# Οκνηρή Αποτίμηση / Αποδείξεις Ορθότητας

Γιάννης Κασσιός

# Δομή Σημερινής Διάλεξης

- Οκνηρή αποτίμηση (lazy evaluation)
  - κανόνες οκνηρής αποτίμησης στη Haskell
  - υπολογισμός οδηγούμενος από δεδομένα
  - άπειρες δομές
    - κατασκευή / χρήση / dovetailing
- Αποδείξεις ορθότητας (proofs of correctness)
  - αποδείξεις με προδιαγραφές vs. testing
  - χειρισμός μη τερματισμού
  - επαγωγή σε ακέραιους και λίστες
    - πεπερασμένες / άπειρες δομές
    - ισχυροποίηση θεωρήματος στην επαγωγή

# Δομή Σημερινής Διάλεξης

- Οκνηρή αποτίμηση (lazy evaluation)
  - κανόνες οκνηρής αποτίμησης στη Haskell
  - υπολογισμός οδηγούμενος από δεδομένα
  - άπειρες δομές
    - κατασκευή / χρήση / dovetailing
- Αποδείξεις ορθότητας (proofs of correctness)
  - αποδείξεις με προδιαγραφές vs. testing
  - χειρισμός μη τερματισμού
  - επαγωγή σε ακέραιους και λίστες
    - πεπερασμένες / άπειρες δομές
    - ισχυροποίηση θεωρήματος στην επαγωγή

# Δομή Σημερινής Διάλεξης

- Οκνηρή αποτίμηση (lazy evaluation)
  - κανόνες οκνηρής αποτίμησης στη Haskell
  - υπολογισμός οδηγούμενος από δεδομένα
  - άπειρες δομές
    - κατασκευή / χρήση / dovetailing
- Αποδείξεις ορθότητας (proofs of correctness)
  - αποδείξεις με προδιαγραφές vs. testing
  - χειρισμός μη τερματισμού
  - επαγωγή σε ακέραιους και λίστες
    - πεπερασμένες / άπειρες δομές
    - ισχυροποίηση θεωρήματος στην επαγωγή

# Δομή Σημερινής Διάλεξης

- Οκνηρή αποτίμηση (lazy evaluation)
  - κανόνες οκνηρής αποτίμησης στη Haskell
  - υπολογισμός οδηγούμενος από δεδομένα
  - άπειρες δομές
    - κατασκευή / χρήση / dovetailing
- Αποδείξεις ορθότητας (proofs of correctness)
  - αποδείξεις με προδιαγραφές vs. testing
  - χειρισμός μη τερματισμού
  - επαγωγή σε ακέραιους και λίστες
    - πεπερασμένες / άπειρες δομές
    - ισχυροποίηση θεωρήματος στην επαγωγή

# Δομή Σημερινής Διάλεξης

- Οκνηρή αποτίμηση (lazy evaluation)
  - κανόνες οκνηρής αποτίμησης στη Haskell
  - υπολογισμός οδηγούμενος από δεδομένα
  - άπειρες δομές
    - κατασκευή / χρήση / dovetailing
- Αποδείξεις ορθότητας (proofs of correctness)
  - αποδείξεις με προδιαγραφές vs. testing
  - χειρισμός μη τερματισμού
  - επαγωγή σε ακέραιους και λίστες
    - πεπερασμένες / άπειρες δομές
    - ισχυροποίηση θεωρήματος στην επαγωγή

# Δομή Σημερινής Διάλεξης

- Οκνηρή αποτίμηση (lazy evaluation)
  - κανόνες οκνηρής αποτίμησης στη Haskell
  - υπολογισμός οδηγούμενος από δεδομένα
  - άπειρες δομές
    - κατασκευή / χρήση / dovetailing
- Αποδείξεις ορθότητας (proofs of correctness)
  - αποδείξεις με προδιαγραφές vs. testing
  - χειρισμός μη τερματισμού
  - επαγωγή σε ακέραιους και λίστες
    - πεπερασμένες / άπειρες δομές
    - ισχυροποίηση θεωρήματος στην επαγωγή

# Δομή Σημερινής Διάλεξης

- Οκνηρή αποτίμηση (lazy evaluation)
  - κανόνες οκνηρής αποτίμησης στη Haskell
  - υπολογισμός οδηγούμενος από δεδομένα
  - άπειρες δομές
    - κατασκευή / χρήση / dovetailing
- Αποδείξεις ορθότητας (proofs of correctness)
  - αποδείξεις με προδιαγραφές vs. testing
  - χειρισμός μη τερματισμού
  - επαγωγή σε ακέραιους και λίστες
    - πεπερασμένες / άπειρες δομές
    - ισχυροποίηση θεωρήματος στην επαγωγή

# Δομή Σημερινής Διάλεξης

- Οκνηρή αποτίμηση (lazy evaluation)
  - κανόνες οκνηρής αποτίμησης στη Haskell
  - υπολογισμός οδηγούμενος από δεδομένα
  - άπειρες δομές
    - κατασκευή / χρήση / dovetailing
- Αποδείξεις ορθότητας (proofs of correctness)
  - αποδείξεις με προδιαγραφές vs. testing
  - χειρισμός μη τερματισμού
  - επαγωγή σε ακέραιους και λίστες
    - πεπερασμένες / άπειρες δομές
    - ισχυροποίηση θεωρήματος στην επαγωγή

# Οκνηρή Αποτίμηση
## Γενικά

- Οκνηρή αποτίμηση: μία δομή δεδομένων δεν κατασκευάζεται ολόκληρη όταν ορίζεται ή όταν περνάει σαν όρισμα σε μία συνάρτηση
- Μέρη της δομής κατασκευάζονται μόνο όταν χρειάζονται
- Βασικό κίνητρο: διαχωρισμός του ορισμού από τη χρήση μίας δομής

# Οκνηρή Αποτίμηση
## Γενικά

- Οκνηρή αποτίμηση: μία δομή δεδομένων δεν κατασκευάζεται ολόκληρη όταν ορίζεται ή όταν περνάει σαν όρισμα σε μία συνάρτηση
- Μέρη της δομής κατασκευάζονται μόνο όταν χρειάζονται
- Βασικό κίνητρο: διαχωρισμός του ορισμού από τη χρήση μίας δομής

# Οκνηρή Αποτίμηση
## Γενικά

- Οκνηρή αποτίμηση: μία δομή δεδομένων δεν κατασκευάζεται ολόκληρη όταν ορίζεται ή όταν περνάει σαν όρισμα σε μία συνάρτηση
- Μέρη της δομής κατασκευάζονται μόνο όταν χρειάζονται
- Βασικό κίνητρο: διαχωρισμός του ορισμού από τη χρήση μίας δομής

- Αποτίμηση `f x`:
  - Αποτίμηση της `f`
  - Τμηματική αποτίμηση της `x` για ταίριασμα μοτίβων
  - Πέρασμα της τμηματικά αποτιμημένης `x` στον κατάλληλο ορισμό
- Παράδειγμα:

```
head (x:_) = x
l = [n^2 | n<-[1..100]]
```

Αποτίμηση της `head l`:

- Για να ταιριάξει η `l` με το μοτίβο `x:_`

```
l = 1:[n^2 | n<-[2..100]]
```

- Πέρασμα παραμέτρων στην `head`: `x=1`
- Αποτίμηση του σώματος και επιστροφή `1`
- Η `l` παραμένει τμηματικά αποτιμημένη

- Αποτίμηση `f x`:
  - Αποτίμηση της `f`
  - Τμηματική αποτίμηση της `x` για ταίριασμα μοτίβων
  - Πέρασμα της τμηματικά αποτιμημένης `x` στον κατάλληλο ορισμό
- Παράδειγμα:

```
head (x:_) = x
l = [n^2 | n<-[1..100]]
```

Αποτίμηση της `head l`:

- Για να ταιριάξει η `l` με το μοτίβο `x:_`

```
l = 1:[n^2 | n<-[2..100]]
```

- Πέρασμα παραμέτρων στην `head`: `x=1`
- Αποτίμηση του σώματος και επιστροφή `1`
- Η `l` παραμένει τμηματικά αποτιμημένη

- Αποτίμηση `f x`:
  - Αποτίμηση της `f`
  - Τμηματική αποτίμηση της `x` για ταίριασμα μοτίβων
  - Πέρασμα της τμηματικά αποτιμημένης `x` στον κατάλληλο ορισμό
- Παράδειγμα:

```
head (x:_) = x
l = [n^2 | n<-[1..100]]
```

  Αποτίμηση της `head l`:
  - Για να ταιριάξει η `l` με το μοτίβο `x:_`

```
l = 1:[n^2 | n<-[2..100]]
```

  - Πέρασμα παραμέτρων στην `head`: `x=1`
  - Αποτίμηση του σώματος και επιστροφή `1`
  - Η `l` παραμένει τμηματικά αποτιμημένη

- Αποτίμηση `f x`:
  - Αποτίμηση της `f`
  - Τμηματική αποτίμηση της `x` για ταίριασμα μοτίβων
  - Πέρασμα της τμηματικά αποτιμημένης `x` στον κατάλληλο ορισμό
- Παράδειγμα:

  ```
  head (x:_) = x
  l = [n^2 | n<-[1..100]]
  ```

  Αποτίμηση της `head l`:
  - Για να ταιριάξει η `l` με το μοτίβο `x:_`

    ```
    l = 1:[n^2 | n<-[2..100]]
    ```
  - Πέρασμα παραμέτρων στην `head`: `x=1`
  - Αποτίμηση του σώματος και επιστροφή `1`
  - Η `l` παραμένει τμηματικά αποτιμημένη

- Αποτίμηση `f x`:
  - Αποτίμηση της `f`
  - Τμηματική αποτίμηση της `x` για ταίριασμα μοτίβων
  - Πέρασμα της τμηματικά αποτιμημένης `x` στον κατάλληλο ορισμό
- Παράδειγμα:

  ```
  head (x:_) = x
  l = [n^2 | n<-[1..100]]
  ```

  Αποτίμηση της `head l`:
  - Για να ταιριάξει η `l` με το μοτίβο `x:_`

    ```
    l = 1:[n^2 | n<-[2..100]]
    ```
  - Πέρασμα παραμέτρων στην `head`: `x=1`
  - Αποτίμηση του σώματος και επιστροφή `1`
  - Η `l` παραμένει τμηματικά αποτιμημένη

- Αποτίμηση `f x`:
  - Αποτίμηση της `f`
  - Τμηματική αποτίμηση της `x` για ταίριασμα μοτίβων
  - Πέρασμα της τμηματικά αποτιμημένης `x` στον κατάλληλο ορισμό
- Παράδειγμα:

  ```
  head (x:_) = x
  l = [n^2 | n<-[1..100]]
  ```

  Αποτίμηση της `head l`:
  - Για να ταιριάξει η `l` με το μοτίβο `x:_`

    ```
    l = 1:[n^2 | n<-[2..100]]
    ```
  - Πέρασμα παραμέτρων στην `head`: `x=1`
  - Αποτίμηση του σώματος και επιστροφή `1`
  - Η `l` παραμένει τμηματικά αποτιμημένη

- Αποτίμηση `f x`:
  - Πέρασμα εκφράσεων και όχι τιμών
  - Μερικές πραγματικές παράμετροι δεν αποτιμώνται καθόλου
- Παράδειγμα:

  ```
  c b t e = if b then t else e
  ```

  Αποτίμηση της `c condition (1+2) (3+4)`:
  - Πέρασμα παραμέτρων: `b=condition`, `t=(1+2)`, `e=(3+4)`
  - Αν `condition` τότε θα αποτιμηθεί μόνο η `t`, αλλιώς μόνο η `e`

- Αποτίμηση `f x`:
  - Πέρασμα εκφράσεων και όχι τιμών
  - Μερικές πραγματικές παράμετροι δεν αποτιμώνται καθόλου
- Παράδειγμα:

  ```
  c b t e = if b then t else e
  ```

