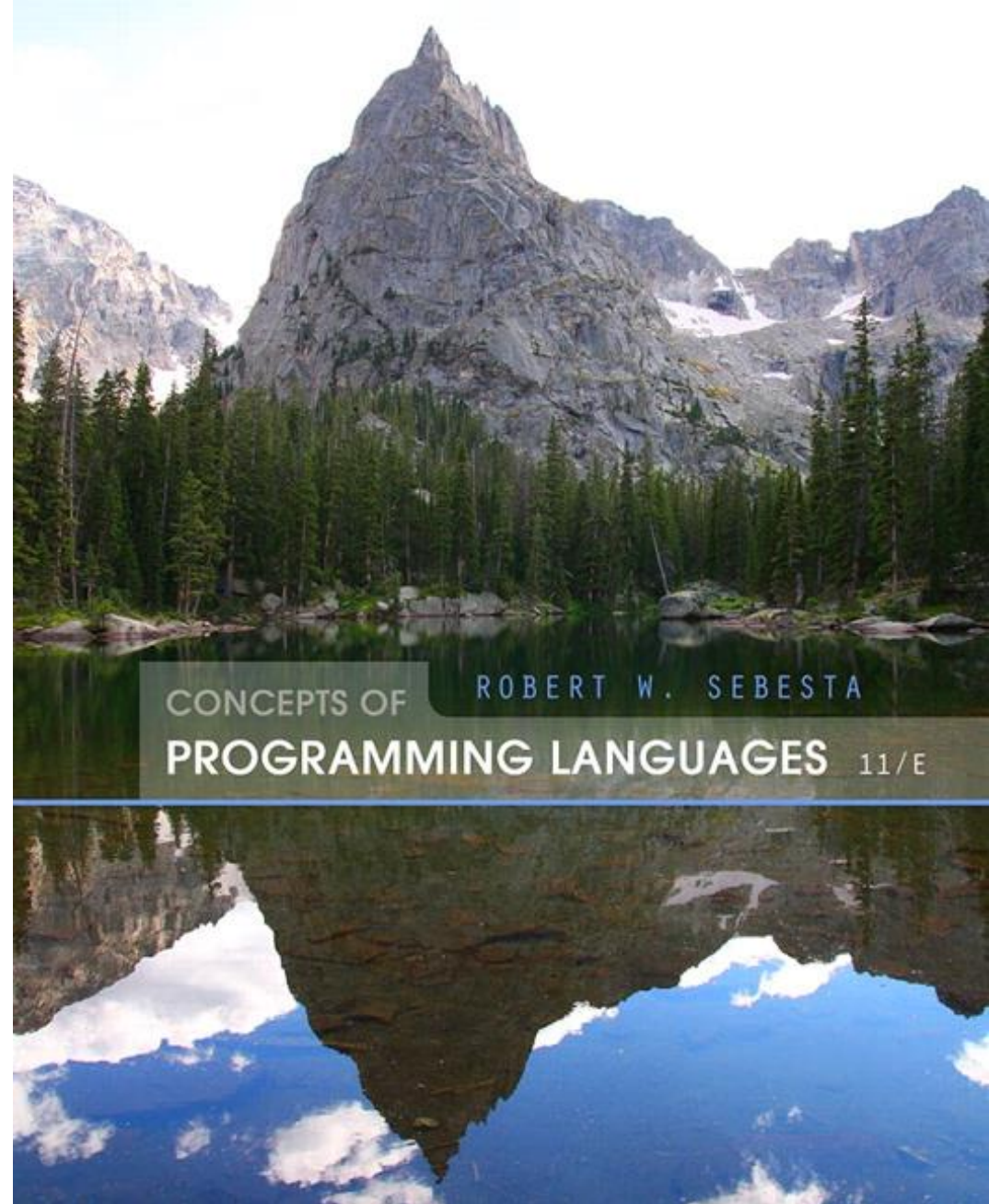


# Κεφάλαιο 15

## Γλώσσες Συναρτησιακού Προγραμματισμού

Γκόγκος Χρήστος  
Τμήμα Πληροφορικής και Τηλεπικοινωνιών (Αρτα)  
Πανεπιστήμιο Ιωαννίνων



# Θέματα Κεφαλαίου 15

---

- Εισαγωγή
- Μαθηματικές συναρτήσεις
- Θεμελιώδεις αρχές των συναρτησιακών γλωσσών προγραμματισμού
- Η πρώτη συναρτησιακή γλώσσα: Lisp
- Εισαγωγή στην Scheme
- Common Lisp
- ML
- Haskell
- F#
- Υποστήριξη για συναρτησιακό προγραμματισμό σε κυρίως προστακτικές γλώσσες
- Σύγκριση συναρτησιακών και προστακτικών γλωσσών

# Εισαγωγή

---

- Ο σχεδιασμός των προστακτικών γλωσσών βασίζεται στην *αρχιτεκτονική von Neumann*
  - Η επιδίωξη υψηλής αποδοτικότητας αποτέλεσε τον κυρίαρχο παράγοντα ανάπτυξης των προστακτικών γλωσσών, και λιγότερο η καταλληλότητα της γλώσσας στην ανάπτυξη λογισμικού
- Ο σχεδιασμός των συναρτησιακών γλωσσών βασίζεται στις μαθηματικές συναρτήσεις
  - Οι συναρτησιακές γλώσσες διαθέτουν ένα στέρεο θεωρητικό υπόβαθρο που είναι πλησιέστερα προς τον χρήστη, αλλά μακριά από την αρχιτεκτονική των μηχανών στις οποίες τα προγράμματα εκτελούνται

# Μαθηματικές συναρτήσεις

---

- Μια μαθηματική συνάρτηση είναι μια αντιστοίχιση μελών ενός συνόλου, που ονομάζεται πεδίο ορισμού, σε ένα άλλο σύνολο, που ονομάζεται πεδίο τιμών
- Μια *λάμδα έκφραση* καθορίζει τις παραμέτρους και την αντιστοίχιση της συνάρτησης με την ακόλουθη μορφή

$$\lambda(x) \quad x * x * x$$

για τη συνάρτηση  $\text{cube}(x) = x * x * x$

# Λάμδα Εκφράσεις

---

- Οι λάμδα εκφράσεις περιγράφουν ανώνυμες συναρτήσεις
- Οι λάμδα εκφράσεις εφαρμόζονται σε παραμέτρους τοποθετώντας τις παραμέτρους μετά την έκφραση

