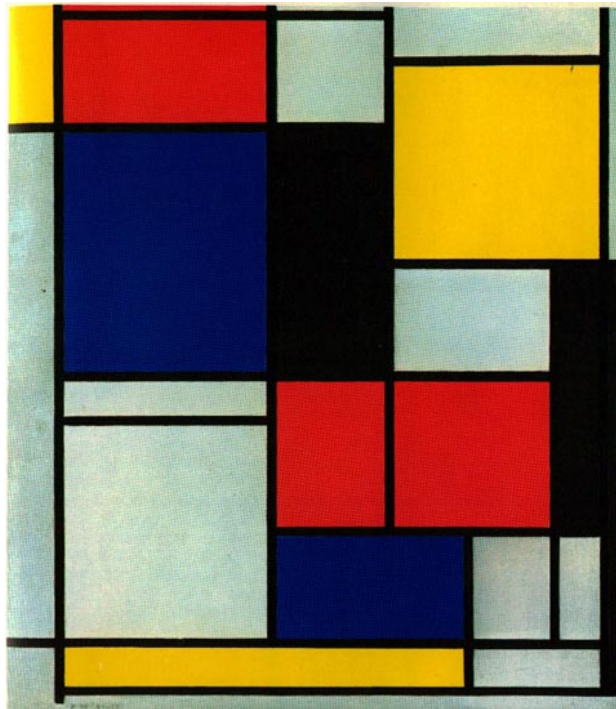


# Εισαγωγή στους Τύπους



Piet Mondrian, *Composition with Blue, Yellow, Black, and Red*, 1922

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

Νίκος Παπασπύρου <[nickie@softlab.ntua.gr](mailto:nickie@softlab.ntua.gr)>

# Περιεχόμενα

---

- Τύποι στις γλώσσες προγραμματισμού
  - Τι είναι οι τύποι;
  - Κατηγορίες και αναπαραστάσεις των τύπων
- Χρήση και χρησιμότητα των τύπων
  - Γλώσσες με στατικό και γλώσσες και δυναμικό σύστημα τύπων
  - Έλεγχος τύπων
    - κατά το χρόνο μετάφρασης (compile-time checking)
    - κατά το χρόνο εκτέλεσης (run-time checking)
  - Η έννοια της συντηρητικής ανάλυσης προγραμμάτων
  - Ισοδυναμία τύπων

## Τι είναι οι τύποι;

---

- Ένας τύπος είναι ένα σύνολο από τιμές
- Όταν ορίζουμε ότι μια μεταβλητή έχει ένα συγκεκριμένο τύπο, δηλώνουμε ότι οι τιμές της μεταβλητής θα είναι πάντα στοιχεία του συγκεκριμένου συνόλου

`int n;`

- **Άρα ένας τύπος είναι ένα σύνολο από τιμές**
  - που έχουν μια συγκεκριμένη κοινή αναπαράσταση
  - και μία συλλογή από λειτουργίες που μπορούν να εφαρμοστούν σε αυτές τις τιμές
- Το ποια σύνολα θεωρούνται ή δεν θεωρούνται τύποι εξαρτάται από τη γλώσσα

# Πρωτόγονοι και κατασκευαζόμενοι τύποι

---

- Κάθε τύπος που μπορεί να χρησιμοποιηθεί αλλά δε μπορεί να οριστεί από ένα πρόγραμμα μιας γλώσσας είναι ένας **πρωτόγονος τύπος** της γλώσσας
  - Πρωτόγονοι τύποι της ML: `int`, `real`, `char`
  - Ένα πρόγραμμα ML δε μπορεί να ορίσει έναν τύπο που να δουλεύει σαν τον προκαθορισμένο τύπο `int`
- Κάθε τύπος που μπορεί να οριστεί από ένα πρόγραμμα (με βάση πρωτόγονους ή ήδη ορισμένους τύπους) είναι ένας **κατασκευαζόμενος τύπος**
  - Π.χ. κατασκευαζόμενος τύπος στην ML: `int list`
  - Ορίζεται με χρήση του πρωτόγονου τύπου `int` και του κατασκευαστή τύπων `list`

# Πρωτόγονοι τύποι

---

- Ο ορισμός της κάθε γλώσσας είναι αυτός που καθορίζει ποιοι είναι πρωτόγονοι τύποι της γλώσσας
- Κάποιες γλώσσες ορίζουν τους πρωτόγονους τύπους πιο αυστηρά από κάποιες άλλες:
  - Π.χ. η Java ορίζει τους πρωτόγονους τύπους επακριβώς
  - Από την άλλη μεριά, π.χ. η C και η ML αφήνουν κάποια περιθώρια ελευθερίας στον ορισμό των πρωτόγονων τύπων μεταξύ διαφορετικών υλοποιήσεων της γλώσσας

