Εισαγωγή στο Συναρτησιακό Προγραμματισμό

Γιάννης Κασσιός

Καλωσορίσατε στο Μάθημα

Συναρτησιακός Προγραμματισμός

- Διδάσκων: Γιάννης Κασσιός
- e-mail: `kassios@di.uoa.gr`
- Web Site μαθήματος:
  `http://www.di.uoa.gr/~kassios/courses/fp`

Δομή του Μαθήματος

- Η φιλοσοφία του Συναρτησιακού Προγραμματισμού
- Η γλώσσα Haskell
- λ-λογισμός (τα μαθηματικά του Σ.Π.)
- Θέματα υλοποίησης Σ.Π.

Το μάθημα αυτό διδάσκεται για πρώτη φορά και είναι ακόμα υπό κατασκευή. Η δομή αυτή μπορεί να τροποποιηθεί.

Δομή του Μαθήματος

- Η φιλοσοφία του Συναρτησιακού Προγραμματισμού
- Η γλώσσα Haskell
- λ-λογισμός (τα μαθηματικά του Σ.Π.)
- Θέματα υλοποίησης Σ.Π.

Το μάθημα αυτό διδάσκεται για πρώτη φορά και είναι ακόμα υπό κατασκευή. Η δομή αυτή μπορεί να τροποποιηθεί.

Δομή του Μαθήματος

- Η φιλοσοφία του Συναρτησιακού Προγραμματισμού
- Η γλώσσα Haskell
- λ-λογισμός (τα μαθηματικά του Σ.Π.)
- Θέματα υλοποίησης Σ.Π.

Το μάθημα αυτό διδάσκεται για πρώτη φορά και είναι ακόμα υπό κατασκευή. Η δομή αυτή μπορεί να τροποποιηθεί.

Δομή του Μαθήματος

- Η φιλοσοφία του Συναρτησιακού Προγραμματισμού
- Η γλώσσα Haskell
- λ-λογισμός (τα μαθηματικά του Σ.Π.)
- Θέματα υλοποίησης Σ.Π.

Το μάθημα αυτό διδάσκεται για πρώτη φορά και είναι ακόμα υπό κατασκευή. Η δομή αυτή μπορεί να τροποποιηθεί.

Δομή του Μαθήματος

- Η φιλοσοφία του Συναρτησιακού Προγραμματισμού
- Η γλώσσα Haskell
- λ-λογισμός (τα μαθηματικά του Σ.Π.)
- Θέματα υλοποίησης Σ.Π.

Το μάθημα αυτό διδάσκεται για πρώτη φορά και είναι ακόμα υπό κατασκευή. Η δομή αυτή μπορεί να τροποποιηθεί.

Σημερινή Διάλεξη

- Τι είναι ο Σ.Π.
- Γιατί ο Σ.Π. είναι σημαντικός
  - Καθαρότητα κώδικα / αποδείξεις ορθότητας
  - Ιδιώματα που ευνοούν τη τμηματοποίηση κώδικα
  - ...
- Σ.Π. vs. Προστακτικός Προγραμματισμός
- Ανακεφαλαίωση

Σημερινή Διάλεξη

- Τι είναι ο Σ.Π.
- Γιατί ο Σ.Π. είναι σημαντικός
  - Καθαρότητα κώδικα / αποδείξεις ορθότητας
  - Ιδιώματα που ευνοούν τη τμηματοποίηση κώδικα
    - Συναρτήσεις υψηλότερης τάξης
    - Οκνηρή αποτίμηση
- Σ.Π. vs. Προστακτικός Προγραμματισμός
- Ανακεφαλαίωση

Σημερινή Διάλεξη

- Τι είναι ο Σ.Π.
- Γιατί ο Σ.Π. είναι σημαντικός
  - Καθαρότητα κώδικα / αποδείξεις ορθότητας
  - Ιδιώματα που ευνοούν τη τμηματοποίηση κώδικα
    - Συναρτήσεις υψηλότερης τάξης
    - Οκνηρή αποτίμηση
- Σ.Π. vs. Προστακτικός Προγραμματισμός
- Ανακεφαλαίωση

Σημερινή Διάλεξη

- Τι είναι ο Σ.Π.
- Γιατί ο Σ.Π. είναι σημαντικός
  - Καθαρότητα κώδικα / αποδείξεις ορθότητας
  - Ιδιώματα που ευνοούν τη τμηματοποίηση κώδικα
    - Συναρτήσεις υψηλότερης τάξης
    - Οκνηρή αποτίμηση
- Σ.Π. vs. Προστακτικός Προγραμματισμός
- Ανακεφαλαίωση

Σημερινή Διάλεξη

- Τι είναι ο Σ.Π.
- Γιατί ο Σ.Π. είναι σημαντικός
  - Καθαρότητα κώδικα / αποδείξεις ορθότητας
  - Ιδιώματα που ευνοούν τη τμηματοποίηση κώδικα
    - Συναρτήσεις υψηλότερης τάξης
    - Οκνηρή αποτίμηση
- Σ.Π. vs. Προστακτικός Προγραμματισμός
- Ανακεφαλαίωση

Σημερινή Διάλεξη

- Τι είναι ο Σ.Π.
- Γιατί ο Σ.Π. είναι σημαντικός
  - Καθαρότητα κώδικα / αποδείξεις ορθότητας
  - Ιδιώματα που ευνοούν τη τμηματοποίηση κώδικα
    - Συναρτήσεις υψηλότερης τάξης
    - Οκνηρή αποτίμηση
- Σ.Π. vs. Προστακτικός Προγραμματισμός
- Ανακεφαλαίωση

Σημερινή Διάλεξη

- Τι είναι ο Σ.Π.
- Γιατί ο Σ.Π. είναι σημαντικός
  - Καθαρότητα κώδικα / αποδείξεις ορθότητας
  - Ιδιώματα που ευνοούν τη τμηματοποίηση κώδικα
    - Συναρτήσεις υψηλότερης τάξης
    - Οκνηρή αποτίμηση
- Σ.Π. vs. Προστακτικός Προγραμματισμός
- Ανακεφαλαίωση

Σημερινή Διάλεξη

- Τι είναι ο Σ.Π.
- Γιατί ο Σ.Π. είναι σημαντικός
  - Καθαρότητα κώδικα / αποδείξεις ορθότητας
  - Ιδιώματα που ευνοούν τη τμηματοποίηση κώδικα
    - Συναρτήσεις υψηλότερης τάξης
    - Οκνηρή αποτίμηση
- Σ.Π. vs. Προστακτικός Προγραμματισμός
- Ανακεφαλαίωση

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο

- Παριστάνει τον υπολογισμό σα μία αλληλουχία *εντολών*
- Οι εντολές αλλάζουν την *κατάσταση* του προγράμματος
  - π.χ. η αλλαγή του περιεγχομένου μίας θέσης μνήμης
  - *παρενέργεια*: μία αλλαγή στην κατάσταση του προγράμματος
- Το μοντέλο περιγράφει τον υπολογισμό με όρους *πολύ κοντά* στον υπολογιστή
  - σχετικά "χαμηλού επιπέδου" μοντέλο
  - π.χ. η γλώσσα μηχανής
- Πολλές αφαιρετικές έννοιες κάνουν το μοντέλο "υψηλότερου επιπέδου"
  - π.χ. η έννοια της μεταβλητής
  - το υψηλό επίπεδο είναι πιο πρακτικό για πολύπλοκα προγράμματα

Παραδοσιακό Προστακτικό Μοντέλο - Παράδειγμα

Υπολογισμός παραγοντικού στην Pascal:

```
factorial:=1;
while (n > 0) do begin
  factorial := factorial * n;
  n:=n-1
end
```

- Εκφράζει τον υπολογισμό σαν αλληλουχία εντολών
  - *πώς* υπολογίζουμε, όχι *τι*
- Βασίζεται στις παρενέργειες

Παραδοσιακό Προστακτικό Μοντέλο - Παράδειγμα

Υπολογισμός παραγοντικού στην Pascal:

```
factorial:=1;
while (n > 0) do begin
  factorial := factorial * n;
  n:=n-1
end
```

