

# Εισαγωγή στη Γλώσσα ML



Juan Miró

Κωστής Σαγώνας <[kostis@cs.ntua.gr](mailto:kostis@cs.ntua.gr)>  
Ζωή Παρασκευοπούλου <[zoepar@softlab.ntua.gr](mailto:zoepar@softlab.ntua.gr)>

# Συναρτησιακός Προγραμματισμός

---

- Υπολογισμός μέσω ορισμού και εφαρμογών συναρτήσεων  
Π.χ.:
  - ορισμός:  
$$f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2)$$
  - εφαρμογή:  
$$f(8) \text{ -- αποτίμηση --> } 21$$
- Βασίζεται στο μαθηματικό μοντέλο του λ-λογισμού (Church)
- Κάποια προγράμματα μπορούν να γραφούν με πολύ κομψό, σύντομο και σαφή τρόπο
- Αποφυγή συγκεκριμένων κατηγοριών προγραμματιστικών σφαλμάτων

# Συναρτησιακός vs. Προστακτικός Προγραμματισμός

- Προστακτικός προγραμματισμός
  - Έμφαση στο πώς θα υπολογιστεί κάτι
  - Ακολουθίες εντολών που μεταβάλουν την κατάσταση (state) του προγράμματος
- Συναρτησιακός προγραμματισμός
  - Έμφαση στο τι θα υπολογιστεί
  - Εκφράσεις (expressions) που αποτιμούνται σε τιμές (values)
  - Υπολογισμός μέσω εφαρμογών συναρτήσεων

```
int f(int n) {  
  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
  
    int prev2 = 0; // f(0)  
    int prev1 = 1; // f(1)  
    int current;  
  
    for (int i = 2; i <= n; i++) {  
        current = prev1 + prev2;  
        prev2 = prev1;  
        prev1 = current;  
    }  
    return current;  
}
```

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2)$$

# Διαφάνεια αναφοράς (referential transparency)

---

- Σε μία *αμιγώς συναρτησιακή* γλώσσα προγραμματισμού, **η αποτίμηση μιας συνάρτησης δίνει πάντα το ίδιο αποτέλεσμα για τις ίδιες τιμές των παραμέτρων της**
- Η σημαντική αυτή ιδιότητα δεν ισχύει κατ' ανάγκη στις γλώσσες προστακτικού προγραμματισμού
- Στον προστακτικό προγραμματισμό αυτό συμβαίνει λόγω:
  - Μεταβλητών που ορίζονται και αλλάζουν τιμές εκτός του σώματος της συνάρτησης (global variables)
  - Εξάρτησης από την κατάσταση (state) του υπολογισμού
  - Άλλων παρενεργειών (side-effects) που μπορεί να υπάρχουν στο πρόγραμμα



# Παράδειγμα σε C

```
bool flag; // global state

int f(int n) {
    int result;

    if (flag) {
        result = n;
    } else {
        result = 2 * n;
    }

    flag = !flag;
    return result;
}

int main(void) {

    flag = true;
    printf("%d\n", f(1));
    printf("%d\n", f(1));
    return 0;
}
```

Τι τυπώνει το πρόγραμμα;

1 και μετά 2

- Περύεργο διότι η f είναι συνάρτηση!
- Στα μαθηματικά, οι συναρτήσεις εξαρτώνται μόνο από τα ορίσματά τους

# Μεταβλητές και “μεταβλητές”

---

- Στην καρδιά του προβλήματος είναι το γεγονός ότι η μεταβλητή **flag** επηρεάζει την τιμή της **f**
- Ειδικότερα, η συμπεριφορά οφείλεται στην ανάθεση  
$$\text{flag} = \text{!flag};$$
- Σε μια γλώσσα χωρίς πολλαπλές αναθέσεις μεταβλητών δεν υπάρχουν τέτοια προβλήματα!
- Στις συναρτησιακές γλώσσες, οι μεταβλητές είναι ονόματα για συγκεκριμένες τιμές, δεν είναι ονόματα για συγκεκριμένες θέσεις μνήμης
- Μπορούμε να τις θεωρήσουμε «όχι πολύ μεταβλητές»

# Η γλώσσα ML (Meta Language)

---

- Γλώσσα συναρτησιακού προγραμματισμού γενικής χρήσης με *στατικό σύστημα σύστημα τύπων*
- Υποστηρίζει διαδραστική χρήση (REPL) και μεταγλώττιση σε εκτελέσιμα αρχεία
- Συνδυάζει τα παρακάτω στοιχεία:
  - Βασισμένη στο λ-λογισμό και στην αποτίμηση εκφράσεων
  - Συναρτήσεις υψηλής τάξης (higher-order functions)
  - Αυτόματη διαχείριση μνήμης (με χρήση συλλογής σκουπιδιών)
  - Αφηρημένους τύπους δεδομένων (abstract data types)
  - Σύστημα αρθρωμάτων (module system)
  - Εξαιρέσεις (exceptions)
- Σχετικές γλώσσες: OCaml, F#, Haskell, Scala, Rust, ...