π.χ.,  $(\lambda (x) x * x * x) (2)$

που αποτιμάται στην τιμή 8

# Συναρτησιακές Μορφές

---

- Μια συνάρτηση υψηλής τάξης (higher-order function, ή *functional form*), είναι μια συνάρτηση που είτε δέχεται άλλες συναρτήσεις ως παραμέτρους ή επιστρέφει μια συνάρτηση ως αποτέλεσμα, ή και τα δύο

# Σύνθεση συναρτήσεων

---

- Η σύνθεση (composition) είναι μια συναρτησιακή μορφή που δέχεται δύο συναρτήσεις ως παραμέτρους και επιστρέφει μια συνάρτηση η τιμή της οποίας είναι η συνάρτηση της πρώτης παραμέτρου με παράμετρο την τιμή της συνάρτησης της δεύτερης παραμέτρου

Μορφή:  $h \equiv f \circ g$

που σημαίνει  $h(x) \equiv f(g(x))$

Για  $f(x) \equiv x + 2$  και  $g(x) \equiv 3 * x$ ,

Η  $h \equiv f \circ g$  παράγει  $(3 * x) + 2$

# Εφαρμογή-σε-όλα

---

- Η εφαρμογή-σε-όλα (apply-to-all) είναι μια συναρτησιακή μορφή που παίρνει μια απλή συνάρτηση ως παράμετρο και επιστρέφει μια λίστα τιμών που λαμβάνεται εφαρμόζοντας τη δεδομένη συνάρτηση σε κάθε στοιχείο της 2<sup>ης</sup> παραμέτρου που είναι μια λίστα:

Μορφή:  $\alpha$

Για  $h(x) \equiv x * x$

$\alpha(h, (2, 3, 4))$  επιστρέφει  $(4, 9, 16)$



# Θεμελιώδεις αρχές των συναρτησιακών γλωσσών προγραμματισμού

---

- Σχεδιαστικός στόχος των γλωσσών συναρτησιακού προγραμματισμού είναι η μίμηση των μαθηματικών συναρτήσεων στο μεγαλύτερο δυνατό βαθμό
- Η βασική υπολογιστική διαδικασία είναι θεμελιωδώς διαφορετική στις συναρτησιακές γλώσσες προγραμματισμού από ότι στις γλώσσες προσακτικού προγραμματισμού
  - Σε μια προσακτική γλώσσα τα αποτελέσματα των λειτουργιών αποθηκεύονται σε μεταβλητές για μελλοντική χρήση
  - Η διαχείριση των μεταβλητών είναι ένα συνεχές ζήτημα καθώς και πηγή πολυπλοκότητας στις προσακτικές γλώσσες
- Στις γλώσσες συναρτησιακού προγραμματισμού, οι μεταβλητές δεν είναι απαραίτητες, όπως και στα μαθηματικά
- *Referential Transparency (διαφάνεια αναφοράς)* – Σε μια γλώσσα συναρτησιακού προγραμματισμού, η αποτίμηση μιας συνάρτησης δίνει πάντα το ίδιο αποτέλεσμα εφόσον χρησιμοποιούνται οι ίδιες παράμετροι

# Τύποι και δομές δεδομένων της Lisp

---

- *Κατηγορίες αντικειμένων δεδομένων:*  
αρχικά μόνο άτομα και λίστες
- *Μορφή λίστας:* εντός παρενθέσεων συλλογές υπολιστών και/ή ατόμων  
π.χ., (A B (C D) E)
- Αρχικά, η Lisp ήταν μια γλώσσα χωρίς τύπους
- Οι λίστες της Lisp αποθηκεύονται εσωτερικά ως απλά συνδεδεμένες λίστες

# Lisp Ερμηνεία

---

- Ο συμβολισμός λάμδα χρησιμοποιείται για να καθορίσει συναρτήσεις και ορισμούς συναρτήσεων. Η κλήση συναρτήσεων και τα δεδομένα έχουν την ίδια μορφή.  
π.χ., Αν η λίστα (A B C) ερμηνευτεί ως δεδομένα τότε είναι μια απλή λίστα τριών ατόμων, A, B, και C  
Αν η λίστα ερμηνευτεί ως κλήση συνάρτησης σημαίνει ότι η συνάρτηση με όνομα A καλείται με δύο παραμέτρους, B και C
- Ο πρώτος διερμηνευτής της Lisp προέκυψε ως επίδειξη της καθολικότητας των υπολογιστικών δυνατοτήτων που παρείχε ο συμβολισμός της γλώσσας

# Προέλευση της Scheme

---

- Στα μέσα της δεκαετίας του 1970 σχεδιάστηκε η Scheme ως μια καθαρότερη, μοντέρνα (για την εποχή) και απλούστερη διάλεκτος της Lisp
- Χρησιμοποιεί μόνο στατική εμφάνιση
- Οι συναρτήσεις είναι οντότητες πρώτης-τάξης
  - Μπορούν να είναι τιμές εκφράσεων και στοιχεία λιστών
  - Μπορούν να ανατίθενται σε μεταβλητές, να μεταβιβάζονται ως παράμετροι, και να επιστρέφονται από συναρτήσεις

# Ο διερμηνευτής της Scheme

---

- Ο χρήστης μπορεί να αλληλοεπιδράσει με το διερμηνευτή της Scheme μέσω του read-evaluate-print loop (REPL)
  - Αυτή η μορφή αλληλεπίδρασης με διερμηνευτή χρησιμοποιείται επίσης στην Python και στην Ruby
- Οι εκφράσεις διερμηνεύονται με τη συνάρτηση `EVAL`
- Τα κυριολεκτικά αποτιμώνται στους εαυτούς τους

# Αποτίμηση Συναρτήσεων

---

- Οι παράμετροι αποτιμώνται, όχι σε κάποια συγκεκριμένη σειρά
- Οι τιμές των παραμέτρων αντικαθίστανται στο σώμα της συνάρτησης
- Το σώμα της συνάρτησης αποτιμάται
- Η τιμή της τελευταίας έκφρασης στο σώμα της συνάρτησης είναι η τιμή της συνάρτησης

# Βασικές Συναρτήσεις & LAMBDA Εκφράσεις

---

- Βασικές αριθμητικές συναρτήσεις: +, -, \*, /, ABS, SQRT, REMAINDER, MIN, MAX

π.χ., (+ 5 2) παράγει 7

- Lambda Εκφράσεις

- Η μορφή τους βασίζεται στο λ συμβολισμό

π.χ., (LAMBDA (x) (\* x x))