- Εκφράζει τον υπολογισμό σαν αλληλουχία εντολών
  - *πώς* υπολογίζουμε, όχι *τι*
- Βασίζεται στις παρενέργειες

Παραδοσιακό Προστακτικό Μοντέλο - Παράδειγμα

Υπολογισμός παραγοντικού στην Pascal:

```
factorial:=1;
while (n > 0) do begin
  factorial := factorial * n;
  n:=n-1
end
```

- Εκφράζει τον υπολογισμό σαν αλληλουχία εντολών
  - *πώς* υπολογίζουμε, όχι *τι*
- Βασίζεται στις παρενέργειες

Παραδοσιακό Προστακτικό Μοντέλο - Παράδειγμα

Υπολογισμός παραγοντικού στην Pascal:

```
factorial:=1;
while (n > 0) do begin
  factorial := factorial * n;
  n:=n-1
end
```

- Εκφράζει τον υπολογισμό σαν αλληλουχία εντολών
  - *πώς* υπολογίζουμε, όχι *τι*
- Βασίζεται στις παρενέργειες

Παραδοσιακό Προστακτικό Μοντέλο - Παράδειγμα

Υπολογισμός παραγοντικού στην Pascal:

```
factorial:=1;
while (n > 0) do begin
  factorial := factorial * n;
  n:=n-1
end
```

- Εκφράζει τον υπολογισμό σαν αλληλουχία εντολών
  - *πώς* υπολογίζουμε, όχι *τι*
- Βασίζεται στις παρενέργειες

Παραδοσιακό Προστακτικό Μοντέλο - Παράδειγμα

Υπολογισμός παραγοντικού στην Pascal:

```
factorial:=1;
while (n > 0) do begin
  factorial := factorial * n;
  n:=n-1
end
```

- Εκφράζει τον υπολογισμό σαν αλληλουχία εντολών
  - *πώς* υπολογίζουμε, όχι *τι*
- Βασίζεται στις παρενέργειες

Συναρτησιακός Προγραμματισμός

- Εγκαταλείπουμε το *πώς*, εκφράζουμε το *τι*
- Δεν ασχολούμαστε με: μνήμη, κατάσταση προγράμματος, εντολές, παρενέργειες
- Περιγράφουμε τον υπολογισμό ως *μαθηματική συνάρτηση (function)*
  - παράμετροι: *είσοδος* του υπολογισμού
  - επιστρεφόμενες τιμές: *έξοδος* του υπολογισμού
- Υψηλότερου επιπέδου αφαίρεση από αυτές του Προστακτικού Προγραμματισμού

Συναρτησιακός Προγραμματισμός

- Εγκαταλείπουμε το *πώς*, εκφράζουμε το *τι*
- Δεν ασχολούμαστε με: μνήμη, κατάσταση προγράμματος, εντολές, παρενέργειες
- Περιγράφουμε τον υπολογισμό ως *μαθηματική συνάρτηση (function)*
  - παράμετροι: *είσοδος* του υπολογισμού
  - επιστρεφόμενες τιμές: *έξοδος* του υπολογισμού
- Υψηλότερου επιπέδου αφαίρεση από αυτές του Προστακτικού Προγραμματισμού

Συναρτησιακός Προγραμματισμός

- Εγκαταλείπουμε το *πώς*, εκφράζουμε το *τι*
- Δεν ασχολούμαστε με: μνήμη, κατάσταση προγράμματος, εντολές, παρενέργειες
- Περιγράφουμε τον υπολογισμό ως *μαθηματική συνάρτηση (function)*
  - παράμετροι: *είσοδος* του υπολογισμού
  - επιστρεφόμενες τιμές: *έξοδος* του υπολογισμού
- Υψηλότερου επιπέδου αφαίρεση από αυτές του Προστακτικού Προγραμματισμού

Συναρτησιακός Προγραμματισμός

- Εγκαταλείπουμε το *πώς*, εκφράζουμε το *τι*
- Δεν ασχολούμαστε με: μνήμη, κατάσταση προγράμματος, εντολές, παρενέργειες
- Περιγράφουμε τον υπολογισμό ως *μαθηματική συνάρτηση (function)*
  - παράμετροι: *είσοδος* του υπολογισμού
  - επιστρεφόμενες τιμές: *έξοδος* του υπολογισμού
- Υψηλότερου επιπέδου αφαίρεση από αυτές του Προστακτικού Προγραμματισμού

Συναρτησιακός Προγραμματισμός - Παράδειγμα

Υπολογισμός παραγοντικού στην Haskell:

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

- Εκφράζει τον υπολογισμό σαν ορισμό μίας συνάρτησης
  - *τι* υπολογίζουμε, όχι *πώς*
- Δεν υπάρχουν αναφορές σε μνήμη/παρενέργειες

Συναρτησιακός Προγραμματισμός - Παράδειγμα

Υπολογισμός παραγοντικού στην Haskell:

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

- Εκφράζει τον υπολογισμό σαν ορισμό μίας συνάρτησης
  - *τι* υπολογίζουμε, όχι *πώς*
- Δεν υπάρχουν αναφορές σε μνήμη/παρενέργειες

Συναρτησιακός Προγραμματισμός - Παράδειγμα

Υπολογισμός παραγοντικού στην Haskell:

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

- Εκφράζει τον υπολογισμό σαν ορισμό μίας συνάρτησης
  - *τι* υπολογίζουμε, όχι *πώς*
- Δεν υπάρχουν αναφορές σε μνήμη/παρενέργειες

Συναρτησιακός Προγραμματισμός - Παράδειγμα

Υπολογισμός παραγοντικού στην Haskell:

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

- Εκφράζει τον υπολογισμό σαν ορισμό μίας συνάρτησης
  - *τι* υπολογίζουμε, όχι *πώς*
- Δεν υπάρχουν αναφορές σε μνήμη/παρενέργειες

Συναρτησιακός Προγραμματισμός - Παράδειγμα

Υπολογισμός παραγοντικού στην Haskell:

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

- Εκφράζει τον υπολογισμό σαν ορισμό μίας συνάρτησης
  - *τι* υπολογίζουμε, όχι *πώς*
- Δεν υπάρχουν αναφορές σε μνήμη/παρενέργειες

Συναρτησιακός Προγραμματισμός - Παράδειγμα

Υπολογισμός παραγοντικού στην Haskell:

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

- Εκφράζει τον υπολογισμό σαν ορισμό μίας συνάρτησης
  - *τι* υπολογίζουμε, όχι *πώς*
- Δεν υπάρχουν αναφορές σε μνήμη/παρενέργειες

Μαθηματικές Συναρτήσεις

- Οι συναρτήσεις του Σ.Π. είναι καθαρά μαθηματικές συναρτήσεις
  - η τιμή επιστροφής μίας συνάρτησης εξαρτάται *αποκλειστικά* από τις παραμέτρους
  - δύο κλήσεις στην ίδια συνάρτηση με τις ίδιες παραμέτρους επιστρέφουν το ίδιο αποτέλεσμα
- Ο όρος "συνάρτηση" είναι καταχρηστικός αν εμπεριέχονται παρενέργειες, όπως π.χ. στις C/C++ και Pascal
- Η παρακάτω "συνάρτηση" Pascal *δεν* είναι μαθηματική συνάρτηση:

```
var somevar:integer;
function p():integer;
begin
  somevar:=somevar+1;
  p:=somevar
end
```

Μαθηματικές Συναρτήσεις

- Οι συναρτήσεις του Σ.Π. είναι καθαρά μαθηματικές συναρτήσεις
  - η τιμή επιστροφής μίας συνάρτησης εξαρτάται *αποκλειστικά* από τις παραμέτρους
  - δύο κλήσεις στην ίδια συνάρτηση με τις ίδιες παραμέτρους επιστρέφουν το ίδιο αποτέλεσμα
- Ο όρος "συνάρτηση" είναι καταχρηστικός αν εμπεριέχονται παρενέργειες, όπως π.χ. στις C/C++ και Pascal
- Η παρακάτω "συνάρτηση" Pascal *δεν* είναι μαθηματική συνάρτηση:

```
var somevar:integer;
function p():integer;
begin
  somevar:=somevar+1;
  p:=somevar
end
```

