ΕΙΣΑΓΩΓΗ ΣΤΗΝ ΕΠΙΣΤΗΜΗ ΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

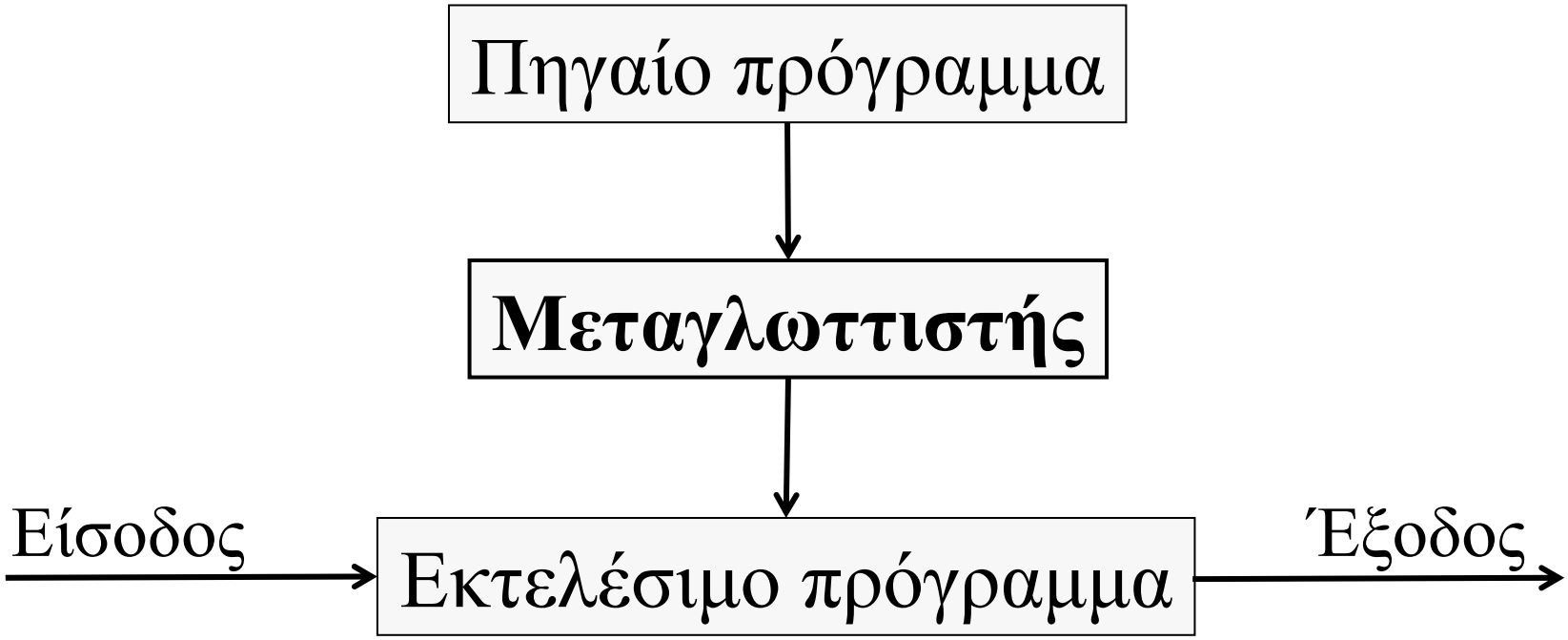
Επιμέλεια και παρουσίαση: **Κωστής Σαγώνας** (kostis@cs.ntua.gr)

Αρχικές διαφάνειες: **Νίκος Παπασπύρου** (nickie@softlab.ntua.gr)

Γλώσσες Προγραμματισμού: Θεωρητικό Υπόβαθρο και Μοντέλα

- ✓Λεκτική και συντακτική ανάλυση γλωσσών
- ✓Συναρτησιακός προγραμματισμός
- ✓Λογικός προγραμματισμός
- ✓Αντικειμενοστρεφής προγραμματισμός

Μετάφραση προγραμμάτων



Μετάφραση προγραμμάτων

Σύνταξη και σημασιολογία προγραμμάτων

- **Σύνταξη γλωσσών προγραμματισμού:** πως δείχνουν τα προγράμματα στο χρήστη, τι μορφή και τι δομή έχουν
  - Η σύνταξη συνήθως ορίζεται με χρήση κάποιας τυπικής γραμματικής
- **Σημασιολογία γλωσσών προγραμματισμού:** τι σημαίνουν τα προγράμματα, ποια είναι η συμπεριφορά τους
  - Η σημασιολογία είναι πιο δύσκολη να ορισθεί από τη σύνταξη – διάφοροι τρόποι ορισμού της

Σύνταξη και σημασιολογία: παραδείγματα

- Φράση λεκτικά λάθος

  οπα πάς οπα χύςέ φαγ επα χιάφα κή

- Φράση λεκτικά ορθή αλλά συντακτικά λάθος

  ο παπάς ο φακή έφαγε παχιά παχύς

- Φράση συντακτικά (και λεκτικά) ορθή αλλά νοηματικά (και σημασιολογικά) λάθος

  ο παπάς ο παχιά έφαγε παχύς φακή

- Φράση συντακτικά και σημασιολογικά ορθή

  η φακή η παχιά έφαγε παχύ παπά

- Φράση συντακτικά και σημασιολογικά ορθή

  ο παπάς ο παχύς έφαγε παχιά φακή

Λεκτική ανάλυση γλωσσών

- Η λεκτική ανάλυση δεν είναι τετριμμένο πρόβλημα γιατί οι γλώσσες προγραμματισμού συνήθως είναι πιο περίπλοκες λεκτικά από τα Ελληνικά

```
*p->f++ = -.12345e-6
```

```
float x, y, z;
float * p = &z;

x = y/*p;
```

Δήλωση λεκτικών μονάδων

- Πως δηλώνονται οι λεκτικές μονάδες;
  - Λέξεις κλειδιά – μέσω συμβολοσειρών (strings)
  - Πως ορίζονται τα ονόματα των μεταβλητών;
  - Πως ορίζονται οι αριθμοί κινητής υποδιαστολής;
- Κανονικές εκφράσεις (regular expressions)
  - Ένας εύχρηστος τρόπος να ορίσουμε ακολουθίες από χαρακτήρες
  - Χρησιμοποιούνται ευρέως: grep, awk, perl, κ.λπ.