# Παράδειγμα: πρωτόγονοι τύποι ακεραίων

---

## C:

`char`  
`unsigned char`  
`short int`  
`unsigned short int`  
`int`  
`unsigned int`  
`long int`  
`unsigned long int`  
`long long`

Δεν υπάρχει προκαθορισμένη υλοποίηση, αλλά οι “μεγαλύτεροι” τύποι πρέπει να έχουν τουλάχιστον το εύρος των “μικρότερων” τύπων

## Java:

`byte` (1-byte signed)  
`char` (2-byte unsigned)  
`short` (2-byte signed)  
`int` (4-byte signed)  
`long` (8-byte signed)

## Scheme:

`integer`

Ακέραιοι “απείρου” εύρους

## Haskell:

`int` (4-byte signed)  
`Integer` (“άπειρο” εύρος)

# Θέματα σχεδιασμού

---

- Σε ποια σύνολα αντιστοιχούν οι πρωτόγονοι τύποι;
  - Τι είναι μέρος του ορισμού της γλώσσας, τι επαφίεται στη διακριτική ευχέρεια της υλοποίησης της γλώσσας;
  - Εάν χρειαστεί, πως μπορεί ένα πρόγραμμα να ανακαλύψει πληροφορίες σχετικές με τα μέλη του συνόλου;
    - (`INT_MAX` στη C, `Int.maxInt` στην ML, ...)
- Τι λειτουργίες υποστηρίζονται και πώς;
  - Λεπτομερείς ορισμοί περί στρογγυλοποίησης, εξαιρέσεων, κ.λπ.
- Η επιλογή της αναπαράστασης είναι καθοριστική για κάποιες από τις αποφάσεις

# Κατασκευαζόμενοι τύποι

---

- Πρόσθετοι τύποι οι οποίοι ορίζονται με χρήση της γλώσσας
- Παραδείγματα: απαριθμήσεις, πλειάδες, πίνακες, συμβολοσειρές, λίστες, ενώσεις, υποτύποι, και τύποι συναρτήσεων
- Για κάθε έναν από αυτούς, υπάρχει στενή σχέση μεταξύ του πώς ορίζονται τα *σύνολα* στα μαθηματικά και του πώς ορίζονται οι *τύποι* στις γλώσσες προγραμματισμού



# Απαριθμήσεις (enumerations)

---

- Στα μαθηματικά μπορούμε ορίσουμε ένα σύνολο απλώς με την απαρίθμηση των μελών του:

$$S = \{ a, b, c \}$$

- Πολλές γλώσσες υποστηρίζουν τύπους **απαρίθμησης**:

**C:** `enum coin {penny, nickel, dime, quarter};`

**Ada:** `type GENDER is (MALE, FEMALE);`

**Pascal:** `type primaryColors = (red, green, blue);`

**ML:** `datatype day = M | Tu | W | Th | F | Sa | Su;`

- Ορίζουν ένα νέο τύπο (= σύνολο τιμών)
- Ορίζουν επίσης μια συλλογή από ονοματισμένες σταθερές αυτού του τύπου (= στοιχεία του συνόλου)

## Αναπαράσταση τιμών μιας απαρίθμησης

---

- Ένας συνήθης τρόπος αναπαράστασης απαριθμήσεων είναι η χρησιμοποίηση μικρών ακεραίων για τις τιμές
- Η αναπαράσταση μπορεί να είναι εμφανής στον προγραμματιστή, όπως π.χ. στη C:

```
enum coin { penny = 1, nickel = 5, dime = 10, quarter = 25 };  
  
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',  
              NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
```

# Λειτουργίες απαριθμήσεων

---

- Έλεγχος ισότητας, φυσικά:

```
fun isWeekend x = (x = Sa orelse x = Su);
```

- Εάν η “ακέραια φύση” της αναπαράστασης των απαριθμήσεων είναι εμφανής, η γλώσσα συνήθως επιτρέπει κάποιες από ή όλες τις λειτουργίες που επιτρέπονται σε ακεραίους:

**Pascal:**

```
for c := red to blue do p(c)
```

**C:**

```
int x = penny + nickel + dime;
```

## Πλειάδες (tuples)

---

- Το καρτεσιανό γινόμενο δύο ή περισσότερων συνόλων ορίζει σύνολα από πλειάδες:

$$S = X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$$

- Κάποιες γλώσσες υποστηρίζουν **καθαρές πλειάδες**:

```
fun get1 (x : real * real) = #1 x;
```

- Πολλές άλλες υποστηρίζουν **εγγραφές (records)**, που είναι πλειάδες τα πεδία των οποίων έχουν ονόματα:

```
C: struct complex {  
    double rp;  
    double ip;  
};
```

```
type complex = {  
    rp:real,  
    ip:real  
};  
fun getip (x : complex) = #ip x;
```