η x είναι μια φραγμένη μεταβλητή

- Οι lambda εκφράσεις μπορούν να εφαρμοστούν σε παραμέτρους

π.χ., ((LAMBDA (x) (\* x x)) 7)

- Οι LAMBDA εκφράσεις μπορούν να έχουν οποιοδήποτε αριθμό παραμέτρων

(LAMBDA (a b x) (+ (\* a x x) (\* b x)))

# Μια ειδική συνάρτηση: DEFINE

---

- DEFINE – Δύο μορφές:
  1. Για δέσμευση ενός συμβόλου σε μια έκφραση  
π.χ., (DEFINE pi 3.141593)  
Παράδειγμα χρήσης: (DEFINE two\_pi (\* 2 pi))  
Τα σύμβολα αυτά δεν είναι μεταβλητές - είναι σαν τα ονόματα στις δηλώσεις `final` της Java
  2. Για να δεσμευθούν ονόματα σε λάμδα εκφράσεις (το LAMBDA υπονοείται)  
π.χ., (DEFINE (square x) (\* x x))  
Παράδειγμα χρήσης: (square 5)
- Η διαδικασία αποτίμησης του DEFINE είναι διαφορετική! Η πρώτη παράμετρος δεν αποτιμάται ποτέ. Η δεύτερη παράμετρος αποτιμάται και προσδένεται στην πρώτη παράμετρο



# Συναρτήσεις εξόδου

---

- Συνήθως δεν χρειάζονται, διότι ο διερμηνευτής πάντα εμφανίζει το αποτέλεσμα της αποτίμησης στο κορυφαίο επίπεδο
- Η Scheme έχει την `PRINTF`, που είναι παρόμοια με τη συνάρτηση `printf` της C
- Σημείωση: Η είσοδος και η έξοδος δεν αποτελούν μέρη του μοντέλου καθαρού συναρτησιακού προγραμματισμού, διότι οι λειτουργίες εισόδου αλλάζουν την κατάσταση του προγράμματος και οι λειτουργίες εξόδου προκαλούν παράπλευρες συνέπειες (side-effects)

# Συναρτήσεις Αριθμητικών Κατηγορημάτων

---

- #T (ή #t) σημαίνει αληθής και #F (ή #f) σημαίνει ψευδής (μερικές φορές το () χρησιμοποιείται για το ψευδές)
- =, <>, >, <, >=, <=
- EVEN?, ODD?, ZERO?, NEGATIVE?
- Η συνάρτηση NOT αντιστρέφει τη λογική τιμή μιας Boolean έκφρασης

# Ροή ελέγχου

---

- Επιλογή– ειδική μορφή, `IF`

```
(IF predicate then_exp else_exp)
```

```
(IF (<> count 0)
    (/ sum count)
)
```

- Η συνάρτηση `COND` :

```
(DEFINE (leap? year)
  (COND
    ((ZERO? (MODULO year 400)) #T)
    ((ZERO? (MODULO year 100)) #F)
    (ELSE (ZERO? (MODULO year 4))))
))
```

# Συναρτήσεις λιστών

---

- QUOTE – δέχεται μια παράμετρο – επιστρέφει την παράμετρο χωρίς αποτίμηση
  - Η QUOTE χρειάζεται διότι ο διερμηνευτής της Scheme, ο EVAL, πάντοτε αποτιμά τις παραμέτρους πριν τις εφαρμόσει στη συνάρτηση. Η QUOTE χρησιμοποιείται έτσι ώστε να αποφευχθεί η αποτίμηση των παραμέτρων όταν αυτό δεν είναι επιθυμητό
  - QUOTE μπορεί να αντικατασταθεί με τον τελεστή της αποστρόφου  
ΤΟ ' (A B) είναι ισοδύναμο με ΤΟ (QUOTE (A B))

# Συναρτήσεις λιστών (συνέχεια)

---

- Παραδείγματα:

(CAR '(A B) C D) επιστρέφει (A B)

(CAR 'A) είναι λάθος

(CDR '(A B) C D) επιστρέφει (C D)

(CDR 'A) είναι λάθος

(CDR '(A)) επιστρέφει ()

(CONS '() '(A B)) επιστρέφει (() A B)

(CONS '(A B) '(C D)) επιστρέφει ((A B) C D)

(CONS 'A 'B) επιστρέφει (A . B) (ένα ζεύγος με τελεία)

# Συναρτήσεις λιστών (συνέχεια)

---

- `LIST` είναι μια συνάρτηση για δημιουργία λιστών από ένα οποιοδήποτε πλήθος παραμέτρων

`(LIST 'apple 'orange 'grape)` επιστρέφει

`(apple orange grape)`

# Η συνάρτηση κατηγορημα: EQ?

---

- Η EQ? δέχεται δύο εκφράσεις ως παραμέτρους (συνήθως δύο atoms), επιστρέφει #T αν και οι δύο παράμετροι έχουν την ίδια τιμή δείκτη, αλλιώς επιστρέφει #F

(EQ? 'A 'A) παράγει #T

(EQ? 'A 'B) παράγει #F

(EQ? 'A '(A B)) παράγει #F

(EQ? '(A B) '(A B)) παράγει #T ή #F

(EQ? 3.4 (+ 3 0.4)) παράγει #T ή #F

# Η συνάρτηση κατηγορημα: EQV?

---

- Η EQV? είναι παρόμοια με την EQ?, με τη διαφορά ότι λειτουργεί τόσο για συμβολικά όσο και για αριθμητικά atoms, πρόκειται για σύγκριση τιμής, και όχι για σύγκριση δεικτών

(EQV? 3 3) παράγει #T

(EQV? 'A 3) παράγει #F

(EQV? 3.4 (+ 3 0.4)) παράγει #T

(EQV? 3.0 3) παράγει #F (οι αριθμοί κινητής υποδιαστολής και οι ακέραιοι είναι διαφορετικοί)



# Οι συναρτήσεις: LIST? ΚΑΙ NULL?

---

- Η `LIST?` δέχεται μια παράμετρο, επιστρέφει `#T` αν η παράμετρος είναι μια λίστα, αλλιώς `#F`  
(`LIST? '()`) παράγει `#T`
- Η `NULL?` δέχεται μια παράμετρο, επιστρέφει `#T` αν η παράμετρος είναι η άδεια λίστα, αλλιώς επιστρέφει `#F`  
(`NULL? '(())`) παράγει `#F`