  Αποτίμηση της `c condition (1+2) (3+4)`:
  - Πέρασμα παραμέτρων: `b=condition, t=(1+2), e=(3+4)`
  - Αν `condition` τότε θα αποτιμηθεί μόνο η `t`, αλλιώς μόνο η `e`

- Αποτίμηση `f x`:
  - Πέρασμα εκφράσεων και όχι τιμών
  - Μερικές πραγματικές παράμετροι δεν αποτιμώνται καθόλου
- Παράδειγμα:

  ```
  c b t e = if b then t else e
  ```

  Αποτίμηση της `c condition (1+2) (3+4)`:
  - Πέρασμα παραμέτρων: `b=condition, t=(1+2), e=(3+4)`
  - Αν `condition` τότε θα αποτιμηθεί μόνο η `t`, αλλιώς μόνο η `e`

- Αποτίμηση `f x`:
  - Πέρασμα εκφράσεων και όχι τιμών
  - Μερικές πραγματικές παράμετροι δεν αποτιμώνται καθόλου
- Παράδειγμα:

  ```
  c b t e = if b then t else e
  ```

  Αποτίμηση της `c condition (1+2) (3+4)`:
  - Πέρασμα παραμέτρων: `b=condition, t=(1+2), e=(3+4)`
  - Αν `condition` τότε θα αποτιμηθεί μόνο η `t`, αλλιώς μόνο η `e`

- Κάθε πραγματική παράμετρος αποτιμάται το πολύ μία φορά
- Παράδειγμα:

```
f x = x + x
```

Αποτίμηση της `f (5+5) (5+5)`:

- Πέρασμα παραμέτρων: `x=(5+5)`
- Το σώμα γίνεται `(5+5)+(5+5)`
- Η Haskell θυμάται ότι οι δύο υποεκφράσεις αναφέρονται στην `x`
- Η αποτίμηση της `x` θα γίνει μόνο μία φορά: `10+10`

- Κάθε πραγματική παράμετρος αποτιμάται το πολύ μία φορά
- Παράδειγμα:

  ```
  f x = x + x
  ```

  Αποτίμηση της `f (5+5) (5+5)`:

  - Πέρασμα παραμέτρων: `x=(5+5)`
  - Το σώμα γίνεται `(5+5)+(5+5)`
  - Η Haskell θυμάται ότι οι δύο υποεκφράσεις αναφέρονται στην `x`
  - Η αποτίμηση της `x` θα γίνει μόνο μία φορά: `10+10`

- Κάθε πραγματική παράμετρος αποτιμάται το πολύ μία φορά
- Παράδειγμα:

  ```
  f x = x + x
  ```

  Αποτίμηση της `f (5+5) (5+5)`:

  - Πέρασμα παραμέτρων: `x=(5+5)`
  - Το σώμα γίνεται `(5+5)+(5+5)`
  - Η Haskell θυμάται ότι οι δύο υποεκφράσεις αναφέρονται στην `x`
  - Η αποτίμηση της `x` θα γίνει μόνο μία φορά: `10+10`

- Κάθε πραγματική παράμετρος αποτιμάται το πολύ μία φορά
- Παράδειγμα:

  ```
  f x = x + x
  ```

  Αποτίμηση της `f (5+5) (5+5)`:

  - Πέρασμα παραμέτρων: `x=(5+5)`
  - Το σώμα γίνεται `(5+5)+(5+5)`
  - Η Haskell θυμάται ότι οι δύο υποεκφράσεις αναφέρονται στην `x`
  - Η αποτίμηση της `x` θα γίνει μόνο μία φορά: `10+10`

- Γενικότερα: κάθε επώνυμο δεδομένο αποτιμάται το πολύ μία φορά
- Παράδειγμα:

```
fibpair n = (   fst (fibpair (n-1))
              + snd (fibpair (n-1))
            , fst (fibpair (n-1))   )
```

vs.

```
fibpair n = (fst p + snd p , fst p)
   where p = fibpair (n-1)
```

- Γενικότερα: κάθε επώνυμο δεδομένο αποτιμάται το πολύ μία φορά
- Παράδειγμα:

```
fibpair n = (   fst (fibpair (n-1))
              + snd (fibpair (n-1))
            , fst (fibpair (n-1))   )
```

vs.

```
fibpair n = (fst p + snd p , fst p)
   where p = fibpair (n-1)
```

- `True || x` αποτιμάται σε `True` και `False && x` σε `False` χωρίς αποτίμηση του `x`
- Δεν ισχύει για `x || True` και `x && False`
  - αποτίμηση από αριστερά προς τα δεξιά

- `True || x` αποτιμάται σε `True` και `False && x` σε `False` χωρίς αποτίμηση του `x`
- Δεν ισχύει για `x || True` και `x && False`
  - αποτίμηση από αριστερά προς τα δεξιά

- Υπολογισμός οδηγούμενος από δεδομένα (data-driven computation): υπολογισμός μέσω ορισμού δεδομένων
- Οι δομές που ορίζουμε δεν κατασκευάζονται στην πράξη
- Παραδείγματα:

```
lSearch x l = foldr (||) False (map (x==) l)
minList = head.iSort
```

- Υπολογισμός οδηγούμενος από δεδομένα (data-driven computation): υπολογισμός μέσω ορισμού δεδομένων
- Οι δομές που ορίζουμε δεν κατασκευάζονται στην πράξη
- Παραδείγματα:

```
lSearch x l = foldr (||) False (map (x==) l)

minList = head.iSort
```

- Υπολογισμός οδηγούμενος από δεδομένα (data-driven computation): υπολογισμός μέσω ορισμού δεδομένων
- Οι δομές που ορίζουμε δεν κατασκευάζονται στην πράξη
- Παραδείγματα:

```
lSearch x l = foldr (||) False (map (x==) l)
minList = head.iSort
```

- Υπολογισμός οδηγούμενος από δεδομένα (data-driven computation): υπολογισμός μέσω ορισμού δεδομένων
- Οι δομές που ορίζουμε δεν κατασκευάζονται στην πράξη
- Παραδείγματα:

```
lSearch x l = foldr (||) False (map (x==) l)

minList = head.iSort
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `lSearch 2 [1,2,3]`

```
  lSearch 2 [1,2,3]
= foldr (||) False (map (2==) [1,2,3])
= foldr (||) False (map (2==) (1:[2,3]))
= foldr (||) False ((2==1):(map (2==) [2,3]))
= 2==1 || (foldr False (||) (map (2==) [2,3]))
= False || (foldr False (||) (map (2==) [2,3]))
= foldr False (||) (map (2==) [2,3])
= ...
= 2==2 || (foldr False (||) (map (2==) [3]))
= True || (foldr False (||) (map (2==) [3]))
= True
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

# Οκνηρή Αποτίμηση

## Υπολογισμός Οδηγούμενος από Δεδομένα: Αποτίμηση της `minList [3,2,1]`

```
  minList [3,2,1]
= head (iSort [3,2,1])
= ...
= head (ins 3 (ins 2 (ins 1 [])))
= head (ins 3 (ins 2 [1]))
= head (ins 3 (1: (ins 2 [])))
= head (1: (ins 3 (ins 2 [])))
= 1
```

- Παραδείγματα που έχουμε δει:

```
listOfOnes = 1:listOfOnes
```

```
nat = 0:  [n+1 | n<-nat]
```

- Άπειρες λίστες της Haskell:

```
[n..]=n:[n+1..]
```

```
[n,m..]=n:[m,2*m-n..]
```

- Παραδείγματα που έχουμε δει:

```
listOfOnes = 1:listOfOnes
```

```
nat = 0:  [n+1 | n<-nat]
```

- Άπειρες λίστες της Haskell:

```
[n..]=n:[n+1..]
```

```
[n,m..]=n:[m,2*m-n..]
```

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:
  `[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]`

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]
```

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]
```

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]
```

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]
```

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]
```

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]
```

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]
```

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

  `[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]`

- Πρόβλημα: παραγωγή της λίστας όλων των πρώτων αριθμών
- Λύση:
  - Ξεκινάμε από τη λίστα `[2..]`
  - Η κεφαλή της λίστας είναι πρώτος αριθμός
  - Φιλτράρουμε από τη λίστα όλα τα πολλαπλάσια της κεφαλής εκτός από την κεφαλή
  - Επαναλαμβάνουμε στην ουρά της λίστας
- Διαδικασία:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10 ..]
```

- Φιλτράρισμα πολλαπλάσιων της κεφαλής μιας λίστας:
  ```
  f (x:xs) = [y | y<-xs , y`mod`x /= 0]
  ```
- Θέλουμε να κρατήσουμε την κεφαλή:
  ```
  f (x:xs) = x:[y | y<-xs , y`mod`x /= 0]
  ```
- Θέλουμε να επαναλάβουμε για την ουρά της λίστας:
  ```
  f (x:xs) = x:f[y | y<-xs , y`mod`x /= 0]
  ```
- Θα πούμε τη συνάρτησή μας `sieve` (κόσκινο)
  ```
  sieve (x:xs) = x: sieve[y | y<-xs , y`mod`x /= 0]
  ```
- Λίστα πρώτων αριθμών: ξεκινάμε από την άπειρη λίστα `[2..]`
  ```
  primes = sieve [2..]
  ```
- Έλεγχος για πρώτο αριθμό:
  ```
  isPrime n = head (dropWhile (<n) primes) == n
  ```

# Οκνηρή Αποτίμηση

## Άπειρες Λίστες: το Κόσκινο του Ερατοσθένη-II

- Φιλτράρισμα πολλαπλάσιων της κεφαλής μιας λίστας:

```
f (x:xs) = [y | y<-xs , y`mod`x /= 0]
```

- Θέλουμε να κρατήσουμε την κεφαλή:

```
f (x:xs) = x:[y | y<-xs , y`mod`x /= 0]
```

- Θέλουμε να επαναλάβουμε για την ουρά της λίστας:

```
f (x:xs) = x:f[y | y<-xs , y`mod`x /= 0]
```

- Θα πούμε τη συνάρτησή μας `sieve` (κόσκινο)

```
sieve (x:xs) = x: sieve[y | y<-xs , y`mod`x /= 0]
```

- Λίστα πρώτων αριθμών: ξεκινάμε από την άπειρη λίστα `[2..]`

```
primes = sieve [2..]
```

- Έλεγχος για πρώτο αριθμό:

```
isPrime n = head (dropWhile (<n) primes) == n
```

# Οκνηρή Αποτίμηση

## Άπειρες Λίστες: το Κόσκινο του Ερατοσθένη-II

- Φιλτράρισμα πολλαπλάσιων της κεφαλής μιας λίστας:

```
f (x:xs) = [y | y<-xs , y`mod`x /= 0]
```

- Θέλουμε να κρατήσουμε την κεφαλή:

```
f (x:xs) = x:[y | y<-xs , y`mod`x /= 0]
```

- Θέλουμε να επαναλάβουμε για την ουρά της λίστας:

```
f (x:xs) = x:f[y | y<-xs , y`mod`x /= 0]
```

- Θα πούμε τη συνάρτησή μας `sieve` (κόσκινο)

```
sieve (x:xs) = x: sieve[y | y<-xs , y`mod`x /= 0]
```

- Λίστα πρώτων αριθμών: ξεκινάμε από την άπειρη λίστα `[2..]`

```
primes = sieve [2..]
```

- Έλεγχος για πρώτο αριθμό:

```
isPrime n = head (dropWhile (<n) primes) == n
```

# Οκνηρή Αποτίμηση

## Άπειρες Λίστες: το Κόσκινο του Ερατοσθένη-II

- Φιλτράρισμα πολλαπλάσιων της κεφαλής μιας λίστας:

```
f (x:xs) = [y | y<-xs , y`mod`x /= 0]
```