# Γιατί εξετάζουμε την ML;

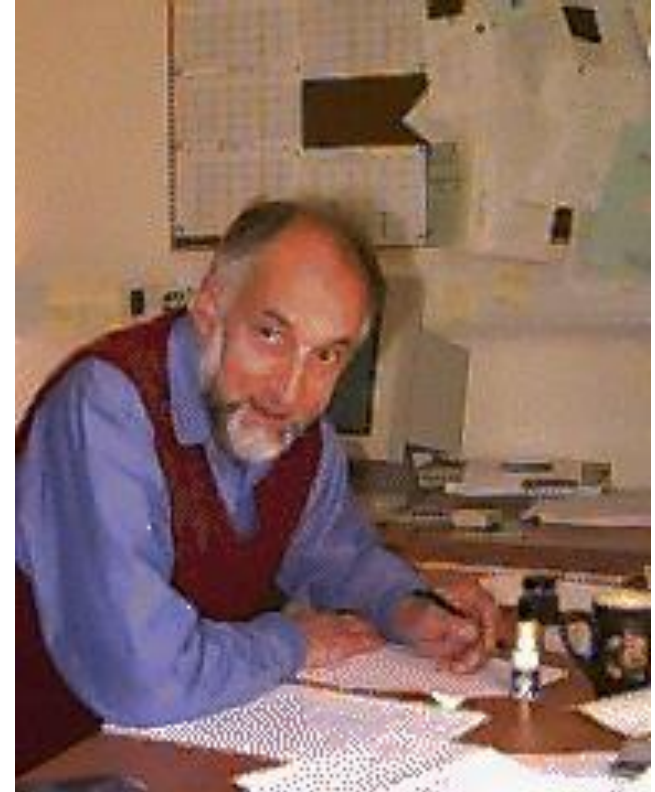
---

- Αυστηρό, στατικό σύστημα τύπων
  - Στατική ασφάλεια τύπων: *“Well-typed programs can’t go wrong”*
  - Συμπερασμός τύπων (type inference)
  - Πολυμορφισμός και γενικός προγραμματισμός (generic programming)
- Δομές ελέγχου ροής
  - Εξαιρέσεις
  - Αναδρομή “ουράς” (tail recursion) και συνέχειες (continuations)
- Θέματα σχεδιασμού και υλοποίησης
  - Στατική εμβέλεια και δομή κατά μπλοκ
  - Εγγραφές ενεργοποίησης συναρτήσεων (function activation records) και υλοποίηση συναρτήσεων υψηλής τάξης



# Σύντομη ιστορία της γλώσσας ML

- Robin Milner (ACM Turing Award)
- Logic for Computable Functions
  - interactive theorem prover
  - μηχανική απόδειξη θεωρημάτων, αποδείξεις ελέγχονται από τον υπολογιστή
- Μεταγλώσσα του συστήματος LCF
  - Η γλώσσα στην οποία γράφονται οι αποδείξεις
  - Αυστηρό σύστημα τύπων + αφηρημένοι τύποι δεδομένων = **μόνο** έγκυρα θεωρήματα
- Θα χρησιμοποιήσουμε την υλοποίηση SML/NJ (Standard ML of New Jersey)



# Βασικοί τύποι της ML

---

- Booleans και τελεστές τους
  - `true, false : bool`
  - `andalso, orelse, not`
- Ακέραιοι και τελεστές τους
  - `0, 1, 2, ... : int`
  - `+, -, *, mod, div, ~` (μοναδιαίο μείον)
- Συμβολοσειρές και τελεστές τους
  - `"Robin Milner" : string`
  - `^` (συνένωση συμβολοσειρών)
- Αριθμοί κινητής υποδιαστολής και τελεστές τους
  - `1.0, 2.56, 3.14159, ...`
  - `+, -, *, /, ~`

Οι τελεστές είναι αριστερά προσηταιριστικοί, με προτεραιότητες  $\{+,-\} < \{*,/,div,mod\} < \{~\}$ .