# Αναπαράσταση των πλειάδων

---

- Η συνήθης αναπαράσταση των πλειάδων είναι τα πεδία τους να διατάσσονται το ένα μετά το άλλο στη μνήμη
- Αλλά υπάρχουν πολλές λεπτομέρειες:
  - Με ποια σειρά;
  - Επιτρέπονται “τρύπες” για ευθυγράμμιση (alignment) των πεδίων (π.χ. σε αρχές διαφορετικών λέξεων) στη μνήμη;
  - Είναι κάτι από όλα αυτά ορατό στον προγραμματιστή;

## Παράδειγμα: ANSI C

---

The members of a structure have addresses increasing in the order of their declarations. A non-field member of a structure is aligned at an addressing boundary depending on its type; therefore, there may be unnamed holes in a structure. If a pointer to a structure is cast to the type of a pointer to its first member, the result refers to the first member...

Adjacent field members of structures are packed into implementation-dependent storage units in an implementation-dependent direction...

*The C Programming Language*, 2nd ed.  
Brian W. Kernighan and Dennis M. Ritchie

# Λειτουργίες πλειάδων

---

- Επιλογή στοιχείων, φυσικά:

**C:** `x.ip`  
**ML:** `#ip x`

- Άλλες λειτουργίες ανάλογα με το ποιο/πόσο μέρος της αναπαράστασης είναι εμφανές στον προγραμματιστή:

```
C: double y = *((double *) &x);  
struct person {  
    char *firstname;  
    char *lastname;  
} p1 = {"dennis", "ritchie"};
```

# Ανύσματα (vectors)

---

- Ανύσματα γνωστού (σταθερού) μήκους:

$$S = X^n = \{(x_1, \dots, x_n) \mid \forall i . x_i \in X\}$$

- Ανύσματα αγνώστου μήκους:

$$S = X^* = \bigcup_i X^i$$

- Τύποι σχετικοί με τα ανύσματα:

- Πίνακες, συμβολοσειρές και λίστες
- Είναι σαν τις πλειάδες αλλά συνήθως έχουν τον ίδιο τύπο στοιχείων και πολλές παραλλαγές από γλώσσα σε γλώσσα
- Ένα παράδειγμα διαφοροποίησης: δείκτες (indices) σε πίνακες
  - Ποιες είναι οι τιμές των δεικτών;
  - Καθορίζεται το μέγεθος των πινάκων στο χρόνο μεταγλώττισης ή στο χρόνο εκτέλεσης;



# Τιμές δεικτών

---

- Στη C, C++, Java και C#:
  - Το πρώτο στοιχείο ενός πίνακα `a` είναι το `a[0]`
  - Οι δείκτες είναι πάντα ακέραιοι που αρχίζουν από το 0
- Η Pascal είναι πιο ευέλικτη:
  - Μπορεί να χρησιμοποιηθούν διάφοροι τύποι δεικτών: ακέραιοι, χαρακτήρες, απαριθμήσεις, υποδιαστήματα (subranges)
  - Ο αρχικός δείκτης καθορίζεται από τον προγραμματιστή
  - Ο τελικός δείκτης καθορίζεται επίσης από τον προγραμματιστή: όμως το μέγεθος ενός πίνακα πρέπει να είναι γνωστό κατά το χρόνο μετάφρασης του προγράμματος

## Παράδειγμα πίνακα σε Pascal

---

```
type
  LetterCount = array['a'..'z'] of Integer;
var
  Counts: LetterCount;

begin
  Counts['a'] = 1
  ...
```

# Σχεδιαστικά θέματα σχετικά με τα ανύσματα

---

- Τι είδους δείκτες μπορούν να έχουν οι πίνακες;
- Πρέπει το μέγεθος των πινάκων να είναι καθορισμένο κατά το χρόνο μετάφρασης;
- Μπορεί να αλλάξει (π.χ. επεκταθεί) το μέγεθος δυναμικά;
- Μπορούν οι πίνακες να έχουν πολλές διαστάσεις;
- Είναι ένας πίνακας με πολλές διαστάσεις ισοδύναμος με έναν πίνακα από πίνακες;
- Πώς διατάσσονται τα στοιχεία ενός πίνακα στη μνήμη;
- Οι συμβολοσειρές έχουν το δικό τους τύπο ή είναι πίνακες από bytes; Υπάρχει τύπος λίστας;

## Ενώσεις (unions)

---

- Η ένωση δύο συνόλων δίνει ένα καινούριο σύνολο

$$S = X \cup Y$$

- Πολλές γλώσσες υποστηρίζουν τύπους ενώσεων:

**C:** `union element {  
 int i;  
 double d;  
};`

**ML:** `datatype element =  
 I of int |  
 R of real;`

- Η αναπαράσταση των ενώσεων μπορεί να είναι ή μπορεί να μην είναι εμφανής στον προγραμματιστή:

`sizeof(u) == max(sizeof(u.i), sizeof(u.d))`

## Αυστηρότητα τύπων και ενώσεις

---

- Στην ML, το μόνο που μπορεί να κάνουμε σε μία ένωση είναι να εξάγουμε τα περιεχόμενά της
- Και στην περίπτωση αυτή πρέπει να πούμε τι μπορεί να γίνει με τις τιμές ενώσεων διαφορετικών τύπων:

```
datatype element =  
  I of int |  
  R of real;  
  
fun getReal (R x) = x  
  | getReal (I x) = real x;
```

## Ενώσεις χωρίς αυστηρό σύστημα τύπων

---

- Σε μερικές γλώσσες οι λεπτομέρειες της υλοποίησης των ενώσεων είναι εμφανείς στον προγραμματιστή
- Σε κάποιες περιπτώσεις τα προγράμματα μπορούν να εκμεταλλευθούν το γεγονός ότι ο τύπος κάποιας τιμής χάνεται:

```
union element {  
    int i;  
    double d;  
};  
  
union element e;  
e.i = 100;  
double x = e.d;
```

## Τι λέει η ANSI C για τις ενώσεις;

---

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most one of the members can be stored in a union at any time. If a pointer to a union is cast to the type of a pointer to a member, the result refers to that member.

In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member.

*The C Programming Language*, 2nd ed.  
Brian W. Kernighan and Dennis M. Ritchie

## Υποσύνολα (subsets)

---

- Μπορούμε να ορίσουμε το υποσύνολο που ορίζεται με βάση ένα κατηγορημα  $P$  :

$$S = \{x \in X \mid P(x)\}$$

- Κάποιες γλώσσες υποστηρίζουν **υποτύπους (subtypes)**, με λίγη ή περισσότερη γενικότητα
  - Λίγη γενικότητα: Ορισμός υποδιαστημάτων στην Pascal  
`type digit = 0..9;`
  - Κάμπωση γενικότητα: Ορισμός υποτύπων στην Ada  
`subtype DIGIT is INTEGER range 0..9;`  
`subtype WEEKDAY is DAY range MON..FRI;`
  - Πολλή γενικότητα: Ορισμός τύπων στη Lisp μέσω κατηγορημάτων



# Παράδειγμα: Ada Subtypes

---

```
type DEVICE is (PRINTER, DISK);

type PERIPHERAL(Unit: DEVICE) is
  record
    HoursWorking: INTEGER;
    case Unit is
      when PRINTER =>
        Line_count: INTEGER;
      when DISK =>
        Cylinder: INTEGER;
        Track: INTEGER;
    end case;
  end record;

subtype DISK_UNIT is PERIPHERAL(DISK);
```

## Παράδειγμα: Ορισμός τύπων στη Lisp

---

```
(declare (type integer x))
```

```
(declare (type (or null cons) x))
```

```
(declare (type (and number (not integer)) x))
```

```
(declare (type (and integer (satisfies evenp)) x))
```

# Αναπαράσταση των τιμών ενός υποτύπου

---

- Συνήθως, ο υποτύπος χρησιμοποιεί την ίδια εσωτερική αναπαράσταση με τον υπερτύπο (supertype) του
- Ερωτήσεις:
  - Μπορεί να χρησιμοποιηθεί κάποια πιο “φθηνή” αναπαράσταση; Π.χ. οι δύο παρακάτω τύποι χρειάζονται τον ίδιο χώρο αποθήκευσης;  
`x: 1..9`  
`x: Integer?`
  - Οι υποτύποι επιβάλλονται και ελέγχονται από το μεταγλωττιστή;  
`x := 10`  
`x := x + 1?`

## Λειτουργίες υποτύπων

---

- Συνήθως οι επιτρεπόμενες λειτουργίες είναι όλες οι λειτουργίες που είναι διαθέσιμες και για τον υπερτύπο
- Καθώς και κάποιες άλλες που δεν έχουν τόσο πολύ νόημα στον υπερτύπο:

```
function toDigit (X: Digit) : Char;
```



### Θέμα για περισυλλογή

Ένας υποτύπος είναι ένα υποσύνολο όλων των τιμών ενός υπερτύπου, αλλά συνήθως υποστηρίζει ένα υπερσύνολο από λειτουργίες.