Λεκτικές μονάδες με κανονικές εκφράσεις

Παραδείγματα:

- **'0' – ταιριάζει μόνο με το χαρακτήρα 0 (μηδέν)**
- **'0'|'1' – ταιριάζει με μηδέν ή με ένα**
- **'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' – ταιριάζει με ψηφία**
- **[0-9] – το ίδιο με το παραπάνω αλλά σε πιο συμπαγή μορφή**
- **[0-9]* – σειρά από ψηφία (πιθανώς κενή)**

Θέματα σχεδιασμού λεκτικών μονάδων

- Ακέραιοι αριθμοί (π.χ. **10**)
  - Οι αρνητικοί ακέραιοι είναι μία λεκτική μονάδα ή όχι;
- Χαρακτήρες (π.χ. **'a'**)
  - Πως αναπαρίστανται οι μη εκτυπώσιμοι χαρακτήρες ή το **'** ;
- Αριθμοί κινητής υποδιαστολής (π.χ. **3.14e-5**)
- Συμβολοσειρές (π.χ. **"hello world"**)
  - Πώς αναπαρίσταται ο χαρακτήρας **"**;

Λεκτικές μονάδες με κανονικές εκφράσεις

- Το αλφάβητο

  Σ = { ,=,+,-,*,/,(,),>,<,a,…z,0,…,9,A,…Z}

- Ονοματίζουμε κάποιες κανονικές εκφράσεις
  - **LPAR ::=** (
  - **PLUS ::=** +
  - **letter ::=** A | B | … | Z | a | b | … | z
  - **digit ::=** 0 | 1 | … | 9
  - **ID ::= letter** (**letter** | **digit**)*
  - **INT ::=** ((1 | … | 9) **digit***) | 0

Λεκτική ανάλυση (Scanning)

- Τεμαχίζει ένα πρόγραμμα σε μια ακολουθία από λεκτικές μονάδες (tokens)

```
foo = a + bar2(42);
```

ID	EQUALS	ID	PLUS	ID	LPAR	INT	RPAR	SEMI

- Ουσιαστικά απλοποιεί τη λειτουργία του συντακτικού αναλυτή
- Οι λεκτικοί αναλυτές είναι πιο γρήγοροι από τους συντακτικούς αναλυτές διότι έχουν να κάνουν με απλούστερη γραμματική

Μια γραμματική για τα Αγγλικά

Μια πρόταση αποτελείται από μια ουσιαστική φράση, ένα ρήμα, και μια ουσιαστική φράση	<*S*> ::= <*NP*> <*V*> <*NP*>
Μια ουσιαστική φράση αποτελείται από ένα άρθρο και ένα ουσιαστικό	<*NP*> ::= <*A*> <*N*>
Ρήματα είναι τα εξής…	<*V*> ::= **`loves`** | **`hates`** | **`eats`**
Άρθρα είναι τα εξής…	<*A*> ::= **`a`** | **`the`**
Ουσιαστικά είναι τα εξής…	<*N*> ::= **`dog`** | **`cat`** | **`rat`**

Πως δουλεύει μια γραμματική

- Μια γραμματική είναι ένα σύνολο κανόνων που ορίζουν πως κατασκευάζεται ένα **συντακτικό δένδρο**
- Ξεκινάμε βάζοντας το <S> στη ρίζα του δένδρου
- Οι κανόνες της γραμματικής λένε πως μπορούμε να προσθέσουμε παιδιά σε κάθε σημείο του δένδρου
- Για παράδειγμα, ο κανόνας

  <S> ::= <NP> <V> <NP>

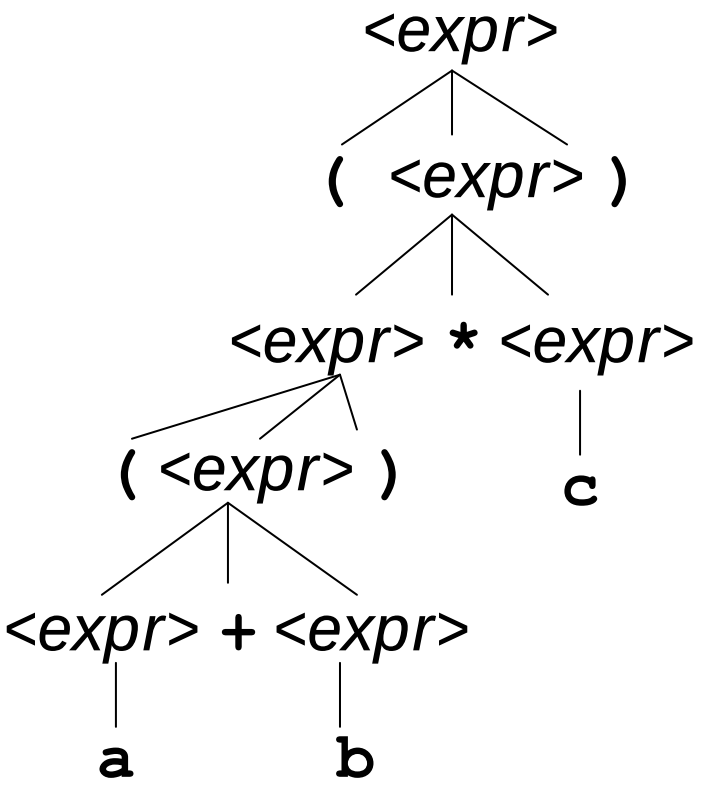
  λέει ότι μπορούμε να προσθέσουμε κόμβους <NP>, <V>, και <NP>, με αυτή τη σειρά, ως παιδιά του κόμβου <S>