# Η γλώσσα ML μέσα από παραδείγματα

```
% sml
```

```
Standard ML of New Jersey, v110.XX
```

```
- 42;
val it = 42 : int
- 2 + 3;
val it = 5 : int
```

Εκφράσεις

↓ αποτίμηση

Τιμές

- «Δέσιμο» τιμής με όνομα:

```
- val x = 20 + 22;
val x = 42 : int
- val y = x + 1
val y = 5 : int
```

# ML και τύποι

---

- Η ML ελέγχει **στατικά** (compile time) αν μια έκφραση έχει τον σωστό τύπο
- $e : t$  σημαίνει «η έκφραση  $e$  έχει τύπο  $t$ »
- Με αυτό τον τρόπο, αποτρέπει στατικά κάποιες κατηγορίες προγραμματιστικών σφαλμάτων
  - “Well-typed programs can’t go wrong” (Milner, 1978)
- Π.χ.
  - `40 + 2` (well-typed)
  - `40 + "2"` (ill-typed)
  - `if (even 42) then 11 else "hello ML!"` (ill-typed)
- Η ML μας επιτρέπει να παραλείψουμε τους τύπους στους ορισμούς μας. **Αυτό δεν σημαίνει ότι δεν υπάρχουν.**
- Τους καταλαβαίνει μόνη της! (συμπερασμός τύπων)

# Η γλώσσα ML μέσα από παραδείγματα

---

```
- 1 = 2;
```

```
val it = false : bool
```

```
- 1 <> 2 andalso true <> false;
```

```
val it = true : bool
```

```
- true = false orelse 1 <= 2;
```

```
val it = true : bool
```

```
- "Robin" > "Milner";
```

```
val it = true : bool
```

```
- 2.56 < 3.14;
```

```
val it = true : bool
```

```
- 2.56 = 3.14;
```

```
stdIn: Error: operator and operand don't agree
```

```
operator domain: 'Z * 'Z
```

```
operand:          real * real
```

# Υπερφόρτωση τελεστών (operator overloading)

```
- 6 * 7
val it = 42 : int
- 6.0 * 7.0;
val it = 42.0 : real
- 2.0 * 21;
stdIn: Error: operator and operand don't agree
  operator domain: real * real
  operand:          real * int
in expression:     2.0 * 21
```

- Ο τελεστής \* (και άλλοι όπως ο +) είναι **υπερφορτωμένοι**
- Έχουν διαφορετική ερμηνεία σε ζεύγη ακεραίων και διαφορετική σε ζεύγη αριθμών κινητής υποδιαστολής
- Η ML δεν κάνει αυτόματη μετατροπή από ακεραίους σε πραγματικούς αριθμούς (όπως π.χ. κάνει η C)

# Πλειάδες (tuples)

- Σύνθετος τύπος δεδομένων  $a_1 * \dots * a_n$
- Ακολουθία  $n$  αντικειμένων  $(x_1, \dots, x_n)$  με τύπους  $x_1 : a_1, \dots, x_n : a_n$

```
- (42, 11);
```

```
val it = (42, 11) : int * int
```

```
- val z = ("42", 11, true);
```

```
val z = ("42", 11, true) : string * int * bool
```

- Προσπέλαση στοιχείων

```
- val z2 = #2 z;
```

```
val it = 11 : int
```

```
- val z1 = #3 z;
```

```
val it = true : bool
```

```
- val (z1, z2, z3) = z;
```

```
val z1 = "42" : string
```

```
val z2 = 11 : int
```

```
val z3 = true : bool
```

(Συνήθως το προτιμάμε)

# Συναρτήσεις

---

- Ένας τύπος δεδομένων όπως όλοι οι άλλοι!
- Μια συνάρτηση που παίρνει ένα όρισμα τύπου **a** και επιστρέφει ένα αποτέλεσμα τύπου **b** έχει τύπο **a -> b**
- Ορισμός συνάρτησης

```
- fun square x = x * x;  
val square = fn : int -> int
```

- Κλήση συνάρτησης

```
- square 5;  
val it = 25 : int
```



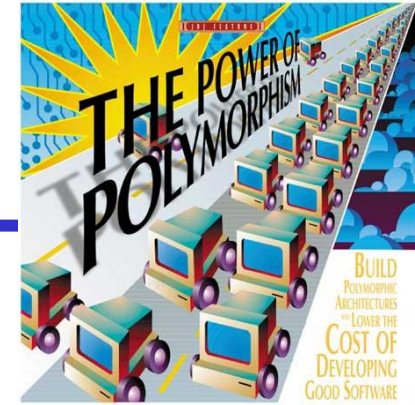
# Πλειάδες και συναρτήσεις

---

- Μπορούμε να χρησιμοποιήσουμε πλειάδες (tuples) ως ορίσματα ή αποτελέσματα συναρτήσεων

```
- fun max (a,b) = if a > b then a else b;
val max = fn : int * int -> int
- max (17,42);
val it = 42 : int
- fun reverse (x,y) = (y,x);
val reverse = fn : 'a * 'b -> 'b * 'a
- reverse (17,42);
val it = (42,17) : int * int
- max (reverse (17,42));
val it = 42 : int
```

# Πολυμορφισμός



- Η συνάρτηση `reverse` έχει έναν ενδιαφέροντα **τύπο**

```
- fun reverse (x,y) = (y,x) ;  
val reverse = fn : 'a * 'b -> 'b * 'a
```