## Κάποια λόγια για τις κλάσεις (classes)

---

- Μια από τις βασικές ιδέες του αντικειμενοστρεφούς προγραμματισμού
- Μια **κλάση** είναι ένα είδος τύπου: κάποια δεδομένα και οι λειτουργίες τους σε “συσκευασία πακέτου”
- Μια υποκλάση είναι ένας υποτύπος: περιλαμβάνει μόνο ένα υποσύνολο των αντικειμένων, αλλά υποστηρίζει ένα υπερσύνολο των λειτουργιών
- Περισσότερα για τις κλάσεις στις διαλέξεις των γλωσσών αντικειμενοστρεφούς προγραμματισμού

# Συναρτήσεις (functions)

---

- Το σύνολο των συναρτήσεων με κάποιο πεδίο ορισμού και τιμών:

$$S = D \rightarrow R = \{f \mid \text{dom } f = D \wedge \text{ran } f = R\}$$

- Οι περισσότερες γλώσσες υποστηρίζουν την έννοια των συναρτήσεων:

```
C: int f(char a, char b) {  
    return a == b;  
}
```

```
ML: fun f(a:char, b:char) = (a = b);
```

# Λειτουργίες συναρτήσεων

---

- Φυσικά, κλήση συναρτήσεων
- Μέχρι στιγμής, το θεωρήσαμε ως δεδομένο ότι οι υπόλοιποι τύποι μπορούσαν να περαστούν ως παράμετροι, να επιστραφούν ως αποτελέσματα, να ανατεθούν σε μεταβλητές, κ.λπ.
- Με τις συναρτήσεις δε μπορούμε να το θεωρήσουμε ως δεδομένο: οι περισσότερες γλώσσες δεν υποστηρίζουν κάτι περισσότερο από κλήση συναρτήσεων!

# Χρησιμότητα των τύπων



# Χρησιμότητα των τύπων

---

- Βοηθούν στην οργάνωση και τεκμηρίωση προγραμμάτων
  - Διαφορετικοί τύποι για διαφορετικές έννοιες
    - Αναπαριστούν έννοιες από το πεδίο του προβλήματος
  - Υποδηλώνουν την προτιθέμενη χρήση των μεταβλητών
    - Οι τύποι μπορούν να ελεγχθούν αυτόματα και να μείνουν σε συμφωνία με αλλαγές στο πρόγραμμα, σε αντίθεση με τα σχόλια
- Υποδεικνύουν και ανιχνεύουν κάποιου είδους σφάλματα
  - Ο έλεγχος τύπων μπορεί να ανιχνεύσει υπολογισμούς χωρίς σημασιολογία, όπως π.χ. `3 + true - "Bill"`
- Υποστηρίζουν τη βελτιστοποίηση (optimization)
  - Παράδειγμα: οι `short` ακέραιοι αποθηκεύονται σε λιγότερα bits
  - Αποφυγή ελέγχου εξαιρέσεων ή υπερχείλισης

# Σφάλματα τύπων (type errors)

---

- Όπως είδαμε, ένας τύπος ορίζεται από:
  - Τρόπους εισαγωγής τιμών για τον τύπο
  - Τρόπους χρησιμοποίησης των τιμών για την παραγωγή νέων τύπων τιμών
- Κάτω από αυτό το πρίσμα
  - Κάθε τύπος συσχετίζεται με ένα σύνολο από λειτουργίες (δηλαδή, με κάποιους τελεστές)
  - Κάθε λειτουργία ορίζεται πάνω σε στοιχεία ενός συγκεκριμένου τύπου (δηλαδή, σε ένα συγκεκριμένο σύνολο τιμών)
  - Ένα **σφάλμα τύπου** λαμβάνει χώρα όταν η λειτουργία πάει να εφαρμοστεί σε ορίσματα εκτός του πεδίου ορισμού της (δηλαδή, σε ορίσματα διαφορετικού τύπου)

## Επισημειώσεις τύπων

---

- Πολλές γλώσσες απαιτούν, ή τουλάχιστον επιτρέπουν, επισημειώσεις τύπων σε μεταβλητές, συναρτήσεις, κ.λπ.
- Χρησιμοποιούνται από τον προγραμματιστή για την παροχή στατικής πληροφορίας τύπων στο σύστημα υλοποίησης της γλώσσας
- Οι επισημειώσεις είναι επίσης ένα είδος τεκμηρίωσης, που καθιστά τα προγράμματα πιο εύκολα αναγνώσιμα από άλλους
- Μέρος της σύνταξης της γλώσσας συνήθως έχει σχέση με τον ορισμό τύπων (π.χ. `*`, `->` και `list` στην ML)

## Εγγενείς τύποι (intrinsic types)

---

- Κάποιες γλώσσες χρησιμοποιούν ονομαστικές συμβάσεις για τον ορισμό των τύπων μεταβλητών
  - Σε διαλέκτους της BASIC: `s$` ορίζει μια συμβολοσειρά
  - Σε διαλέκτους της Fortran: `I` ορίζει έναν ακέραιο
- Όπως και οι ρητές επισημειώσεις τύπων, οι εγγενείς τύποι προμηθεύουν με πληροφορία τύπων τόσο την υλοποίηση της γλώσσας όσο και τον αναγνώστη του προγράμματος

# Απλός συμπερασμός τύπων

---

- Οι περισσότερες γλώσσες υλοποιούν κάποιες απλές μορφές συμπερασμού τύπων
- Οι σταθερές έχουν τύπο που καθορίζεται στατικά  
Π.χ. στη Java: Η σταθερά `10` έχει τύπο `int`, η `10L` έχει τύπο `long`
- Οι εκφράσεις με τη σειρά τους μπορεί να έχουν στατικά καθοριζόμενους τύπους, που συμπεραίνονται εύκολα από τους τελεστές και τους τύπους των ορισμάτων τους  
Π.χ. στη Java: εάν ο `a` είναι `double`, τότε η έκφραση `a*0` έχει τύπο `double (0.0)`
- Η γλώσσα ML πηγαίνει την παραπάνω ιδέα στα άκρα...

