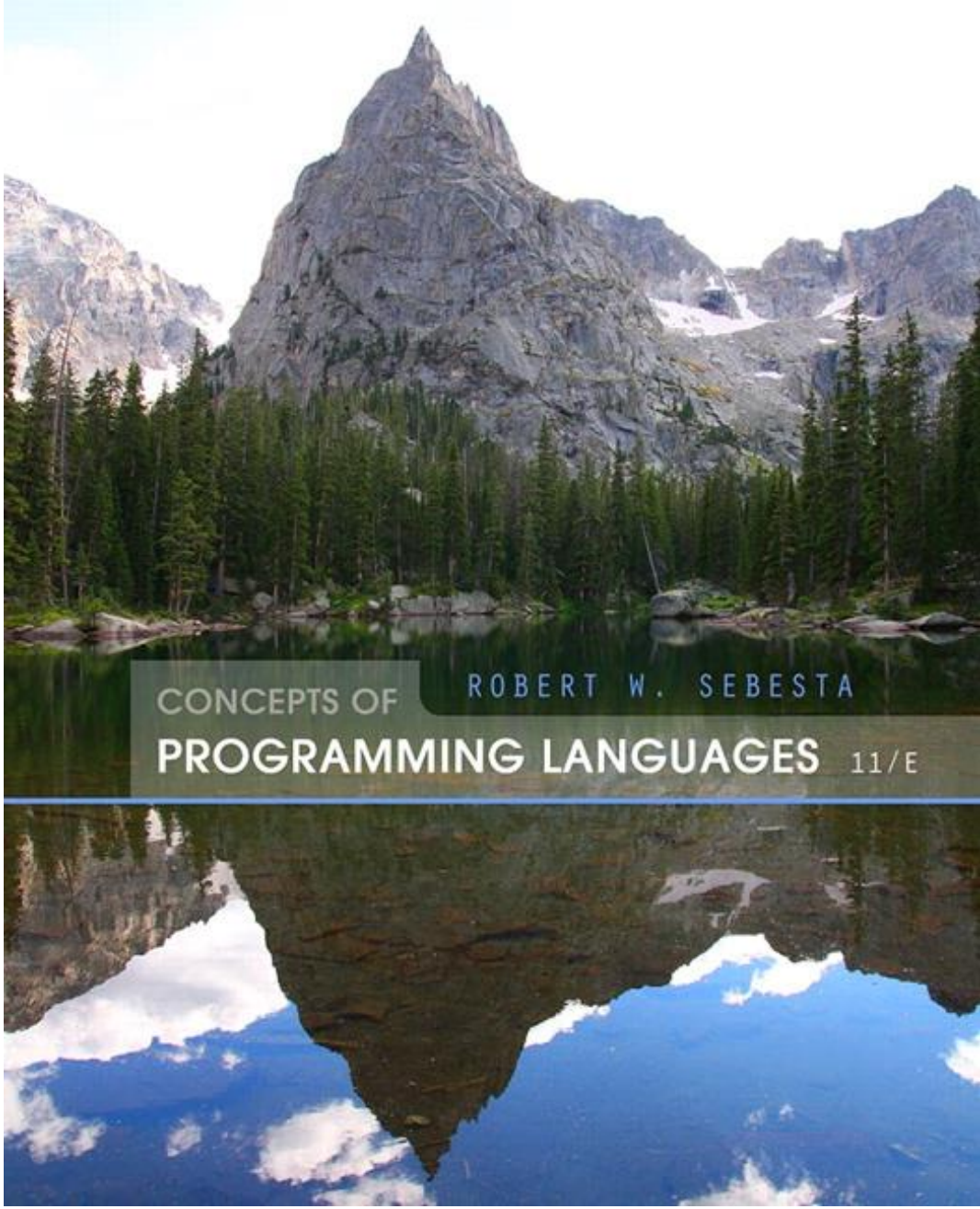
Κεφάλαιο 15

Γλώσσες Συναρτησιακού Προγραμματισμού

Γκόγκος Χρήστος
Τμήμα Πληροφορικής και Τηλεπικοινωνιών (Άρτα)
Πανεπιστήμιο Ιωαννίνων



ISBN 0-321-49362-1

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 ανάπτυξής των προστακτικών γλωσσών, και λιγότερο η καταλληλόλητα 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

  `λ(x) x * x * x`

  για τη συνάρτηση `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., `(λ(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 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 ≡ f ° g`