- Αυτό σημαίνει ότι μπορούμε να αντιστρέψουμε μια δυάδα με στοιχεία οποιουδήποτε τύπου

```
- reverse (42,3.14) ;  
val it = (3.14,42) : real * int  
- reverse ("foo", (1,2)) ;  
val it = ((1,2), "foo") : (int * int) * string
```

# Currying

```
- fun max a b =  
=   if a > b then a else b;  
val max = fn : int -> int -> int  
- max 17 5;  
val it = 17 : int  
- max 10 42;  
val it = 42 : int
```



Haskell B. Curry

-> δεξιά προσεταιριστικό

- Προσέξτε τον τύπο

`int -> int -> int`

- Λέει ότι η `max` είναι μια συνάρτηση που παίρνει έναν ακέραιο και επιστρέφει μια συνάρτηση που παίρνει έναν ακέραιο και επιστρέφει έναν ακέραιο



## Μερική εφαρμογή (partial application)

---

- Μια συνάρτηση που είναι curried μπορεί να εφαρμοστεί μερικώς. Το αποτέλεσμα της μερικής εφαρμογής είναι μια άλλη συνάρτηση.

```
- fun max a b = if a > b then a else b;  
val max = fn : int -> int -> int  
- val max_five = max 5;  
val max_five = fn : int -> int  
- max_five 42;  
val it = 42 : int  
- max_five 3;  
val it = 5 : int
```

# Συναρτήσεις πρώτης τάξης

---

- Στην ML, οι συναρτήσεις είναι αντικείμενα πρώτης τάξης τα οποία μπορούμε να τα διαχειριστούμε όπως όλα τα άλλα αντικείμενα (π.χ. τους ακεραίους)

```
- fun square x = x * x;  
val square = fn : int -> int  
- square;  
val it = fn : int -> int
```

- Μπορούμε να τις περάσουμε ως παραμέτρους ή να τις επιστρέψουμε ως ορίσματα!

# Συναρτήσεις πρώτης τάξης (παράδειγμα)

---

- Η συνάρτηση `app` παίρνει ως πρώτο όρισμα μια συνάρτηση και την εφαρμόζει στο δεύτερο όρισμα

```
- fun compose f g x = f (g x);  
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

- Παράδειγμα χρήσης:

```
- fun add x y = x + y;  
val add = fn : int -> int  
- val add15 = compose (add 10) (add 5)  
val add5 = fn : int -> int  
- add15 6;  
val it = 21 : int
```

# Αναδρομή

- Επειδή δεν υπάρχουν μεταβλητές με την παραδοσιακή έννοια, τα προγράμματα χρησιμοποιούν **αναδρομή** για να εκφράσουν επανάληψη

```
- fun sum n =  
=   if n = 0 then 0 else sum (n-1) + n;  
val sum = fn : int -> int  
- sum 2;  
val it = 3 : int  
- sum 3;  
val it = 6 : int  
- sum 4;  
val it = 10 : int
```

## Αναδρομή

- Επειδή δεν υπάρχουν μεταβλητές με την παραδοσιακή έννοια, τα προγράμματα χρησιμοποιούν **αναδρομή** για να εκφράσουν επανάληψη

```
- fun sum n =  
=   if n = 0 then 0 else sum (n-1) + n;  
val sum = fn : int -> int  
- sum 2;  
val it = 3 : int  
- sum 3;  
val it = 6 : int  
- sum 4;  
val it = 10 : int
```

### Αναδρομή

Επειδή δεν υπάρχουν μεταβλητές με την παραδοσιακή έννοια, τα προγράμματα χρησιμοποιούν **αναδρομή** για να εκφράσουν επανάληψη

```
- fun sum n =  
=   if n = 0 then 0 else sum (n-1) + n;  
val sum = fn : int -> int  
- sum 2;  
val it = 3 : int  
- sum 3;  
val it = 6 : int  
- sum 4;  
val it = 10 : int
```

Εισαγωγή στη γλώσσα ML

11



## Τελεστής ύψωσης σε δύναμη

- Μπορούμε επίσης να ορίσουμε νέους αριθμητικούς τελεστές ως συναρτήσεις

```
- fun x ^ y =  
=   if y = 0 then 1 else x * (x ^ (y-1));  
val ^ = fn : int * int -> int  
- 2 ^ 2;  
val it = 4 : int  
- 2 ^ 3;  
val it = 8 : int  
- 2 ^ 4;  
val it = 16 : int
```



## Επαναχρησιμοποίηση αποτελεσμάτων

- Αν δεν έχουμε μεταβλητές, είμαστε αναγκασμένοι να επαναλάβουμε εκφράσεις (και υπολογισμούς)