# Έλεγχος τύπων: Στατικός και δυναμικός

# Έλεγχος χρόνου μετάφρασης έναντι εκτέλεσης

---

- Οι γλώσσες Lisp, Prolog, Python, Ruby, ... ελέγχουν τους τύπους στο χρόνο εκτέλεσης του προγράμματος
  - Π.χ. στη Lisp σε μια έκφραση (`car x`) ελέγχουμε εάν το `x` είναι λίστα πριν πάρουμε το `car` του `x`
- Οι γλώσσες ML και Haskell ελέγχουν τους τύπους κατά το χρόνο μετάφρασης του προγράμματος
  - Π.χ. σε μια κλήση `f(x)` πρέπει να έχουμε  $f: A \rightarrow B$  και  $x \in A$
- Ένα από τα κύρια “πάρε-δώσε” του σχεδιασμού γλωσσών
  - Και οι δύο τρόποι αποφεύγουν τα σφάλματα
  - Ο δυναμικός έλεγχος καθυστερεί την εκτέλεση
  - Ο στατικός έλεγχος περιορίζει την εκφραστικότητα
    - Λίστες σε Lisp: τα στοιχεία μπορούν να έχουν διαφορετικούς τύπους
    - Λίστες σε ML: όλα τα στοιχεία πρέπει να έχουν τον ίδιο τύπο

# Στατικός έλεγχος τύπων

---

- Στο στατικό έλεγχο τύπων, ο τύπος της κάθε έκφρασης καθορίζεται πριν την εκτέλεση του προγράμματος
- Ο μεταγλωττιστής τυπώνει μηνύματα λάθους όταν οι τύποι που εξάγονται στατικά τον κάνουν να καταλάβει κάποια ασυνέπεια ή ασυμφωνία στη χρήση των τύπων
  - Όσον αφορά σε τελεστές: `1 + "abc"`
  - Όσον αφορά σε συναρτήσεις: `round("abc")`
  - Όσον αφορά σε εντολές της γλώσσας: `if "abc" then ...`
- Πολλές μοντέρνες γλώσσες προγραμματισμού έχουν στατικά (ή ως επί το πλείστον στατικά) συστήματα τύπων



## Δυναμικός έλεγχος τύπων (dynamic typing)

---

- Σε πολλές άλλες γλώσσες (π.χ. Lisp, Scheme, Smalltalk, Erlang, Prolog, Python, Ruby, ...), τα προγράμματα δεν ελέγχονται στατικά για πιθανά σφάλματα τύπων
- Ελέγχονται όμως **δυναμικά** για τέτοιου είδους σφάλματα
- Με άλλα λόγια, κατά το χρόνο εκτέλεσης, η υλοποίηση της γλώσσας ελέγχει ότι τα ορίσματα των τελεστών είναι τύπων οι οποίοι είναι συμβατοί με τους τελεστές

## Παράδειγμα: Lisp

---

- Μια συνάρτηση σε Lisp που προσθέτει δύο αριθμούς:  

```
(defun f (a b) (+ a b))
```
- Θα εγείρει εξαίρεση εάν ο `a` ή ο `b` δεν είναι αριθμοί
- Κλήσεις με ορίσματα λάθος τύπου, π.χ. `(f nil nil)`, δεν υποδεικνύονται κατά το χρόνο μεταγλώττισης
- Στο δυναμικό έλεγχο τύπων τα παραπάνω λάθη πιάνονται κατά το χρόνο εκτέλεσης του προγράμματος
  - Κατά κάποιο τρόπο, στις γλώσσες με δυναμικό έλεγχο τύπων οι τύποι χρησιμοποιούνται παραπάνω από αυτές με στατικό έλεγχο, διότι οι τύποι πρέπει να ελεγχθούν κατά το χρόνο εκτέλεσης
  - Αυτό σημαίνει ότι η υλοποίηση της γλώσσας πρέπει να διατηρήσει πληροφορία για τον τύπο της κάθε μεταβλητής

# Εκφραστικότητα

---

- Στη Lisp, είναι δυνατό να γράψουμε τη συνάρτηση

```
(lambda (x) (cond ((less x 10) x) (T (car x))))
```

Κάποιες χρήσεις της συνάρτησης μπορεί να οδηγήσουν σε σφάλματα τύπων, κάποιες άλλες όμως όχι