Μαθηματικές Συναρτήσεις

- Οι συναρτήσεις του Σ.Π. είναι καθαρά μαθηματικές συναρτήσεις
  - η τιμή επιστροφής μίας συνάρτησης εξαρτάται *αποκλειστικά* από τις παραμέτρους
  - δύο κλήσεις στην ίδια συνάρτηση με τις ίδιες παραμέτρους επιστρέφουν το ίδιο αποτέλεσμα
- Ο όρος "συνάρτηση" είναι καταχρηστικός αν εμπεριέχονται παρενέργειες, όπως π.χ. στις C/C++ και Pascal
- Η παρακάτω "συνάρτηση" Pascal *δεν* είναι μαθηματική συνάρτηση:

```
var somevar:integer;
function p():integer;
begin
  somevar:=somevar+1;
  p:=somevar
end
```

Μαθηματικές Συναρτήσεις

- Οι συναρτήσεις του Σ.Π. είναι καθαρά μαθηματικές συναρτήσεις
  - η τιμή επιστροφής μίας συνάρτησης εξαρτάται *αποκλειστικά* από τις παραμέτρους
  - δύο κλήσεις στην ίδια συνάρτηση με τις ίδιες παραμέτρους επιστρέφουν το ίδιο αποτέλεσμα
- Ο όρος "συνάρτηση" είναι καταχρηστικός αν εμπεριέχονται παρενέργειες, όπως π.χ. στις C/C++ και Pascal
- Η παρακάτω "συνάρτηση" Pascal *δεν* είναι μαθηματική συνάρτηση:

```
var somevar:integer;
function p():integer;
begin
  somevar:=somevar+1;
  p:=somevar
end
```

Γιατί Σ.Π.;

Ιδιότητες *καλού κώδικα* για κατασκευή πολύπλοκου λογισμικού

- *Καθαρός* και *ευανάγνωστος*
- *Εύχρηστος* από μαθηματικές θεωρίες απόδειξης ορθότητας
- *Τμηματοποιημένος (modularized)*
  - δηλ. διαχωρισμένος σε μικρά ανεξάρτητα μεταξύ τους κομμάτια

Αυτοί είναι οι στόχοι των αφαιρέσεων υψηλού επιπέδου. Ο Σ.Π. χρησιμεύει σαν ένα μέσο για να επιτευχθούν τέτοιοι στόχοι

Γιατί Σ.Π.;

Ιδιότητες *καλού κώδικα* για κατασκευή πολύπλοκου λογισμικού

- *Καθαρός* και *ευανάγνωστος*
- *Εύχρηστος* από μαθηματικές θεωρίες απόδειξης ορθότητας
- *Τμηματοποιημένος (modularized)*
  - δηλ. διαχωρισμένος σε μικρά ανεξάρτητα μεταξύ τους κομμάτια

Αυτοί είναι οι στόχοι των αφαιρέσεων υψηλού επιπέδου. Ο Σ.Π. χρησιμεύει σαν ένα μέσο για να επιτευχθούν τέτοιοι στόχοι

Γιατί Σ.Π.;

Ιδιότητες *καλού κώδικα* για κατασκευή πολύπλοκου λογισμικού

- *Καθαρός* και *ευανάγνωστος*
- *Εύχρηστος* από μαθηματικές θεωρίες απόδειξης ορθότητας
- *Τμηματοποιημένος (modularized)*
  - δηλ. διαχωρισμένος σε μικρά ανεξάρτητα μεταξύ τους κομμάτια

Αυτοί είναι οι στόχοι των αφαιρέσεων υψηλού επιπέδου. Ο Σ.Π. χρησιμεύει σαν ένα μέσο για να επιτευχθούν τέτοιοι στόχοι

Γιατί Σ.Π.;

Ιδιότητες *καλού κώδικα* για κατασκευή πολύπλοκου λογισμικού

- *Καθαρός* και *ευανάγνωστος*
- *Εύχρηστος* από μαθηματικές θεωρίες απόδειξης ορθότητας
- *Τμηματοποιημένος (modularized)*
  - δηλ. διαχωρισμένος σε μικρά ανεξάρτητα μεταξύ τους κομμάτια

Αυτοί είναι οι στόχοι των αφαιρέσεων υψηλού επιπέδου. Ο Σ.Π. χρησιμεύει σαν ένα μέσο για να επιτευχθούν τέτοιοι στόχοι

Γιατί Σ.Π.;

Ιδιότητες *καλού κώδικα* για κατασκευή πολύπλοκου λογισμικού

- *Καθαρός* και *ευανάγνωστος*
- *Εύχρηστος* από μαθηματικές θεωρίες απόδειξης ορθότητας
- *Τμηματοποιημένος (modularized)*
  - δηλ. διαχωρισμένος σε μικρά ανεξάρτητα μεταξύ τους κομμάτια

Αυτοί είναι οι στόχοι των αφαιρέσεων υψηλού επιπέδου. Ο Σ.Π. χρησιμεύει σαν ένα μέσο για να επιτευχθούν τέτοιοι στόχοι

Καθαρότητα/Αναγνωσιμότητα/Αποδειξιμότητα

Κώδικας Pascal:

```
if f(1)=f(1) then print("Yes") else print("No")
```

- Λόγω παρενεργειών, η ανακλαστική ιδιότητα $x = x$ δεν ισχύει
  - ο παραπάνω κώδικας δεν απλοποιείται σε `print("Yes")`
- Δε μπορούμε να χρησιμοποιήσουμε θεμελειώδη μαθηματικά στις αποδείξεις ορθότητας
- Οι τυπικές μέθοδοι για προστακτικό προγραμματισμό φροντίζουν να διαχωρίζουν τις *εντολές* από τις *εκφράσεις*
  - οι εκφράσεις δεν έχουν παρενέργειες
- Η απλότητα του παραπάνω κώδικα είναι παραπλανητική

Καθαρότητα/Αναγνωσιμότητα/Αποδειξιμότητα

Κώδικας Pascal:

```
if f(1)=f(1) then print("Yes") else print("No")
```

- Λόγω παρενεργειών, η ανακλαστική ιδιότητα $x = x$ δεν ισχύει
  - ο παραπάνω κώδικας δεν απλοποιείται σε `print("Yes")`
- Δε μπορούμε να χρησιμοποιήσουμε θεμελειώδη μαθηματικά στις αποδείξεις ορθότητας
- Οι τυπικές μέθοδοι για προστακτικό προγραμματισμό φροντίζουν να διαχωρίζουν τις *εντολές* από τις *εκφράσεις*
  - οι εκφράσεις δεν έχουν παρενέργειες
- Η απλότητα του παραπάνω κώδικα είναι παραπλανητική

Καθαρότητα/Αναγνωσιμότητα/Αποδειξιμότητα

Κώδικας Pascal:

```
if f(1)=f(1) then print("Yes") else print("No")
```

- Λόγω παρενεργειών, η ανακλαστική ιδιότητα $x = x$ δεν ισχύει
  - ο παραπάνω κώδικας δεν απλοποιείται σε `print("Yes")`
- Δε μπορούμε να χρησιμοποιήσουμε θεμελειώδη μαθηματικά στις αποδείξεις ορθότητας
- Οι τυπικές μέθοδοι για προστακτικό προγραμματισμό φροντίζουν να διαχωρίζουν τις *εντολές* από τις *εκφράσεις*
  - οι εκφράσεις δεν έχουν παρενέργειες
- Η απλότητα του παραπάνω κώδικα είναι παραπλανητική