```
fun f x =  
  g(square(max(x, 4))) +  
  (if x < 1 then 1  
   else g(square(max(x, 4))))
```

- Μια μέθοδος για να γράψουμε πιο εύκολα την παραπάνω συνάρτηση είναι με χρήση μιας βοηθητικής συνάρτησης

```
fun f1(a,b) = b + (if a < 1 then 1 else b)  
fun f x = f1(x, g(square(max(x, 4))))
```

## Η έκφραση let

---

- Ένας πιο εύκολος τρόπος είναι ο ορισμός ενός τοπικού ονόματος για την επαναχρησιμοποιούμενη έκφραση

```
fun f x =  
  let  
    val gg = g(square(max(x,4)))  
  in  
    gg + (if x < 1 then 1 else gg)  
  end
```

Δήλωση  
ονόματος

Εμβέλεια

## Η έκφραση let

---

- Ένας πιο εύκολος τρόπος είναι ο ορισμός ενός τοπικού ονόματος για την επαναχρησιμοποιούμενη έκφραση

```
fun f x =  
  let  
    val y = square x  
    val z = max(y, 42)  
  in  
    y + z + (if x < 1 then y else z)  
  end
```

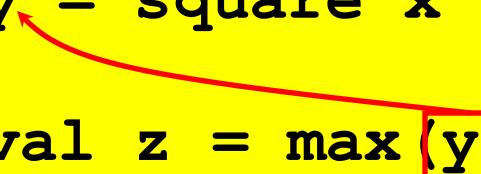
Δήλωση y

Εμβέλεια

# Εμφωλευμένα let

---

```
fun f x =  
  let  
    val y = square x  
  in  
    let val z = max(y, 42)  
    in  
      y + z + (if x < 1 then y else z)  
    end  
  end  
end
```



# Η έκφραση let δεν είναι ανάθεση

```
- let
=   val a = 2
=   in
=     (let
=       val a = a + 2
=       in
=         a
=       end,
=       a)
= end;
val it = (4,2) : int * int
```

Οι εκφράσεις let  
μπορούν να είναι  
φωλιασμένες

# Σύνθετοι τύποι δεδομένων στην ML

---

- Προγράμματα που επεξεργάζονται μόνο βαθμωτά δεδομένα (scalars - χωρίς δομή) δεν είναι πολύ χρήσιμα
- Οι συναρτησιακές γλώσσες είναι ιδανικές για την διαχείριση σύνθετων τύπων δεδομένων
- Έχουμε ήδη δει **πλειάδες**, που είναι σύνθετοι τύποι δεδομένων για την αναπαράσταση ενός ορισμένου αριθμού αντικειμένων (πιθανώς διαφορετικών τύπων)
- Η ML έχει επίσης **λίστες** που είναι ακολουθίες αντικειμένων μεταβλητού μήκους του ίδιου όμως τύπου

# Λίστες

---

- Τύπος: `t list` όπου `t` οποιοσδήποτε τύπος

```
- [1, 2, 3, 42, 11, 2];
```

```
val it = [1, 2, 3, 42, 11, 2] : int list
```

```
- [false, true, false];
```

```
val it = [false, true, false] : bool list
```

- Οι λίστες περιέχουν στοιχεία **ίδιου τύπου**

```
- [1, true, 42];
```

```
stdin:1.2-8.3 Error: operator and operand  
do not agree [overload - bad  
instantiation]
```

# Κατασκευή λιστών

---

- Μπορούμε να κατασκευάσουμε μια κενή λίστα

```
- [];  
val it = [] : 'a list
```

- Για κάθε λίστα τύπου `list t` μπορούμε να προσθέσουμε ένα στοιχείο τύπου `t` στην αρχή της με τον τελεστή `::` (προφέρεται `cons`)

```
- val x = 1::[];  
val x = [1] : int list  
- val y = 2::x;  
val y = [2,1] : int list  
- 0::1::2::3::[];  
val it = [0,1,2,3] : int list
```



## Συνένωση λιστών

---

- Τελεστής συνένωσης @

```
- [1,2] @ [3,4];
```

```
val it = [1,2,3,4] : int list
```

```
- ["ML", "is"] @ ["fun"]
```

```
val it = ["ML","is","fun"] : string list
```

- Η συνένωση δύο λιστών δεν είναι το ίδιο με το cons

```
- [1,2] :: [3,4];
```

```
stdin: Error: operator and operand don't agree
```

```
operator domain: int list * int list list
```

```
operand:          int list * int list
```

```
in expression:
```

```
(1 :: 2 :: nil) :: 3 :: 4 :: nil
```

# Άλλες συναρτήσεις για λίστες

---

```
- null [];  
val it = true : bool  
- null [1,2];  
val it = false : bool  
- val l = [1,2,3,4];  
val l = [1,2,3,4] : int list  
- hd l;  
val it = 1 : int  
- tl l;  
val it = [2,3,4] : int list  
- length l;  
val it = 4 : int  
- nil;  
val it = [] : 'a list
```

# Ορισμός συναρτήσεων για λίστες

---

- Πρόσθεση μιας τιμής σε όλα τα στοιχεία μιας λίστας

