing = Nothing
liftMaybe g (Just x) = Just (g x)
```

- Καλύτερη υλοποίηση της `myfunc`:

```
myfunc x y = liftMaybe (\z->z+1) (x`mydiv`y)
```

- Διοχέτευση λάθους από συνάρτηση με `Maybe`:

```
myfunc :: Int -> Int -> Maybe Int
myfunc x y =
  if division == Nothing then Nothing
  else Just (z + 1)
  where division = x`mydiv`y
        Just z = division
```

- Ορίζουμε συνάρτηση `liftMaybe`:

```
liftMaybe :: (a->b) -> Maybe a -> Maybe b
liftMaybe g Nothing = Nothing
liftMaybe g (Just x) = Just (g x)
```

- Καλύτερη υλοποίηση της `myfunc`:

```
myfunc x y = liftMaybe (\z->z+1) (x`mydiv`y)
```

- Διοχέτευση λάθους από συνάρτηση με `Maybe`:

```
myfunc :: Int -> Int -> Maybe Int
myfunc x y =
  if division == Nothing then Nothing
  else Just (z + 1)
  where division = x`mydiv`y
        Just z = division
```

- Ορίζουμε συνάρτηση `liftMaybe`:

```
liftMaybe :: (a->b) -> Maybe a -> Maybe b
liftMaybe g Nothing = Nothing
liftMaybe g (Just x) = Just (g x)
```

- Καλύτερη υλοποίηση της `myfunc`:

```
myfunc x y = liftMaybe (\z->z+1) (x`mydiv`y)
```

- Συνάρτηση `maybe` της Haskell:

```
maybe :: a -> (b->a) -> Maybe b -> a
maybe n f Nothing = n
maybe n f (Just x) = f x
```

- Ανάλογη του `try/catch`
- Συνάρτηση που "παγιδεύει" το λάθος:

```
mydivcatching :: Int->Int->Int
mydivcatching x y = maybe (-1) id (x`mydiv`y)
```

- Συνάρτηση `maybe` της Haskell:

```
maybe :: a -> (b->a) -> Maybe b -> a
maybe n f Nothing = n
maybe n f (Just x) = f x
```

- Ανάλογη του `try/catch`
- Συνάρτηση που "παγιδεύει" το λάθος:

```
mydivcatching :: Int->Int->Int
mydivcatching x y = maybe (-1) id (x`mydiv`y)
```

- Συνάρτηση `maybe` της Haskell:

```
maybe :: a -> (b->a) -> Maybe b -> a
maybe n f Nothing = n
maybe n f (Just x) = f x
```

- Ανάλογη του `try/catch`
- Συνάρτηση που "παγιδεύει" το λάθος:

```
mydivcatching :: Int->Int->Int
mydivcatching x y = maybe (-1) id (x`mydiv`y)
```

Αλγεβρικοί Τύποι και Πολυμορφισμός

Ο τύπος Either

- Άθροισμα τύπων (union στη C) a και b:

```
data Either a b = Left a | Right b
    deriving (Eq,Ord,Read,Show)
```

- Παράδειγμα: εναλλακτικές υλοποιήσεις του τύπου "σύνολα φυσικών":

```
type SetofNats = Either [Int] [Bool]
element :: Int->SetofNats->Bool
element x (Left li) = lSearch x li
element x (Right lb) = lb!!x
```

- Παραδείγματα:

```
> element 3 (Left[2,1,2,3])
True
> element 3 (Right[True,False,False,False,True])
False
```

Αλγεβρικοί Τύποι και Πολυμορφισμός

Ο τύπος Either

- Άθροισμα τύπων (union στη C) a και b:

```
data Either a b = Left a | Right b
    deriving (Eq,Ord,Read,Show)
```

- Παράδειγμα: εναλλακτικές υλοποιήσεις του τύπου "σύνολα φυσικών":

```
type SetofNats = Either [Int] [Bool]
element :: Int->SetofNats->Bool
element x (Left li) = lSearch x li
element x (Right lb) = lb!!x
```

- Παραδείγματα:

```
> element 3 (Left[2,1,2,3])
True
> element 3 (Right[True,False,False,False,True])
False
```

- Άθροισμα τύπων (union στη C) a και b:

```
data Either a b = Left a | Right b
    deriving (Eq,Ord,Read,Show)
```

- Παράδειγμα: εναλλακτικές υλοποιήσεις του τύπου "σύνολα φυσικών":

```
type SetofNats = Either [Int] [Bool]
element :: Int->SetofNats->Bool
element x (Left li) = lSearch x li
element x (Right lb) = lb!!x
```

- Παραδείγματα:

```
> element 3 (Left[2,1,2,3])
True
> element 3 (Right[True,False,False,False,True])
False
```

Αναδρομικοί Τύποι
Εισαγωγή

- Αναδρομικός τύπος (recursive type): εμφανίζεται στον ορισμό του
- Αναδρομική δομή φυσικών:
  `Nat = Zero | Succ Nat`
  (δεν υλοποιούνται έτσι στη Haskell)
- Αναδρομική δομή λιστών:
  `List a = EmptyList | Cons a (List a)`
  (διαφορετική σύνταξη στη Haskell: `EmptyList` γράφεται `[]` και `Cons` γράφεται `(:)`)
- Ομοίως μπορούμε να ορίσουμε τους δικούς μας αναδρομικούς τύπους

Αναδρομικοί Τύποι
Εισαγωγή

- Αναδρομικός τύπος (recursive type): εμφανίζεται στον ορισμό του
- Αναδρομική δομή φυσικών:
  `Nat = Zero | Succ Nat`
  (δεν υλοποιούνται έτσι στη Haskell)
- Αναδρομική δομή λιστών:
  `List a = EmptyList | Cons a (List a)`
  (διαφορετική σύνταξη στη Haskell: `EmptyList` γράφεται `[]` και `Cons` γράφεται `(:)`)
- Ομοίως μπορούμε να ορίσουμε τους δικούς μας αναδρομικούς τύπους

Αναδρομικοί Τύποι
Εισαγωγή

- Αναδρομικός τύπος (recursive type): εμφανίζεται στον ορισμό του
- Αναδρομική δομή φυσικών:
  `Nat = Zero | Succ Nat`
  (δεν υλοποιούνται έτσι στη Haskell)
- Αναδρομική δομή λιστών:
  `List a = EmptyList | Cons a (List a)`
  (διαφορετική σύνταξη στη Haskell: `EmptyList` γράφεται `[]` και `Cons` γράφεται `(:)`)
- Ομοίως μπορούμε να ορίσουμε τους δικούς μας αναδρομικούς τύπους

- Αναδρομικός τύπος (recursive type): εμφανίζεται στον ορισμό του
- Αναδρομική δομή φυσικών:
  `Nat = Zero | Succ Nat`
  (δεν υλοποιούνται έτσι στη Haskell)
- Αναδρομική δομή λιστών:
  `List a = EmptyList | Cons a (List a)`
  (διαφορετική σύνταξη στη Haskell: `EmptyList` γράφεται `[]` και `Cons` γράφεται `(:)`)
- Ομοίως μπορούμε να ορίσουμε τους δικούς μας αναδρομικούς τύπους

- Ένα δυαδικό δέντρο τύπου a:
  - το κενό δέντρο
  - ένας κόμβος με πληροφορία τύπου a και δύο δυαδικά δέντρα
- Το κενό δέντρο έχει τη σημασία των 0 και []
  - επιτρέπει τη μη ύπαρξη πληροφορίας
  - διευκολύνει τον αναδρομικό ορισμό
  - καλύτερος σχεδιασμός να συμπεριλαμβάνουμε τετριμμένες περιπτώσεις
- Στη Haskell:

```
data BTree a
   = EmptyBTree | BNode a (BTree a) (BTree a)
   deriving (Eq, Show)
```

- Ένα δυαδικό δέντρο τύπου a:
  - το κενό δέντρο
  - ένας κόμβος με πληροφορία τύπου a και δύο δυαδικά δέντρα
- Το κενό δέντρο έχει τη σημασία των 0 και []
  - επιτρέπει τη μη ύπαρξη πληροφορίας
  - διευκολύνει τον αναδρομικό ορισμό
  - καλύτερος σχεδιασμός να συμπεριλαμβάνουμε τετριμμένες περιπτώσεις
- Στη Haskell:

```
data BTree a
   = EmptyBTree | BNode a (BTree a) (BTree a)
   deriving (Eq, Show)
```

- Ένα δυαδικό δέντρο τύπου `a`:
  - το κενό δέντρο
  - ένας κόμβος με πληροφορία τύπου `a` και δύο δυαδικά δέντρα
- Το κενό δέντρο έχει τη σημασία των `0` και `[]`
  - επιτρέπει τη μη ύπαρξη πληροφορίας
  - διευκολύνει τον αναδρομικό ορισμό
  - καλύτερος σχεδιασμός να συμπεριλαμβάνουμε τετριμμένες περιπτώσεις
- Στη Haskell:

```
data BTree a
   = EmptyBTree | BNode a (BTree a) (BTree a)
   deriving (Eq, Show)
```

- Ένα δυαδικό δέντρο τύπου `a`:
  - το κενό δέντρο
  - ένας κόμβος με πληροφορία τύπου `a` και δύο δυαδικά δέντρα
- Το κενό δέντρο έχει τη σημασία των `0` και `[]`
  - επιτρέπει τη μη ύπαρξη πληροφορίας
  - διευκολύνει τον αναδρομικό ορισμό
  - καλύτερος σχεδιασμός να συμπεριλαμβάνουμε τετριμμένες περιπτώσεις
- Στη Haskell:

```
data BTree a
   = EmptyBTree | BNode a (BTree a) (BTree a)
   deriving (Eq, Show)
```

- Ένα δυαδικό δέντρο τύπου `a`:
  - το κενό δέντρο
  - ένας κόμβος με πληροφορία τύπου `a` και δύο δυαδικά δέντρα
- Το κενό δέντρο έχει τη σημασία των `0` και `[]`
  - επιτρέπει τη μη ύπαρξη πληροφορίας
  - διευκολύνει τον αναδρομικό ορισμό
  - καλύτερος σχεδιασμός να συμπεριλαμβάνουμε τετριμμένες περιπτώσεις
- Στη Haskell:

```
data BTree a
   = EmptyBTree | BNode a (BTree a) (BTree a)
   deriving (Eq, Show)
```

- Ένα δυαδικό δέντρο τύπου `a`:
  - το κενό δέντρο
  - ένας κόμβος με πληροφορία τύπου `a` και δύο δυαδικά δέντρα
- Το κενό δέντρο έχει τη σημασία των `0` και `[]`
  - επιτρέπει τη μη ύπαρξη πληροφορίας
  - διευκολύνει τον αναδρομικό ορισμό
  - καλύτερος σχεδιασμός να συμπεριλαμβάνουμε τετριμμένες περιπτώσεις
- Στη Haskell:

```
data BTree a
   = EmptyBTree | BNode a (BTree a) (BTree a)
   deriving (Eq, Show)
```

- Ένα δυαδικό δέντρο τύπου `a`:
  - το κενό δέντρο
  - ένας κόμβος με πληροφορία τύπου `a` και δύο δυαδικά δέντρα
- Το κενό δέντρο έχει τη σημασία των `0` και `[]`
  - επιτρέπει τη μη ύπαρξη πληροφορίας
  - διευκολύνει τον αναδρομικό ορισμό
  - καλύτερος σχεδιασμός να συμπεριλαμβάνουμε τετριμμένες περιπτώσεις
- Στη Haskell:

```
data BTree a
   = EmptyBTree | BNode a (BTree a) (BTree a)
   deriving (Eq, Show)
```

- Ένα δυαδικό δέντρο τύπου `a`:
  - το κενό δέντρο
  - ένας κόμβος με πληροφορία τύπου `a` και δύο δυαδικά δέντρα
- Το κενό δέντρο έχει τη σημασία των `0` και `[]`
  - επιτρέπει τη μη ύπαρξη πληροφορίας
  - διευκολύνει τον αναδρομικό ορισμό
  - καλύτερος σχεδιασμός να συμπεριλαμβάνουμε τετριμμένες περιπτώσεις
- Στη Haskell:

```
data BTree a
   = EmptyBTree | BNode a (BTree a) (BTree a)
   deriving (Eq, Show)
```

Μία συνάρτηση `map` για δυαδικά δέντρα:

```
mapBTree :: (a->b) -> BTree a -> BTree b
mapBTree f EmptyBTree = EmptyBTree
mapBTree f (BNode root left right)
  = BNode (f root) (mapBTree f left)
    (mapBTree f right)
```

- Έκφραση μίας απλής γλώσσας προγραμματισμού:
  - ακέραια σταθερά
  - ακέραια μεταβλητή
  - μοναδιαίος τελεστής με έκφραση
  - δυαδικός τελεστής με δύο εκφράσεις
- Στη Haskell:

```
data Expr
  =   Constant Int
    | Variable String
    | UnaryExp UnOp Expr
    | BinaryExp BinOp Expr Expr
data UnOp = UPlus | UMinus
data BinOp = Plus | Minus | Mult | Div
```

Αναδρομικοί Τύποι
Εκφράσεις σε μια Γλώσσα Προγραμματισμού

- Έκφραση μίας απλής γλώσσας προγραμματισμού:
  - ακέραια σταθερά
  - ακέραια μεταβλητή
  - μοναδιαίος τελεστής με έκφραση
  - δυαδικός τελεστής με δύο εκφράσεις
- Στη Haskell:

```
data Expr
  =   Constant Int
    | Variable String
    | UnaryExp UnOp Expr
    | BinaryExp BinOp Expr Expr
data UnOp = UPlus | UMinus
data BinOp = Plus | Minus | Mult | Div
```

- Έκφραση μίας απλής γλώσσας προγραμματισμού:
  - ακέραια σταθερά
  - ακέραια μεταβλητή
  - μοναδιαίος τελεστής με έκφραση
  - δυαδικός τελεστής με δύο εκφράσεις
- Στη Haskell:

```
data Expr
  =   Constant Int
    | Variable String
    | UnaryExp UnOp Expr
    | BinaryExp BinOp Expr Expr
data UnOp = UPlus | UMinus
data BinOp = Plus | Minus | Mult | Div
```

- Έκφραση μίας απλής γλώσσας προγραμματισμού:
  - ακέραια σταθερά
  - ακέραια μεταβλητή
  - μοναδιαίος τελεστής με έκφραση
  - δυαδικός τελεστής με δύο εκφράσεις
- Στη Haskell:

```
data Expr
  =   Constant Int
    | Variable String
    | UnaryExp UnOp Expr
    | BinaryExp BinOp Expr Expr
data UnOp = UPlus | UMinus
data BinOp = Plus | Minus | Mult | Div
```

- Έκφραση μίας απλής γλώσσας προγραμματισμού:
  - ακέραια σταθερά
  - ακέραια μεταβλητή
  - μοναδιαίος τελεστής με έκφραση
  - δυαδικός τελεστής με δύο εκφράσεις
- Στη Haskell:

```
data Expr
  =   Constant Int
    | Variable String
    | UnaryExp UnOp Expr
    | BinaryExp BinOp Expr Expr
data UnOp = UPlus | UMinus
data BinOp = Plus | Minus | Mult | Div
```

- Έκφραση μίας απλής γλώσσας προγραμματισμού:
  - ακέραια σταθερά
  - ακέραια μεταβλητή
  - μοναδιαίος τελεστής με έκφραση
  - δυαδικός τελεστής με δύο εκφράσεις
- Στη Haskell:

```
data Expr
  =   Constant Int
    | Variable String
    | UnaryExp UnOp Expr
    | BinaryExp BinOp Expr Expr
data UnOp = UPlus | UMinus
data BinOp = Plus | Minus | Mult | Div
```

Αναπαράσταση της: `-b*(c+a/5)*2` με προτ./προσετ. Haskell:

```
BinaryExp
    Mult
    (BinaryExp
       Mult
       (UnaryExp UMinus (Variable "b"))
       (BinaryExp
          Plus
          (Variable "c")
          (BinaryExp
             Div
             (Variable "a")
             (Constant 5))))
    (Constant 2)
```

- Διερμηνέας `eval`: αποτιμά μία έκφραση σε ένα περιβάλλον που αντιστοιχίζει τιμές σε μεταβλητές:

```
eval :: Env->Expr->Int
```

- Ο τύπος `Env` θα μπορούσε να είναι:

```
type Env = [(String,Int)]
```

- Μερικές περιπτώσεις:

```
eval env (Constant x) = x
eval env (UnaryExp UMinus e) = - eval env e
eval env (BinaryExp Plus e1 e2)
  = eval env e1 + eval env e2
...
```

- Διερμηνέας `eval`: αποτιμά μία έκφραση σε ένα περιβάλλον που αντιστοιχίζει τιμές σε μεταβλητές:

```
eval :: Env->Expr->Int
```

- Ο τύπος `Env` θα μπορούσε να είναι:

```
type Env = [(String,Int)]
```

- Μερικές περιπτώσεις:

```
eval env (Constant x) = x
eval env (UnaryExp UMinus e) = - eval env e
eval env (BinaryExp Plus e1 e2)
  = eval env e1 + eval env e2
...
```

- Διερμηνέας `eval`: αποτιμά μία έκφραση σε ένα περιβάλλον που αντιστοιχίζει τιμές σε μεταβλητές:

```
eval :: Env->Expr->Int
```

- Ο τύπος `Env` θα μπορούσε να είναι:

```
type Env = [(String,Int)]
```

- Μερικές περιπτώσεις:

```
eval env (Constant x) = x
eval env (UnaryExp UMinus e) = - eval env e
eval env (BinaryExp Plus e1 e2)
  = eval env e1 + eval env e2
...
```

- Όπως και στις λίστες, είναι δυνατή η δημιουργία άπειρων δομών
- Ένα άπειρο δέντρο:

```
inftree = inftreestart 0
  where
  inftreestart n =
    BTree n (inftreestart (2*n+1))
            (inftreestart (2*n+2))
```

- Όπως και στις λίστες, είναι δυνατή η δημιουργία άπειρων δομών
- Ένα άπειρο δέντρο:

```
inftree = inftreestart 0
  where
  inftreestart n =
    BTree n (inftreestart (2*n+1))
            (inftreestart (2*n+2))
```

- Έστω σύνολο **U** και $F \in \mathbf{U} \to \mathbf{U}$ και F μονότονη:

$$X \sqsubseteq Y \Rightarrow F\,X \sqsubseteq F\,Y$$

- Θεώρημα Knaster-Tarski:
  - Η F έχει σταθερά σημεία (fixed points), δηλ. σημεία fp ώστε

$$fp = F\,fp$$

  - Υπάρχει ελάχιστο σταθερό σημείο (least fixed point) μF:

$$\mu F = F(\mu F)$$

$$X = F\,X \Rightarrow \mu F \sqsubseteq X$$

  - Ισχύει:

$$FX \sqsubseteq X \Rightarrow \mu F \sqsubseteq X$$

- Έστω σύνολο $\mathbf{U}$ και $F \in \mathbf{U} \to \mathbf{U}$ και F μονότονη:

$$X \sqsubseteq Y \Rightarrow F\,X \sqsubseteq F\,Y$$

- Θεώρημα Knaster-Tarski:
  - Η F έχει σταθερά σημεία (fixed points), δηλ. σημεία fp ώστε

$$fp = F\,fp$$

  - Υπάρχει ελάχιστο σταθερό σημείο (least fixed point) μF:

$$\mu F = F(\mu F)$$

$$X = F\,X \Rightarrow \mu F \sqsubseteq X$$

  - Ισχύει:

$$FX \sqsubseteq X \Rightarrow \mu F \sqsubseteq X$$

- Έστω σύνολο **U** και $F \in \mathbf{U} \to \mathbf{U}$ και F μονότονη:

$$X \sqsubseteq Y \Rightarrow F\,X \sqsubseteq F\,Y$$

- Θεώρημα Knaster-Tarski:
  - Η F έχει σταθερά σημεία (fixed points), δηλ. σημεία fp ώστε

$$fp = F\,fp$$

  - Υπάρχει ελάχιστο σταθερό σημείο (least fixed point) μF:

$$\mu F = F(\mu F)$$

$$X = F\,X \Rightarrow \mu F \sqsubseteq X$$

  - Ισχύει:

$$FX \sqsubseteq X \Rightarrow \mu F \sqsubseteq X$$

- Έστω σύνολο $\mathbf{U}$ και $F \in \mathbf{U} \to \mathbf{U}$ και F μονότονη:

$$X \sqsubseteq Y \Rightarrow F\,X \sqsubseteq F\,Y$$

- Θεώρημα Knaster-Tarski:
  - Η F έχει σταθερά σημεία (fixed points), δηλ. σημεία fp ώστε

$$fp = F\,fp$$

  - Υπάρχει ελάχιστο σταθερό σημείο (least fixed point) μF:

$$\mu F = F(\mu F)$$

$$X = F\,X \Rightarrow \mu F \sqsubseteq X$$

  - Ισχύει:

$$FX \sqsubseteq X \Rightarrow \mu F \sqsubseteq X$$

- Ορίζουμε

$$\mathbb{N} = \mu X.\{0\} \cup \{x + 1 \mid x \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. $\mathbb{Z}$
  - εμείς παίρνουμε την ελάχιστη
- Έστω σύνολο P στοιχείων με κάποια ιδιότητα. Από Knaster-Tarski, για να δείξουμε ότι $\mathbb{N} \sqsubseteq P$ αρκεί:

$$\{0\} \cup \{x + 1 \mid x \in P\} \sqsubseteq P$$

- Ισοδύναμα:

$$0 \in P$$

$$\forall x \in P.\ x + 1 \in P$$

- Ορίζουμε

$$\mathbb{N} = \mu X.\{0\} \cup \{x + 1 \mid x \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. $\mathbb{Z}$
  - εμείς παίρνουμε την ελάχιστη

- Έστω σύνολο P στοιχείων με κάποια ιδιότητα. Από Knaster-Tarski, για να δείξουμε ότι $\mathbb{N} \sqsubseteq P$ αρκεί:

$$\{0\} \cup \{x + 1 \mid x \in P\} \sqsubseteq P$$

- Ισοδύναμα:

$$0 \in P$$

$$\forall x \in P.\ x + 1 \in P$$

- Ορίζουμε

$$\mathbb{N} = \mu X.\{0\} \cup \{x + 1 \mid x \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. $\mathbb{Z}$
  - εμείς παίρνουμε την ελάχιστη

- Έστω σύνολο P στοιχείων με κάποια ιδιότητα. Από Knaster-Tarski, για να δείξουμε ότι $\mathbb{N} \sqsubseteq P$ αρκεί:

$$\{0\} \cup \{x + 1 \mid x \in P\} \sqsubseteq P$$

- Ισοδύναμα:

$$0 \in P$$

$$\forall x \in P.\ x + 1 \in P$$

- Ορίζουμε

$$\mathbb{N} = \mu X.\{0\} \cup \{x + 1 \mid x \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. $\mathbb{Z}$
  - εμείς παίρνουμε την ελάχιστη
- Έστω σύνολο P στοιχείων με κάποια ιδιότητα. Από Knaster-Tarski, για να δείξουμε ότι $\mathbb{N} \sqsubseteq P$ αρκεί:

$$\{0\} \cup \{x + 1 \mid x \in P\} \sqsubseteq P$$

- Ισοδύναμα:

$$0 \in P$$

$$\forall x \in P.\ x + 1 \in P$$

- Ορίζουμε

$$\mathbb{N} = \mu X.\{0\} \cup \{x + 1 \mid x \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. $\mathbb{Z}$
  - εμείς παίρνουμε την ελάχιστη
- Έστω σύνολο P στοιχείων με κάποια ιδιότητα. Από Knaster-Tarski, για να δείξουμε ότι $\mathbb{N} \sqsubseteq P$ αρκεί:

$$\{0\} \cup \{x + 1 \mid x \in P\} \sqsubseteq P$$

- Ισοδύναμα:

$$0 \in P$$

$$\forall x \in P.\ x + 1 \in P$$

- Πεπερασμένες λίστες από σύνολο A:

$$A^\star = \mu X.\{[]\} \cup \{h : t \mid h \in A,\ t \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. A^ω
  - εμείς παίρνουμε την ελάχιστη

- Ομοίως, από Knaster-Tarski, για να δείξουμε $A^\star \sqsubseteq P$:

$$[] \in P$$

$$\forall t \in P. h \in A.\ \ h : t \in P$$