Γραμματική για αριθμητικές εκφράσεις

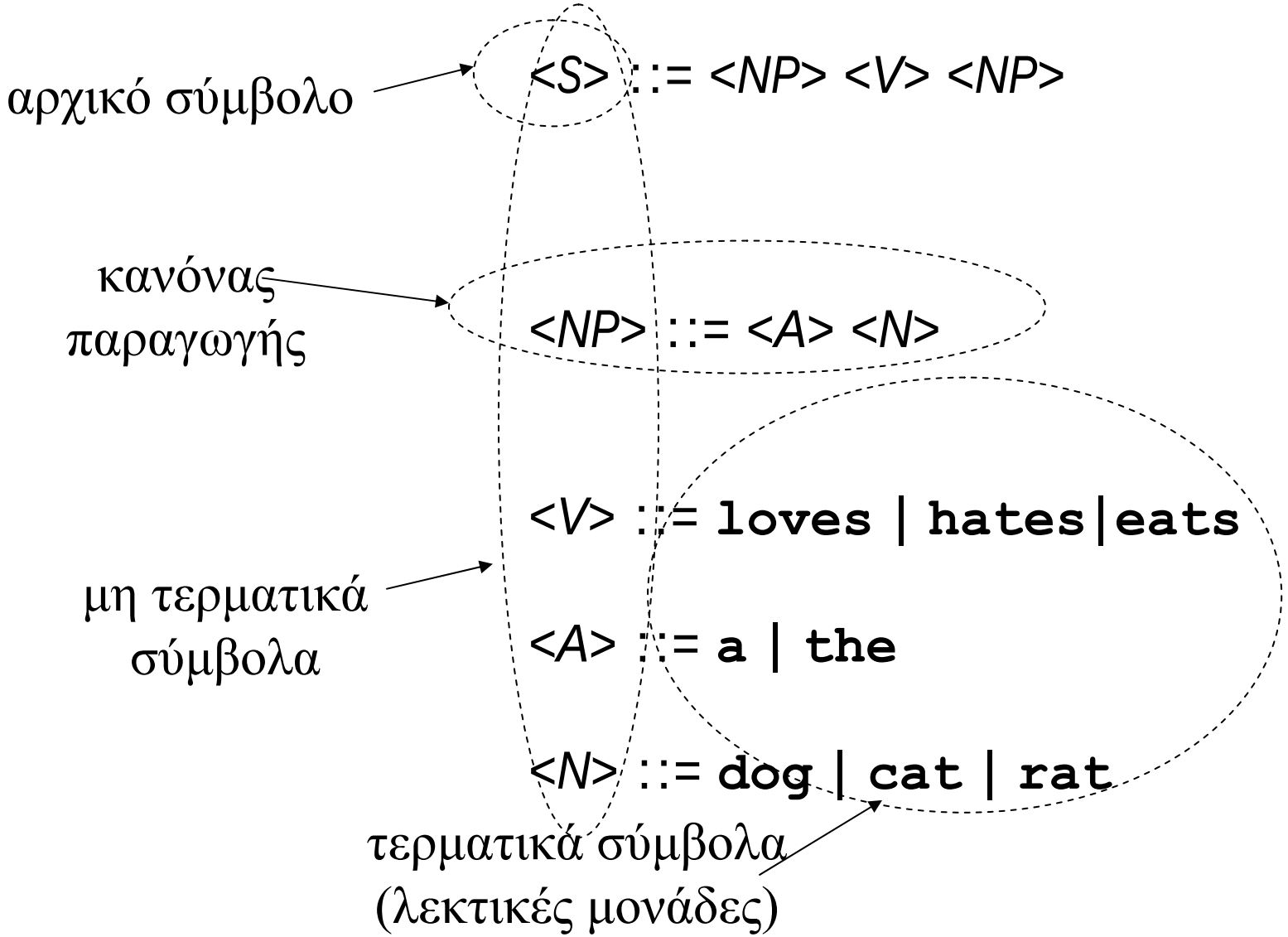
<*expr*> ::= <*expr*> **+** <*expr*>
| <*expr*> ***** <*expr*>
| **(** <*expr*> **)**
| **a** | **b** | **c**

- Μια αριθμητική έκφραση μπορεί να είναι
  - το άθροισμα δύο άλλων εκφράσεων, ή
  - το γινόμενο δύο εκφράσεων, ή
  - μια έκφραση που περικλείεται από παρενθέσεις, ή
  - κάποια από τις μεταβλητές **a**, **b**, ή **c**

Συντακτικό δένδρο



Συστατικά μιας γραμματικής



Ορισμός γραμματικών σε μορφή Backus-Naur

- Μια γραμματική σε μορφή Backus-Naur είναι
  - Ένα σύνολο από **λεκτικές μονάδες (tokens)**
    - Συμβολοσειρές που αποτελούν τα μικρότερα αδιαίρετα κομμάτια της σύνταξης του προγράμματος
  - Ένα σύνολο από μη **τερματικά σύμβολα**
    - Συμβολοσειρές που εγκλείονται σε αγκύλες, π.χ. <*NP*>, και αντιπροσωπεύουν κομμάτια του συντακτικού της γλώσσας
    - Δε συναντιούνται στο πρόγραμμα, αλλά είναι σύμβολα που βρίσκονται στο αριστερό μέρος κάποιων κανόνων της γραμματικής
  - Το **αρχικό σύμβολο** της γραμματικής
    - Ένα συγκεκριμένο μη τερματικό σύμβολο που αποτελεί τη ρίζα του συντακτικού δένδρου για κάθε αποδεκτό από τη γλώσσα πρόγραμμα
  - Ένα σύνολο από **κανόνες παραγωγής**

Κανόνες παραγωγής (1)

- Οι κανόνες παραγωγής χρησιμοποιούνται για την κατασκευή του συντακτικού δένδρου
- Κάθε κανόνας έχει τη μορφή **Α ::= Δ**
  - Το αριστερό μέρος **Α** αποτελείται από ένα μη τερματικό σύμβολο
  - Το δεξί μέρος **Δ** είναι μια ακολουθία από τερματικά (λεκτικές μονάδες) και μη τερματικά σύμβολα

Κανόνες παραγωγής (2)