# Παράδειγμα συνάρτησης στη Scheme: `member`

---

- Η `member` δέχεται ένα `atom` και μια απλή λίστα, επιστρέφει `#T` αν το άτομο βρίσκεται στη λίστα, `#F` αλλιώς

```
DEFINE (member atm a_list)
(COND
  ((NULL? a_list) #F)
  ((EQ? atm (CAR lis)) #T)
  ((ELSE (member atm (CDR a_list))))
))
```

# Παράδειγμα συνάρτησης στη Scheme:

## `equalsimp`

---

- Η `equalsimp` δέχεται δύο απλές λίστες ως παραμέτρους, επιστρέφει `#T` αν οι δύο απλές λίστες είναι ίσες, `#F` αλλιώς

```
(DEFINE (equalsimp list1 list2)
(COND
  ((NULL? list1) (NULL? list2))
  ((NULL? list2) #F)
  ((EQ? (CAR list1) (CAR list2))
   (equalsimp(CDR list1) (CDR list2)))
  (ELSE #F)
))
```

# Παράδειγμα συνάρτησης στη Scheme : `equal`

---

- Η `equal` δέχεται δύο γενικές λίστες ως παραμέτρους, επιστρέφει `#T` αν οι δύο λίστες είναι ίσες, `#F` αλλιώς

```
(DEFINE (equal list1 list2)
  (COND
    ((NOT (LIST? list1)) (EQ? list1 list2))
    ((NOT (LIST? list2)) #F)
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((equal (CAR list1) (CAR list2))
     (equal (CDR list1) (CDR list2)))
    (ELSE #F)
  ))
```

# Παράδειγμα συνάρτησης στη Scheme: `append`

---

- Η `append` δέχεται δύο λίστες ως παραμέτρους, επιστρέφει την πρώτη λίστα με τα στοιχεία της δεύτερης λίστας προσαρτημένα στο τέλος της

```
(DEFINE (append list1 list2)
  (COND
    ((NULL? list1) list2)
    (ELSE (CONS (CAR list1)
                 (append (CDR list1) list2))))
))
```

# Παράδειγμα συνάρτησης στη Scheme: LET

---

- Η `LET` συζητήθηκε στο Κεφάλαιο 5
- Η `LET` είναι ένας σύντομος τρόπος να γραφεί μια έκφραση `LAMBDA` που εφαρμόζεται σε μια παράμετρο

```
(LET ((alpha 7)) (* 5 alpha))
```

είναι το ίδιο με:

```
((LAMBDA (alpha) (* 5 alpha)) 7)
```

# Παράδειγμα με το LET

---

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
      (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
    (LIST (+ minus_b_over_2a root_part_over_2a)
          (- minus_b_over_2a root_part_over_2a))
  ))
```

# Αναδρομή ουράς στη Scheme

---

- Ορισμός: Μια συνάρτηση είναι *tail recursive* αν η αναδρομική κλήση της είναι η τελευταία λειτουργία της συνάρτησης
- Μια συνάρτηση με αναδρομή ουράς μπορεί αυτόματα να μετατραπεί από το μεταγλωττιστή έτσι ώστε να χρησιμοποιεί επανάληψη, και συνεπώς να εκτελείται ταχύτερα
- Ο ορισμός της γλώσσας Scheme καθορίζει ότι τα Scheme συστήματα θα πρέπει να μετατρέπουν όλες τις συναρτήσεις tail recursive έτσι ώστε να χρησιμοποιούν επανάληψη



# Αναδρομή ουράς στη Scheme – συνέχεια

---

- Παράδειγμα επαναδιατύπωσης συνάρτησης έτσι ώστε να γίνει tail recursive, χρησιμοποιώντας μια βοηθητική συνάρτηση

Original: 

```
(DEFINE (factorial n)
  (IF (<= n 0)
      1
      (* n (factorial (- n 1)))
  ))
```

Tail recursive: 

```
(DEFINE (facthelper n factpartial)
  (IF (<= n 0)
      factpartial
      facthelper((- n 1) (* n factpartial)))
  ))
(DEFINE (factorial n)
  (facthelper n 1))
```

# Η συναρτησιακή μορφή: Σύνθεση

---

- Σύνθεση (composition)

- Αν  $h$  είναι η σύνθεση της  $f$  και της  $g$ ,  $h(x) = f(g(x))$

```
(DEFINE (g x) (* 3 x))
```

```
(DEFINE (f x) (+ 2 x))
```

```
(DEFINE h x) (+ 2 (* 3 x)) (Η σύνθεση)
```

- Στη Scheme, η συνάρτηση σύνθεσης `compose` μπορεί να γραφεί ως εξής:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
```

```
((compose CAR CDR) '(a b) c d) παράγει c
```

```
(DEFINE (third a_list)
```

```
((compose CAR (compose CDR CDR)) a_list))
```

είναι ισοδύναμο με το `CADDR`

# Η συναρτησιακή μορφή: Εφαρμογή-σε-όλα

---

- Εφαρμογή-σε-όλα: μια μορφή της στη Scheme είναι η `map`
  - Εφαρμόζει τη συνάρτηση που δίνεται σε όλα τα στοιχεία μιας δεδομένης λίστας:

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list))
                 (map fun (CDR a_list))))
  ))
```

```
(map (LAMBDA (num) (* num num num)) '(3 4 2 6))
παράγει (27 64 8 216)
```

# Συναρτήσεις που δημιουργούν κώδικα

---

- Η Scheme δίνει τη δυνατότητα να ορίσουμε συναρτήσεις που δημιουργούν κώδικα Scheme και στη συνέχεια ζητούν τη διερμηνεία του
- Αυτό είναι εφικτό διότι ο ίδιος ο διερμηνευτής είναι διαθέσιμος προς τον προγραμματιστή μέσω της συνάρτησης  
EVAL

# Πρόσθεση μιας λίστας αριθμών

---