Καθαρότητα/Αναγνωσιμότητα/Αποδειξιμότητα

Κώδικας Pascal:

```
if f(1)=f(1) then print("Yes") else print("No")
```

- Λόγω παρενεργειών, η ανακλαστική ιδιότητα $x = x$ δεν ισχύει
  - ο παραπάνω κώδικας δεν απλοποιείται σε `print("Yes")`
- Δε μπορούμε να χρησιμοποιήσουμε θεμελειώδη μαθηματικά στις αποδείξεις ορθότητας
- Οι τυπικές μέθοδοι για προστακτικό προγραμματισμό φροντίζουν να διαχωρίζουν τις *εντολές* από τις *εκφράσεις*
  - οι εκφράσεις δεν έχουν παρενέργειες
- Η απλότητα του παραπάνω κώδικα είναι παραπλανητική

Καθαρότητα/Αναγνωσιμότητα/Αποδειξιμότητα

Κώδικας Pascal:

```
if f(1)=f(1) then print("Yes") else print("No")
```

- Λόγω παρενεργειών, η ανακλαστική ιδιότητα $x = x$ δεν ισχύει
  - ο παραπάνω κώδικας δεν απλοποιείται σε `print("Yes")`
- Δε μπορούμε να χρησιμοποιήσουμε θεμελειώδη μαθηματικά στις αποδείξεις ορθότητας
- Οι τυπικές μέθοδοι για προστακτικό προγραμματισμό φροντίζουν να διαχωρίζουν τις *εντολές* από τις *εκφράσεις*
  - οι εκφράσεις δεν έχουν παρενέργειες
- Η απλότητα του παραπάνω κώδικα είναι παραπλανητική

Τμηματοποίηση Κώδικα

Κλασσική τμηματοποίηση σε προστακτικό προγραμματισμό:

- πακέτα, τμήματα, κλάσεις

Ιδιώματα του συναρτησιακού προγραμματισμού που δίνουν διαφορετικές δυνατότητες τμηματοποίησης:

- Συναρτήσεις *ως τιμές πρώτης κατηγορίας (first-class values)*
  - *συναρτήσεις υψηλότερης τάξης (higher order functions)*
- *Οκνηρή αποτίμηση (lazy evaluation)*

Τμηματοποίηση Κώδικα

Κλασσική τμηματοποίηση σε προστακτικό προγραμματισμό:

- πακέτα, τμήματα, κλάσεις

Ιδιώματα του συναρτησιακού προγραμματισμού που δίνουν διαφορετικές δυνατότητες τμηματοποίησης:

- Συναρτήσεις *ως τιμές πρώτης κατηγορίας (first-class values)*
  - *συναρτήσεις υψηλότερης τάξης (higher order functions)*
- *Οκνηρή αποτίμηση (lazy evaluation)*

Τμηματοποίηση Κώδικα

Κλασσική τμηματοποίηση σε προστακτικό προγραμματισμό:

- πακέτα, τμήματα, κλάσεις

Ιδιώματα του συναρτησιακού προγραμματισμού που δίνουν διαφορετικές δυνατότητες τμηματοποίησης:

- Συναρτήσεις *ως τιμές πρώτης κατηγορίας (first-class values)*
  - *συναρτήσεις υψηλότερης τάξης (higher order functions)*
- *Οκνηρή αποτίμηση (lazy evaluation)*

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας

Σε μια γλώσσα που υποστηρίζει αυτό το ιδίωμα:

- Συναρτήσεις μπορούν να περάσουν ως παράμετροι σε άλλες συναρτήσεις
- Συναρτήσεις μπορούν να επιστραφούν ως αποτέλεσμα από άλλες συναρτήσεις

Μας οδηγεί στην έννοια της *συνάρτησης υψηλότερης τάξης*:

- Μια συνάρτηση που έχει ως παράμετρο(υς) άλλη(ες) συνάρτηση(εις)

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας

Σε μια γλώσσα που υποστηρίζει αυτό το ιδίωμα:

- Συναρτήσεις μπορούν να περάσουν ως παράμετροι σε άλλες συναρτήσεις
- Συναρτήσεις μπορούν να επιστραφούν ως αποτέλεσμα από άλλες συναρτήσεις

Μας οδηγεί στην έννοια της *συνάρτησης υψηλότερης τάξης*:

- Μια συνάρτηση που έχει ως παράμετρο(υς) άλλη(ες) συνάρτηση(εις)

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας

Σε μια γλώσσα που υποστηρίζει αυτό το ιδίωμα:

- Συναρτήσεις μπορούν να περάσουν ως παράμετροι σε άλλες συναρτήσεις
- Συναρτήσεις μπορούν να επιστραφούν ως αποτέλεσμα από άλλες συναρτήσεις

Μας οδηγεί στην έννοια της *συνάρτησης υψηλότερης τάξης*:

- Μια συνάρτηση που έχει ως παράμετρο(υς) άλλη(ες) συνάρτηση(εις)

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας

Σε μια γλώσσα που υποστηρίζει αυτό το ιδίωμα:

- Συναρτήσεις μπορούν να περάσουν ως παράμετροι σε άλλες συναρτήσεις
- Συναρτήσεις μπορούν να επιστραφούν ως αποτέλεσμα από άλλες συναρτήσεις

Μας οδηγεί στην έννοια της *συνάρτησης υψηλότερης τάξης*:

- Μια συνάρτηση που έχει ως παράμετρο(υς) άλλη(ες) συνάρτηση(εις)

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Παράδειγμα συσσωρευτικού υπολογισμού

- Συνάρτηση που υπολογίζει το άθροισμα των στοιχείων μίας λίστας ακεραίων:

```
sum::[Int]->Int
sum l
  | l==[]     = 0
  | otherwise = head l + sum (tail l)
```

- Η συνάρτηση αυτή δίνεται από ένα *συσσωρευτικό υπολογισμό*
  - εφαρμογή της συνάρτησης `(+)` συσσωρευτικά σε όλα τα στοιχεία της λίστας
  - ξεκινάμε τη συσσώρευση από το 0 (για κενή λίστα)
- Άλλες συναρτήσεις που μπορούν να υπολογιστούν έτσι:
  - το γινόμενο όλων των αριθμών της λίστας
  - ο μέγιστος/ελάχιστος όλων των αριθμών της λίστας
  - ...

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Παράδειγμα συσσωρευτικού υπολογισμού

- Συνάρτηση που υπολογίζει το άθροισμα των στοιχείων μίας λίστας ακεραίων:

```
sum::[Int]->Int
sum l
  | l==[]     = 0
  | otherwise = head l + sum (tail l)
```

- Η συνάρτηση αυτή δίνεται από ένα *συσσωρευτικό υπολογισμό*
  - εφαρμογή της συνάρτησης `(+)` συσσωρευτικά σε όλα τα στοιχεία της λίστας
  - ξεκινάμε τη συσσώρευση από το 0 (για κενή λίστα)
- Άλλες συναρτήσεις που μπορούν να υπολογιστούν έτσι:
  - το γινόμενο όλων των αριθμών της λίστας
  - ο μέγιστος/ελάχιστος όλων των αριθμών της λίστας
  - ...

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Παράδειγμα συσσωρευτικού υπολογισμού

- Συνάρτηση που υπολογίζει το άθροισμα των στοιχείων μίας λίστας ακεραίων:

```
sum::[Int]->Int
sum l
  | l==[]     = 0
  | otherwise = head l + sum (tail l)
```

- Η συνάρτηση αυτή δίνεται από ένα *συσσωρευτικό υπολογισμό*
  - εφαρμογή της συνάρτησης `(+)` συσσωρευτικά σε όλα τα στοιχεία της λίστας
  - ξεκινάμε τη συσσώρευση από το 0 (για κενή λίστα)
- Άλλες συναρτήσεις που μπορούν να υπολογιστούν έτσι:
  - το γινόμενο όλων των αριθμών της λίστας
  - ο μέγιστος/ελάχιστος όλων των αριθμών της λίστας
  - ...