- Κάθε κανόνας προσδιορίζει έναν πιθανό τρόπο κατασκευής του συντακτικού υποδένδρου που
  - έχει ως ρίζα του το μη τερματικό σύμβολο στο αριστερό μέρος **Α** του κανόνα και
  - ως παιδιά αυτής της ρίζας (με την ίδια σειρά εμφάνισης) έχει τα σύμβολα στο δεξί μέρος **Δ** του κανόνα

Κατασκευή συντακτικών δένδρων

- Αρχίζουμε την κατασκευή βάζοντας το αρχικό σύμβολο της γραμματικής στη ρίζα του δένδρου
- Προσθέτουμε παιδιά σε κάθε μη τερματικό σύμβολο, χρησιμοποιώντας *κάποιον* από τους *κανόνες παραγωγής* της γλώσσας για το συγκεκριμένο μη τερματικό
- Η διαδικασία τερματίζει όταν όλα τα φύλλα του δένδρου αποτελούνται από λεκτικές μονάδες
- Η συμβολοσειρά που αντιστοιχεί στο δένδρο βρίσκεται διαβάζοντας τα φύλλα του δένδρου από αριστερά προς τα δεξιά

Παραδείγματα

```
<expr> ::= <expr> + <expr>
         | <expr> * <expr>
         | ( <expr> )
         | a | b | c
```

- Τα συντακτικά δένδρα για τις παρακάτω εκφράσεις

```
a+b        (a+b)        (a+(b))        a*b+c
```

- Η κατασκευή των συντακτικών δένδρων είναι δουλειά του συντακτικού αναλυτή (parser) ενός μεταγλωττιστή
- Υπάρχουν διάφοροι αποδοτικοί αλγόριθμοι και εργαλεία για ημιαυτόματη κατασκευή του συντακτικού αναλυτή

Ορισμός γλωσσών

- Για να ορίσουμε τη σύνταξη των γλωσσών προγραμματισμού χρησιμοποιούμε γραμματικές
- Η γλώσσα που ορίζεται από μια γραμματική είναι το σύνολο των συμβολοσειρών για τα οποία η γραμματική μπορεί να παράξει συντακτικά δένδρα
- Τις περισσότερες φορές το σύνολο αυτό είναι άπειρο (παρόλο που η γραμματική είναι πεπερασμένη)
- Η κατασκευή μιας γραμματικής για μια γλώσσα μοιάζει λίγο με προγραμματισμό…

Παράδειγμα κατασκευής γραμματικής (1)

Συνήθως γίνεται με χρήση της τεχνικής “διαίρει και βασίλευε” (divide and conquer)

- **Παράδειγμα**: κατασκευή της γλώσσας των δηλώσεων της C:
  - αρχικά, η δήλωση έχει ένα όνομα τύπου
  - στη συνέχεια μια ακολουθία από μεταβλητές που διαχωρίζονται με κόμματα (όπου κάθε μεταβλητή μπορεί να πάρει μια αρχική τιμή)
  - και στο τέλος ένα ερωτηματικό (semicolon)

```
float a;
short a, b, c;
int a = 1, b, c = 1 + 2;
```

Παράδειγμα κατασκευής γραμματικής (2)

- Αρχικά ας αγνοήσουμε την πιθανή ύπαρξη αρχικοποιητών:

```
<var-decl> ::= <type-name> <declarator-list> ;
```

- Ο κανόνας για τα ονόματα των πρωτογενών τύπων (primitive types) είναι απλούστατος:

```
<type-name> ::= short | int | long | char |
                float | double
```

Σημείωση: δεν παίρνουμε υπόψη τύπους πινάκων

Παράδειγμα κατασκευής γραμματικής (3)

- Η ακολουθία των μεταβλητών που διαχωρίζονται με κόμματα έχει ως εξής:

```
<declarator-list> ::= <declarator>
                    | <declarator> , <declarator-list>
```

- Όπου ξανά, έχουμε προς το παρόν αγνοήσει τους πιθανούς αρχικοποιητές των μεταβλητών

Παράδειγμα κατασκευής γραμματικής (4)

- Οι δηλωτές μεταβλητών, με ή χωρίς αρχικοποιήσεις, ορίζονται ως:

```
<declarator> ::= <variable-name>
               | <variable-name> =  <expr>
```

- Για ολόκληρη τη C:
  - Πρέπει να επιτρέψουμε και ζεύγη από αγκύλες μετά το όνομα των μεταβλητών για τη δήλωση των πινάκων
  - Πρέπει επίσης να ορίσουμε και τη σύνταξη των αρχικοποιητών πινάκων
  - (Φυσικά θέλουμε και ορισμούς για τα μη τερματικά σύμβολα <*variable-name*> και <*expr*>)

ΚΥΡΙΑ ΜΟΝΤΕΛΑ ΓΛΩΣΣΩΝ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Κύρια προγραμματιστικά μοντέλα (1)

- Προστακτικός προγραμματισμός (*imperative programming*)
  - FORTRAN, Algol, COBOL, BASIC, C, Pascal, Modula-2, Ada
- Συναρτησιακός προγραμματισμός (*functional programming*)
  - LISP, ML, Scheme, Miranda, Haskell, Erlang
- Λογικός προγραμματισμός (*logic programming*)
  - Prolog

Κύρια προγραμματιστικά μοντέλα (2)

- Αντικειμενοστρεφής προγραμματισμός (*object-oriented programming*)
  - Simula, Smalltalk, C++, Eiffel, Java, C#
- Προγραμματιμός «σεναρίων» (*scripting*)
  - Perl, Python, Ruby, JavaScript, PHP
- Παράλληλος/ταυτόχρονος/κατανεμημένος προγραμματισμός (*parallel/concurrent/distributed progr.*)
  - OCCAM, Concurrent C, Ada, Java, C#, Erlang

ΣΥΝΑΡΤΗΣΙΑΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Διαφάνεια αναφοράς (referential transparency)