```
((DEFINE (adder a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (EVAL (CONS '+ a_list))))
))
```

- Η παράμετρος είναι μια λίστα αριθμών προς πρόσθεση, ο `adder` εισάγει τον τελεστή `+` ανάμεσα στα στοιχεία της λίστας και την αποτιμά. Ειδικότερα:
  - Χρησιμοποιεί το `CONS` για να εισάγει τον ατομικό όρο `+` στη λίστα των αριθμών.
  - Πρέπει το `+` να προηγείται το `'` έτσι ώστε να αποφευχθεί η αποτίμηση
  - Υποβάλλει τη νέα λίστα στο `EVAL` για αποτίμηση

# Common Lisp

---

- Συνδυάζει διάφορα χαρακτηριστικά των πλέον δημοφιλών διαλέκτων της Lisp που υπήρχαν στη δεκαετία του 1980
- Αποτελεί μια μεγάλη και σύνθετη γλώσσα – το αντίθετο της Scheme
- Ορισμένα χαρακτηριστικά της Common Lisp:
  - εγγραφές
  - πίνακες
  - μιγαδικοί αριθμοί
  - λεκτικά χαρακτήρων
  - Ισχυρές δυνατότητες I/O
  - πακέτα με έλεγχο πρόσβασης
  - προστακτικές εντολές ελέγχου

# Common Lisp (συνέχεια)

---

- Μακροεντολές
  - Λειτουργούν σε δύο βήματα:
    - Επέκταση της μακροεντολής
    - Αποτίμηση της μακροεντολής που έχει επεκταθεί
- Ορισμένες από τις προκαθορισμένες συναρτήσεις της Common Lisp είναι μακροεντολές
- Οι χρήστες μπορούν να ορίσουν τις δικές τους μακροεντολές με το `DEFMACRO`

# Common Lisp (συνέχεια)

---

- Τελεστής backquote (``)
  - Λειτουργεί παρόμοια με το QUOTE της Scheme με τη διαφορά ότι ορισμένα τμήματα της παραμέτρου που δέχεται μπορούν να εξαιρεθούν τοποθετώντας πριν από αυτά κόμματα

```(a (* 3 4) c)` αποτιμάται σε `(a (* 3 4) c)`

```(a , (* 3 4) c)` αποτιμάται σε `(a 12 c)`



# Common Lisp (συνέχεια)

---

- Μακροεντολές reader

- Lisp implementations have a front end called the *reader* that transforms Lisp into a code representation. Then macro calls are expanded into the code representation.
- Μια reader μακροεντολή είναι μια ειδική μορφή μακροεντολής που επεκτείνεται στη βάση ανάγνωσης
- A reader macro is a definition of a single character, which is expanded into its Lisp definition
- Ένα παράδειγμα μιας reader μακροεντολής είναι ο χαρακτήρας της αποστρόφου, που επεκτείνεται σε μια κλήση του `QUOTE`
- Οι χρήστες μπορούν να ορίζουν τις δικές τους reader μακροεντολές ως μια μορφή συντόμευσης

# Common Lisp (συνέχεια)

---

- Η Common Lisp διαθέτει ένα τύπο δεδομένων συμβόλων (παρόμοια με την Ruby)
  - Οι δεσμευμένες λέξεις είναι σύμβολα που αποτιμώνται στον εαυτό τους
  - Τα σύμβολα είναι είτε φραγμένα είτε μη φραγμένα
    - Τα σύμβολα των παραμέτρων είναι φραγμένα όσο η συνάρτηση αποτιμάται
    - Τα σύμβολα που είναι ονόματα μεταβλητών προστακτικού στυλ και στα οποία έχουν ανατεθεί τιμές είναι φραγμένα
    - Όλα τα άλλα σύμβολα είναι μη φραγμένα

# ML

---

- Η ML είναι μια συναρτησιακή γλώσσα στατικής εμβέλειας με συντακτικό πλησιέστερα στην Pascal παρά στην Lisp
- Δηλώσεις τύπων χρήστη, αλλά επίσης και *type inferencing* για να καθορίσει τον τύπο των μη δηλωμένων μεταβλητών
- Είναι strongly typed (ενώ η Scheme είναι ουσιαστικά χωρίς τύπους) και δεν έχει μετατροπές τύπων (type coercions)
- Δεν διαθέτει μεταβλητές προσακτικού στυλ
- Τα αναγνωριστικά της είναι χωρίς τύπους ονόματα για τιμές
- Υποστηρίζει χειρισμό εξαιρέσεων και έναν τρόπο (module facility) υλοποίησης αφηρημένων τύπων δεδομένων
- Περιέχει λίστες και λειτουργίες πάνω σε λίστες

# Ιδιαίτερα χαρακτηριστικά της ML

- Ένας πίνακας που ονομάζεται περιβάλλον αποτίμησης (*evaluation environment*) αποθηκεύει τα ονόματα όλων των αναγνωριστικών ενός προγράμματος μαζί με τους τύπους τους (μπορεί να θεωρηθεί σαν ένας πίνακας συμβόλων χρόνου εκτέλεσης)
- Μορφή δήλωσης συνάρτησης:

**fun** *name* (*formal parameters*) = *expression*;

Π.χ., **fun** cube (*x* : **int**) = *x* \* *x* \* *x*;

- Ο τύπος μπορεί να προστεθεί στην τιμή επιστροφής, όπως στη συνέχεια:

**fun** cube (*x*) : **int** = *x* \* *x* \* *x*;

- Αν δεν καθοριστεί τύπος τότε χρησιμοποιείται το ως προκαθορισμένος το

**int** (προκαθορισμένο για αριθμητικές τιμές)

- Δεν επιτρέπονται υπερφορτωμένες συναρτήσεις του χρήστη, οπότε αν επιθυμούσαμε μια συνάρτηση *cube* για πραγματικές παραμέτρους, θα έπρεπε να επιλεγεί διαφορετικό όνομα

# Ιδιαίτερα χαρακτηριστικά της ML

## (συνέχεια)

---

- Η επιλογή στην ML:

```
if expression then then_expression  
else else_expression
```

όπου η πρώτη έκφραση πρέπει να αποτιμάται σε μια Boolean τιμή

- Το ταίριασμα προτύπων (pattern matching) χρησιμοποιείται έτσι ώστε η συνάρτηση να εφαρμόζεται σε διαφορετικές μορφές παραμέτρων:

```
fun fact(0) = 1  
  | fact(1) = 1  
  | fact(n : int) : int = n * fact(n - 1)
```

# Ιδιαίτερα χαρακτηριστικά της ML (συνέχεια)

---

- Λίστες

Οι λίστες κυριολεκτικών ορίζονται σε αγκύλες

`[3, 5, 7]`

`[]` είναι η άδεια λίστα

`CONS` είναι ο δυαδικός infix τελεστής, `::`

`4 :: [3, 5, 7]`, που αποτιμάται σε `[4, 3, 5, 7]`

`CAR` είναι ο μοναδιαίος τελεστής `hd`

`CDR` είναι ο μοναδιαίος τελεστής `tl`

```
fun length([]) = 0
```

```
| length(h :: t) = 1 + length(t);
```

```
fun append([], lis2) = lis2
```

```
| append(h :: t, lis2) = h :: append(t, lis2);
```

# Ιδιαίτερα χαρακτηριστικά της ML (συνέχεια)

---

- Η εντολή `val` προσδένει ένα όνομα με μια τιμή (παρόμοια με το `DEFINE` στη Scheme)