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Παράδειγμα συσσωρευτικού υπολογισμού

- Συνάρτηση που υπολογίζει το άθροισμα των στοιχείων μίας λίστας ακεραίων:

```
sum::[Int]->Int
sum l
  | l==[]     = 0
  | otherwise = head l + sum (tail l)
```

- Η συνάρτηση αυτή δίνεται από ένα *συσσωρευτικό υπολογισμό*
  - εφαρμογή της συνάρτησης `(+)` συσσωρευτικά σε όλα τα στοιχεία της λίστας
  - ξεκινάμε τη συσσώρευση από το 0 (για κενή λίστα)
- Άλλες συναρτήσεις που μπορούν να υπολογιστούν έτσι:
  - το γινόμενο όλων των αριθμών της λίστας
  - ο μέγιστος/ελάχιστος όλων των αριθμών της λίστας
  - ...

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Παράδειγμα συσσωρευτικού υπολογισμού

- Συνάρτηση που υπολογίζει το άθροισμα των στοιχείων μίας λίστας ακεραίων:

```
sum::[Int]->Int
sum l
  | l==[]      = 0
  | otherwise = head l + sum (tail l)
```

- Η συνάρτηση αυτή δίνεται από ένα *συσσωρευτικό υπολογισμό*
  - εφαρμογή της συνάρτησης (+) συσσωρευτικά σε όλα τα στοιχεία της λίστας
  - ξεκινάμε τη συσσώρευση από το 0 (για κενή λίστα)
- Άλλες συναρτήσεις που μπορούν να υπολογιστούν έτσι:
  - το γινόμενο όλων των αριθμών της λίστας
  - ο μέγιστος/ελάχιστος όλων των αριθμών της λίστας
  - ...

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Παράδειγμα συσσωρευτικού υπολογισμού

- Συνάρτηση που υπολογίζει το άθροισμα των στοιχείων μίας λίστας ακεραίων:

```
sum::[Int]->Int
sum l
  | l==[]      = 0
  | otherwise = head l + sum (tail l)
```

- Η συνάρτηση αυτή δίνεται από ένα *συσσωρευτικό υπολογισμό*
  - εφαρμογή της συνάρτησης `(+)` συσσωρευτικά σε όλα τα στοιχεία της λίστας
  - ξεκινάμε τη συσσώρευση από το 0 (για κενή λίστα)
- Άλλες συναρτήσεις που μπορούν να υπολογιστούν έτσι:
  - το γινόμενο όλων των αριθμών της λίστας
  - ο μέγιστος/ελάχιστος όλων των αριθμών της λίστας
  - ...

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Παράδειγμα συσσωρευτικού υπολογισμού

- Συνάρτηση που υπολογίζει το άθροισμα των στοιχείων μίας λίστας ακεραίων:

```
sum::[Int]->Int
sum l
  | l==[]      = 0
  | otherwise = head l + sum (tail l)
```

- Η συνάρτηση αυτή δίνεται από ένα *συσσωρευτικό υπολογισμό*
  - εφαρμογή της συνάρτησης `(+)` συσσωρευτικά σε όλα τα στοιχεία της λίστας
  - ξεκινάμε τη συσσώρευση από το 0 (για κενή λίστα)
- Άλλες συναρτήσεις που μπορούν να υπολογιστούν έτσι:
  - το γινόμενο όλων των αριθμών της λίστας
  - ο μέγιστος/ελάχιστος όλων των αριθμών της λίστας
  - ...

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση (+)

```
reduce::
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι `0`
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

- Θα απομονώσουμε την έννοια "συσσωρευτικός υπολογισμός" σε μια νέα συνάρτηση με όνομα `reduce`
- Η `reduce` θα παίρνει σαν παραμέτρους την αρχική τιμή και τη συνάρτηση συσσώρευσης. Στη `sum`:
  - η αρχική τιμή είναι 0
  - η συνάρτηση συσσώρευσης είναι η πρόσθεση `(+)`

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]       = i
  | otherwise   = f (head l) (reduce i f (tail l))
```

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Συσσωρευτικός υπολογισμός αθροίσματος/γινομένου με τη `reduce`

- Όλοι οι συσσωρευτικοί υπολογισμοί είναι συναρτήσεις που μπορούν να δημιουργηθούν από τη `reduce`
- Άθροιση των στοιχείων μιας λίστας: εφαρμόζουμε τη `reduce` με αρχική τιμή `0` και με συνάρτηση συσσώρευσης την πρόσθεση

```
add::Int->Int->Int
add x y = x+y
sum::[Int]->Int
sum = reduce 0 add
```

- Στη Haskell, η πρόσθεση συμβολίζεται `(+)`

```
sum::[Int]->Int
sum = reduce 0 (+)
```

- Ομοίως για συσσωρευτικό πολλαπλασιασμό:

```
mult::[Int]->Int
mult = reduce 1 (*)
```

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Συσσωρευτικός υπολογισμός αθροίσματος/γινομένου με τη `reduce`

- Όλοι οι συσσωρευτικοί υπολογισμοί είναι συναρτήσεις που μπορούν να δημιουργηθούν από τη `reduce`
- Άθροιση των στοιχείων μιας λίστας: εφαρμόζουμε τη `reduce` με αρχική τιμή `0` και με συνάρτηση συσσώρευσης την πρόσθεση

```
add::Int->Int->Int
add x y = x+y
sum::[Int]->Int
sum = reduce 0 add
```

- Στη Haskell, η πρόσθεση συμβολίζεται `(+)`

```
sum::[Int]->Int
sum = reduce 0 (+)
```

- Ομοίως για συσσωρευτικό πολλαπλασιασμό:

```
mult::[Int]->Int
mult = reduce 1 (*)
```

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Συσσωρευτικός υπολογισμός αθροίσματος/γινομένου με τη `reduce`

- Όλοι οι συσσωρευτικοί υπολογισμοί είναι συναρτήσεις που μπορούν να δημιουργηθούν από τη `reduce`
- Άθροιση των στοιχείων μιας λίστας: εφαρμόζουμε τη `reduce` με αρχική τιμή 0 και με συνάρτηση συσσώρευσης την πρόσθεση

```
add::Int->Int->Int
add x y = x+y
sum::[Int]->Int
sum = reduce 0 add
```

- Στη Haskell, η πρόσθεση συμβολίζεται `(+)`

```
sum::[Int]->Int
sum = reduce 0 (+)
```

- Ομοίως για συσσωρευτικό πολλαπλασιασμό:

```
mult::[Int]->Int
mult = reduce 1 (*)
```

Συναρτήσεις ως Τιμές Πρώτης Κατηγορίας
Συσσωρευτικός υπολογισμός αθροίσματος/γινομένου με τη `reduce`

- Όλοι οι συσσωρευτικοί υπολογισμοί είναι συναρτήσεις που μπορούν να δημιουργηθούν από τη `reduce`
- Άθροιση των στοιχείων μιας λίστας: εφαρμόζουμε τη `reduce` με αρχική τιμή `0` και με συνάρτηση συσσώρευσης την πρόσθεση

```
add::Int->Int->Int
add x y = x+y
sum::[Int]->Int
sum = reduce 0 add
```

- Στη Haskell, η πρόσθεση συμβολίζεται `(+)`

```
sum::[Int]->Int
sum = reduce 0 (+)
```

- Ομοίως για συσσωρευτικό πολλαπλασιασμό:

```
mult::[Int]->Int
mult = reduce 1 (*)
```

- Διαχωρίσαμε το μηχανισμό συσσωρευτικού υπολογισμού από τη συνάρτηση συσσώρευσης
  - κάθε νέος συσσωρευτικός υπολογισμός μπορεί να δημιουργηθεί αμέσως επαναχρησιμοποιώντας τη `reduce`
  - τα κλασσικά σχήματα τμηματοποίησης δεν επιτρέπουν τέτοια τμηματοποίηση
  - στις προστακτικές γλώσσες: πρόβλημα με την απόδειξη ορθότητας
- Η `reduce` είναι συνάρτηση ανώτερης τάξης