- Σε μία γλώσσα συναρτησιακού 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
- Στις προστακτικές γλώσσες 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 Pascal

```
program example(output)
var flag:boolean;

function f(n:int): int
begin
  if flag then f := n
          else f := 2*n;
  flag := not flag
end

begin
  flag := true;
  writeln(f(1)+f(2));
  writeln(f(2)+f(1));
end
```

Τι τυπώνει το πρόγραμμα;

5 και μετά 4

- Περίεργο διότι περιμένουμε ότι

  f(1)+ f(2) = f(2)+ f(1)

- Στα μαθηματικά οι συναρτήσεις εξαρτώνται μόνο από τα ορίσματά τους

Συναρτησιακός προγραμματισμός (1)

- Πλεονεκτήματα
  - Συντομία *(2-10 φορές μικρότερος κώδικας)*
  - Ευκολία στην κατανόηση
  - Λιγότερα σφάλματα εκτέλεσης
  - Επαναχρησιμοποίηση, αφαίρεση, δόμηση
  - Αυτόματη διαχείριση μνήμης

Παράδειγμα: QuickSort σε Haskell

```
qsort [] = []
qsort (x:xs) = qsort lt ++ [x] ++ qsort ge
  where lt = [y | y <- xs, y < x]
        ge = [y | y <- xs, y >= x]
```

Συναρτησιακός προγραμματισμός (2)

- Μειονεκτήματα
  - Μειωμένη απόδοση
  - Μεγαλύτερες απαιτήσεις μνήμης
- Όχι μειονεκτήματα ☺
  αλλαγή φιλοσοφίας στον προγραμματισμό
  - Όχι μεταβλητές, όχι εντολές ανάθεσης
  - Εκφράσεις και συναρτήσεις
- Τα παραδείγματα που ακολουθούν είναι σε Haskell http://www.haskell.org/

Δηλώσεις και εξαγωγή τύπων

- Δήλωση συναρτήσεων

```
inc n = n + 1
f t = t * inc t
```

- Δήλωση τιμών

```
x = f 6
y = f (f 2)
```

- Συμπερασμός τύπων (*type inference*)
  - Οι τύποι υπολογίζονται αυτόματα

```
inc, f :: Int -> Int
x, y   :: Int
```

Υπολογισμοί τιμών

- Υπολογισμός τιμής

```
f t = t * inc t
```

```
x → f 6 → 6 * inc 6
  → 6*(6+1) → 6*7 → 42
```

- Το αποτέλεσμα είναι ανεξάρτητο της σειράς των επιμέρους υπολογισμών (*υπό κ.σ.*)

```
y → f (f 2) → f (2 * inc 2)
  → f (2*(2+1)) → f (2*3) → f 6
  → 6 * inc 6 → 6*(6+1) → 6*7 → 42

y → f (f 2) → f 2 * inc (f 2)
  → (2 * inc 2) * inc (2 * inc 2)
  → (2*(2+1)) * inc (2*(2+1))
  → (2*(2+1)) * (2*(2+1)+1) → … → 42
```

Τοπικές δηλώσεις

- Με χρήση του **`let`**

```
x = let inc n = n+1
        f t = t * inc t
    in  f 6
```

- ... ή με χρήση του **`where`**

```
x = f 6
  where inc n = n+1
        f t = t * inc t
```

- Οι τοπικές δηλώσεις ακολουθούν κανόνες εμβέλειας όπως π.χ. της Pascal

Πλειάδες τιμών

- Συναρτήσεις με "πολλές" παραμέτρους

```
add :: (Int, Int) -> Int
add (x, y) = x+y
```

- ... και "πολλά" αποτελέσματα

```
solve2eq :: (Double, Double, Double)
                  -> (Double, Double)
solve2eq (a, b, c) =
   let d  = b*b - 4.0*a*c
       x1 = (-b - sqrt(d)) / (2.0*a)
       x2 = (-b + sqrt(d)) / (2.0*a)
   in  (x1, x2)
```

Αναδρομή

- Στο συναρτησιακό προγραμματισμό είναι ο κύριος τρόπος επαναληπτικών υπολογισμών
  - Υπολογισμός παραγοντικού

```
factorial n =
  if n <= 1 then 1
            else n * factorial (n-1)
```

  - Υπολογισμός Μ.Κ.Δ. (αλγόριθμος Ευκλείδη)

```
gcd (n, 0) = n
gcd (n, m) = gcd(m, n `mod` m)
```

αν m≠0

pattern matching στις παραμέτρους

Συναρτήσεις υψηλής τάξης

- Συναρτήσεις που παίρνουν ως παραμέτρους άλλες συναρτήσεις

```
twice :: (Int -> Int, Int) -> Int
twice (f, x) = f (f x)

inc n = n + 1
plus2 x = twice (inc, x)
```

- ... ή που έχουν ως αποτέλεσμα συναρτήσεις

```
plusN :: Int -> (Int -> Int)
plusN x = let f y = x + y
          in  f
```

Ανώνυμες συναρτήσεις

- "Η συνάρτηση που απεικονίζει κάθε *n* στο *n*+1" $\lambda n.\ n+1$

```
\n -> n+1
```

- Παράδειγμα

```
twice :: (Int -> Int, Int) -> Int
twice (f, x) = f (f x)

plus2 :: Int -> Int
plus2 x = twice (\n -> n+1, x)

plusN :: Int -> (Int -> Int)
plusN x = \y -> x + y
```

Παραμέτρων συνέχεια (1)

- "Currying" *(Haskell B. Curry)*
  - Μια συνάρτηση με δύο παραμέτρους ισοδυναμεί με μια συνάρτηση που δέχεται την πρώτη παράμετρο και επιστρέφει μια συνάρτηση που δέχεται τη δεύτερη

```
add :: (Int, Int) -> Int
add (x, y) = x+y

add' :: Int -> (Int -> Int)
add' x = \y -> x+y
```

Curried version

```
add (x, y) == (add' x) y
```

Παραμέτρων συνέχεια (2)

- Απλούστερη γραφή curried συναρτήσεων

```
add :: Int -> Int -> Int
add x y = x+y

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)
```

- Με τις curried συναρτήσεις επιτρέπεται η "μερική εφαρμογή"

```
twice (add 20) 2 → add 20 (add 20 2)
  → add 20 (20+2) → 20+(20+2) → 42
```