```
val distance = time * speed;
```

- Όπως και με το `DEFINE`, το `val` δεν είναι παρόμοιο με τις εντολές ανάθεσης στις προστακτικές γλώσσες
- Αν υπάρχουν δύο εντολές `val` για το ίδιο αναγνωριστικό, η πρώτη κρύβεται από τη δεύτερη
- Οι εντολές `val` συχνά χρησιμοποιούνται σε δομές `let`

```
let
```

```
  val radius = 2.7
```

```
  val pi = 3.14159
```

```
in
```

```
  pi * radius * radius
```

```
end;
```

# Ιδιαιτέρα χαρακτηριστικά της ML

## (συνέχεια)

---

- `filter`
    - Είναι μια συνάρτηση υψηλότερης τάξης `higher-order` για φιλτράρισμα σε λίστες
    - Δέχεται μια συνάρτηση κατηγορημα ως παράμετρο, συχνά με τη μορφή μιας λάμδα έκφρασης
    - Οι λάμδα εκφράσεις ορίζονται όπως οι συναρτήσεις, εκτός από το ότι χρησιμοποιείται η δεσμευμένη λέξη `fn`
- ```
filter(fn(x) => x < 100, [25, 1, 711, 50, 100]);
```
- Που επιστρέφει [25, 1, 50]



# Ιδιαίτερα χαρακτηριστικά της ML

## (συνέχεια)

---

- `map`
  - Είναι μια συνάρτηση υψηλότερης τάξης που δέχεται μια παράμετρο, μια συνάρτηση
  - Εφαρμόζει την παράμετρο συνάρτηση σε κάθε στοιχείο μιας λίστας και επιστρέφει μια λίστα αποτελεσμάτων

```
fun cube x = x * x * x;
```

```
val cubeList = map cube;
```

```
val newList = cubeList [1, 3, 5];
```

Που θέτει την `newList` σε `[1, 27, 125]`

- Εναλλακτικά: με χρήση λάμδα έκφρασης

```
val newList = map (fn x => x * x * x, [1, 3, 5]);
```

# Ιδιαιτέρα χαρακτηριστικά της ML

## (συνέχεια)

---

- Σύνθεση συναρτήσεων
  - Χρησιμοποιώντας τον μοναδιαίο τελεστή,  $\circ$

```
val h = g o f;
```

# Ιδιαιτέρα χαρακτηριστικά της ML

## (συνέχεια)

---

- **Currying**

- Οι συναρτήσεις στην ML στην πράξη δέχονται μόνο μια παράμετρο — αν δίνονται περισσότερες, θεωρεί τις παραμέτρους ως πλειάδες (απαιτείται κόμμα)
- Η διαδικασία του *currying* αντικαθιστά μια συνάρτηση με περισσότερες από μια παραμέτρους με μια συνάρτηση με μια παράμετρο που επιστρέφει μια συνάρτηση που δέχεται τις υπόλοιπες παραμέτρους της αρχικής συνάρτησης
- Μια ML συνάρτηση που δέχεται περισσότερες από μια παραμέτρους μπορεί να οριστεί σε μορφή *curried*, αν δεν χρησιμοποιηθούν κόμματα στις παραμέτρους

```
fun add a b = a + b;
```

Μια συνάρτηση με μια παράμετρο, *a*. Επιστρέφει μια συνάρτηση που δέχεται το *b* ως παράμετρο. Κλήση:

```
add 3 5;
```

# Ιδιαίτερα χαρακτηριστικά της ML(συνέχεια)

---

- Μερική αποτίμηση

- Οι curried συναρτήσεις μπορούν να χρησιμοποιηθούν για να δημιουργήσουν νέες συναρτήσεις με μερική αποτίμηση
- Η μερική αποτίμηση σημαίνει ότι η συνάρτηση αποτιμάται με πραγματικές παραμέτρους για μια ή περισσότερες από τις αριστερότερες πραγματικές παραμέτρους της

```
fun add5 x add 5 x;
```

Λαμβάνει την πραγματική παράμετρο 5 και αποτιμά τη συνάρτηση `add` με το 5 ως την τιμή της πρώτης της τυπικής παραμέτρου. Επιστρέφει μια συνάρτηση που προσθέτει το 5 στη μοναδική της παράμετρο

```
val num = add5 10; (* θέτει το num σε 15 *)
```

# Haskell

---

- Ομοιότητες με την ML (συντακτικό, στατική εμβέλεια, με ισχυρούς κανόνες τύπων, συμπερασμό τύπων - type inference, ταίριασμα προτύπων - pattern matching)
- Διαφορές από την ML (και από τις περισσότερες συναρτησιακές γλώσσες) καθώς είναι αμιγώς συναρτησιακή (δηλαδή, χωρίς μεταβλητές, χωρίς εντολές εκχώρησης, και χωρίς παράπλευρες συνέπειες οποιουδήποτε τύπου)

```
fact 0 = 1
```

```
fact 1 = 1
```

```
fact n = n * fact (n - 1)
```

```
fib 0 = 1
```

```
fib 1 = 1
```

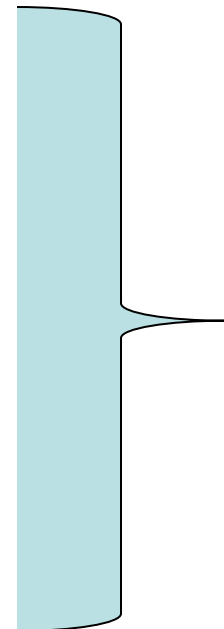
```
fib n = fib (n - 1) + fib (n - 2)
```

# Ορισμοί συναρτήσεων καθορίζοντας διάφορα διαστήματα για τις παραμέτρους

---

```
fact n
| n == 0 = 1
| n == 1 = 1
| n > 0 = n * fact (n - 1)
```

```
sub n
| n < 10      = 0
| n > 100     = 2
| otherwise  = 1
```



Guards (φρουροί)