- Θέλουμε να κρατήσουμε την κεφαλή:

```
f (x:xs) = x:[y | y<-xs , y`mod`x /= 0]
```

- Θέλουμε να επαναλάβουμε για την ουρά της λίστας:

```
f (x:xs) = x:f[y | y<-xs , y`mod`x /= 0]
```

- Θα πούμε τη συνάρτησή μας `sieve` (κόσκινο)

```
sieve (x:xs) = x: sieve[y | y<-xs , y`mod`x /= 0]
```

- Λίστα πρώτων αριθμών: ξεκινάμε από την άπειρη λίστα `[2..]`

```
primes = sieve [2..]
```

- Έλεγχος για πρώτο αριθμό:

```
isPrime n = head (dropWhile (<n) primes) == n
```

- Φιλτράρισμα πολλαπλάσιων της κεφαλής μιας λίστας:
  ```
  f (x:xs) = [y | y<-xs , y`mod`x /= 0]
  ```
- Θέλουμε να κρατήσουμε την κεφαλή:
  ```
  f (x:xs) = x:[y | y<-xs , y`mod`x /= 0]
  ```
- Θέλουμε να επαναλάβουμε για την ουρά της λίστας:
  ```
  f (x:xs) = x:f[y | y<-xs , y`mod`x /= 0]
  ```
- Θα πούμε τη συνάρτησή μας `sieve` (κόσκινο)
  ```
  sieve (x:xs) = x: sieve[y | y<-xs , y`mod`x /= 0]
  ```
- Λίστα πρώτων αριθμών: ξεκινάμε από την άπειρη λίστα `[2..]`
  ```
  primes = sieve [2..]
  ```
- Έλεγχος για πρώτο αριθμό:
  ```
  isPrime n = head (dropWhile (<n) primes) == n
  ```

- Φιλτράρισμα πολλαπλάσιων της κεφαλής μιας λίστας:
```
f (x:xs) = [y | y<-xs , y`mod`x /= 0]
```
- Θέλουμε να κρατήσουμε την κεφαλή:
```
f (x:xs) = x:[y | y<-xs , y`mod`x /= 0]
```
- Θέλουμε να επαναλάβουμε για την ουρά της λίστας:
```
f (x:xs) = x:f[y | y<-xs , y`mod`x /= 0]
```
- Θα πούμε τη συνάρτησή μας `sieve` (κόσκινο)
```
sieve (x:xs) = x: sieve[y | y<-xs , y`mod`x /= 0]
```
- Λίστα πρώτων αριθμών: ξεκινάμε από την άπειρη λίστα `[2..]`
```
primes = sieve [2..]
```
- Έλεγχος για πρώτο αριθμό:
```
isPrime n = head (dropWhile (<n) primes) == n
```

# Οκνηρή Αποτίμηση
## Άπειρες Λίστες: Πυθαγόρειες Τριάδες

- Πρόβλημα: παραγωγή της λίστας όλων των τριάδων θετικών ακεραίων `(x,y,z)` ώστε `x^2+y^2 == z^2`
- Προτεινόμενη λύση:

```
pythagorean =
 [(x,y,z)
 | x<-[1..], y<-[1..], z<-[1..], x^2+y^2 == z^2
 ]
```

- Πρόβλημα: η `pythagorean!!0` δεν τερματίζει!
  - Έκφραση διαχωρισμού: ψάχνει πρώτα όλα τα `z` για `x=1` και `y=1`
  - Η λίστα των `z` είναι άπειρη: τα `x` και `y` δε θα προχωρήσουν ποτέ
- Πολλές άπειρες λίστες μπορούν να δημιουργήσουν πρόβλημα

# Οκνηρή Αποτίμηση

## Άπειρες Λίστες: Πυθαγόρειες Τριάδες

- Πρόβλημα: παραγωγή της λίστας όλων των τριάδων θετικών ακεραίων `(x,y,z)` ώστε `x^2+y^2 == z^2`
- Προτεινόμενη λύση:

```
pythagorean =
 [(x,y,z)
 | x<-[1..], y<-[1..], z<-[1..], x^2+y^2 == z^2
 ]
```

- Πρόβλημα: η `pythagorean!!0` δεν τερματίζει!
  - Έκφραση διαχωρισμού: ψάχνει πρώτα όλα τα `z` για `x=1` και `y=1`
  - Η λίστα των `z` είναι άπειρη: τα `x` και `y` δε θα προχωρήσουν ποτέ
- Πολλές άπειρες λίστες μπορούν να δημιουργήσουν πρόβλημα

# Οκνηρή Αποτίμηση

## Άπειρες Λίστες: Πυθαγόρειες Τριάδες

- Πρόβλημα: παραγωγή της λίστας όλων των τριάδων θετικών ακεραίων `(x,y,z)` ώστε `x^2+y^2 == z^2`
- Προτεινόμενη λύση:

```
pythagorean =
 [(x,y,z)
 | x<-[1..], y<-[1..], z<-[1..], x^2+y^2 == z^2
 ]
```

- Πρόβλημα: η `pythagorean!!0` δεν τερματίζει!
  - Έκφραση διαχωρισμού: ψάχνει πρώτα όλα τα `z` για `x=1` και `y=1`
  - Η λίστα των `z` είναι άπειρη: τα `x` και `y` δε θα προχωρήσουν ποτέ
- Πολλές άπειρες λίστες μπορούν να δημιουργήσουν πρόβλημα

# Οκνηρή Αποτίμηση
## Άπειρες Λίστες: Πυθαγόρειες Τριάδες

- Πρόβλημα: παραγωγή της λίστας όλων των τριάδων θετικών ακεραίων `(x,y,z)` ώστε `x^2+y^2 == z^2`
- Προτεινόμενη λύση:

```
pythagorean =
 [(x,y,z)
 | x<-[1..], y<-[1..], z<-[1..], x^2+y^2 == z^2
 ]
```

- Πρόβλημα: η `pythagorean!!0` δεν τερματίζει!
  - Έκφραση διαχωρισμού: ψάχνει πρώτα όλα τα `z` για `x=1` και `y=1`
  - Η λίστα των `z` είναι άπειρη: τα `x` και `y` δε θα προχωρήσουν ποτέ
- Πολλές άπειρες λίστες μπορούν να δημιουργήσουν πρόβλημα

# Οκνηρή Αποτίμηση
## Dovetailing

- Έχουμε πολλές άπειρες λίστες `l0,l1,...` και θέλουμε να πάρουμε μία άπειρη λίστα με όλους τους συνδυασμούς των στοιχείων από τις `l0,l1,...`
- Dovetailing
  - συνδυάζουμε τα στοιχεία με άθροισμα δεικτών `0` και μετά `1,2,...` κτλ.
- Παράδειγμα: `[2,..]` και `[-1,-2,..]`
  - βήμα 0: δείκτες `(0,0)`
  - βήμα 1: δείκτες `(0,1),(1,0)`
  - βήμα 2: δείκτες `(0,2),(1,1),(2,0)`
  - ...

`[(2,-1), (2,-2),(3,-1), (2,-3),(3,-2),(4,-1), ...]`

# Οκνηρή Αποτίμηση
## Dovetailing

- Έχουμε πολλές άπειρες λίστες `l0,l1,...` και θέλουμε να πάρουμε μία άπειρη λίστα με όλους τους συνδυασμούς των στοιχείων από τις `l0,l1,...`
- Dovetailing
  - συνδυάζουμε τα στοιχεία με άθροισμα δεικτών `0` και μετά `1,2,...` κτλ.
- Παράδειγμα: `[2,..]` και `[-1,-2,..]`
  - βήμα 0: δείκτες `(0,0)`
  - βήμα 1: δείκτες `(0,1),(1,0)`
  - βήμα 2: δείκτες `(0,2),(1,1),(2,0)`
  - ...

`[(2,-1), (2,-2),(3,-1), (2,-3),(3,-2),(4,-1),`
`...]`

# Οκνηρή Αποτίμηση
## Dovetailing

- Έχουμε πολλές άπειρες λίστες `l0,l1,...` και θέλουμε να πάρουμε μία άπειρη λίστα με όλους τους συνδυασμούς των στοιχείων από τις `l0,l1,...`
- Dovetailing
  - συνδυάζουμε τα στοιχεία με άθροισμα δεικτών `0` και μετά `1,2,...` κτλ.
- Παράδειγμα: `[2,..]` και `[-1,-2,..]`
  - βήμα 0: δείκτες `(0,0)`
  - βήμα 1: δείκτες `(0,1),(1,0)`
  - βήμα 2: δείκτες `(0,2),(1,1),(2,0)`
  - ...

```
[(2,-1), (2,-2),(3,-1), (2,-3),(3,-2),(4,-1),
...]
```

# Οκνηρή Αποτίμηση
## Dovetailing

- Έχουμε πολλές άπειρες λίστες `l0,l1,...` και θέλουμε να πάρουμε μία άπειρη λίστα με όλους τους συνδυασμούς των στοιχείων από τις `l0,l1,...`
- Dovetailing
  - συνδυάζουμε τα στοιχεία με άθροισμα δεικτών `0` και μετά `1,2,...` κτλ.
- Παράδειγμα: `[2,..]` και `[-1,-2,..]`
  - βήμα 0: δείκτες `(0,0)`
  - βήμα 1: δείκτες `(0,1),(1,0)`
  - βήμα 2: δείκτες `(0,2),(1,1),(2,0)`
  - ...

```
[(2,-1), (2,-2),(3,-1), (2,-3),(3,-2),(4,-1),
...]
```

# Οκνηρή Αποτίμηση
## Dovetailing

- Έχουμε πολλές άπειρες λίστες `l0,l1,...` και θέλουμε να πάρουμε μία άπειρη λίστα με όλους τους συνδυασμούς των στοιχείων από τις `l0,l1,...`
- Dovetailing
  - συνδυάζουμε τα στοιχεία με άθροισμα δεικτών `0` και μετά `1,2,...` κτλ.
- Παράδειγμα: `[2,..]` και `[-1,-2,..]`
  - βήμα 0: δείκτες `(0,0)`
  - βήμα 1: δείκτες `(0,1),(1,0)`
  - βήμα 2: δείκτες `(0,2),(1,1),(2,0)`
  - ...

```
[(2,-1), (2,-2),(3,-1), (2,-3),(3,-2),(4,-1),
...]
```

# Οκνηρή Αποτίμηση
## Dovetailing

- Έχουμε πολλές άπειρες λίστες `l0,l1,...` και θέλουμε να πάρουμε μία άπειρη λίστα με όλους τους συνδυασμούς των στοιχείων από τις `l0,l1,...`
- Dovetailing
  - συνδυάζουμε τα στοιχεία με άθροισμα δεικτών `0` και μετά `1,2,...` κτλ.
- Παράδειγμα: `[2,..]` και `[-1,-2,..]`
  - βήμα 0: δείκτες `(0,0)`
  - βήμα 1: δείκτες `(0,1),(1,0)`
  - βήμα 2: δείκτες `(0,2),(1,1),(2,0)`
  - ...

```
[(2,-1), (2,-2),(3,-1), (2,-3),(3,-2),(4,-1),
...]
```

# Οκνηρή Αποτίμηση
## Dovetailing

- Έχουμε πολλές άπειρες λίστες `l0,l1,...` και θέλουμε να πάρουμε μία άπειρη λίστα με όλους τους συνδυασμούς των στοιχείων από τις `l0,l1,...`
- Dovetailing
  - συνδυάζουμε τα στοιχεία με άθροισμα δεικτών `0` και μετά `1,2,...` κτλ.
- Παράδειγμα: `[2,..]` και `[-1,-2,..]`
  - βήμα 0: δείκτες `(0,0)`
  - βήμα 1: δείκτες `(0,1),(1,0)`
  - βήμα 2: δείκτες `(0,2),(1,1),(2,0)`
  - ...

```
[(2,-1), (2,-2),(3,-1), (2,-3),(3,-2),(4,-1),
...]
```