Λίστες (1)

- Ακολουθίες ομοειδών στοιχείων

```
digits :: [Int]
digits = [0,1,2,3,4]
```

```
== 0:1:2:3:4:[]
```

```
prices :: [(String, Float)]
prices = [("μήλο",   3.0),
          ("αχλάδι", 2.2),
          ("ανανάς", 5.5)]
```

- Παραδείγματα με λίστες
  - Εύρεση μήκους

```
length [] = 0
length (x:xs) = 1 + length xs
```

Λίστες (2)

- Παραδείγματα με λίστες (συνέχεια)
  - Συνένωση δύο λιστών

```
concat [] ys = ys
concat (x:xs) ys = x : concat xs ys
```

  - Αναστροφή λίστας

```
reverse [] = []
reverse (x:xs) =
    concat (reverse xs) [x]
```

  - Αναστροφή λίστας (*καλύτερη υλοποίηση*)

```
reverse xs = rev xs []
  where rev [] ys = ys
        rev (x:xs) ys = rev xs (x:ys)
```

Λίστες (3)

- Παραδείγματα με λίστες και συναρτήσεις υψηλής τάξης
  - Εφαρμογή μιας συνάρτησης σε όλα τα στοιχεία μιας λίστας

```
map f [] = []
map f (x:xs) = f x : map f xs
```

  - "Φιλτράρισμα" των στοιχείων μιας λίστας

```
filter f [] = []
filter f (x:xs) =
  if f x then x : filter f xs
         else filter f xs
```

Παραμετρικός πολυμορφισμός (1)

- Ποιος ο τύπος της ταυτοτικής συνάρτησης;

```
id x = x
```

- Μπορεί να δεχθεί παραμέτρους κάθε τύπου

```
id 42       → 42        :: Int
id "Hello"  → "Hello"   :: String
id inc      → inc       :: Int -> Int
```

- Είναι μια πολυμορφική συνάρτηση

```
id :: a -> a
```
(*για κάθε τύπο* **a**)

Παραμετρικός πολυμορφισμός (2)

- Περισσότερες πολυμορφικές συναρτήσεις

```
length  :: [a] -> Int
concat  :: [a] -> [a] -> [a]
reverse :: [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
```

  - και με περισσότερους "άγνωστους" τύπους

```
map :: (a -> b) -> [a] -> [b]
```

  - ένα ακόμα σύνθετο παράδειγμα

```
zip :: [a] -> [b] -> [(a, b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Ορισμοί τύπων (1)

- Απλές απαριθμήσεις

```
data Light = Red | Green | Yellow

next :: Light -> Light
next Green = Yellow
next Yellow = Red
next Red = Green
```

- Πιο σύνθετοι τύποι δεδομένων

```
data Number = NInteger Int
            | NReal    Double
            | NComplex Double Double

neg (NInteger n) = NInteger (-n)
neg (NReal r) = NReal (-r)
neg (NComplex x y) = NComplex (-x) (-y)
```

Ορισμοί τύπων (2)

- Αναδρομικά ορισμένοι τύποι
  - Συνδεδεμένες μονομορφικές λίστες

    ```
    data IntList = Nil | Cons Int IntList
    ```

    ή και πολυμορφικές λίστες

    ```
    data List a = Nil | Cons a (List a)
    ```

  - Παραδείγματα

    ```
    sum :: List Int -> Int
    sum Nil = 0
    sum (Cons x xs) = x + sum xs

    length :: List a -> Int
    length Nil = 0
    length (Cons x xs) = 1 + length xs
    ```

Ορισμοί τύπων (3)

- Αναδρομικά ορισμένοι τύποι (συνέχεια)
  - Πολυμορφικά δυαδικά δέντρα

```
data Tree a = Nil
            | Node a (Tree a) (Tree a)
```

  - Μέτρηση κόμβων

```
count :: Tree a -> Int
count Nil = 0
count (Node a left right) =
  1 + count left + count right
```

Ορισμοί τύπων (4)

- Αναδρομικά ορισμένοι τύποι (συνέχεια)
  - Διάσχιση κατά βάθος

```
preorder :: Tree a -> [a]
preorder Nil = []
preorder (Node a left right) =
  a : preorder left ++ preorder right
```

  - Διάσχιση κατά πλάτος

```
traverseBF :: Tree a -> [a]
traverseBF t = trav [t]
  where trav [] = []
        trav (Nil : ts) = trav ts
        trav (Node a left right : ts) =
          a : trav (ts ++ [left, right])
```

Ορισμοί τύπων (5)

- Αναδρομικά ορισμένοι τύποι (συνέχεια)
  - Διάσχιση κατά βάθος (*καλύτερη υλοποίηση*)

```
preorder t = trav t []
  where trav Nil ts = ts
        trav (Node a left right) ts =
          a : trav left (trav right ts)
```

  - Διάσχιση κατά πλάτος (*καλύτερη υλοποίηση*)

```
traverseBF t = trav [t] []
  where
    trav [] [] = []
    trav [] ys = trav (reverse ys) []
    trav (Nil : xs) ys = trav xs ys
    trav (Node a left right : xs) ys =
      a : trav xs (right : left : ys)
```

Πρόθυμη και οκνηρή αποτίμηση (1)

- Πρόθυμη αποτίμηση (*eager evaluation*)
  - Οι υπολογισμοί γίνονται όσο πιο νωρίτερα δυνατόν
  - Οι παράμετροι των συναρτήσεων αποτιμώνται πριν την κλήση
  - π.χ. LISP, ML, Scheme, Erlang
- Οκνηρή αποτίμηση (*lazy evaluation*)
  - Οι υπολογισμοί γίνονται όσο πιο αργότερα δυνατόν, δηλαδή μόνο όταν χρειαστεί το αποτέλεσμά τους (*call by need evaluation*)
  - Οι παράμετροι των συναρτήσεων αποτιμώνται την πρώτη φορά που θα χρειαστεί η τιμή
  - π.χ. Miranda, Haskell

Πρόθυμη και οκνηρή αποτίμηση (2)

- Παράδειγμα: άπειρη αναδρομή