- Πεπερασμένες λίστες από σύνολο A:

$$A^\star = \mu X.\{[]\} \cup \{h : t \mid h \in A,\ t \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. A^ω
  - εμείς παίρνουμε την ελάχιστη

- Ομοίως, από Knaster-Tarski, για να δείξουμε $A^\star \sqsubseteq P$:

$$[] \in P$$

$$\forall t \in P. h \in A.\ \ h : t \in P$$

- Πεπερασμένες λίστες από σύνολο A:

$$A^\star = \mu X.\{[]\} \cup \{h : t \mid h \in A,\ t \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. A^ω
  - εμείς παίρνουμε την ελάχιστη

- Ομοίως, από Knaster-Tarski, για να δείξουμε $A^\star \sqsubseteq P$:

$$[] \in P$$

$$\forall t \in P. h \in A.\ \ h : t \in P$$

- Πεπερασμένες λίστες από σύνολο A:

$$A^\star = \mu X.\{[]\} \cup \{h : t \mid h \in A,\ t \in X\}$$

  - ο αναδρομικός ορισμός έχει και άλλες λύσεις π.χ. A^ω
  - εμείς παίρνουμε την ελάχιστη
- Ομοίως, από Knaster-Tarski, για να δείξουμε $A^\star \sqsubseteq P$:

$$[] \in P$$

$$\forall t \in P. h \in A.\ \ h : t \in P$$

Αποδείξεις Ορθότητας
Επαγωγή για Κάθε Αλγεβρικό Τύπο

- Για να αποδείξουμε ότι κάθε πεπερασμένη δομή που ανήκει σε αλγεβρικό τύπο, ικανοποιεί τη συνθήκη `P`:
  - αποδεικνύουμε βάση επαγωγής για κάθε κατασκευαστή χωρίς αναδρομή
  - αποδεικνύουμε βήμα επαγωγής για κάθε κατασκευαστή με αναδρομή
- Επαγωγή σε πεπερασμένα δυαδικά δέντρα: για να δείξω ότι

$$\forall \texttt{t::(BTree a)}_{\texttt{df}}\texttt{. p t}$$

αρκεί

  - `p EmptyTree`
  - Για `x::a` και `l,r::(BTree a)`$_{\texttt{df}}$

    `p l && p r` $\Rightarrow$ `p(BNode x l r)`

Αποδείξεις Ορθότητας
Επαγωγή για Κάθε Αλγεβρικό Τύπο

- Για να αποδείξουμε ότι κάθε πεπερασμένη δομή που ανήκει σε αλγεβρικό τύπο, ικανοποιεί τη συνθήκη `P`:
  - αποδεικνύουμε βάση επαγωγής για κάθε κατασκευαστή χωρίς αναδρομή
  - αποδεικνύουμε βήμα επαγωγής για κάθε κατασκευαστή με αναδρομή
- Επαγωγή σε πεπερασμένα δυαδικά δέντρα: για να δείξω ότι

$$\forall \texttt{t::(BTree a)}_{\texttt{df}}\texttt{. p t}$$

αρκεί

- `p EmptyTree`
- Για `x::a` και `l,r::(BTree a)`$_{\texttt{df}}$

`p l && p r ⇒ p(BNode x l r)`

Αποδείξεις Ορθότητας
Επαγωγή για Κάθε Αλγεβρικό Τύπο

- Για να αποδείξουμε ότι κάθε πεπερασμένη δομή που ανήκει σε αλγεβρικό τύπο, ικανοποιεί τη συνθήκη `P`:
  - αποδεικνύουμε βάση επαγωγής για κάθε κατασκευαστή χωρίς αναδρομή
  - αποδεικνύουμε βήμα επαγωγής για κάθε κατασκευαστή με αναδρομή
- Επαγωγή σε πεπερασμένα δυαδικά δέντρα: για να δείξω ότι

$$\forall \texttt{t::(BTree a)}_{\texttt{df}}\texttt{. p t}$$

αρκεί

- `p EmptyTree`
- Για `x::a` και `l,r::(BTree a)`$_{\texttt{df}}$

  `p l && p r` $\Rightarrow$ `p(BNode x l r)`

Αποδείξεις Ορθότητας
Επαγωγή για Κάθε Αλγεβρικό Τύπο

- Για να αποδείξουμε ότι κάθε πεπερασμένη δομή που ανήκει σε αλγεβρικό τύπο, ικανοποιεί τη συνθήκη `P`:
  - αποδεικνύουμε βάση επαγωγής για κάθε κατασκευαστή χωρίς αναδρομή
  - αποδεικνύουμε βήμα επαγωγής για κάθε κατασκευαστή με αναδρομή
- Επαγωγή σε πεπερασμένα δυαδικά δέντρα: για να δείξω ότι

$$\forall \texttt{t::(BTree a)}_{\texttt{df}}\texttt{. p t}$$

αρκεί

- `p EmptyTree`
- Για `x::a` και `l,r::(BTree a)`$_{\texttt{df}}$

  `p l && p r` $\Rightarrow$ `p(BNode x l r)`

Αποδείξεις Ορθότητας
Επαγωγή για Κάθε Αλγεβρικό Τύπο

- Για να αποδείξουμε ότι κάθε πεπερασμένη δομή που ανήκει σε αλγεβρικό τύπο, ικανοποιεί τη συνθήκη `P`:
  - αποδεικνύουμε βάση επαγωγής για κάθε κατασκευαστή χωρίς αναδρομή
  - αποδεικνύουμε βήμα επαγωγής για κάθε κατασκευαστή με αναδρομή
- Επαγωγή σε πεπερασμένα δυαδικά δέντρα: για να δείξω ότι

$$\forall \texttt{t::(BTree a)}_{\texttt{df}}\texttt{. p t}$$

αρκεί

  - `p EmptyTree`
  - Για `x::a` και `l,r::(BTree a)`$_{\texttt{df}}$

    `p l && p r` $\Rightarrow$ `p(BNode x l r)`

- Για να αποδείξουμε ότι κάθε πεπερασμένη δομή που ανήκει σε αλγεβρικό τύπο, ικανοποιεί τη συνθήκη `P`:
  - αποδεικνύουμε βάση επαγωγής για κάθε κατασκευαστή χωρίς αναδρομή
  - αποδεικνύουμε βήμα επαγωγής για κάθε κατασκευαστή με αναδρομή
- Επαγωγή σε πεπερασμένα δυαδικά δέντρα: για να δείξω ότι

$$\forall \texttt{t::(BTree a)}_{\texttt{df}}\texttt{. p t}$$

αρκεί

  - `p EmptyTree`
  - Για `x::a` και `l,r::(BTree a)`$_{\texttt{df}}$

    `p l && p r` $\Rightarrow$ `p(BNode x l r)`

Αποδείξεις Ορθότητας

height και nodes - I