- Διαχωρίσαμε το μηχανισμό συσσωρευτικού υπολογισμού από τη συνάρτηση συσσώρευσης
  - κάθε νέος συσσωρευτικός υπολογισμός μπορεί να δημιουργηθεί αμέσως επαναχρησιμοποιώντας τη `reduce`
  - τα κλασσικά σχήματα τμηματοποίησης δεν επιτρέπουν τέτοια τμηματοποίηση
  - στις προστακτικές γλώσσες: πρόβλημα με την απόδειξη ορθότητας
- Η `reduce` είναι συνάρτηση ανώτερης τάξης

- Διαχωρίσαμε το μηχανισμό συσσωρευτικού υπολογισμού από τη συνάρτηση συσσώρευσης
  - κάθε νέος συσσωρευτικός υπολογισμός μπορεί να δημιουργηθεί αμέσως επαναχρησιμοποιώντας τη `reduce`
  - τα κλασσικά σχήματα τμηματοποίησης δεν επιτρέπουν τέτοια τμηματοποίηση
  - στις προστακτικές γλώσσες: πρόβλημα με την απόδειξη ορθότητας
- Η `reduce` είναι συνάρτηση ανώτερης τάξης

- Διαχωρίσαμε το μηχανισμό συσσωρευτικού υπολογισμού από τη συνάρτηση συσσώρευσης
  - κάθε νέος συσσωρευτικός υπολογισμός μπορεί να δημιουργηθεί αμέσως επαναχρησιμοποιώντας τη `reduce`
  - τα κλασσικά σχήματα τμηματοποίησης δεν επιτρέπουν τέτοια τμηματοποίηση
  - στις προστακτικές γλώσσες: πρόβλημα με την απόδειξη ορθότητας
- Η `reduce` είναι συνάρτηση ανώτερης τάξης

- Διαχωρίσαμε το μηχανισμό συσσωρευτικού υπολογισμού από τη συνάρτηση συσσώρευσης
  - κάθε νέος συσσωρευτικός υπολογισμός μπορεί να δημιουργηθεί αμέσως επαναχρησιμοποιώντας τη `reduce`
  - τα κλασσικά σχήματα τμηματοποίησης δεν επιτρέπουν τέτοια τμηματοποίηση
  - στις προστακτικές γλώσσες: πρόβλημα με την απόδειξη ορθότητας
- Η `reduce` είναι συνάρτηση ανώτερης τάξης

Μπορούμε να ορίσουμε συναρτήσεις δίνοντας σε μία συνάρτηση λιγότερα ορίσματα από αυτά που παίρνει:

- η reduce έχει 3 ορίσματα:
  `reduce::Int->(Int->Int->Int)->[Int]->Int`
- ο ορισμός της sum της περνάει 2 ορίσματα:
  `sum=reduce 0 (+)`
- προκύπτει μία συνάρτηση που περιμένει τα υπόλοιπα ορίσματα:
  `sum::[Int]->Int`
- ανεπίτρεπτο στις κλασσικές προστακτικές γλώσσες

Μπορούμε να ορίσουμε συναρτήσεις δίνοντας σε μία συνάρτηση λιγότερα ορίσματα από αυτά που παίρνει:

- η reduce έχει 3 ορίσματα:
  `reduce::Int->(Int->Int->Int)->[Int]->Int`
- ο ορισμός της sum της περνάει 2 ορίσματα:
  `sum=reduce 0 (+)`
- προκύπτει μία συνάρτηση που περιμένει τα υπόλοιπα ορίσματα:
  `sum::[Int]->Int`
- ανεπίτρεπτο στις κλασσικές προστακτικές γλώσσες

Μπορούμε να ορίσουμε συναρτήσεις δίνοντας σε μία συνάρτηση λιγότερα ορίσματα από αυτά που παίρνει:

- η reduce έχει 3 ορίσματα:
  `reduce::Int->(Int->Int->Int)->[Int]->Int`
- ο ορισμός της sum της περνάει 2 ορίσματα:
  `sum=reduce 0 (+)`
- προκύπτει μία συνάρτηση που περιμένει τα υπόλοιπα ορίσματα:
  `sum::[Int]->Int`
- ανεπίτρεπτο στις κλασσικές προστακτικές γλώσσες

Μπορούμε να ορίσουμε συναρτήσεις δίνοντας σε μία συνάρτηση λιγότερα ορίσματα από αυτά που παίρνει:

- η reduce έχει 3 ορίσματα:
  `reduce::Int->(Int->Int->Int)->[Int]->Int`
- ο ορισμός της sum της περνάει 2 ορίσματα:
  `sum=reduce 0 (+)`
- προκύπτει μία συνάρτηση που περιμένει τα υπόλοιπα ορίσματα:
  `sum::[Int]->Int`
- ανεπίτρεπτο στις κλασσικές προστακτικές γλώσσες

Οκνηρή Αποτίμηση

- Στην *πρόθυμη αποτίμηση* (eager evaluation), μία δομή δεδομένων κατασκευάζεται όταν οριστεί
- Στην *οκνηρή αποτίμηση* (lazy evaluation), η δομή *δεν* κατασκευάζεται με τον ορισμό της
  - κάθε τμήμα της δομής κατασκευάζεται μόνο όταν χρειαστεί σε υπολογισμό
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
- Απλοϊκό παράδειγμα ορισμού δομής δεδομένων στη Haskell:

```
listOfOnes::[Int]
listOfOnes = [1] ++ listOfOnes
```

(ο `(++)` είναι ο τελεστής συνένωσης λιστών)

- Πρόκειται για μη πεπερασμένη δομή
  - πρόθυμη αποτίμηση $\rightarrow$ ατέρμων βρόγχος

Οκνηρή Αποτίμηση

- Στην *πρόθυμη αποτίμηση* (eager evaluation), μία δομή δεδομένων κατασκευάζεται όταν οριστεί
- Στην *οκνηρή αποτίμηση* (lazy evaluation), η δομή *δεν* κατασκευάζεται με τον ορισμό της
  - κάθε τμήμα της δομής κατασκευάζεται μόνο όταν χρειαστεί σε υπολογισμό
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
- Απλοϊκό παράδειγμα ορισμού δομής δεδομένων στη Haskell:

```
listOfOnes::[Int]
listOfOnes = [1] ++ listOfOnes
```

(ο `(++)` είναι ο τελεστής συνένωσης λιστών)

- Πρόκειται για μη πεπερασμένη δομή
  - πρόθυμη αποτίμηση $\rightarrow$ ατέρμων βρόγχος

Οκνηρή Αποτίμηση

- Στην *πρόθυμη αποτίμηση* (eager evaluation), μία δομή δεδομένων κατασκευάζεται όταν οριστεί
- Στην *οκνηρή αποτίμηση* (lazy evaluation), η δομή *δεν* κατασκευάζεται με τον ορισμό της
  - κάθε τμήμα της δομής κατασκευάζεται μόνο όταν χρειαστεί σε υπολογισμό
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
- Απλοϊκό παράδειγμα ορισμού δομής δεδομένων στη Haskell:

```
listOfOnes::[Int]
listOfOnes = [1] ++ listOfOnes
```

(ο `(++)` είναι ο τελεστής συνένωσης λιστών)

- Πρόκειται για μη πεπερασμένη δομή
  - πρόθυμη αποτίμηση $\rightarrow$ ατέρμων βρόγχος

Οκνηρή Αποτίμηση

- Στην *πρόθυμη αποτίμηση* (eager evaluation), μία δομή δεδομένων κατασκευάζεται όταν οριστεί
- Στην *οκνηρή αποτίμηση* (lazy evaluation), η δομή *δεν* κατασκευάζεται με τον ορισμό της
  - κάθε τμήμα της δομής κατασκευάζεται μόνο όταν χρειαστεί σε υπολογισμό
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
- Απλοϊκό παράδειγμα ορισμού δομής δεδομένων στη Haskell:

```
listOfOnes::[Int]
listOfOnes = [1] ++ listOfOnes
```

(ο `(++)` είναι ο τελεστής συνένωσης λιστών)