  που σημαίνει `h (x) ≡ f ( g (x))`

  Για `f (x) ≡ x + 2` και `g (x) ≡ 3 * x`,

  Η `h ≡ f ° g` παράγει `(3 * x)+ 2`

Apply-to-all

- Η 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 2ης παραμέτρου που είναι μια λίστα:

  Μορφή: α

  Για `h(x) ≡ x * x`

  α`(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 προστακτικού προγραμματισμού
  - Σε μια προστακτική γλώσσα τα αποτελέσματα των λειτουργιών αποθηκεύονται σε μεταβλητές για μελλοντική χρήση
  - Η διαχείριση των μεταβλητών είναι ένα συνεχές ζήτημα καθώς και πηγή πολυπλοκότητας στις προστακτικές 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 και απλούστερη διάλεκτος της Lisp
- Χρησιμοποιεί μόνο στατική εμβέλεια
- 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

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 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 κατηγόρημα: EQ?

- EQ? takes two expressions as parameters (usually two atoms); it returns #T if both parameters have the same pointer value; otherwise #F

  (EQ? 'A 'A) yields #T

  (EQ? 'A 'B) yields #F

  (EQ? 'A '(A B)) yields #F

  (EQ? '(A B) '(A B)) yields #T or #F

  (EQ? 3.4 (+ 3 0.4))) yields #T or #F

Predicate Function: `EQV?`

- `EQV?` is like `EQ?`, except that it works for both symbolic and numeric atoms; it is a value comparison, not a pointer comparison

  `(EQV? 3 3)` yields `#T`

  `(EQV? 'A 3)` yields `#F`

  `(EQV 3.4 (+ 3 0.4))` yields `#T`

  `(EQV? 3.0 3)` yields `#F` (floats and integers are different)

Predicate Functions: LIST? and NULL?

- LIST? takes one parameter; it returns #T if the parameter is a list; otherwise #F

  (LIST? '()) yields #T
- NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise #F

  (NULL? '(())) yields #F

Example Scheme Function: member

- member takes an atom and a simple list; returns #T if the atom is in the list; #F otherwise

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

Example Scheme Function: `equalsimp`

- `equalsimp` takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `#F` otherwise

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

Example Scheme Function: `equal`

- `equal` takes two general lists as parameters; returns `#T` if the two lists are equal; `#F` otherwise

```
(DEFINE (equal list1 list2)
  (COND
    ((NOT (LIST? list1))(EQ? list1 list2))
    ((NOT (LIST? lis2)) #F)
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((equal (CAR list1) (CAR list2))
         (equal (CDR list1) (CDR list2)))
    (ELSE #F)
))
```

Example Scheme Function: `append`

- `append` takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

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

Example Scheme Function: LET

- Recall that LET was discussed in Chapter 5
- LET is actually shorthand for a LAMBDA expression applied to a parameter

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

is the same as:

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

LET Example

```
(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))
))
```

Tail Recursion in Scheme

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language definition requires that Scheme language systems convert all tail recursive functions to use iteration

Tail Recursion in Scheme – continued

- Example of rewriting a function to make it tail recursive, using helper a function

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))
```

Functional Form – Composition

- Composition
  - If `h` is the composition of `f` and `g`, `h(x) = f(g(x))`

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
(DEFINE h x) (+ 2 (* 3 x))) (The composition)
```

  - In Scheme, the functional composition function `compose` can be written:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
((compose CAR CDR) '((a b) c d)) yields c
 (DEFINE (third a_list)
    ((compose CAR (compose CDR CDR)) a_list))
```

 is equivalent to `CADDR`

Functional Form – Apply-to-All

- Apply to All – one form in Scheme is `map`
  - Applies the given function to all elements of the given list;

```
(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))` yields
`(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 διερμηνευτής είναι μια διαθέσιμη προς τον προγραμματιστή συνάρτηση, η `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 του `+` να προηγείται το ' έτσι ώστε να αποφευχθεί η αποτίμηση
  - Υποβάλει 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

- A static-scoped functional language with syntax that is closer to Pascal than to Lisp
- Uses type declarations, but also does *type inferencing* to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Does not have imperative-style variables
- Its identifiers are untyped names for values
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations

ML Specifics

- A table called the *evaluation environment* stores the names of all identifiers in a program, along with their types (like a run-time symbol table)
- Function declaration form:

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

  e.g., `fun cube(x : int) = x * x * x;`

  - The type could be attached to return value, as in

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

  - With no type specified, it would default to `int` (the default for numeric values)
  - User-defined overloaded functions are not allowed, so if we wanted a `cube` function for real parameters, it would need to have a different name

ML Specifics (continued)

- ML selection

  `if` *expression* **`then`** *then_expression*

  `else` *else_expression*

  where the first expression must evaluate to a Boolean value

- Pattern matching is used to allow a function to operate on different parameter forms

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

ML Specifics (continued)

- Lists
  Literal lists are specified in brackets
  `[3, 5, 7]`
  `[]` is the empty list
  `CONS` is the binary infix operator, `::`
  `4 :: [3, 5, 7]`, which evaluates to `[4, 3, 5, 7]`
  `CAR` is the unary operator `hd`
  `CDR` is the unary operator `tl`

```
fun length([]) = 0
|   length(h :: t) = 1 + length(t);

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

ML Specifics (continued)

- The **val** statement binds a name to a value (similar to DEFINE in Scheme)

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

  - As is the case with DEFINE, **val** is nothing like an assignment statement in an imperative language
  - If there are two **val** statements for the same identifier, the first is hidden by the second
  - **val** statements are often used in **let** constructs

```
let
   val radius = 2.7
   val pi = 3.14159
in
   pi * radius * radius
end;
```

ML Specifics (continued)

- `filter`
  - A higher-order filtering function for lists
  - Takes a predicate function as its parameter, often in the form of a lambda expression
  - Lambda expressions are defined like functions, except with the reserved word **`fn`**

```
filter(fn(x) => x < 100, [25, 1, 711, 50, 100]);
```

This returns `[25, 1, 50]`

ML Specifics (continued)

- `map`
  - A higher-order function that takes a single parameter, a function
  - Applies the parameter function to each element of a list and returns a list of results

```
fun cube x = x * x * x;
val cubeList = map cube;
val newList = cubeList [1, 3, 5];
```

This sets `newList` to `[1, 27, 125]`

  - Alternative: use a lambda expression

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

ML Specifics (continued)

- Function Composition
  - Use the unary operator, ∘

```
val h = g ∘ f;
```

ML Specifics (continued)

- Currying
  - ML functions actually take just one parameter—if more are given, it considers the parameters a tuple (commas required)
  - Process of *currying* replaces a function with more than one parameter with a function with one parameter that returns a function that takes the other parameters of the original function
  - An ML function that takes more than one parameter can be defined in curried form by leaving out the commas in the parameters

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

A function with one parameter, `a`. Returns a function that takes `b` as a parameter. Call: `add 3 5;`

ML Specifics (continued)

- Partial Evaluation
  - Curried functions can be used to create new functions by partial evaluation
  - Partial evaluation means that the function is evaluated with actual parameters for one or more of the leftmost actual parameters

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

Takes the actual parameter `5` and evaluates the `add` function with `5` as the value of its first formal parameter. Returns a function that adds `5` to its single parameter

```
val num = add5 10;   (* sets num to 15 *)
```

Haskell

- Ομοιότητες με την ML (συντακτικό, στατική εμβέλεια, με ισχυρούς κανόνες τύπων, συμπερασμό τύπων, αντιστοίχιση μοτίβων)
- Διαφορές από την ML (και από τις περισσότερες συναρτησιακές γλώσσες) καθώς είναι αμιγώς συναρτησιακή (π.χ., 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 + 2) = fib (n + 1) + fib n
```

Ορισμοί συναρτήσεων με διαφορετικά διαστήματα παραμέτρων

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


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


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: `#`

  π.χ., `#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)

- Μοτίβα παραμέτρων

```
product [] = 1
product (a:x) = a * product x
```

  - Παραγοντικό:

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

- List Comprehensions

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

Τα list comprehension 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 – *άπειρες λίστες*
- Οκνηρή αποτίμηση – Υπολογισμός μόνο των τιμών που είναι απαραίτητες

  `positives = [0..]`
- 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
```

- Ωστόσο, 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 unions, 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 γίνουν 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 evensTimesFive = 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 >> 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 εισέρχεται και στις προστακτικές γλώσσες
  - Ανώνυμες συναρτήσεις (λάμδα εκφράσεις)
    - JavaScript: το όνομα της συνάρτησης δεν συμπεριλαμβάνεται στον ορισμό της συνάρτησης
    - C#: `i => (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 οκνηρής αποτίμησης που υποστηρίζει άπειρες λίστες και list comprehensions
- 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