# Οκνηρή Αποτίμηση
## Dovetailing για 3 λίστες

- Τριάδες στοιχείων στο βήμα `n`:

```
[(l1!!i1), (l2!!i2), (l3!!i3)
| i1<-[0..n], i2<-[0..n], i3<-[0..n],
  i1+i2+i3 == n]
```

- Βοηθητική συνάρτηση `dovetail3aux` (για δημιουργία άπειρης λίστας):

```
dovetail3aux n l1 l2 l3 =
    [((l1!!i1), (l2!!i2), (l3!!i3))
    | i1<-[0..n], i2<-[0..n], i3<-[0..n],
      i1+i2+i3 == n]
  ++ dovetail3aux (n+1) l1 l2 l3
```

- Συνάρτηση `dovetail3` (για αρχικοποίηση από το `0`):

```
dovetail = dovetail3aux 0
```

- Τριάδες στοιχείων στο βήμα `n`:

```
[(l1!!i1), (l2!!i2), (l3!!i3)
| i1<-[0..n], i2<-[0..n], i3<-[0..n],
  i1+i2+i3 == n]
```

- Βοηθητική συνάρτηση `dovetail3aux` (για δημιουργία άπειρης λίστας):

```
dovetail3aux n l1 l2 l3 =
    [((l1!!i1), (l2!!i2), (l3!!i3))
    | i1<-[0..n], i2<-[0..n], i3<-[0..n],
      i1+i2+i3 == n]
  ++ dovetail3aux (n+1) l1 l2 l3
```

- Συνάρτηση `dovetail3` (για αρχικοποίηση από το `0`):

```
dovetail = dovetail3aux 0
```

- Τριάδες στοιχείων στο βήμα `n`:

```
[(l1!!i1), (l2!!i2), (l3!!i3)
| i1<-[0..n], i2<-[0..n], i3<-[0..n],
  i1+i2+i3 == n]
```

- Βοηθητική συνάρτηση `dovetail3aux` (για δημιουργία άπειρης λίστας):

```
dovetail3aux n l1 l2 l3 =
    [((l1!!i1), (l2!!i2), (l3!!i3))
    | i1<-[0..n], i2<-[0..n], i3<-[0..n],
      i1+i2+i3 == n]
  ++ dovetail3aux (n+1) l1 l2 l3
```

- Συνάρτηση `dovetail3` (για αρχικοποίηση από το `0`):

```
dovetail = dovetail3aux 0
```

```
pythagorean =
  [(x,y,z)
  | (x,y,z)<-dovetail3 [1..] [1..] [1..],
    x^2+y^2==z^2]
```

Ερώτηση: `take 10 pythagorean`

Απάντηση:

```
[(3,4,5),(4,3,5),(6,8,10),(8,6,10),(5,12,13),
(12,5,13),(9,12,15),(12,9,15),(8,15,17),(15,8,17)]
```

- Απόδειξη ορθότητας: μαθηματική απόδειξη ότι ένα πρόγραμμα ικανοποιεί μία προδιαγραφή (specification)
  - προδιαγραφή: επιθυμητή ιδιότητα εκφρασμένη σε τυπική γλώσσα
- Testing: τρέχουμε το πρόγραμμα για μερικά δεδομένα και εξετάζουμε την έξοδο
- Testing: αυτοματοποιημένο, βρίσκει λάθη, ένδειξη αλλά όχι εγγύηση ορθότητας
- Αποδείξεις: ανθρώπινη προσπάθεια, αποδεικνύει την ικανοποίηση μιας προδιαγραφής για όλες τις εισόδους

- Απόδειξη ορθότητας: μαθηματική απόδειξη ότι ένα πρόγραμμα ικανοποιεί μία προδιαγραφή (specification)
  - προδιαγραφή: επιθυμητή ιδιότητα εκφρασμένη σε τυπική γλώσσα
- Testing: τρέχουμε το πρόγραμμα για μερικά δεδομένα και εξετάζουμε την έξοδο
- Testing: αυτοματοποιημένο, βρίσκει λάθη, ένδειξη αλλά όχι εγγύηση ορθότητας
- Αποδείξεις: ανθρώπινη προσπάθεια, αποδεικνύει την ικανοποίηση μιας προδιαγραφής για όλες τις εισόδους

- Απόδειξη ορθότητας: μαθηματική απόδειξη ότι ένα πρόγραμμα ικανοποιεί μία προδιαγραφή (specification)
  - προδιαγραφή: επιθυμητή ιδιότητα εκφρασμένη σε τυπική γλώσσα
- Testing: τρέχουμε το πρόγραμμα για μερικά δεδομένα και εξετάζουμε την έξοδο
- Testing: αυτοματοποιημένο, βρίσκει λάθη, ένδειξη αλλά όχι εγγύηση ορθότητας
- Αποδείξεις: ανθρώπινη προσπάθεια, αποδεικνύει την ικανοποίηση μιας προδιαγραφής για όλες τις εισόδους

- Απόδειξη ορθότητας: μαθηματική απόδειξη ότι ένα πρόγραμμα ικανοποιεί μία προδιαγραφή (specification)
  - προδιαγραφή: επιθυμητή ιδιότητα εκφρασμένη σε τυπική γλώσσα
- Testing: τρέχουμε το πρόγραμμα για μερικά δεδομένα και εξετάζουμε την έξοδο
- Testing: αυτοματοποιημένο, βρίσκει λάθη, ένδειξη αλλά όχι εγγύηση ορθότητας
- Αποδείξεις: ανθρώπινη προσπάθεια, αποδεικνύει την ικανοποίηση μιας προδιαγραφής για όλες τις εισόδους

# Αποδείξεις Ορθότητας
## Χειρισμός Μη Τερματισμού

- Τιμή υπολογισμού που δεν τερματίζει: "μη ορισμένη" `undef`

  `undef = undef`
  - Η `undef` ανήκει σε όλους τους τύπους
- Ορισμένη βασική τιμή `x`: `x`$\neq$`undef`
- Λίστες με `undef`:

  `undef` `2:undef` `2:3:undef`

  `undef:[2,3]` `[1,undef,3]` `undef:undef`

  `[0,[1,undef],2]` `[0,[1]:undef,2]`
- Πεπερασμένη ορισμένη λίστα: `l`
  - `length l` ορισμένο
  - για κάθε φυσικό αριθμό `i`, ώστε `i<length l`, το `l!!i` είναι ορισμένη βασική τιμή ή πεπερασμένη ορισμένη λίστα
- Συμβολισμός: $T_d$ $[T]_f$ $[T]_{df}$

# Αποδείξεις Ορθότητας
## Χειρισμός Μη Τερματισμού

- Τιμή υπολογισμού που δεν τερματίζει: "μη ορισμένη" `undef`

  `undef = undef`

  - Η `undef` ανήκει σε όλους τους τύπους
- Ορισμένη βασική τιμή `x`: `x`$\neq$`undef`
- Λίστες με `undef`:

  `undef` `2:undef` `2:3:undef`

  `undef:[2,3]` `[1,undef,3]` `undef:undef`

  `[0,[1,undef],2]` `[0,[1]:undef,2]`
- Πεπερασμένη ορισμένη λίστα: `l`
  - `length l` ορισμένο
  - για κάθε φυσικό αριθμό `i`, ώστε `i<length l`, το `l!!i` είναι ορισμένη βασική τιμή ή πεπερασμένη ορισμένη λίστα
- Συμβολισμός: $T_d$ $[T]_f$ $[T]_{df}$

# Αποδείξεις Ορθότητας
## Χειρισμός Μη Τερματισμού

- Τιμή υπολογισμού που δεν τερματίζει: "μη ορισμένη" `undef`

  `undef = undef`

  - Η `undef` ανήκει σε όλους τους τύπους
- Ορισμένη βασική τιμή `x`: `x`$\neq$`undef`
- Λίστες με `undef`:

  `undef` `2:undef` `2:3:undef`

  `undef:[2,3]` `[1,undef,3]` `undef:undef`

  `[0,[1,undef],2]` `[0,[1]:undef,2]`
- Πεπερασμένη ορισμένη λίστα: `l`
  - `length l` ορισμένο
  - για κάθε φυσικό αριθμό `i`, ώστε `i<length l`, το `l!!i` είναι ορισμένη βασική τιμή ή πεπερασμένη ορισμένη λίστα
- Συμβολισμός: $T_d$ $[T]_f$ $[T]_{df}$

# Αποδείξεις Ορθότητας
## Χειρισμός Μη Τερματισμού

- Τιμή υπολογισμού που δεν τερματίζει: "μη ορισμένη" `undef`

  `undef = undef`
  - Η `undef` ανήκει σε όλους τους τύπους
- Ορισμένη βασική τιμή `x`: `x`$\neq$`undef`
- Λίστες με `undef`:

  `undef` `2:undef` `2:3:undef`

  `undef:[2,3]` `[1,undef,3]` `undef:undef`

  `[0,[1,undef],2]` `[0,[1]:undef,2]`
- Πεπερασμένη ορισμένη λίστα: `l`
  - `length l` ορισμένο
  - για κάθε φυσικό αριθμό `i`, ώστε `i<length l`, το `l!!i` είναι ορισμένη βασική τιμή ή πεπερασμένη ορισμένη λίστα
- Συμβολισμός: $T_d$ $[T]_f$ $[T]_{df}$

# Αποδείξεις Ορθότητας
## Χειρισμός Μη Τερματισμού

- Τιμή υπολογισμού που δεν τερματίζει: "μη ορισμένη" `undef`

  `undef = undef`

  - Η `undef` ανήκει σε όλους τους τύπους
- Ορισμένη βασική τιμή `x`: `x`$\neq$`undef`
- Λίστες με `undef`:

  `undef` `2:undef` `2:3:undef`

  `undef:[2,3]` `[1,undef,3]` `undef:undef`

  `[0,[1,undef],2]` `[0,[1]:undef,2]`
- Πεπερασμένη ορισμένη λίστα: `l`
  - `length l` ορισμένο
  - για κάθε φυσικό αριθμό `i`, ώστε `i<length l`, το `l!!i` είναι ορισμένη βασική τιμή ή πεπερασμένη ορισμένη λίστα
- Συμβολισμός: $T_d$ $[T]_f$ $[T]_{df}$

# Αποδείξεις Ορθότητας
## Χειρισμός Μη Τερματισμού

- Τιμή υπολογισμού που δεν τερματίζει: "μη ορισμένη" `undef`

  `undef = undef`
  - Η `undef` ανήκει σε όλους τους τύπους
- Ορισμένη βασική τιμή `x`: `x`$\neq$`undef`
- Λίστες με `undef`:

  `undef` `2:undef` `2:3:undef`

  `undef:[2,3]` `[1,undef,3]` `undef:undef`

  `[0,[1,undef],2]` `[0,[1]:undef,2]`
- Πεπερασμένη ορισμένη λίστα: `l`
  - `length l` ορισμένο
  - για κάθε φυσικό αριθμό `i`, ώστε `i<length l`, το `l!!i` είναι ορισμένη βασική τιμή ή πεπερασμένη ορισμένη λίστα
- Συμβολισμός: $\mathtt{T}_{\mathtt{d}}$ $[\mathtt{T}]_{\mathtt{f}}$ $[\mathtt{T}]_{\mathtt{df}}$

- Κατηγόρημα (predicate): `p :: ...->Bool`$_{\mathtt{d}}$
  - δε γράφουμε μόνο Haskell, πχ. $\forall \Rightarrow$ κτλ.
- Επαγωγή (induction) φυσικών αριθμών. Αν:
  - Βάση επαγωγής (induction base): `p 0`
  - Βήμα επαγωγής (induction step):
    `∀n<-[0..]. p n⇒p(n+1)`