```
- fun addto (l,v) =  
=   if null l then nil  
=   else hd l + v :: addto (tl l,v);  
val addto = fn : int list * int -> int list
```

```
- addto ([1,2,3],2);  
val it = [3,4,5] : int list  
- addto ([1,2,3],~2);  
val it = [~1,0,1] : int list
```

# Συναρτήσεις υψηλής τάξης για λίστες

---

- Εφαρμογή μιας οποιασδήποτε συνάρτησης σε όλα τα στοιχεία μιας λίστας

```
- fun map f l =  
=   if null l then nil  
=   else f (hd l) :: map f (tl l);  
val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

- Χρησιμοποιώντας την `map` μπορούμε να εκφράσουμε μετασχηματισμούς λιστών

## Συναρτήσεις υψηλής τάξης για λίστες

---

```
- fun add2 x = x + 2;  
val add2 = fn : int -> int  
- map add2 [10,11,12];  
val it = [12,13,14] : int list
```

```
- fun mul2 x = x * 2;  
val mul2 = fn : int -> int  
- map mul2 [10,11,12];  
val it = [20,22,24] : int list
```

```
- fun even x = x mod 2 = 0;  
val even = fn : int -> bool  
- map even [1,2,3,4,5,6];  
val it = [false,true,false,true,false,true]  
: bool list
```

# Ανώνυμες συναρτήσεις (λ-εκφράσεις)

- Μια λάμδα έκφραση είναι μια συνάρτηση χωρίς όνομα

```
- val add2 = fn x => x + 2;  
val add2 = fn : int -> int  
- add2 10;  
val it = 12 : int  
- fun add2' x = x + 2;  
val add2 = fn : int -> int  
- add2' 10;  
val it = 12 : int
```



- Οι ορισμοί `add2` και `add2'` είναι ισοδύναμοι

## Ανώνυμες συναρτήσεις (λ-εκφράσεις)

---

- Χρήσιμες για συναρτήσεις που χρησιμοποιούμε μια φορά

```
- map (fun x => x + 2) [10,11,12];  
val it = [12,13,14] : int list
```

```
- map (fun x => x mod 2 = 0) [1,2,3,4,5,6];  
val it = [false,true,false,true,false,true]  
: bool list
```

# Αναδρομικές λάμδα εκφράσεις

---

- Πώς καλούμε αναδρομικά κάτι το οποίο δεν έχει όνομα;
- Του δίνουμε ένα!

```
- let
=   val rec f =
=     fn x => if null x then nil
=           else (hd x + 3) :: f (tl x)
= in
=   f
= end
= [1,2,3,4];
val it = [4,5,6,7] : int list
```



## Ταίριασμα προτύπων (pattern matching)

- Στα μαθηματικά, οι συναρτήσεις πολλές φορές ορίζονται με διαφορετικές εκφράσεις βάσει κάποιων συνθηκών

$$f(x) = \begin{cases} x & \text{εάν } x \geq 0 \\ -x & \text{εάν } x < 0 \end{cases}$$

- Οι συναρτήσεις της ML δε διαφέρουν και επιτρέπουν τον ορισμό κατά περιπτώσεις και την αποφυγή της χρήσης `if`

```
fun map f [] = []  
  | map f l  = f (hd l) :: map f (tl l)
```

- Όμως, ο ορισμός ανά περιπτώσεις είναι ευαίσθητος ως προς τη σειρά εμφάνισης των συναρτησιακών προτάσεων

```
fun map f l  = f (hd l) :: map f (tl l)  
  | map f [] = []
```



## Καλύτερος ορισμός μέσω ταιριάσματος προτύπων

- Το πρότυπο `_` ταιριάζει με όλα τα αντικείμενα
- Το πρότυπο `h :: t` ταιριάζει με μια λίστα και δένει
  - τη μεταβλητή `h` με την κεφαλή της λίστας και
  - τη μεταβλητή `t` με την ουρά της λίστας

```
fun map _ [] = []  
  | map f (h::t) = f h :: map f t
```



## Χρήση σταθερών ως πρότυπα

---

```
- fun is_zero 0 = "yes";  
stdIn: Warning: match nonexhaustive  
      0 => ...  
val is_zero = fn : int -> string  
- is_zero 0;  
val it = "yes" : string
```

- Κάθε σταθερά ενός τύπου που υποστηρίζει ισότητα μπορεί να χρησιμοποιηθεί ως πρότυπο
- Αλλά δεν μπορούμε να γράψουμε

```
fun is_zero 0.0 = "yes";
```