```
square x = x * x
```

- Καθώς η Haskell υποστηρίζει τον πολυμορφισμό το παραπάνω λειτουργεί με οποιοδήποτε αριθμητικό τύπο για το  $x$

# Λίστες της Haskell

---

- Συμβολισμός για τις λίστες: Τοποθέτηση των στοιχείων σε αγκύλες  
π.χ., `directions = ["north", "south", "east", "west"]`
- Μήκος: `length`  
π.χ., `length directions` επιστρέφει 4
- Δημιουργία αριθμητικών σειρών με τον τελεστή `..`  
π.χ., `[2, 4..10]` επιστρέφει `[2, 4, 6, 8, 10]`
- Η συνένωση γίνεται με τον τελεστή `++`  
π.χ., `[1, 3] ++ [5, 7]` επιστρέφει `[1, 3, 5, 7]`
- `CONS`, `CAR`, `CDR` μέσω του τελεστή `:`  
π.χ., `1:[3, 5, 7]` επιστρέφει `[1, 3, 5, 7]`

# Haskell (συνέχεια)

---

- Τάιριασμα προτύπων (pattern matching)

```
product [] = 1
```

```
product (a:x) = a * product x
```

- Εναλλακτική υλοποίηση παραγοντικού:

```
fact n = product [1..n]
```

- Περιφραστικές λίστες (list comprehensions)

```
[n * n * n | n <- [1..50]]
```

Οι περιφραστικές λίστες μπορούν να περιέχουν και ελέγχους, όπως στη συνέχεια

```
factors n = [i | i <- [1..n `div` 2], n `mod` i == 0]
```

Τα `backticks` καθορίζουν ότι η συνάρτηση χρησιμοποιείται ως ενδοθεματικός τελεστής



# Γρήγορη ταξινόμηση (quicksort)

---

```
sort [] = []
sort (h:t) =
    sort [b | b <- t, b <= h]
    ++ [h] ++
    sort [b | b <- t, b > h]
```

Η παραπάνω υλοποίηση της quicksort δείχνει τη συντομία με την οποία μπορούν να επιλυθούν προβλήματα στην Haskell

# Οκνηρή αποτίμηση (lazy evaluation)

---

- Μια γλώσσα είναι *αυστηρή* (*strict*) αν απαιτεί την πλήρη αποτίμηση όλων των πραγματικών παραμέτρων
- Μια γλώσσα είναι *χαλαρή* (*nonstrict*) αν δεν απαιτεί την πλήρη αποτίμηση όλων των πραγματικών παραμέτρων
- Οι χαλαρές γλώσσες είναι αποδοτικότερες και προσφέρουν ορισμένες ενδιαφέρουσες δυνατότητες – (π.χ. *άπειρες λίστες*)
- Οκνηρή αποτίμηση – Υπολογισμός μόνο των τιμών που είναι απαραίτητες

```
positives = [1..]
```

- Παράδειγμα: Έλεγχος εάν το 16 είναι τετραγωνικός αριθμός
- Η συνάρτηση `member` θα μπορούσε να γραφεί ως εξής:

```
member b [] = False
member b (a:x) = (a == b) || member b x
squares = [n * n | n <- [0..]]
member 16 squares
```

# Η συνάρτηση member (ξανά)

---

```
member b [] = False
```

```
member b (a:x) = (a == b) || member b x
```

- Η παραπάνω έκδοση της member λειτουργεί σωστά μόνο αν η παράμετρος του στοιχείου που αναζητείται υπάρχει στην άπειρη λίστα squares – αν αυτό δεν συμβαίνει, θα συνεχίσει να δημιουργεί τιμές για πάντα. Η ακόλουθη έκδοση θα λειτουργεί σε όλες τις περιπτώσεις:

```
member2 b (a:x)
```

```
  | a < b = member2 b x
```

```
  | a == b = True
```

```
  | otherwise = False
```

# F#

---

- Βασίζεται στην Ocaml, που είναι απόγονος της ML και της Haskell
- Βασικά είναι μια συναρτησιακή γλώσσα, αλλά έχει και προστακτικά χαρακτηριστικά και υποστηρίζει Αντικειμενοστραφή Προγραμματισμό
- Διαθέτει ένα πλήρες IDE, μια εκτεταμένη βιβλιοθήκη με βοηθητικά προγράμματα, και επιτρέπει τη διαλειτουργικότητα με άλλες .NET γλώσσες
- Περιέχει πλειάδες, λίστες, discriminated ενώσεις, εγγραφές, και τόσο mutable όσο και immutable πίνακες
- Υποστηρίζει γενικές ακολουθίες, των οποίων οι τιμές μπορούν να δημιουργηθούν με generators και μέσω επαναλήψεων

# F# (συνέχεια)

---

- Ακολουθίες

```
let x = seq {1..4};;
```

- Η δημιουργία των τιμών της ακολουθίας γίνεται οκνηρά (lazy)

```
let y = seq {0..10000000};;
```

Θέτει το `y` στο `[0; 1; 2; 3; ...]`

- Το προκαθορισμένο βήμα είναι 1, αλλά μπορεί να είναι και οποιοσδήποτε αριθμός

```
let seq1 = seq {1..2..7}
```

Θέτει το `seq1` σε `[1; 3; 5; 7]`

- Iterators – δεν είναι οκνηροί για λίστες και πίνακες

```
let cubes = seq {for i in 1..4 -> (i, i * i * i)};;
```

Θέτει το `cubes` σε `[(1, 1); (2, 8); (3, 27); (4, 64)]`

# F# (συνέχεια)

---

- Συναρτήσεις

- Αν έχουν όνομα, ορίζονται με το `let` - αν είναι λάμδα εκφράσεις, ορίζονται με το `fun`  
(`fun a b -> a / b`)
- Δεν υπάρχει διαφορά ανάμεσα σε ένα όνομα που ορίζεται με το `let` και σε μια συνάρτηση χωρίς παραμέτρους
- Το εύρος στο οποίο εκτείνεται μια συνάρτηση ορίζεται από την εσοχή

```
let f =  
    let pi = 3.14159  
    let twoPi = 2.0 * pi;;
```

# F# (συνέχεια)

---

- Συναρτήσεις (συνέχεια)

- Αν μια συνάρτηση είναι αναδρομική, ο ορισμός της θα πρέπει να περιέχει τη δεσμευμένη λέξη `rec`
- Τα ονόματα στις συναρτήσεις μπορούν να βγουν εκτός εμβέλειας (outscoped), οπότε τερματίζεται η εμβέλειά τους

```
let x4 =  
    let x = x * x  
    let x = x * x
```

Το πρώτο `let` στο σώμα της συνάρτησης δημιουργεί μια νέα έκδοση του `x` - αυτό τερματίζει την εμβέλεια της παραμέτρου - Το δεύτερο `let` στο σώμα της συνάρτησης δημιουργεί ένα άλλο `x`, τερματίζοντας την εμβέλεια του δεύτερου `x`

# F# (συνέχεια)

---

- Συναρτησιακοί Τελεστές
  - Pipeline (`|>`)
  - Ένας δυαδικός τελεστής που στέλνει την τιμή του αριστερού ορίσματος στην τελευταία παράμετρο της κλήσης (στο δεξιό όρισμα)