Τότε: `∀n<-[0..]. p n`

- Κατηγόρημα (predicate): `p :: ...->Bool`$_{\mathtt{d}}$
  - δε γράφουμε μόνο Haskell, πχ. $\forall \Rightarrow$ κτλ.
- Επαγωγή (induction) φυσικών αριθμών. Αν:
  - Βάση επαγωγής (induction base): `p 0`
  - Βήμα επαγωγής (induction step):
    `∀n<-[0..]. p n⇒p(n+1)`
    - `p n`: επαγωγική υπόθεση (induction hypothesis)

Τότε: `∀n<-[0..]. p n`

- Κατηγόρημα (predicate): `p :: ...->Bool`$_d$
  - δε γράφουμε μόνο Haskell, πχ. $\forall \Rightarrow$ κτλ.
- Επαγωγή (induction) φυσικών αριθμών. Αν:
  - Βάση επαγωγής (induction base): `p 0`
  - Βήμα επαγωγής (induction step):
    `∀n<-[0..]. p n⇒p(n+1)`
    - `p n`: επαγωγική υπόθεση (induction hypothesis)

Τότε: `∀n<-[0..]. p n`

- Κατηγόρημα (predicate): `p :: ...->Bool`$_d$
  - δε γράφουμε μόνο Haskell, πχ. $\forall \Rightarrow$ κτλ.
- Επαγωγή (induction) φυσικών αριθμών. Αν:
  - Βάση επαγωγής (induction base): `p 0`
  - Βήμα επαγωγής (induction step):
    $\forall$`n<-[0..]. p n`$\Rightarrow$`p(n+1)`
    - `p n`: επαγωγική υπόθεση (induction hypothesis)

Τότε: $\forall$`n<-[0..]. p n`

- Κατηγόρημα (predicate): `p :: ...->Bool`$_{\texttt{d}}$
  - δε γράφουμε μόνο Haskell, πχ. $\forall \Rightarrow$ κτλ.
- Επαγωγή (induction) φυσικών αριθμών. Αν:
  - Βάση επαγωγής (induction base): `p 0`
  - Βήμα επαγωγής (induction step):
    `∀n<-[0..]. p n⇒p(n+1)`
    - `p n`: επαγωγική υπόθεση (induction hypothesis)

Τότε: `∀n<-[0..]. p n`

- Κατηγόρημα (predicate): `p :: ...->Bool`$_d$
  - δε γράφουμε μόνο Haskell, πχ. $\forall \Rightarrow$ κτλ.
- Επαγωγή (induction) φυσικών αριθμών. Αν:
  - Βάση επαγωγής (induction base): `p 0`
  - Βήμα επαγωγής (induction step):
    $\forall$`n<-[0..]. p n`$\Rightarrow$`p(n+1)`
    - `p n`: επαγωγική υπόθεση (induction hypothesis)

  Τότε: $\forall$`n<-[0..]. p n`

# Αποδείξεις Ορθότητας

## Επαγωγή σε Ακεραίους - II

- Επαγωγή ορισμένων ακεραίων. Αν:
  - Βάση επαγωγής: $\texttt{p 0}$
  - Βήμα επαγωγής:

    $\forall \texttt{n::Int}_{\texttt{d}}\texttt{.}\quad \texttt{p n} \Rightarrow \texttt{p(n+1)}$

    $\forall \texttt{n::Int}_{\texttt{d}}\texttt{.}\quad \texttt{p n} \Rightarrow \texttt{p(n-1)}$

  Τότε: $\forall \texttt{n::Int}_{\texttt{d}}\texttt{.}\quad \texttt{p n}$
- Επαγωγή ακεραίων. Αν:
  - Βάση επαγωγής: $\texttt{p 0}$ και $\texttt{p undef}$
  - Βήμα επαγωγής:

    $\forall \texttt{n::Int}_{\texttt{d}}\texttt{.}\quad \texttt{p n} \Rightarrow \texttt{p(n+1)}$

    $\forall \texttt{n::Int}_{\texttt{d}}\texttt{.}\quad \texttt{p n} \Rightarrow \texttt{p(n-1)}$

  Τότε: $\forall \texttt{n::Int.}\quad \texttt{p n}$

- Επαγωγή ορισμένων ακεραίων. Αν:
  - Βάση επαγωγής: `p 0`
  - Βήμα επαγωγής:

    $\forall$`n::Int`$_d$`. p n`$\Rightarrow$`p(n+1)`

    $\forall$`n::Int`$_d$`. p n`$\Rightarrow$`p(n-1)`

  Τότε: $\forall$`n::Int`$_d$`. p n`
- Επαγωγή ακεραίων. Αν:
  - Βάση επαγωγής: `p 0` και `p undef`
  - Βήμα επαγωγής:

    $\forall$`n::Int`$_d$`. p n`$\Rightarrow$`p(n+1)`

    $\forall$`n::Int`$_d$`. p n`$\Rightarrow$`p(n-1)`

  Τότε: $\forall$`n::Int. p n`

# Αποδείξεις Ορθότητας

## Παραγοντικό - I

- ```
  factorial 0 = 1
  factorial n = n*factorial(n-1)
  ```
- Προδιαγραφή: `∀n<-[0..]. p n`, όπου
  `p n = ∀k<-[1..n]. factorial n ‘mod‘ k == 0`
- Απόδειξη με επαγωγή σε φυσικούς. Βάση επαγωγής:

```
    p 0
=   (∀k<-[1..0]. factorial n ‘mod‘ k == 0)
=   (∀k<-[]. factorial n ‘mod‘ k == 0)
=   True
```
```

# Αποδείξεις Ορθότητας

## Παραγοντικό - I

- `factorial 0 = 1`

  `factorial n = n*factorial(n-1)`
- Προδιαγραφή: `∀n<-[0..]. p n`, όπου

  `p n = ∀k<-[1..n]. factorial n ‘mod‘ k == 0`
- Απόδειξη με επαγωγή σε φυσικούς. Βάση επαγωγής:

```
p 0
= (∀k<-[1..0]. factorial n ‘mod‘ k == 0)
= (∀k<-[]. factorial n ‘mod‘ k == 0)
= True
```

# Αποδείξεις Ορθότητας

## Παραγοντικό - I

- `factorial 0 = 1`

  `factorial n = n*factorial(n-1)`
- Προδιαγραφή: `∀n<-[0..]. p n`, όπου

  `p n = ∀k<-[1..n]. factorial n ‘mod‘ k == 0`
- Απόδειξη με επαγωγή σε φυσικούς. Βάση επαγωγής:

```
    p 0
=   (∀k<-[1..0]. factorial n ‘mod‘ k == 0)
=   (∀k<-[]. factorial n ‘mod‘ k == 0)
=   True
```

# Αποδείξεις Ορθότητας

## Παραγοντικό - I

- `factorial 0 = 1`

  `factorial n = n*factorial(n-1)`
- Προδιαγραφή: `∀n<-[0..]. p n`, όπου

  `p n = ∀k<-[1..n]. factorial n ‘mod‘ k == 0`
- Απόδειξη με επαγωγή σε φυσικούς. Βάση επαγωγής:

```
    p 0
=   (∀k<-[1..0]. factorial n ‘mod‘ k == 0)
=   (∀k<-[]. factorial n ‘mod‘ k == 0)
=   True
```

# Αποδείξεις Ορθότητας

## Παραγοντικό - I

- `factorial 0 = 1`

  `factorial n = n*factorial(n-1)`
- Προδιαγραφή: `∀n<-[0..]. p n`, όπου

  `p n = ∀k<-[1..n]. factorial n ‘mod‘ k == 0`
- Απόδειξη με επαγωγή σε φυσικούς. Βάση επαγωγής:

```
    p 0
=   (∀k<-[1..0]. factorial n ‘mod‘ k == 0)
=   (∀k<-[]. factorial n ‘mod‘ k == 0)
=   True
```

# Αποδείξεις Ορθότητας

## Παραγοντικό - I

- `factorial 0 = 1`

  `factorial n = n*factorial(n-1)`
- Προδιαγραφή: `∀n<-[0..]. p n`, όπου

  `p n = ∀k<-[1..n]. factorial n ‘mod‘ k == 0`
- Απόδειξη με επαγωγή σε φυσικούς. Βάση επαγωγής:

$$
\begin{aligned}
& \texttt{p 0} \\
= \quad & (\forall \texttt{k<-[1..0].} \quad \texttt{factorial n ‘mod‘ k == 0}) \\
= \quad & (\forall \texttt{k<-[].} \quad \texttt{factorial n ‘mod‘ k == 0}) \\
= \quad & \texttt{True}
\end{aligned}
$$

Βήμα επαγωγής. Επαγωγική υπόθεση:

```
∀k<-[1..n].  factorial n `mod` k  ==  0
```

Απόδειξη:

```
    p(n+1)
=   (∀k<-[1..n+1]. factorial(n+1) `mod` k  ==  0)
=   (   factorial(n+1) `mod` (n+1) == 0
    &&  ∀k<-[1..n]. factorial(n+1) `mod` k == 0
    )
=   (∀k<-[1..n]. ((n+1)*(factorial n))`mod`k == 0)
          επαγ. υπόθ.
=   True
```

Βήμα επαγωγής. Επαγωγική υπόθεση:

```
∀k<-[1..n].  factorial n `mod` k  ==  0
```

Απόδειξη:

```
    p(n+1)
=   (∀k<-[1..n+1]. factorial(n+1) `mod` k  ==  0)
=   (   factorial(n+1) `mod` (n+1) == 0
    &&  ∀k<-[1..n]. factorial(n+1) `mod` k == 0
    )
=   (∀k<-[1..n]. ((n+1)*(factorial n))`mod`k == 0)
          επαγ. υπόθ.
=   True
```

Βήμα επαγωγής. Επαγωγική υπόθεση:

```
∀k<-[1..n].  factorial n `mod` k  ==  0
```

Απόδειξη:

```
    p(n+1)
=   (∀k<-[1..n+1]. factorial(n+1) `mod` k  ==  0)
=   (   factorial(n+1) `mod` (n+1) == 0
    &&  ∀k<-[1..n]. factorial(n+1) `mod` k == 0
    )
=   (∀k<-[1..n]. ((n+1)*(factorial n))`mod`k == 0)
                επαγ. υπόθ.
=   True
```

Βήμα επαγωγής. Επαγωγική υπόθεση:

```
∀k<-[1..n].  factorial n `mod` k  ==  0
```

Απόδειξη:

```
    p(n+1)
=   (∀k<-[1..n+1]. factorial(n+1) `mod` k  ==  0)
=   (   factorial(n+1) `mod` (n+1) == 0
    &&  ∀k<-[1..n]. factorial(n+1) `mod` k == 0
    )
=   (∀k<-[1..n]. ((n+1)*(factorial n))`mod`k == 0)
          επαγ. υπόθ.
=   True
```

- Επαγωγή πεπερασμένων ορισμένων λιστών. Αν:
  - Βάση επαγωγής: `p []`
  - Βήμα επαγωγής:
    $\forall$`x::T`$_{\texttt{df}}$`, xs::[T]`$_{\texttt{df}}$`. p xs` $\Rightarrow$`p(x:xs)`

  Τότε: $\forall$`l::[T]`$_{\texttt{df}}$`. p l`
- Επαγωγή πεπερασμένων λιστών. Αν:
  - Βάση επαγωγής: `p []` και `p undef`
  - Βήμα επαγωγής:
    $\forall$`x::T`$_{\texttt{f}}$`, xs::[T]`$_{\texttt{f}}$`. p xs` $\Rightarrow$`p(x:xs)`

  Τότε: $\forall$`l::[T]`$_{\texttt{f}}$`. p l`

- Επαγωγή πεπερασμένων ορισμένων λιστών. Αν:
  - Βάση επαγωγής: `p []`
  - Βήμα επαγωγής:
    $\forall$`x::T`$_{\texttt{df}}$`, xs::[T]`$_{\texttt{df}}$`. p xs` $\Rightarrow$`p(x:xs)`

  Τότε: $\forall$`l::[T]`$_{\texttt{df}}$`. p l`
- Επαγωγή πεπερασμένων λιστών. Αν:
  - Βάση επαγωγής: `p []` και `p undef`
  - Βήμα επαγωγής:
    $\forall$`x::T`$_{\texttt{f}}$`, xs::[T]`$_{\texttt{f}}$`. p xs` $\Rightarrow$`p(x:xs)`

  Τότε: $\forall$`l::[T]`$_{\texttt{f}}$`. p l`

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `reverse` - I