- Ο στατικός έλεγχος τύπων είναι πάντα συντηρητικός

```
if (big-hairy-boolean-expression)
  then ((lambda (x) ... ) 5)
  else ((lambda (x) ... ) nil)
```

Δε μπορούμε να αποφασίσουμε κατά το χρόνο μεταγλώττισης εάν θα υπάρξει κάποιο σφάλμα τύπου

# Στατικός έλεγχος τύπων έναντι δυναμικού

---

- Δεν είναι όλα άσπρα ή μαύρα...
- Γλώσσες με στατικό έλεγχο τύπων πολλές φορές χρησιμοποιούν κάποιο δυναμικό έλεγχο
  - Π.χ. λόγω της ύπαρξης υποτύπων (subtyping), ειδικά στις αντικειμενοστρεφείς γλώσσες προγραμματισμού
  - Π.χ. λόγω υπερφορτωμένων τελεστών
- Γλώσσες με δυναμικό έλεγχο τύπων πολλές φορές χρησιμοποιούν κάποιο στατικό έλεγχο
  - Τύποι για (μέρη) προγραμμάτων σε Lisp, μπορούν να εξαχθούν με χρήση δηλώσεων τύπων και τοπικού συμπερασμού τύπων
  - Πολλοί μεταγλωττιστές της Lisp χρησιμοποιούν την πληροφορία αυτή για την αποφυγή ελέγχων τύπων κατά το χρόνο εκτέλεσης και για την παραγωγή καλύτερου (ταχύτερου) κώδικα μηχανής

# Ρητοί έλεγχοι τύπων κατά το χρόνο εκτέλεσης

---

- Κάποιες γλώσσες επιτρέπουν στον προγραμματιστή να ελέγξει τύπους κατά την εκτέλεση του προγράμματος:
  - Π.χ. η Java επιτρέπει τον έλεγχο του τύπου κάποιου αντικειμένου με χρήση του τελεστή `instanceof`
  - Π.χ. η Modula-3 επιτρέπει τη διακλάδωση (branch) με βάση τον τύπο κάποιου αντικειμένου με χρήση της έκφρασης `typecase`
- Τα παραπάνω απαιτούν ότι η πληροφορία για τύπους είναι παρούσα κατά το χρόνο εκτέλεσης, παρόλο που η γλώσσα έχει ως επί το πλείστον στατικό σύστημα τύπων

# Αυστηρά και ασθενή συστήματα τύπων

---

- Η επιδίωξη του ελέγχου των τύπων είναι η αποφυγή της εφαρμογής τελεστών σε ορίσματα των οποίων ο τύπος δεν έχει νόημα για το συγκεκριμένο τελεστή
- Σε κάποιες γλώσσες, όπως στην ML και στη Java, ο έλεγχος τύπων είναι αρκετά εξονυχιστικός και εξασφαλίζει σε μεγάλο βαθμό το παραπάνω—λέμε ότι η γλώσσα έχει **αυστηρό σύστημα τύπων (strong typing)**
- Σε πολλές γλώσσες (όπως π.χ. η C) αυτό δε συμβαίνει: το σύστημα τύπων είναι ασθενές (**weak**) και διάτρητο
  - Αυτό παρέχει ευελιξία στον προγραμματιστή
  - Αλλά δεν προσφέρει αρκετή ασφάλεια

# Σχετική ασφάλεια τύπων των γλωσσών

---

**Μη ασφαλείς:** Η οικογένεια BCPL συμπεριλαμβανομένης της C και της C++

- “Σουλουπώματα” τύπων (type casts), αριθμητική με δείκτες, ...

**Περίπου ασφαλείς:** Η οικογένεια της Algol, Pascal, Ada

- Ξεκρέμαστοι δείκτες
  - Δέσμευσε ένα δείκτη *p* σε έναν ακέραιο, αποδέσμευσε τη μνήμη που δείχνει ο *p*, και στη συνέχεια χρησιμοποίησε την τιμή που δείχνει ο *p*
  - Καμία γλώσσα με ρητή αποδέσμευση μνήμης δεν προσφέρει πλήρη ασφάλεια τύπων!

**Ασφαλείς:** Η οικογένεια της Lisp, η ML, η Smalltalk, και οι Java, C#

- Lisp, Smalltalk, Prolog, Erlang, Ruby: γλώσσες δυναμικών τύπων
- ML, Haskell, Java, C#: γλώσσες στατικών τύπων (statically typed)