- Πρόκειται για μη πεπερασμένη δομή
  - πρόθυμη αποτίμηση → ατέρμων βρόγχος

Οκνηρή Αποτίμηση

- Στην *πρόθυμη αποτίμηση* (eager evaluation), μία δομή δεδομένων κατασκευάζεται όταν οριστεί
- Στην *οκνηρή αποτίμηση* (lazy evaluation), η δομή *δεν* κατασκευάζεται με τον ορισμό της
  - κάθε τμήμα της δομής κατασκευάζεται μόνο όταν χρειαστεί σε υπολογισμό
- Η Haskell υποστηρίζει οκνηρή αποτίμηση
- Απλοϊκό παράδειγμα ορισμού δομής δεδομένων στη Haskell:

```
listOfOnes::[Int]
listOfOnes = [1] ++ listOfOnes
```

(ο `(++)` είναι ο τελεστής συνένωσης λιστών)

- Πρόκειται για μη πεπερασμένη δομή
  - πρόθυμη αποτίμηση $\rightarrow$ ατέρμων βρόγχος

Οκνηρή Αποτίμηση

Τμηματική αποτίμηση της `listOfOnes`

- Στη Haskell δεν υπάρχει πρόβλημα. Ο ορισμός της `listOfOnes` δεν κατασκευάζει τη λίστα
- Τμήματα της λίστας θα κατασκευαστούν μόνο μόλις χρειαστούν στον υπολογισμό
  - αν ρωτήσουμε: `head listOfOnes`
    η Haskell θα απαντήσει: `1`
  - αν ρωτήσουμε: `head (tail (tail listOfOnes))`
    η Haskell θα απαντήσει: `1`

Οκνηρή Αποτίμηση

Τμηματική αποτίμηση της `listOfOnes`

- Στη Haskell δεν υπάρχει πρόβλημα. Ο ορισμός της `listOfOnes` δεν κατασκευάζει τη λίστα
- Τμήματα της λίστας θα κατασκευαστούν μόνο μόλις χρειαστούν στον υπολογισμό
  - αν ρωτήσουμε: `head listOfOnes`
    η Haskell θα απαντήσει: `1`
  - αν ρωτήσουμε: `head (tail (tail listOfOnes))`
    η Haskell θα απαντήσει: `1`

- Στη Haskell δεν υπάρχει πρόβλημα. Ο ορισμός της `listOfOnes` δεν κατασκευάζει τη λίστα
- Τμήματα της λίστας θα κατασκευαστούν μόνο μόλις χρειαστούν στον υπολογισμό
  - αν ρωτήσουμε: `head listOfOnes`
    η Haskell θα απαντήσει: `1`
  - αν ρωτήσουμε: `head (tail (tail listOfOnes))`
    η Haskell θα απαντήσει: `1`

Οκνηρή Αποτίμηση
Χρησιμότητα

- Η οκνηρή αποτίμηση διαχωρίζει την κατασκευή μίας (πιθανώς πολύπλοκης ή και άπειρης) δομής δεδομένων από τη χρήση της
- Κατασκευαστής δομής: αδιαφορεί για τη χρήση της και το μέγεθός της, αφού ξέρει ότι μόνο τα χρήσιμα τμήματα θα κατασκευαστούν
- Χρήστης δομής: αδιαφορεί για την κατασκευή της δομής. Μπορεί να τη θεωρήσει κατασκευασμένη και να επικεντρωθεί στη χρήση της
- Αντίθετα στην πρόθυμη αποτίμηση είμαστε αναγκασμένοι να κάνουμε και τα δύο στο ίδιο κομμάτι κώδικα

- Η οκνηρή αποτίμηση διαχωρίζει την κατασκευή μίας (πιθανώς πολύπλοκης ή και άπειρης) δομής δεδομένων από τη χρήση της
- Κατασκευαστής δομής: αδιαφορεί για τη χρήση της και το μέγεθός της, αφού ξέρει ότι μόνο τα χρήσιμα τμήματα θα κατασκευαστούν
- Χρήστης δομής: αδιαφορεί για την κατασκευή της δομής. Μπορεί να τη θεωρήσει κατασκευασμένη και να επικεντρωθεί στη χρήση της
- Αντίθετα στην πρόθυμη αποτίμηση είμαστε αναγκασμένοι να κάνουμε και τα δύο στο ίδιο κομμάτι κώδικα

Οκνηρή Αποτίμηση
Χρησιμότητα

- Η οκνηρή αποτίμηση διαχωρίζει την κατασκευή μίας (πιθανώς πολύπλοκης ή και άπειρης) δομής δεδομένων από τη χρήση της
- Κατασκευαστής δομής: αδιαφορεί για τη χρήση της και το μέγεθός της, αφού ξέρει ότι μόνο τα χρήσιμα τμήματα θα κατασκευαστούν
- Χρήστης δομής: αδιαφορεί για την κατασκευή της δομής. Μπορεί να τη θεωρήσει κατασκευασμένη και να επικεντρωθεί στη χρήση της
- Αντίθετα στην πρόθυμη αποτίμηση είμαστε αναγκασμένοι να κάνουμε και τα δύο στο ίδιο κομμάτι κώδικα

- Η οκνηρή αποτίμηση διαχωρίζει την κατασκευή μίας (πιθανώς πολύπλοκης ή και άπειρης) δομής δεδομένων από τη χρήση της
- Κατασκευαστής δομής: αδιαφορεί για τη χρήση της και το μέγεθός της, αφού ξέρει ότι μόνο τα χρήσιμα τμήματα θα κατασκευαστούν
- Χρήστης δομής: αδιαφορεί για την κατασκευή της δομής. Μπορεί να τη θεωρήσει κατασκευασμένη και να επικεντρωθεί στη χρήση της
- Αντίθετα στην πρόθυμη αποτίμηση είμαστε αναγκασμένοι να κάνουμε και τα δύο στο ίδιο κομμάτι κώδικα

Δέντρο καταστάσεων ένος παιχνιδιού (π.χ. σκάκι)

- το δέντρο καταστάσεων είναι μία τεράστια δομή
- μπορούμε να την κατασκευάσουμε ανεξάρτητα από τη χρήση της
- η χρήση της μπορεί να είναι από έναν αλγόριθμο τεχνητής νοημοσύνης που παίζει το παιχνίδι
- η πολυπλοκότητα των δύο αλγορίθμων δεν αναμιγνύεται

Περισσότερα για τη χρησιμότητα του Σ.Π., δείτε:
J. Hughes. Why functional Programming Matters. *Computer Journal*, 32(2),98-107, 1989

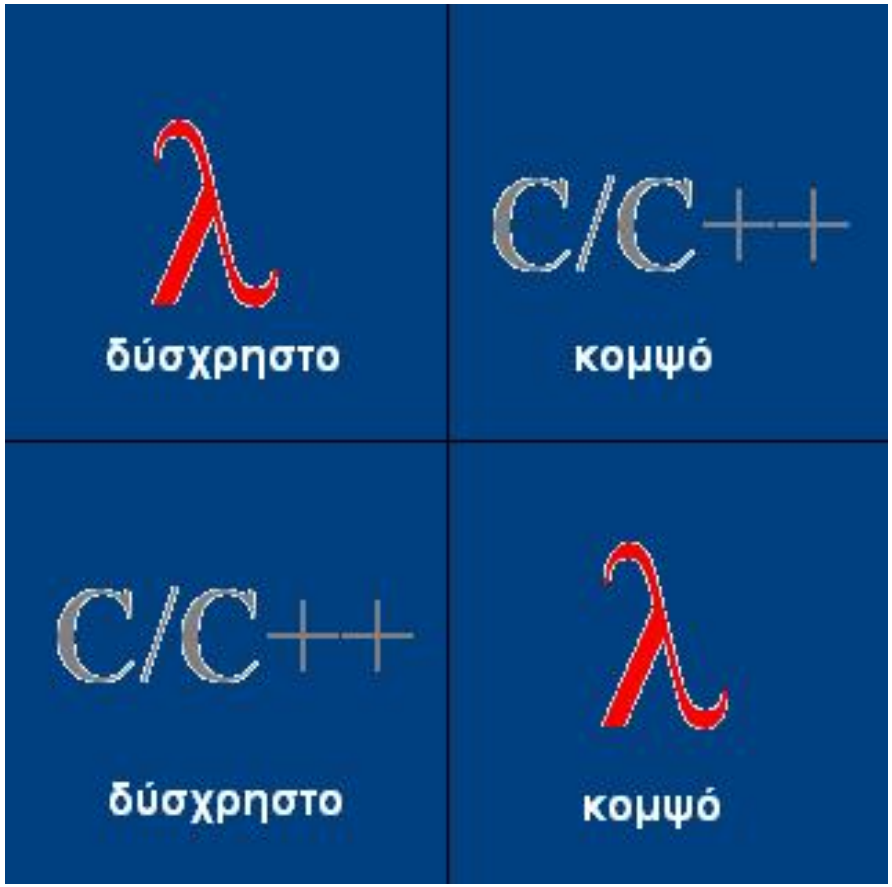
Δέντρο καταστάσεων ένος παιχνιδιού (π.χ. σκάκι)