```
height :: BTree a -> Int
height EmptyBTree = 0
height (BNode _ l r) = max(height l)(height r) + 1

nodes :: BTree a -> Int
nodes EmptyBTree = 0
nodes (BNode _ l r) = nodes l + nodes r + 1
```

Προδιαγραφή (p t):

```
2^(height t) > nodes t
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
    2^(height (BNode _ l r))
=   2^(max(height l)(height r)+1)
=   2*2^(max(height l)(height r))
=     2^(max(height l)(height r))
    +  2^(max(height l)(height r))
>=  2^(height l) + 2^(height r)      επαγ.υπόθ.
>=  nodes l + 1 + 2^(height r)       επαγ.υπόθ.
>   nodes l + 1 + nodes r
=   nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
    2^(height (BNode _ l r))
=   2^(max(height l)(height r)+1)
=   2*2^(max(height l)(height r))
=     2^(max(height l)(height r))
    +  2^(max(height l)(height r))
>=  2^(height l) + 2^(height r)      επαγ.υπόθ.
>=  nodes l + 1 + 2^(height r)       επαγ.υπόθ.
>   nodes l + 1 + nodes r
=   nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
      2^(height (BNode _ l r))
=     2^(max(height l)(height r)+1)
=     2*2^(max(height l)(height r))
=       2^(max(height l)(height r))
      +  2^(max(height l)(height r))
>=    2^(height l) + 2^(height r)      επαγ.υπόθ.
>=    nodes l + 1 + 2^(height r)       επαγ.υπόθ.
>     nodes l + 1 + nodes r
=     nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
      2^(height (BNode _ l r))
=     2^(max(height l)(height r)+1)
=     2*2^(max(height l)(height r))
=       2^(max(height l)(height r))
      +  2^(max(height l)(height r))
>=    2^(height l) + 2^(height r)        επαγ.υπόθ.
>=    nodes l + 1 + 2^(height r)         επαγ.υπόθ.
>     nodes l + 1 + nodes r
=     nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
      2^(height (BNode _ l r))
=     2^(max(height l)(height r)+1)
=     2*2^(max(height l)(height r))
=       2^(max(height l)(height r))
      +  2^(max(height l)(height r))
>=    2^(height l) + 2^(height r)        επαγ.υπόθ.
>=    nodes l + 1 + 2^(height r)         επαγ.υπόθ.
>     nodes l + 1 + nodes r
=     nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
      2^(height (BNode _ l r))
=     2^(max(height l)(height r)+1)
=     2*2^(max(height l)(height r))
=       2^(max(height l)(height r))
      +  2^(max(height l)(height r))
>=    2^(height l) + 2^(height r)        επαγ.υπόθ.
>=    nodes l + 1 + 2^(height r)         επαγ.υπόθ.
>     nodes l + 1 + nodes r
=     nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
      2^(height (BNode _ l r))
=     2^(max(height l)(height r)+1)
=     2*2^(max(height l)(height r))
=       2^(max(height l)(height r))
      +  2^(max(height l)(height r))
>=    2^(height l) + 2^(height r)        επαγ.υπόθ.
>=    nodes l + 1 + 2^(height r)         επαγ.υπόθ.
>     nodes l + 1 + nodes r
=     nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
      2^(height (BNode _ l r))
=     2^(max(height l)(height r)+1)
=     2*2^(max(height l)(height r))
=       2^(max(height l)(height r))
      +  2^(max(height l)(height r))
>=    2^(height l) + 2^(height r)        επαγ.υπόθ.
>=    nodes l + 1 + 2^(height r)         επαγ.υπόθ.
>     nodes l + 1 + nodes r
=     nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
      2^(height (BNode _ l r))
=     2^(max(height l)(height r)+1)
=     2*2^(max(height l)(height r))
=       2^(max(height l)(height r))
      +  2^(max(height l)(height r))
>=    2^(height l) + 2^(height r)        επαγ.υπόθ.
>=    nodes l + 1 + 2^(height r)         επαγ.υπόθ.
>     nodes l + 1 + nodes r
=     nodes(BNode _ l r)
```

Αποδείξεις Ορθότητας

height και nodes - II

Βάση: `p EmptyTree = 2^0 > 0 = True`

Επαγ. υπόθ. `p l && p r`

Επαγ. βήμα:

```
      2^(height (BNode _ l r))
=     2^(max(height l)(height r)+1)
=     2*2^(max(height l)(height r))
=       2^(max(height l)(height r))
      +  2^(max(height l)(height r))
>=    2^(height l) + 2^(height r)        επαγ.υπόθ.
>=    nodes l + 1 + 2^(height r)         επαγ.υπόθ.
>     nodes l + 1 + nodes r
=     nodes(BNode _ l r)
```

Επισκόπηση - I

- Οι αλγεβρικοί τύποι
  - δημιουργούν απαριθμήσεις, τύπους-αθροίσματα
  - λύνουν το πρόβλημα της έκθετης δομής
  - επιτρέπουν την εισαγωγή αναδρομής στους τύπους
- Οι αλγεβρικοί τύποι ορίζονται με τη βοήθεια ενός ή πολλών κατασκευαστών.
  - τύποι απαρίθμησης = μόνο μηδενιαίοι κατασκευαστές
  - οι κατασκευαστές λύνουν το πρόβλημα της έκθετης δομής
  - οι κατασκευαστές μπαίνουν σε μοτίβα
- Η δήλωση `deriving` διευκολύνει τη δήλωση αλγεβρικών τύπων ως στιγμιότυπα συγκεκριμένων κλάσεων της Haskell
- Οι αλγεβρικοί τύποι μπορεί να είναι πολυμορφικοί
  - τύπος `Maybe a` για χειρισμό λαθών
  - τύπος `Either a b` για άθροισμα τύπων (`union`)

Επισκόπηση - I

- Οι αλγεβρικοί τύποι
  - δημιουργούν απαριθμήσεις, τύπους-αθροίσματα
  - λύνουν το πρόβλημα της έκθετης δομής
  - επιτρέπουν την εισαγωγή αναδρομής στους τύπους
- Οι αλγεβρικοί τύποι ορίζονται με τη βοήθεια ενός ή πολλών κατασκευαστών.
  - τύποι απαρίθμησης = μόνο μηδενιαίοι κατασκευαστές
  - οι κατασκευαστές λύνουν το πρόβλημα της έκθετης δομής
  - οι κατασκευαστές μπαίνουν σε μοτίβα
- Η δήλωση `deriving` διευκολύνει τη δήλωση αλγεβρικών τύπων ως στιγμιότυπα συγκεκριμένων κλάσεων της Haskell
- Οι αλγεβρικοί τύποι μπορεί να είναι πολυμορφικοί
  - τύπος `Maybe a` για χειρισμό λαθών
  - τύπος `Either a b` για άθροισμα τύπων (`union`)

Επισκόπηση - I

- Οι αλγεβρικοί τύποι
  - δημιουργούν απαριθμήσεις, τύπους-αθροίσματα
  - λύνουν το πρόβλημα της έκθετης δομής
  - επιτρέπουν την εισαγωγή αναδρομής στους τύπους
- Οι αλγεβρικοί τύποι ορίζονται με τη βοήθεια ενός ή πολλών κατασκευαστών.
  - τύποι απαρίθμησης = μόνο μηδενιαίοι κατασκευαστές
  - οι κατασκευαστές λύνουν το πρόβλημα της έκθετης δομής
  - οι κατασκευαστές μπαίνουν σε μοτίβα
- Η δήλωση `deriving` διευκολύνει τη δήλωση αλγεβρικών τύπων ως στιγμιότυπα συγκεκριμένων κλάσεων της Haskell
- Οι αλγεβρικοί τύποι μπορεί να είναι πολυμορφικοί
  - τύπος `Maybe a` για χειρισμό λαθών
  - τύπος `Either a b` για άθροισμα τύπων (`union`)

Επισκόπηση - I

- Οι αλγεβρικοί τύποι
  - δημιουργούν απαριθμήσεις, τύπους-αθροίσματα
  - λύνουν το πρόβλημα της έκθετης δομής
  - επιτρέπουν την εισαγωγή αναδρομής στους τύπους
- Οι αλγεβρικοί τύποι ορίζονται με τη βοήθεια ενός ή πολλών κατασκευαστών.
  - τύποι απαρίθμησης = μόνο μηδενιαίοι κατασκευαστές
  - οι κατασκευαστές λύνουν το πρόβλημα της έκθετης δομής
  - οι κατασκευαστές μπαίνουν σε μοτίβα
- Η δήλωση `deriving` διευκολύνει τη δήλωση αλγεβρικών τύπων ως στιγμιότυπα συγκεκριμένων κλάσεων της Haskell
- Οι αλγεβρικοί τύποι μπορεί να είναι πολυμορφικοί
  - τύπος `Maybe a` για χειρισμό λαθών
  - τύπος `Either a b` για άθροισμα τύπων (`union`)

Επισκόπηση - II

- Επιτρέπεται αναδρομή στους αλγεβρικούς τύπους
  - οι φυσικοί αριθμοί και οι λίστες είναι ήδη υποστηριζόμενοι
  - φτιάξαμε το πολυμορφικό δυαδικό δέντρο
  - φτιάξαμε το δέντρο εκφράσεων μιας γλώσσας προγραμματισμού (τυπική εφαρμογή Haskell)
- Εισαγωγή τετριμμένων περιπτώσεων (`0`, `[]`, `EmptyBTree`)
  - επιτρέπει μηδενική πληροφορία
  - διευκολύνει τον ορισμό των τύπων
- Επιτρέπονται άπειρες δομές, π.χ. άπειρα δέντρα στους αναδρομικούς τύπους
- Όλες οι επαγωγές είναι συνέπεια του θ. Knaster-Tarski
  - μη αναδρομικός κατασκευαστής $\Rightarrow$ βάση
  - αναδρομικός κατασκευαστής $\Rightarrow$ βήμα

Επισκόπηση - II

- Επιτρέπεται αναδρομή στους αλγεβρικούς τύπους
  - οι φυσικοί αριθμοί και οι λίστες είναι ήδη υποστηριζόμενοι
  - φτιάξαμε το πολυμορφικό δυαδικό δέντρο
  - φτιάξαμε το δέντρο εκφράσεων μιας γλώσσας προγραμματισμού (τυπική εφαρμογή Haskell)
- Εισαγωγή τετριμμένων περιπτώσεων (`0`, `[]`, `EmptyBTree`)
  - επιτρέπει μηδενική πληροφορία
  - διευκολύνει τον ορισμό των τύπων
- Επιτρέπονται άπειρες δομές, π.χ. άπειρα δέντρα στους αναδρομικούς τύπους
- Όλες οι επαγωγές είναι συνέπεια του θ. Knaster-Tarski
  - μη αναδρομικός κατασκευαστής $\Rightarrow$ βάση
  - αναδρομικός κατασκευαστής $\Rightarrow$ βήμα

Επισκόπηση - II

- Επιτρέπεται αναδρομή στους αλγεβρικούς τύπους
  - οι φυσικοί αριθμοί και οι λίστες είναι ήδη υποστηριζόμενοι
  - φτιάξαμε το πολυμορφικό δυαδικό δέντρο
  - φτιάξαμε το δέντρο εκφράσεων μιας γλώσσας προγραμματισμού (τυπική εφαρμογή Haskell)
- Εισαγωγή τετριμμένων περιπτώσεων (`0`, `[]`, `EmptyBTree`)
  - επιτρέπει μηδενική πληροφορία
  - διευκολύνει τον ορισμό των τύπων
- Επιτρέπονται άπειρες δομές, π.χ. άπειρα δέντρα στους αναδρομικούς τύπους
- Όλες οι επαγωγές είναι συνέπεια του θ. Knaster-Tarski
  - μη αναδρομικός κατασκευαστής $\Rightarrow$ βάση
  - αναδρομικός κατασκευαστής $\Rightarrow$ βήμα

Επισκόπηση - II

- Επιτρέπεται αναδρομή στους αλγεβρικούς τύπους
  - οι φυσικοί αριθμοί και οι λίστες είναι ήδη υποστηριζόμενοι
  - φτιάξαμε το πολυμορφικό δυαδικό δέντρο
  - φτιάξαμε το δέντρο εκφράσεων μιας γλώσσας προγραμματισμού (τυπική εφαρμογή Haskell)
- Εισαγωγή τετριμμένων περιπτώσεων (`0`, `[]`, `EmptyBTree`)
  - επιτρέπει μηδενική πληροφορία
  - διευκολύνει τον ορισμό των τύπων
- Επιτρέπονται άπειρες δομές, π.χ. άπειρα δέντρα στους αναδρομικούς τύπους
- Όλες οι επαγωγές είναι συνέπεια του θ. Knaster-Tarski
  - μη αναδρομικός κατασκευαστής $\Rightarrow$ βάση
  - αναδρομικός κατασκευαστής $\Rightarrow$ βήμα