- ```
  reverse [] = []
  reverse (x:xs) = reverse xs ++ [x]
  ```
- Προδιαγραφή:

```
        length l == length(reverse l)
    &&  ∀i<-[0..length l-1].
          (reverse l)!!i == l!!(length l-1-i)
```

- Πρώτο κομμάτι: `length l == length(reverse l).`
  Βάση επαγωγής: `length[] == length(reverse[])`
  τετριμμένη
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `reverse` - I

- ```
  reverse [] = []
  reverse (x:xs) = reverse xs ++ [x]
  ```
- Προδιαγραφή:

```
        length l == length(reverse l)
  &&  ∀i<-[0..length l-1].
        (reverse l)!!i == l!!(length l-1-i)
```

- Πρώτο κομμάτι: `length l == length(reverse l).`
  Βάση επαγωγής: `length[] == length(reverse[])`
  τετριμμένη
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `reverse` - Ι

- ```
  reverse [] = []
  reverse (x:xs) = reverse xs ++ [x]
  ```
- Προδιαγραφή:

```
        length l == length(reverse l)
  &&  ∀i<-[0..length l-1].
        (reverse l)!!i == l!!(length l-1-i)
```

- Πρώτο κομμάτι: `length l == length(reverse l).`
  Βάση επαγωγής: `length[] == length(reverse[])`
  τετριμμένη
```

Βήμα επαγωγής. Επαγ.Υπόθεση:

```
length xs == length(reverse xs)
```

Απόδειξη:

```
    length(reverse(x:xs))
=   length(reverse xs++[x])
=   length(reverse xs) + 1    επαγ. υποθ.
=   length xs + 1
=   length(x:xs)
```

Βήμα επαγωγής. Επαγ.Υπόθεση:

```
length xs == length(reverse xs)
```

Απόδειξη:

```
    length(reverse(x:xs))
=   length(reverse xs++[x])
=   length(reverse xs) + 1    επαγ. υποθ.
=   length xs + 1
=   length(x:xs)
```

Βήμα επαγωγής. Επαγ.Υπόθεση:

```
length xs == length(reverse xs)
```

Απόδειξη:

```
    length(reverse(x:xs))
=   length(reverse xs++[x])
=   length(reverse xs) + 1    επαγ. υποθ.
=   length xs + 1
=   length(x:xs)
```

Βήμα επαγωγής. Επαγ.Υπόθεση:

```
length xs == length(reverse xs)
```

Απόδειξη:

```
    length(reverse(x:xs))
=   length(reverse xs++[x])
=   length(reverse xs) + 1    επαγ. υποθ.
=   length xs + 1
=   length(x:xs)
```

Βήμα επαγωγής. Επαγ.Υπόθεση:

```
length xs == length(reverse xs)
```

Απόδειξη:

```
    length(reverse(x:xs))
=   length(reverse xs++[x])
=   length(reverse xs) + 1    επαγ. υποθ.
=   length xs + 1
=   length(x:xs)
```

Βήμα επαγωγής. Επαγ.Υπόθεση:

```
length xs == length(reverse xs)
```

Απόδειξη:

```
    length(reverse(x:xs))
=   length(reverse xs++[x])
=   length(reverse xs) + 1    επαγ. υποθ.
=   length xs + 1
=   length(x:xs)
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `reverse` - III

Δεύτερο κομμάτι:

```
∀i<-[0..length l-1].
   (reverse l)!!i == xs!!(length l-1-i)
```

Βάση επαγωγής τετριμμένη.

Βήμα επαγωγής. Επαγ. Υπόθεση:

```
∀i<-[0..length xs-1].
   (reverse xs)!!i == xs!!(length xs-1-i)
```

Απόδειξη: (α) `i==length xs`:

```
   reverse(x:xs)!!i
=  (reverse xs++[x])!!(length xs)
=  x
=  (x:xs)!!(length(x:xs)-1-i)
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `reverse` - III

Δεύτερο κομμάτι:

```
∀i<-[0..length l-1].
   (reverse l)!!i == xs!!(length l-1-i)
```

Βάση επαγωγής τετριμμένη.
Βήμα επαγωγής. Επαγ. Υπόθεση:

```
∀i<-[0..length xs-1].
   (reverse xs)!!i == xs!!(length xs-1-i)
```

Απόδειξη: (α) `i==length xs`:

```
   reverse(x:xs)!!i
=  (reverse xs++[x])!!(length xs)
=  x
=  (x:xs)!!(length(x:xs)-1-i)
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `reverse` - III

Δεύτερο κομμάτι:

```
∀i<-[0..length l-1].
   (reverse l)!!i == xs!!(length l-1-i)
```

Βάση επαγωγής τετριμμένη.
Βήμα επαγωγής. Επαγ. Υπόθεση:

```
∀i<-[0..length xs-1].
   (reverse xs)!!i == xs!!(length xs-1-i)
```

Απόδειξη: (α) `i==length xs`:

$$
\begin{aligned}
& \texttt{reverse(x:xs)!!i} \\
= \;& \texttt{(reverse xs++[x])!!(length xs)} \\
= \;& \texttt{x} \\
= \;& \texttt{(x:xs)!!(length(x:xs)-1-i)}
\end{aligned}
$$

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `reverse` - IV

(β) `i<length xs`:

```
    (reverse(x:xs))!!i
=   (reverse xs ++ [x])!!i
=   (reverse xs)!!i                    επαγ. υποθ.
=   xs!!(length xs-1-i)
=   (x:xs)!!(length xs-1-i+1)
=   (x:xs)!!(length(x:xs)-1-i)
```

(β) `i<length xs`:

```
    (reverse(x:xs))!!i
=   (reverse xs ++ [x])!!i
=   (reverse xs)!!i                    επαγ. υποθ.
=   xs!!(length xs-1-i)
=   (x:xs)!!(length xs-1-i+1)
=   (x:xs)!!(length(x:xs)-1-i)
```

(β) `i<length xs`:

```
      (reverse(x:xs))!!i
=     (reverse xs ++ [x])!!i
=     (reverse xs)!!i                    επαγ. υποθ.
=     xs!!(length xs-1-i)
=     (x:xs)!!(length xs-1-i+1)
=     (x:xs)!!(length(x:xs)-1-i)
```

(β) `i<length xs`:

```
    (reverse(x:xs))!!i
=   (reverse xs ++ [x])!!i
=   (reverse xs)!!i                    επαγ. υποθ.
=   xs!!(length xs-1-i)
=   (x:xs)!!(length xs-1-i+1)
=   (x:xs)!!(length(x:xs)-1-i)
```

(β) `i<length xs`:

```
    (reverse(x:xs))!!i
=   (reverse xs ++ [x])!!i
=   (reverse xs)!!i                    επαγ. υποθ.
=   xs!!(length xs-1-i)
=   (x:xs)!!(length xs-1-i+1)
=   (x:xs)!!(length(x:xs)-1-i)
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - I

- ```
  foldr f z [] = z
  foldr f z (x:xs) = f x (foldr f z xs)
  ```
- Έστω `f :: T->T->T`:
  - $\texttt{x::T}_d$ && $\texttt{y::T}_d \Rightarrow \texttt{x`f`y::T}_d$
  - `(x`f`y)`f`z == x`f`(y`f`z)`
  - $\texttt{z::T}_d$ ώστε `z`f`x == x && x`f`z == x`
- Για κάθε $\texttt{xl::[T]}_{df}$, $\texttt{yl::[T]}_{df}$:

  ```
  foldr f z (xl++yl)
  == (foldr f z xl)`f`(foldr f z yl)
  ```
- Παραδείγματα:

  ```
  sum(xl++yl) == sum xl + sum yl
  mult(xl++yl) == mult xl * mult yl
  or(xl++yl) == or xl || or yl
  ```
  ```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - I

- `foldr f z [] = z`

  `foldr f z (x:xs) = f x (foldr f z xs)`
- Έστω `f :: T->T->T`:
  - `x::T`$_d$ `&& y::T`$_d \Rightarrow$ `x‘f‘y::T`$_d$
  - `(x‘f‘y)‘f‘z == x‘f‘(y‘f‘z)`
  - `z::T`$_d$ ώστε `z‘f‘x == x && x‘f‘z == x`
- Για κάθε `xl::[T]`$_{df}$, `yl::[T]`$_{df}$:

  `foldr f z (xl++yl)`

  `== (foldr f z xl)‘f‘(foldr f z yl)`
- Παραδείγματα:

  `sum(xl++yl) == sum xl + sum yl`

  `mult(xl++yl) == mult xl * mult yl`

  `or(xl++yl) == or xl || or yl`

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - I

- `foldr f z [] = z`

  `foldr f z (x:xs) = f x (foldr f z xs)`
- Έστω `f :: T->T->T`:
  - `x::`$\mathtt{T}_d$ `&& y::`$\mathtt{T}_d \Rightarrow$ `x‘f‘y::`$\mathtt{T}_d$
  - `(x‘f‘y)‘f‘z == x‘f‘(y‘f‘z)`
  - `z::`$\mathtt{T}_d$ ώστε `z‘f‘x == x && x‘f‘z == x`
- Για κάθε `xl::[T]`$_{df}$, `yl::[T]`$_{df}$:

  `foldr f z (xl++yl)`

  `== (foldr f z xl)‘f‘(foldr f z yl)`
- Παραδείγματα:

  `sum(xl++yl) == sum xl + sum yl`

  `mult(xl++yl) == mult xl * mult yl`

  `or(xl++yl) == or xl || or yl`

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - I

- `foldr f z [] = z`

  `foldr f z (x:xs) = f x (foldr f z xs)`
- Έστω `f :: T->T->T`:
  - `x::`$\mathtt{T}_d$ `&& y::`$\mathtt{T}_d \Rightarrow$ `x‘f‘y::`$\mathtt{T}_d$
  - `(x‘f‘y)‘f‘z == x‘f‘(y‘f‘z)`
  - `z::`$\mathtt{T}_d$ ώστε `z‘f‘x == x && x‘f‘z == x`
- Για κάθε `xl::[T]`$_{df}$, `yl::[T]`$_{df}$:

  `foldr f z (xl++yl)`

  `== (foldr f z xl)‘f‘(foldr f z yl)`
- Παραδείγματα:

  `sum(xl++yl) == sum xl + sum yl`

  `mult(xl++yl) == mult xl * mult yl`

  `or(xl++yl) == or xl || or yl`

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - I

- `foldr f z [] = z`

  `foldr f z (x:xs) = f x (foldr f z xs)`
- Έστω `f :: T->T->T`:
  - `x::`$\mathtt{T}_d$ `&& y::`$\mathtt{T}_d \Rightarrow$ `x‘f‘y::`$\mathtt{T}_d$
  - `(x‘f‘y)‘f‘z == x‘f‘(y‘f‘z)`
  - `z::`$\mathtt{T}_d$ ώστε `z‘f‘x == x && x‘f‘z == x`
- Για κάθε `xl::[T]`$_{df}$, `yl::[T]`$_{df}$:

  `foldr f z (xl++yl)`

  `== (foldr f z xl)‘f‘(foldr f z yl)`
- Παραδείγματα:

  `sum(xl++yl) == sum xl + sum yl`

  `mult(xl++yl) == mult xl * mult yl`

  `or(xl++yl) == or xl || or yl`

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - I

- `foldr f z [] = z`

  `foldr f z (x:xs) = f x (foldr f z xs)`
- Έστω `f :: T->T->T`:
  - `x::`$\mathtt{T}_d$ `&& y::`$\mathtt{T}_d \Rightarrow$ `x‘f‘y::`$\mathtt{T}_d$
  - `(x‘f‘y)‘f‘z == x‘f‘(y‘f‘z)`
  - `z::`$\mathtt{T}_d$ ώστε `z‘f‘x == x && x‘f‘z == x`
- Για κάθε `xl::[T]`$_{df}$, `yl::[T]`$_{df}$:

  `foldr f z (xl++yl)`

  `== (foldr f z xl)‘f‘(foldr f z yl)`
- Παραδείγματα:

  `sum(xl++yl) == sum xl + sum yl`

  `mult(xl++yl) == mult xl * mult yl`

  `or(xl++yl) == or xl || or yl`

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - I

- `foldr f z [] = z`

  `foldr f z (x:xs) = f x (foldr f z xs)`
- Έστω `f :: T->T->T`:
  - `x::`$\mathtt{T}_d$ `&& y::`$\mathtt{T}_d \Rightarrow$ `x‘f‘y::`$\mathtt{T}_d$
  - `(x‘f‘y)‘f‘z == x‘f‘(y‘f‘z)`
  - `z::`$\mathtt{T}_d$ ώστε `z‘f‘x == x && x‘f‘z == x`
- Για κάθε `xl::[T]`$_{df}$, `yl::[T]`$_{df}$:

  `foldr f z (xl++yl)`

  `== (foldr f z xl)‘f‘(foldr f z yl)`
- Παραδείγματα:

  `sum(xl++yl) == sum xl + sum yl`

  `mult(xl++yl) == mult xl * mult yl`

  `or(xl++yl) == or xl || or yl`

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - II

- Ορίζουμε: `ff = foldr f z`
- Επαγωγή στο `xl`
- Βάση επαγωγής:

$$\texttt{(ff [])`f`(ff y)} = \texttt{z`f`(ff y)} = \texttt{ff([]++y)}$$

- Βήμα επαγωγής. Επαγωγική Υπόθεση:
  `ff(xs++yl) == (ff xs) `f` (ff yl)`
  Απόδειξη:

$$\begin{aligned}
&\texttt{ff((x:xs)++yl)}\\
=\ &\texttt{ff(x:(xs++yl))}\\
=\ &\texttt{x`f`(ff(xs++yl))} && \textit{επαγ. υπόθ.}\\
=\ &\texttt{x`f`(ff xs)`f`(ff yl)}\\
=\ &\texttt{(ff(x:xs)) `f` (ff yl)}
\end{aligned}$$

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - II

- Ορίζουμε: `ff = foldr f z`
- Επαγωγή στο `xl`
- Βάση επαγωγής:

$$\texttt{(ff [])`f`(ff y)} = \texttt{z`f`(ff y)} = \texttt{ff([]++y)}$$

- Βήμα επαγωγής. Επαγωγική Υπόθεση:
  `ff(xs++yl) == (ff xs) `f` (ff yl)`
  Απόδειξη:

$$\begin{aligned}
&\texttt{ff((x:xs)++yl)} \\
=\ &\texttt{ff(x:(xs++yl))} \\
=\ &\texttt{x`f`(ff(xs++yl))} && \textit{επαγ. υπόθ.} \\
=\ &\texttt{x`f`(ff xs)`f`(ff yl)} \\
=\ &\texttt{(ff(x:xs)) `f` (ff yl)}
\end{aligned}$$

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - II

- Ορίζουμε: `ff = foldr f z`
- Επαγωγή στο `xl`
- Βάση επαγωγής:

$$\texttt{(ff [])`f`(ff y)} = \texttt{z`f`(ff y)} = \texttt{ff([]++y)}$$

- Βήμα επαγωγής. Επαγωγική Υπόθεση:
  `ff(xs++yl) == (ff xs) `f` (ff yl)`
  Απόδειξη:

```
    ff((x:xs)++yl)
=   ff(x:(xs++yl))
=   x`f`(ff(xs++yl))          επαγ. υπόθ.
=   x`f`(ff xs)`f`(ff yl)
=   (ff(x:xs)) `f` (ff yl)
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - II

- Ορίζουμε: `ff = foldr f z`
- Επαγωγή στο `xl`
- Βάση επαγωγής:

$$\texttt{(ff [])`f`(ff y)} = \texttt{z`f`(ff y)} = \texttt{ff([]++y)}$$

- Βήμα επαγωγής. Επαγωγική Υπόθεση:
  `ff(xs++yl) == (ff xs) `f` (ff yl)`
  Απόδειξη:

```
    ff((x:xs)++yl)
=   ff(x:(xs++yl))
=   x`f`(ff(xs++yl))          επαγ. υπόθ.
=   x`f`(ff xs)`f`(ff yl)
=   (ff(x:xs)) `f` (ff yl)
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - II

- Ορίζουμε: `ff = foldr f z`
- Επαγωγή στο `xl`
- Βάση επαγωγής:

$$\texttt{(ff [])`f`(ff y)} = \texttt{z`f`(ff y)} = \texttt{ff([]++y)}$$

- Βήμα επαγωγής. Επαγωγική Υπόθεση:
  `ff(xs++yl) == (ff xs) `f` (ff yl)`
  Απόδειξη:

```
    ff((x:xs)++yl)
=   ff(x:(xs++yl))
=   x`f`(ff(xs++yl))          επαγ. υπόθ.
=   x`f`(ff xs)`f`(ff yl)
=   (ff(x:xs)) `f` (ff yl)
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - II

- Ορίζουμε: `ff = foldr f z`
- Επαγωγή στο `xl`
- Βάση επαγωγής:

$$\texttt{(ff [])`f`(ff y)} = \texttt{z`f`(ff y)} = \texttt{ff([]++y)}$$

- Βήμα επαγωγής. Επαγωγική Υπόθεση:
  `ff(xs++yl) == (ff xs) `f` (ff yl)`
  Απόδειξη:

```
    ff((x:xs)++yl)
=   ff(x:(xs++yl))
=   x`f`(ff(xs++yl))          επαγ. υπόθ.
=   x`f`(ff xs)`f`(ff yl)
=   (ff(x:xs)) `f` (ff yl)
```

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - II

- Ορίζουμε: `ff = foldr f z`
- Επαγωγή στο `xl`
- Βάση επαγωγής:

$$\texttt{(ff [])`f`(ff y)} = \texttt{z`f`(ff y)} = \texttt{ff([]++y)}$$

- Βήμα επαγωγής. Επαγωγική Υπόθεση:
  `ff(xs++yl) == (ff xs) `f` (ff yl)`
  Απόδειξη:

$$\begin{aligned}
& \texttt{ff((x:xs)++yl)} \\
= \quad & \texttt{ff(x:(xs++yl))} \\
= \quad & \texttt{x`f`(ff(xs++yl))} && \textit{επαγ. υπόθ.} \\
= \quad & \texttt{x`f`(ff xs)`f`(ff yl)} \\
= \quad & \texttt{(ff(x:xs)) `f` (ff yl)}
\end{aligned}$$

# Αποδείξεις Ορθότητας

## Η Συνάρτηση `foldr` - II

- Ορίζουμε: `ff = foldr f z`
- Επαγωγή στο `xl`
- Βάση επαγωγής:

$$\texttt{(ff [])`f`(ff y)} = \texttt{z`f`(ff y)} = \texttt{ff([]++y)}$$

- Βήμα επαγωγής. Επαγωγική Υπόθεση:
  `ff(xs++yl) == (ff xs) `f` (ff yl)`
  Απόδειξη:

$$\begin{aligned}
& \texttt{ff((x:xs)++yl)} \\
= \quad & \texttt{ff(x:(xs++yl))} \\
= \quad & \texttt{x`f`(ff(xs++yl))} && \textit{επαγ. υπόθ.} \\
= \quad & \texttt{x`f`(ff xs)`f`(ff yl)} \\
= \quad & \texttt{(ff(x:xs)) `f` (ff yl)}
\end{aligned}$$

# Αποδείξεις Ορθότητας
## Άπειρες Λίστες

- Μόνο πεπερασμένες λίστες κατασκευάζονται από `[]` και `:`
- Η επαγωγή λιστών δεν αποδεικνύει πράγματα για άπειρες λίστες
- Παράδειγμα: `length l<-[0..]`
- Ιδιότητες άπειρων λιστών: επαγωγή σε φυσικό δείκτη `n` που τις διατρέχει
  - προτάσεις με `take n l` ή `l!!n`

# Αποδείξεις Ορθότητας
## Άπειρες Λίστες

- Μόνο πεπερασμένες λίστες κατασκευάζονται από `[]` και `:`
- Η επαγωγή λιστών δεν αποδεικνύει πράγματα για άπειρες λίστες
- Παράδειγμα: `length l<-[0..]`
- Ιδιότητες άπειρων λιστών: επαγωγή σε φυσικό δείκτη `n` που τις διατρέχει
  - προτάσεις με `take n l` ή `l!!n`

# Αποδείξεις Ορθότητας
## Άπειρες Λίστες

- Μόνο πεπερασμένες λίστες κατασκευάζονται από `[]` και `:`
- Η επαγωγή λιστών δεν αποδεικνύει πράγματα για άπειρες λίστες
- Παράδειγμα: `length l<-[0..]`
- Ιδιότητες άπειρων λιστών: επαγωγή σε φυσικό δείκτη `n` που τις διατρέχει
  - προτάσεις με `take n l` ή `l!!n`

- Μόνο πεπερασμένες λίστες κατασκευάζονται από `[]` και `:`
- Η επαγωγή λιστών δεν αποδεικνύει πράγματα για άπειρες λίστες
- Παράδειγμα: `length l<-[0..]`
- Ιδιότητες άπειρων λιστών: επαγωγή σε φυσικό δείκτη `n` που τις διατρέχει
  - προτάσεις με `take n l` ή `l!!n`

# Αποδείξεις Ορθότητας
## Ισχυροποίηση Θεωρήματος - I

- Για να αποδείξουμε θεώρημα `P`, μπορεί να χρειαστεί να αποδείξουμε <u>ισχυρότερο</u> θεώρημα `Q` (δηλ. `Q`$\Rightarrow$`P`)
- Η συνάρτηση `mysum`:

```
mysum :: [Int]->Int
mysum = mysumaux 0
 where mysumaux s [] = s
       mysumaux s (x:xs) = mysumaux (s+x) xs
```

- Θέλουμε να δείξουμε:

$\forall$`l::[Int]`$_{\texttt{df}}$. `mysum l = sum l`

- Απόπειρα: επαγωγή στην `l`
  - βάση τετριμμένη
  - επαγωγική υπόθεση: `mysum xs == sum xs`
  - βήμα: `sum(x:xs)=x+sum xs=x+mysum xs`

    και `mysum(x:xs)=mysumaux x xs=` ?

# Αποδείξεις Ορθότητας
## Ισχυροποίηση Θεωρήματος - I

- Για να αποδείξουμε θεώρημα `P`, μπορεί να χρειαστεί να αποδείξουμε <u>ισχυρότερο</u> θεώρημα `Q` (δηλ. `Q`$\Rightarrow$`P`)
- Η συνάρτηση `mysum`:

```
mysum :: [Int]->Int
mysum = mysumaux 0
 where mysumaux s [] = s
       mysumaux s (x:xs) = mysumaux (s+x) xs
```

- Θέλουμε να δείξουμε:
  $\forall$`l::[Int]`$_{\texttt{df}}$. `mysum l = sum l`
- Απόπειρα: επαγωγή στην `l`
  - βάση τετριμμένη
  - επαγωγική υπόθεση: `mysum xs == sum xs`
  - βήμα: `sum(x:xs)=x+sum xs=x+mysum xs`
    και `mysum(x:xs)=mysumaux x xs=` ?