- το δέντρο καταστάσεων είναι μία τεράστια δομή
- μπορούμε να την κατασκευάσουμε ανεξάρτητα από τη χρήση της
- η χρήση της μπορεί να είναι από έναν αλγόριθμο τεχνητής νοημοσύνης που παίζει το παιχνίδι
- η πολυπλοκότητα των δύο αλγορίθμων δεν αναμιγνύεται

Περισσότερα για τη χρησιμότητα του Σ.Π., δείτε:
J. Hughes. Why functional Programming Matters. *Computer Journal*, 32(2),98-107, 1989

Συναρτησιακός vs. Προστακτικός Προγραμματισμός

Ο "σωστός" τρόπος αντίληψης του υπολογισμού είναι υποκειμενικό θέμα



Ανακεφαλαίωση

- Ο Σ.Π. είναι ένα υψηλού επιπέδου μοντέλο υπολογισμού
  - χρησιμοποιεί μαθηματικές συναρτήσεις
- Στόχος: κώδικας καλής ποιότητας
  - καλογραμμένος, ευανάγνωστος, εύχρηστος από θεωρίες απόδειξης, τμηματοποιημένος
- Σημαντικά ιδιώματα:
  - συναρτήσεις ως τιμές πρώτης τάξης
  - οκνηρή αποτίμηση
- Συναρτήσεις ως τιμές πρώτης τάξης
  - βοηθάνε στην τμηματοποίηση κώδικα με νέους τρόπους
- Οκνηρή αποτίμηση
  - διαχωρίζει την κατασκευή από τη χρήση πολύπλοκων δομών δεδομένων
- Βλέπουμε το Σ.Π. ως *συμπληρωματικό* και όχι ως ανταγωνιστικό στον προστακτικό προγραμματισμό.

Ανακεφαλαίωση

- Ο Σ.Π. είναι ένα υψηλού επιπέδου μοντέλο υπολογισμού
  - χρησιμοποιεί μαθηματικές συναρτήσεις
- Στόχος: κώδικας καλής ποιότητας
  - καλογραμμένος, ευανάγνωστος, εύχρηστος από θεωρίες απόδειξης, τμηματοποιημένος
- Σημαντικά ιδιώματα:
  - συναρτήσεις ως τιμές πρώτης τάξης
  - οκνηρή αποτίμηση
- Συναρτήσεις ως τιμές πρώτης τάξης
  - βοηθάνε στην τμηματοποίηση κώδικα με νέους τρόπους
- Οκνηρή αποτίμηση
  - διαχωρίζει την κατασκευή από τη χρήση πολύπλοκων δομών δεδομένων
- Βλέπουμε το Σ.Π. ως *συμπληρωματικό* και όχι ως ανταγωνιστικό στον προστακτικό προγραμματισμό.

Ανακεφαλαίωση

- Ο Σ.Π. είναι ένα υψηλού επιπέδου μοντέλο υπολογισμού
  - χρησιμοποιεί μαθηματικές συναρτήσεις
- Στόχος: κώδικας καλής ποιότητας
  - καλογραμμένος, ευανάγνωστος, εύχρηστος από θεωρίες απόδειξης, τμηματοποιημένος
- Σημαντικά ιδιώματα:
  - συναρτήσεις ως τιμές πρώτης τάξης
  - οκνηρή αποτίμηση
- Συναρτήσεις ως τιμές πρώτης τάξης
  - βοηθάνε στην τμηματοποίηση κώδικα με νέους τρόπους
- Οκνηρή αποτίμηση
  - διαχωρίζει την κατασκευή από τη χρήση πολύπλοκων δομών δεδομένων
- Βλέπουμε το Σ.Π. ως *συμπληρωματικό* και όχι ως ανταγωνιστικό στον προστακτικό προγραμματισμό.

Ανακεφαλαίωση

- Ο Σ.Π. είναι ένα υψηλού επιπέδου μοντέλο υπολογισμού
  - χρησιμοποιεί μαθηματικές συναρτήσεις
- Στόχος: κώδικας καλής ποιότητας
  - καλογραμμένος, ευανάγνωστος, εύχρηστος από θεωρίες απόδειξης, τμηματοποιημένος
- Σημαντικά ιδιώματα:
  - συναρτήσεις ως τιμές πρώτης τάξης
  - οκνηρή αποτίμηση
- Συναρτήσεις ως τιμές πρώτης τάξης
  - βοηθάνε στην τμηματοποίηση κώδικα με νέους τρόπους
- Οκνηρή αποτίμηση
  - διαχωρίζει την κατασκευή από τη χρήση πολύπλοκων δομών δεδομένων
- Βλέπουμε το Σ.Π. ως *συμπληρωματικό* και όχι ως ανταγωνιστικό στον προστακτικό προγραμματισμό.

Ανακεφαλαίωση

- Ο Σ.Π. είναι ένα υψηλού επιπέδου μοντέλο υπολογισμού
  - χρησιμοποιεί μαθηματικές συναρτήσεις
- Στόχος: κώδικας καλής ποιότητας
  - καλογραμμένος, ευανάγνωστος, εύχρηστος από θεωρίες απόδειξης, τμηματοποιημένος
- Σημαντικά ιδιώματα:
  - συναρτήσεις ως τιμές πρώτης τάξης
  - οκνηρή αποτίμηση
- Συναρτήσεις ως τιμές πρώτης τάξης
  - βοηθάνε στην τμηματοποίηση κώδικα με νέους τρόπους
- Οκνηρή αποτίμηση
  - διαχωρίζει την κατασκευή από τη χρήση πολύπλοκων δομών δεδομένων
- Βλέπουμε το Σ.Π. ως *συμπληρωματικό* και όχι ως ανταγωνιστικό στον προστακτικό προγραμματισμό.

Ανακεφαλαίωση

- Ο Σ.Π. είναι ένα υψηλού επιπέδου μοντέλο υπολογισμού
  - χρησιμοποιεί μαθηματικές συναρτήσεις
- Στόχος: κώδικας καλής ποιότητας
  - καλογραμμένος, ευανάγνωστος, εύχρηστος από θεωρίες απόδειξης, τμηματοποιημένος
- Σημαντικά ιδιώματα:
  - συναρτήσεις ως τιμές πρώτης τάξης
  - οκνηρή αποτίμηση
- Συναρτήσεις ως τιμές πρώτης τάξης
  - βοηθάνε στην τμηματοποίηση κώδικα με νέους τρόπους
- Οκνηρή αποτίμηση
  - διαχωρίζει την κατασκευή από τη χρήση πολύπλοκων δομών δεδομένων
- Βλέπουμε το Σ.Π. ως *συμπληρωματικό* και όχι ως ανταγωνιστικό στον προστακτικό προγραμματισμό.

Επικοινωνία

Γιάννης Κασσιός
e-mail: kassios@di.uoa.gr
Web Site μαθήματος:
http://www.di.uoa.gr/~kassios/courses/fp