## Μη εξαντλητικό ταίριασμα προτύπων

---

- Στο προηγούμενο παράδειγμα, ο τύπος της `is_zero` ήταν `int -> string`, αλλά ταυτόχρονα υπήρξε η προειδοποίηση “**Warning: match nonexhaustive**”
- Αυτό σημαίνει ότι η συνάρτηση ορίστηκε με πρότυπα που δεν εξάντλησαν το πεδίο ορισμού της συνάρτησης
- Κατά συνέπεια, είναι δυνατό να υπάρχουν προβλήματα χρόνου εκτέλεσης, όπως:

```
- is_zero 42;  
uncaught exception Match: [nonexhaustive  
                             match failure]  
raised at ...
```

# Κανόνες ταιριάσματος προτύπων στην ML

---

- Το πρότυπο `_` ταιριάζει με οτιδήποτε
- Μια μεταβλητή είναι ένα πρότυπο που ταιριάζει με οποιαδήποτε τιμή και δένει τη μεταβλητή με την τιμή
- Μια σταθερά (ενός τύπου ισότητας) είναι ένα πρότυπο που ταιριάζει μόνο με τη συγκεκριμένη σταθερά
- Μια πλειάδα  $(\mathbf{x}, \mathbf{y}, \dots, \mathbf{z})$  είναι ένα πρότυπο που ταιριάζει με κάθε πλειάδα του ίδιου μεγέθους, της οποίας τα περιεχόμενα ταιριάζουν με τη σειρά τους με τα  $\mathbf{x}, \mathbf{y}, \dots, \mathbf{z}$
- Μια λίστα  $[\mathbf{x}, \mathbf{y}, \dots, \mathbf{z}]$  είναι ένα πρότυπο που ταιριάζει με κάθε λίστα του ίδιου μήκους, της οποίας τα στοιχεία ταιριάζουν με τη σειρά τους με τα  $\mathbf{x}, \mathbf{y}, \dots, \mathbf{z}$
- Ένα `cons h :: t` είναι ένα πρότυπο που ταιριάζει με κάθε μη κενή λίστα, της οποίας η κεφαλή ταιριάζει με το `h` και η ουρά με το `t`

## Παράδειγμα χρήσης ταιριάσματος προτύπων

---

- Παραγοντικό με χρήση `if-then-else`

```
fun fact n =  
  if n = 0 then 1 else n * fact (n-1)
```

- Παραγοντικό με χρήση ταιριάσματος προτύπων

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

- Παρατηρήστε ότι υπάρχει επικάλυψη στα πρότυπα
- Η εκτέλεση δοκιμάζει πρότυπα με τη σειρά που αυτά εμφανίζονται (από πάνω προς τα κάτω)

## Άλλα παραδείγματα

---

- Η παρακάτω δομή είναι πολύ συνηθισμένη σε αναδρομικές συναρτήσεις που επεξεργάζονται λίστες: μία περίπτωση για την κενή λίστα (`nil`) και (τουλάχιστον) μία περίπτωση για όταν η λίστα δεν είναι κενή (`h :: t`).
- Άθροισμα όλων των στοιχείων μιας λίστας

```
fun sum nil = 0
    | sum (h::t) = h + sum t
```

- Αριθμός των στοιχείων μιας λίστας που είναι `true`

```
fun ctrue nil = 0
    | ctrue (true::t) = 1 + ctrue t
    | ctrue (false::t) = ctrue t
```

## Ένας περιορισμός: γραμμικά πρότυπα

---

- Δεν επιτρέπεται η χρήση της ίδιας μεταβλητής περισσότερες από μία φορές στο ίδιο πρότυπο
- Για παράδειγμα, το παρακάτω δεν επιτρέπεται:

```
fun f (a, a) = ... for pairs of equal elements  
  | f (a, b) = ... for pairs of unequal elements
```

- Αντί αυτού πρέπει να χρησιμοποιηθεί το παρακάτω:

```
fun f (a, b) =  
  if a = b then ... for pairs of equal elements  
  else ... for pairs of unequal elements
```



## Συνδυασμός προτύπων και `let`

---

```
fun halve nil = (nil, nil)
  | halve [a] = ([a], nil)
  | halve (a::b::cs) =
    let
      val (x, y) = halve cs
    in
      (a::x, b::y)
    end
```

- Με τη χρήση προτύπων στους ορισμούς ενός `let`, μπορούμε να “αποσυνθέσουμε” εύκολα ένα αποτέλεσμα
- Η παραπάνω συνάρτηση παίρνει ως όρισμα μια λίστα και επιστρέφει ένα ζεύγος από λίστες, η κάθε μία από τις οποίες έχει τα μισά στοιχεία της αρχικής λίστας