```
let myNums = [1; 2; 3; 4; 5]
```

```
let evenTimesFive = myNums
```

```
    |> List.filter (fun n -> n % 2 = 0)
```

```
    |> List.map (fun n -> 5 * n)
```

Η τιμή που επιστρέφει είναι `[10; 20]`



# F# (συνέχεια)

---

- Συναρτησιακοί τελεστές (συνέχεια)

- Σύνθεση (>>)

- Δημιουργεί μια συνάρτηση που εφαρμόζει τον αριστερό της τελεστέο σε μια δοσμένη παράμετρο (μια συνάρτηση) και μετά περνά το αποτέλεσμα που επιστρέφεται από τη συνάρτηση στο δεξί της τελεστέο (μια άλλη συνάρτηση)

Η έκφραση F#  $(f \gg g) \ x$  είναι ισοδύναμη με τη μαθηματική έκφραση  $g(f(x))$

- Curried Συναρτήσεις

```
let add a b = a + b;;
```

```
let add5 = add 5;;
```

# F# (συνέχεια)

---

- Γιατί παρουσιάζει ενδιαφέρον η F#;
  - Έχει βασιστεί σε προηγούμενες συναρτησιακές γλώσσες
  - Υποστηρίζει ιδεατά όλες τις προγραμματιστικές μεθοδολογίες που είναι διαδεδομένες σήμερα
  - Είναι η πρώτη συναρτησιακή γλώσσα που έχει σχεδιαστεί για να υποστηρίζει τη διαλειτουργικότητα με άλλες ευρέως διαδεδομένες γλώσσες
  - Διαθέτει ένα πλήρες IDE και μια βιβλιοθήκη με βοηθητικά προγράμματα

# Υποστήριξη Συναρτησιακού Προγραμματισμού σε Προστακτικές Γλώσσες

---

- Η υποστήριξη για συναρτησιακό προγραμματισμό σταδιακά υιοθετείται και στις προστακτικές γλώσσες
  - Ανώνυμες συναρτήσεις (λάμδα εκφράσεις)
    - JavaScript: το όνομα της συνάρτησης δεν συμπεριλαμβάνεται στον ορισμό της συνάρτησης
    - C#:  $i \Rightarrow (i \% 2) == 0$  (επιστρέφει true ή false ανάλογα με το εάν η παράμετρος είναι άρτια ή περιττή)
    - Python: `lambda a, b : 2 * a - b`

# Υποστήριξη Συναρτησιακού Προγραμματισμού σε Προστακτικές Γλώσσες

(συνέχεια)

---

- Η Python υποστηρίζει τις συναρτήσεις υψηλής τάξης `filter` και `map` (συχνά χρησιμοποιούν εκφράσεις λάμδα ως πρώτη παράμετρο)

```
map(lambda x : x ** 3, [2, 4, 6, 8])
```

Επιστρέφει [8, 64, 216, 512]

- Η Python υποστηρίζει τη μερική αποτίμηση συναρτήσεων

```
from operator import add
```

```
from functools import partial
```

```
add5 = partial (add, 5)
```

(η πρώτη γραμμή κάνει import το add ως συνάρτηση)

Χρήση: `add5(15)`

# Υποστήριξη Συναρτησιακού Προγραμματισμού σε Προστακτικές Γλώσσες (συνέχεια)

---

- Μπλοκς της Ruby

- Πρόκειται για υποπρογράμματα που αποστέλλονται σε μεθόδους, καθιστώντας τα υποπρογράμματα υψηλής τάξης
- Ένα μπλοκ μπορεί να μετατραπεί σε αντικείμενο υποπρογράμματος με `lambda`

```
times = lambda {|a, b| a * b}
```

Χρήση: `x = times.(3, 4)` (θέτει το `x` σε 12)

- Η `times` μπορεί να γίνει `curried` με

```
times5 = times.curry.(5)
```

Χρήση: `x5 = times5.(3)` (θέτει το `x5` σε 15)

# Σύγκριση Συναρτησιακών και Προστακτικών Γλωσσών

---

- Προστακτικές Γλώσσες:
  - Αποδοτική εκτέλεση
  - Πολύπλοκη σημασιολογία
  - Πολύπλοκο συντακτικό
  - Ο ταυτοχρονισμός πρέπει να σχεδιαστεί από τον προγραμματιστή
- Συναρτησιακές Γλώσσες:
  - Απλή σημασιολογία
  - Απλό συντακτικό
  - Λιγότερο αποδοτική εκτέλεση
  - Τα προγράμματα μπορούν αυτόματα να γίνουν ταυτόχρονα

# Σύνοψη

---

- Οι γλώσσες συναρτησιακού προγραμματισμού χρησιμοποιούν εφαρμογή συναρτήσεων, εκφράσεις συνθηκών, αναδρομή, και συναρτησιακές μορφές για να ελέγξουν την εκτέλεση του προγράμματος
- Η Lisp ξεκίνησε ως αμιγώς συναρτησιακή γλώσσα αλλά αργότερα ενσωμάτωσε προστακτικά χαρακτηριστικά
- Η Scheme είναι μια σχετικά απλή διάλεκτος της Lisp που χρησιμοποιεί αποκλειστικά στατική εμβέλεια
- Η Common Lisp είναι μια σύνθετη γλώσσα που βασίζεται στη Lisp
- Η ML είναι μια συναρτησιακή γλώσσα που περιλαμβάνει ένα σύστημα εξαγωγής τύπων, υποστηρίζει στατική εμβέλεια και αυστηρούς κανόνες τύπων
- Η Haskell είναι μια συναρτησιακή γλώσσα σκληρής αποτίμησης που υποστηρίζει άπειρες λίστες και περιφραστικές λίστες
- Η F# είναι μια .NET συναρτησιακή γλώσσα που επιπλέον υποστηρίζει προστακτικό και αντικειμενοστραφή προγραμματισμό
- Ορισμένες κατά βάση προστακτικές γλώσσες πλέον έχουν ενσωματώσει κάποια υποστήριξη για συναρτησιακό προγραμματισμό
- Οι αμιγώς συναρτησιακές γλώσσες προγραμματισμού παρουσιάζουν ορισμένα πλεονεκτήματα σε σχέση με τις προστακτικές γλώσσες, αλλά ακόμα και σήμερα δεν χρησιμοποιούνται ευρέως