```
loop n = loop (n+1)
foo x y = if x == 1 then y else 42
```

  - Πρόθυμη αποτίμηση

```
foo 6+1 (loop 0) → foo 7 (loop (0+1))
 → foo 7 (loop 1) → foo 7 (loop (1+1))
 → foo 7 (loop 2) → …
```
(*δεν τερματίζει*)

  - Οκνηρή αποτίμηση

```
foo 6+1 (loop 0)
 → if 6+1 == 1 then loop 0 else 42
 → if 7 == 1 then loop 0 else 42 → 42
```

Πρόθυμη και οκνηρή αποτίμηση (3)

- Παράδειγμα: άπειρη λίστα πρώτων αριθμών με το κόσκινο του Ερατοσθένη

```
primes :: [Int]
primes = sieve (natsgt 2)
  where
    natsgt n = n : natsgt (n+1)
    sieve (x:xs) =
      x : sieve (filter (ndiv x) xs)
    ndiv x y = y `mod` x /= 0
```

```
primes == [2,3,5,7,11,13,17,19,23,29,…]
```

```
allnats = 0 : map (\n -> n+1) allnats
```

Πέρα από το συναρτησιακό μοντέλο

- "Αγνά" συναρτησιακός προγραμματισμός (*purely functional programming*)
- Παρενέργειες (*side effects*)
  - Μεταβαλλόμενες μεταβλητές(*mutable variables*)
    - πολλαπλές αναθέσεις τιμής
    - προσπελάσεις (ανακλήσεις τιμής)
  - Είσοδος/έξοδος (*input/output*)
    - εκτύπωση σε οθόνη ή σε αρχείο
    - ανάγνωση από το πληκτρολόγιο ή από αρχείο

ΛΟΓΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Λογικός προγραμματισμός

- Κεντρική ιδέα
  - Το πρόγραμμα είναι εκφρασμένο σε μια μορφή συμβολικής λογικής με δήλωση των σχέσεων μεταξύ των δεδομένων που διαχειρίζεται
  - Η εκτέλεση του προγράμματος ισοδυναμεί με τη διεξαγωγή συλλογισμών σε αυτή τη λογική
- Η γλώσσα Prolog
  - W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, 4th edition, Springer-Verlag, New York, 1997.

Κατηγορηματική λογική (1)

(*predicate logic*)

- Προτάσεις
  - Δηλώσεις σε συμβολική μορφή που είναι είτε αληθείς είτε όχι αληθείς
  - Αναφέρονται σε αντικείμενα και σε σχέσεις μεταξύ αυτών
  - Ατομική πρόταση: **`κατηγόρημα(ορίσματα)`**
  - Τελεστές: $\neg$ (όχι) $\wedge$ (και) $\vee$ (ή)
    $\Rightarrow$ (συνεπάγεται) $\Leftarrow$ (προκύπτει από)
    $\Leftrightarrow$ (ισοδυναμεί) $\forall$ (για κάθε) $\exists$ (υπάρχει)

Κατηγορηματική λογική (2)

- Κανονική μορφή και προτάσεις Horn
  - Κάθε πρόταση μπορεί να γραφεί στην παρακάτω κανονική μορφή

$$B_1 \vee B_2 \vee ... \vee B_m \Leftarrow A_1 \wedge A_2 \wedge ... \wedge A_n$$

όπου τα $A_1, A_2, ... A_n$ και $B_1, B_2, ... B_m$ είναι ατομικές προτάσεις

  - Πολλές (αλλά όχι όλες) οι προτάσεις μπορούν να γραφούν σε μορφή *προτάσεως Horn*

$$B \Leftarrow A_1 \wedge A_2 \wedge ... \wedge A_n$$

ή $$A_1 \wedge A_2 \wedge ... \wedge A_n$$

Τύποι δεδομένων της γλώσσας Prolog

- Μεταβλητές: Αρχίζουν με κεφαλαίο γράμμα ή με το χαρακτήρα '_'
- Άτομα: Αρχίζουν με μικρό γράμμα (π.χ. **`john`**) ή περικλείονται από αποστρόφους (π.χ. **`'John'`**)
- Ακέραιοι: **`0`**, **`42`**, **`-42`**, ...
- Πραγματικοί: **`3.1415`**,
- Λίστες: **`[]`**, **`[1,2,3]`**, **`[42,apple]`**
- Δομημένα δεδομένα: **`person(john, 1970)`**, **`student(el03109001,[10,9,10,9,10])`**

Γεγονότα και κανόνες (1)

- Δήλωση γεγονότων

```
male(john).
male(george).
female(mary).
female(jenny).
parent(john,george).
parent(mary,george).
parent(john,jenny).
parent(mary,jenny).
```

- Δήλωση κανόνων

```
father(X,Y) :- parent(X,Y), male(X).
mother(X,Y) :- parent(X,Y), female(X).
```

Γεγονότα και κανόνες (2)

- Δήλωση κανόνων (συνέχεια)

```
human(X) :- male(X).
human(X) :- female(X).
brother(X,Y) :-
  male(X), parent(Z,X), parent(Z,Y).
sister(X,Y) :-
  female(X), parent(Z,X), parent(Z,Y).
```

- Ερωτήσεις – στόχοι (*goals*)

```
?- male(john).
yes
?- male(mary).
no
```

Γεγονότα και κανόνες (3)

- Ερωτήσεις – στόχοι (συνέχεια)

```
?- male(peter).
no
?- male(X).
X = john;
X = george;
no
?- human(X).
X = john;
X = george;
X = mary;
X = jenny;
no
```

Γεγονότα και κανόνες (4)

- Ερωτήσεις – στόχοι (συνέχεια)