## Χρήση της συνάρτησης halve

---

```
- fun halve nil = (nil, nil)
=   | halve [a] = ([a], nil)
=   | halve (a::b::cs) =
=     let
=       val (x, y) = halve cs
=     in
=       (a::x, b::y)
=     end;
val halve = fn : 'a list -> 'a list * 'a list
- halve [1];
val it = ([1],[]) : int list * int list
- halve [1,2];
val it = ([1],[2]) : int list * int list
- halve [1,2,3,4,5,6];
val it = ([1,3,5],[2,4,6]) : int list * int list
```

## Ένα μεγαλύτερο παράδειγμα: Merge Sort

- Η συνάρτηση `halve` διανέμει τα στοιχεία μιας λίστας σε δύο περίπου ίσα κομμάτια
- Είναι το πρώτο βήμα για ταξινόμηση συγχώνευσης
- Η συνάρτηση `merge` συγχωνεύει δύο ταξινομημένες λίστες

```
- fun merge (nil, ys) = ys
=   | merge (xs, nil) = xs
=   | merge (x::xs, y::ys) =
=       if x < y then x :: merge (xs, y::ys)
=       else y :: merge (x::xs, ys);
val merge = fn : int list * int list -> int list
- merge ([2],[1,3]);
val it = [1,2,3] : int list
- merge ([1,3,4,7,8],[2,3,5,6,10]);
val it = [1,2,3,3,4,5,6,7,8,10] : int list
```

# Η συνάρτηση Merge Sort

---

```
fun mergeSort nil = nil
  | mergeSort [a] = [a]
  | mergeSort theList =
    let
      val (x, y) = halve theList
    in
      merge (mergeSort x, mergeSort y)
    end
```

Ο τύπος της παραπάνω συνάρτησης είναι

```
int list -> int list
```

λόγω του τύπου της συνάρτησης `merge`

## Παράδειγμα χρήσης της Merge Sort

---

```
- fun mergeSort nil = nil
=   | mergeSort [a] = [a]
=   | mergeSort theList =
=     let
=       val (x, y) = halve theList
=     in
=       merge (mergeSort x, mergeSort y)
=     end;
val mergeSort = fn : int list -> int list
- mergeSort [4,3,2,1];
val it = [1,2,3,4] : int list
- mergeSort [4,2,3,1,5,3,6];
val it = [1,2,3,3,4,5,6] : int list
```

## Φωλιασμένοι ορισμοί συναρτήσεων

---

- Μπορούμε να ορίσουμε τοπικές συναρτήσεις, ακριβώς όπως ορίζουμε τοπικές μεταβλητές, με χρήση `let`
- Συνήθως αυτό γίνεται για βοηθητικές συναρτήσεις που δε θεωρούνται χρήσιμες από μόνες τους
- Με αυτόν τον τρόπο μπορούμε να κρύψουμε τις συναρτήσεις `halve` και `merge` από το υπόλοιπο πρόγραμμα
- Αυτό έχει και το πλεονέκτημα ότι οι εσωτερικές συναρτήσεις μπορούν να αναφέρονται σε μεταβλητές των εξωτερικών συναρτήσεων

```

(* Sort a list of integers. *)
fun mergeSort nil = nil
  | mergeSort [e] = [e]
  | mergeSort theList =
    let
      (* From the given list make a pair of lists
       * (x,y), where half the elements of the
       * original are in x and half are in y. *)
      fun halve nil = (nil, nil)
        | halve [a] = ([a], nil)
        | halve (a::b::cs) =
          let
            val (x, y) = halve cs
          in
            (a::x, b::y)
          end

      (* Merge two sorted lists of integers into
       * a single sorted list. *)
      fun merge (nil, ys) = ys
        | merge (xs, nil) = xs
        | merge (x::xs, y::ys) =
          if x < y then x :: merge(xs, y::ys)
          else y :: merge(x::xs, ys)

      val (x, y) = halve theList
    in
      merge (mergeSort x, mergeSort y)
    end

```

# Ανακεφαλαίωση της γλώσσας ML

---

- Βασικοί τύποι της ML: `int`, `real`, `bool`, `char`, `string`
- Τελεστές: `~`, `+`, `-`, `*`, `div`, `mod`, `/`, `^`, `::`, `@`, `<`, `>`, `<=`, `>=`, `=`, `<>`, `not`, `andalso`, `orelse`
- Επιλογή μεταξύ δύο: `if ... then ... else`
- Ορισμός συναρτήσεων: `fun`, `fn =>` και τιμών: `val`, `let`
- Κατασκευή (και αποσύνθεση) πλειάδων: `(x, y, ..., z)`
- Κατασκευή λιστών: `[x, y, ..., z]`, `::`, `@`
- Κατασκευαστές τύπων: `*`, `list`, και `->`
- Ταίριασμα προτύπων
- Φωλιασμένες συναρτήσεις