# Αποδείξεις Ορθότητας
## Ισχυροποίηση Θεωρήματος - I

- Για να αποδείξουμε θεώρημα `P`, μπορεί να χρειαστεί να αποδείξουμε ισχυρότερο θεώρημα `Q` (δηλ. `Q`$\Rightarrow$`P`)
- Η συνάρτηση `mysum`:

```
mysum :: [Int]->Int
mysum = mysumaux 0
 where mysumaux s [] = s
       mysumaux s (x:xs) = mysumaux (s+x) xs
```

- Θέλουμε να δείξουμε:

$\forall$`l::[Int]`$_{\texttt{df}}$. `mysum l = sum l`

- Απόπειρα: επαγωγή στην `l`
  - βάση τετριμμένη
  - επαγωγική υπόθεση: `mysum xs == sum xs`
  - βήμα: `sum(x:xs)=x+sum xs=x+mysum xs`
    και `mysum(x:xs)=mysumaux x xs= ?`

- Για να αποδείξουμε θεώρημα `P`, μπορεί να χρειαστεί να αποδείξουμε ισχυρότερο θεώρημα `Q` (δηλ. `Q`$\Rightarrow$`P`)
- Η συνάρτηση `mysum`:

```
mysum :: [Int]->Int
mysum = mysumaux 0
 where mysumaux s [] = s
       mysumaux s (x:xs) = mysumaux (s+x) xs
```

- Θέλουμε να δείξουμε:

$\forall$`l::[Int]`$_{\texttt{df}}$. `mysum l = sum l`

- Απόπειρα: επαγωγή στην `l`
  - βάση τετριμμένη
  - επαγωγική υπόθεση: `mysum xs == sum xs`
  - βήμα: `sum(x:xs)=x+sum xs=x+mysum xs`

    και `mysum(x:xs)=mysumaux x xs=` ?

- Πρόβλημα: χρειαζόμαστε ένα ισχυρότερο θεώρημα που να μιλάει για κάθε `mysumaux s` και όχι μόνο για `mysumaux 0`
- Νέα απόπειρα. Απόδειξη του
  $\forall$`s::Int`$_d$`. mysumaux s l = s+sum l`
  - το παλιό θεώρημα είναι υποπερίπτωση του καινούριου
- Βάση: `mysum s [] == sum s []` τετριμμένη
- Επαγωγική υπόθεση:
  $\forall$`t::Int`$_d$`. mysumaux t xs = t+sum xs`
- Βήμα:

```
    mysumaux s (x:xs)
=   mysumaux (s+x) xs   επαγ. υπόθ. για t=s+x
=   s + x + sum xs
=   s + sum(x:xs)
```

# Αποδείξεις Ορθότητας
## Ισχυροποίηση Θεωρήματος - II

- Πρόβλημα: χρειαζόμαστε ένα ισχυρότερο θεώρημα που να μιλάει για κάθε `mysumaux s` και όχι μόνο για `mysumaux 0`
- Νέα απόπειρα. Απόδειξη του

  $\forall$`s::Int`$_d$`. mysumaux s l = s+sum l`

  - το παλιό θεώρημα είναι υποπερίπτωση του καινούριου
- Βάση: `mysum s [] == sum s []` τετριμμένη
- Επαγωγική υπόθεση:

  $\forall$`t::Int`$_d$`. mysumaux t xs = t+sum xs`
- Βήμα:

```
    mysumaux s (x:xs)
=   mysumaux (s+x) xs   επαγ. υπόθ. για t=s+x
=   s + x + sum xs
=   s + sum(x:xs)
```

# Αποδείξεις Ορθότητας
## Ισχυροποίηση Θεωρήματος - II

- Πρόβλημα: χρειαζόμαστε ένα ισχυρότερο θεώρημα που να μιλάει για κάθε `mysumaux s` και όχι μόνο για `mysumaux 0`
- Νέα απόπειρα. Απόδειξη του

  $\forall$`s::Int`$_d$`. mysumaux s l = s+sum l`

  - το παλιό θεώρημα είναι υποπερίπτωση του καινούριου
- Βάση: `mysum s [] == sum s []` τετριμμένη
- Επαγωγική υπόθεση:

  $\forall$`t::Int`$_d$`. mysumaux t xs = t+sum xs`
- Βήμα:

```
    mysumaux s (x:xs)
=   mysumaux (s+x) xs   επαγ. υπόθ. για t=s+x
=   s + x + sum xs
=   s + sum(x:xs)
```

# Αποδείξεις Ορθότητας
## Ισχυροποίηση Θεωρήματος - II

- Πρόβλημα: χρειαζόμαστε ένα ισχυρότερο θεώρημα που να μιλάει για κάθε `mysumaux s` και όχι μόνο για `mysumaux 0`
- Νέα απόπειρα. Απόδειξη του

  $\forall$`s::Int`$_d$`. mysumaux s l = s+sum l`

  - το παλιό θεώρημα είναι υποπερίπτωση του καινούριου
- Βάση: `mysum s [] == sum s []` τετριμμένη
- Επαγωγική υπόθεση:

  $\forall$`t::Int`$_d$`. mysumaux t xs = t+sum xs`
- Βήμα:

```
    mysumaux s (x:xs)
=   mysumaux (s+x) xs   επαγ. υπόθ. για t=s+x
=   s + x + sum xs
=   s + sum(x:xs)
```

- Πρόβλημα: χρειαζόμαστε ένα ισχυρότερο θεώρημα που να μιλάει για κάθε `mysumaux s` και όχι μόνο για `mysumaux 0`
- Νέα απόπειρα. Απόδειξη του

  $\forall$`s::Int`$_d$`. mysumaux s l = s+sum l`
  - το παλιό θεώρημα είναι υποπερίπτωση του καινούριου
- Βάση: `mysum s [] == sum s []` τετριμμένη
- Επαγωγική υπόθεση:

  $\forall$`t::Int`$_d$`. mysumaux t xs = t+sum xs`
- Βήμα:

```
    mysumaux s (x:xs)
=   mysumaux (s+x) xs   επαγ. υπόθ. για t=s+x
=   s + x + sum xs
=   s + sum(x:xs)
```

- Διαχωρισμός ορισμού από χρήση
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
  - επώνυμα δεδομένα αποτιμώνται το πολύ μία φορά
- Προγραμματισμός οδηγούμενος από τα δεδομένα
- Άπειρες δομές
- Προβλήματα με τη χρήση πολλαπλών άπειρων δομών
  - dovetailing

- Διαχωρισμός ορισμού από χρήση
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
  - επώνυμα δεδομένα αποτιμώνται το πολύ μία φορά
- Προγραμματισμός οδηγούμενος από τα δεδομένα
- Άπειρες δομές
- Προβλήματα με τη χρήση πολλαπλών άπειρων δομών
  - dovetailing

- Διαχωρισμός ορισμού από χρήση
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
  - επώνυμα δεδομένα αποτιμώνται το πολύ μία φορά
- Προγραμματισμός οδηγούμενος από τα δεδομένα
- Άπειρες δομές
- Προβλήματα με τη χρήση πολλαπλών άπειρων δομών
  - dovetailing

- Διαχωρισμός ορισμού από χρήση
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
  - επώνυμα δεδομένα αποτιμώνται το πολύ μία φορά
- Προγραμματισμός οδηγούμενος από τα δεδομένα
- Άπειρες δομές
- Προβλήματα με τη χρήση πολλαπλών άπειρων δομών
  - dovetailing

- Διαχωρισμός ορισμού από χρήση
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
  - επώνυμα δεδομένα αποτιμώνται το πολύ μία φορά
- Προγραμματισμός οδηγούμενος από τα δεδομένα
- Άπειρες δομές
- Προβλήματα με τη χρήση πολλαπλών άπειρων δομών
  - dovetailing

- Εξασφαλίζουν την ικανοποίηση προδιαγραφών για κάθε είσοδο
- Απαιτούν ανθρώπινη προσπάθεια
- Ειδικός τρόπος χειρισμού μη τερματισμού
  - ειδική τιμή `undef` όλων των τύπων
- Βασικός τρόπος απόδειξης στο Σ.Π.: επαγωγή
  - σε φυσικούς, (ορισμένους) ακεραίους, πεπερασμένες (ορισμένες) λίστες
- Η επαγωγή είναι ακατάλληλη για άπειρες δομές
- Πολλές φορές χρειαζόμαστε ισχυροποίηση ενός θεωρήματος για να δουλέψει η επαγωγή

- Εξασφαλίζουν την ικανοποίηση προδιαγραφών για κάθε είσοδο
- Απαιτούν ανθρώπινη προσπάθεια
- Ειδικός τρόπος χειρισμού μη τερματισμού
  - ειδική τιμή `undef` όλων των τύπων
- Βασικός τρόπος απόδειξης στο Σ.Π.: επαγωγή
  - σε φυσικούς, (ορισμένους) ακεραίους, πεπερασμένες (ορισμένες) λίστες
- Η επαγωγή είναι ακατάλληλη για άπειρες δομές
- Πολλές φορές χρειαζόμαστε ισχυροποίηση ενός θεωρήματος για να δουλέψει η επαγωγή

- Εξασφαλίζουν την ικανοποίηση προδιαγραφών για κάθε είσοδο
- Απαιτούν ανθρώπινη προσπάθεια
- Ειδικός τρόπος χειρισμού μη τερματισμού
  - ειδική τιμή `undef` όλων των τύπων
- Βασικός τρόπος απόδειξης στο Σ.Π.: επαγωγή
  - σε φυσικούς, (ορισμένους) ακεραίους, πεπερασμένες (ορισμένες) λίστες
- Η επαγωγή είναι ακατάλληλη για άπειρες δομές
- Πολλές φορές χρειαζόμαστε ισχυροποίηση ενός θεωρήματος για να δουλέψει η επαγωγή

- Εξασφαλίζουν την ικανοποίηση προδιαγραφών για κάθε είσοδο
- Απαιτούν ανθρώπινη προσπάθεια
- Ειδικός τρόπος χειρισμού μη τερματισμού
  - ειδική τιμή `undef` όλων των τύπων
- Βασικός τρόπος απόδειξης στο Σ.Π.: επαγωγή
  - σε φυσικούς, (ορισμένους) ακεραίους, πεπερασμένες (ορισμένες) λίστες
- Η επαγωγή είναι ακατάλληλη για άπειρες δομές
- Πολλές φορές χρειαζόμαστε ισχυροποίηση ενός θεωρήματος για να δουλέψει η επαγωγή

- Εξασφαλίζουν την ικανοποίηση προδιαγραφών για κάθε είσοδο
- Απαιτούν ανθρώπινη προσπάθεια
- Ειδικός τρόπος χειρισμού μη τερματισμού
  - ειδική τιμή `undef` όλων των τύπων
- Βασικός τρόπος απόδειξης στο Σ.Π.: επαγωγή
  - σε φυσικούς, (ορισμένους) ακεραίους, πεπερασμένες (ορισμένες) λίστες
- Η επαγωγή είναι ακατάλληλη για άπειρες δομές
- Πολλές φορές χρειαζόμαστε ισχυροποίηση ενός θεωρήματος για να δουλέψει η επαγωγή

- Εξασφαλίζουν την ικανοποίηση προδιαγραφών για κάθε είσοδο
- Απαιτούν ανθρώπινη προσπάθεια
- Ειδικός τρόπος χειρισμού μη τερματισμού
  - ειδική τιμή `undef` όλων των τύπων
- Βασικός τρόπος απόδειξης στο Σ.Π.: επαγωγή
  - σε φυσικούς, (ορισμένους) ακεραίους, πεπερασμένες (ορισμένες) λίστες
- Η επαγωγή είναι ακατάλληλη για άπειρες δομές
- Πολλές φορές χρειαζόμαστε ισχυροποίηση ενός θεωρήματος για να δουλέψει η επαγωγή