# Εργαλεία στατικής ανάλυσης προγραμμάτων

---

- Τομέας με αρκετή ερευνητική δραστηριότητα πρόσφατα
  - Lint, Purify, Coverity, PreFix, PreFast, Dialyzer, Valgrind, ...
- Δύο κατηγορίες ανάλυσης
  - Συντηρητικές (= Sound for correctness)
    - Εάν η ανάλυση πει ότι το πρόγραμμα είναι σωστό, τότε είναι
    - Εάν η ανάλυση πει “όχι σωστό”, μπορεί να μην υπάρχουν λάθη
  - Μη συντηρητικές (= Sound for errors)
    - Εάν η ανάλυση πει “σωστό”, το πρόγραμμα μπορεί να είναι εσφαλμένο
    - Εάν η ανάλυση βρει κάποιο πρόβλημα, τότε όντως υπάρχει κάποια “ανωμαλία” στο πρόγραμμα
  - Μια καλύτερη κατηγοριοποίηση, κατά τη γνώμη μου:
    - Συντηρητικά εργαλεία χρειάζονται για απόδειξη ορθότητας
    - Μη συντηρητικά εργαλεία για εύρεση σφαλμάτων (bug finding)



# Θέματα ισοδυναμίας τύπων

## Ισοδυναμία τύπων

---

- Πότε δύο τύποι θεωρούνται ίδιοι;
- Σημαντική ερώτηση τόσο για το στατικό όσο και για το δυναμικό έλεγχο τύπων
- Για παράδειγμα, η γλώσσα μπορεί να επιτρέπει μια ανάθεση  $a := b$  μόνο εάν η μεταβλητή  $b$  έχει “τον ίδιο” τύπο με την  $a$
- Διαφορετικές γλώσσες ορίζουν την ισοδυναμία των τύπων με διαφορετικούς τρόπους

# Ισοδυναμίες τύπων

---

- **Ισοδυναμία ονόματος (name equivalence)**: δύο τύποι είναι ίδιοι αν και μόνο αν έχουν το ίδιο όνομα
- **Ισοδυναμία δομής (structural equivalence)**: δύο τύποι είναι ίδιοι αν και μόνο αν έχουν προκύψει από τους ίδιους πρωτόγονους τύπους με χρήση της ίδιας σειράς ίδιων κατασκευαστών τύπων
- (Οι παραπάνω, δεν είναι οι μόνοι τρόποι ορισμού της ισοδυναμίας αλλά είναι οι πιο εύκολοι να εξηγηθούν.)

## Παράδειγμα ισοδυναμίας τύπων

---

```
type irpair1 = int * real;  
type irpair2 = int * real;  
fun f(x:irpair1) = #1 x;
```

- Τι θα συμβεί εάν περάσουμε ως όρισμα στην `f` μια παράμετρο τύπου `irpair2`;
  - Η ισοδυναμία ονόματος δεν το επιτρέπει: `irpair2` και `irpair1` είναι διαφορετικά ονόματα
  - Η ισοδυναμία δομής το επιτρέπει: οι τύποι κατασκευάζονται με τον ίδιο τρόπο
- Στην ML επιτρέπεται: η γλώσσα υποστηρίζει ισοδυναμία δομής

## Παράδειγμα ισοδυναμίας τύπων

---

```
var
  Counts1: array['a'..'z'] of Integer;
  Counts2: array['a'..'z'] of Integer;
```

- Τι θα συμβεί εάν προσπαθήσουμε να αναθέσουμε τον πίνακα `Counts1` στον πίνακα `Counts2`;
  - Η ισοδυναμία ονόματος δεν το επιτρέπει: οι τύποι των πινάκων `Counts1` και `Counts2` δεν έχουν τα ίδια ονόματα
  - Η ισοδυναμία δομής το επιτρέπει: οι τύποι κατασκευάζονται με τον ίδιο τρόπο
- Οι περισσότερες υλοποιήσεις της Pascal δεν επιτρέπουν την παραπάνω ανάθεση

# Συμπερασματικά

---

- Οι τύποι παίζουν σημαντικό ρόλο σε πολλές γλώσσες προγραμματισμού
  - Για την οργάνωση και την τεκμηρίωση ενός προγράμματος
  - Για την αποφυγή σφαλμάτων λογισμικού
  - Για την παροχή πληροφορίας στο μεταγλωττιστή της γλώσσας

# Βασική ερώτηση για τα συστήματα τύπων

---

- Πόσο μέρος της αναπαράστασης κάνει το σύστημα τύπων εμφανές στον προγραμματιστή;
  - Μερικοί προγραμματιστές προτιμούν γλώσσες σαν τη C, διότι
    - Προσφέρουν τη δυνατότητα για απ' ευθείας προσπέλαση στη λεπτομερή αναπαράσταση των τύπων όταν για κάποιο λόγο αυτό είναι επιθυμητό
  - Μερικοί άλλοι προτιμούν γλώσσες σαν την ML, διότι
    - Κρύβουν τις λεπτομέρειες της υλοποίησης των τύπων και
    - Προσφέρουν “καθαρούς” τρόπους χρησιμοποίησης δεδομένων που επιτρέπουν την ανάπτυξη προγραμμάτων χωρίς σφάλματα κάποιου είδους, και διευκολύνουν την απόδειξη της ορθότητάς τους