```
?- brother(george,jenny).
yes
?- mother(X,george).
X = mary;
no
?- sister(X,Y).
X = jenny, Y = george;
X = jenny, Y = jenny;
X = jenny, Y = george;
X = jenny, Y = jenny;
no
```

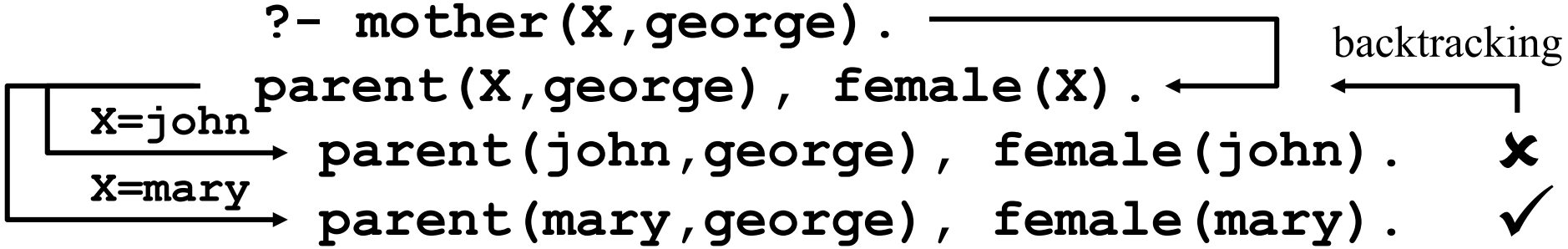
Επίλυση (1)

(*resolution*)

- Αλγόριθμος επίλυσης
  - Η ερώτηση-στόχος συγκρίνεται με τα γεγονότα και τα αριστερά μέλη (κεφαλές) των κανόνων
  - Ενοποίηση (*unification*): αυτή η σύγκριση προκαλεί πιθανώς τη συγκεκριμενοποίηση κάποιων μεταβλητών
  - Αν η σύγκριση επιτύχει για κάποιο γεγονός, ο στόχος έχει ικανοποιηθεί
  - Αν επιτύχει για το αριστερό μέλος κανόνα, οι υποθέσεις προστίθενται στη λίστα των στόχων

Επίλυση (2)

- Αλγόριθμος επίλυσης (συνέχεια)
  - Οπισθοδρόμηση (*backtracking*): αν ένας στόχος αποτύχει, δηλαδή δεν μπορεί να ικανοποιηθεί, ο αλγόριθμος οπισθοδρομεί στον αμέσως προηγούμενο στόχο και προσπαθεί την επίλυση με την αμέσως επόμενη επιλογή γεγονότος ή κανόνα
- Παράδειγμα



Χαρακτηριστικά της Prolog (1)

- Η σειρά εμφάνισης γεγονότων, κανόνων και στόχων είναι καθοριστική

```
ancestor(X,Y) :-
  ancestor(Z,Y), parent(X,Z).
ancestor(X,Y) :- parent(X,Y).
```

- Υπόθεση του κλειστού κόσμου
  - Οι μόνες αληθείς προτάσεις είναι αυτές που αποδεικνύονται βάσει των γνωστών γεγονότων και κανόνων

Χαρακτηριστικά της Prolog (2)

- Το πρόβλημα της άρνησης
  - Μια αρνητική πρόταση **`not`** *A* είναι αληθής όταν ο στόχος *A* δεν μπορεί να ικανοποιηθεί
  - Αυτή η άρνηση δεν ταυτίζεται με τη λογική άρνηση, π.χ. **`not not`** $A \neq A$
- Το πρόβλημα των "προδιαγραφών"
  - Ένα πρόγραμμα ισοδυναμεί με τον ορισμό των προδιαγραφών του σε κατηγορηματική λογική
  - Ο μετασχηματισμός των προδιαγραφών σε αλγόριθμο επίλυσης είναι ένα άλυτο πρόβλημα

Παραδείγματα (1)

- Αριθμητικές πράξεις

```
daysOf(january,Y,31).
daysOf(february,Y,29) :-
   Y mod 400 =:= 0,
   Y mod 4000 =\= 0, !.
daysOf(february,Y,29) :-
   Y mod 4 =:= 0,
   Y mod 100 =\= 0, !.
daysOf(february,Y,28).
daysOf(march,Y,31).
...
daysOf(december,Y,31).

validDate(D,M,Y) :- daysOf(M,Y,X),
                    D>=1, D=<X.
```

Προηγούμενο παράδειγμα με if-then-else

```
daysOf(january,Y,31).
daysOf(february,Y,Z) :-
     ( (Y mod 400 =:= 0,
        Y mod 4000 =\= 0) ->
           Z = 29
     ; (Y mod 4 =:= 0,
        Y mod 100 =\= 0) ->
           Z = 29
     ;  Z = 28
     ).
daysOf(march,Y,31).
  ...
daysOf(december,Y,31).
```

Παραδείγματα (2)

- Λίστες

```
length([],0).
length([_|Xs],N) :-
  length(Xs,M), N is M+1.


member(X,[X|_]).
member(X,[_|Xs]) :-
  member(X,Xs).


append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
   append(Xs,Ys,Zs).
```

Παραδείγματα (3)

- Λίστες (συνέχεια)

```
?- length([1,2,3,4,5],N).
N = 5;
no
?- member(X,[1,2,3]).
X = 1;
X = 2;
X = 3;
no
?- append([1,2,3],[4,5,6],L).
L = [1,2,3,4,5,6];
no
```

Παραδείγματα (4)

- Λίστες (συνέχεια)

```
?- append(L,[4,5],[1,2,3,4,5]).
L = [1,2,3];
no
?- append([1|X],Y,[1,2,3]).
X = [], Y = [2,3];
X = [2], Y = [3];
X = [2,3], Y = [];
no
```

Παραδείγματα (4)

- Λίστες (συνέχεια)

```
?- append(X,Z,[1,2,3]).
X = [], Y = [1,2,3];
X = [1], Y = [2,3];
X = [1,2], Y = [3];
X = [1,2,3], Y = [];
no
```

Παραδείγματα (3)

- Λίστες (συνέχεια)

```
?- member(3,L).
L = [3|_];
L = [_,3|_];
L = [_,_,3|_];
L = [_,_,_,3|_];
L = [_,_,_,_,3|_];
L = [_,_,_,_,_,3|_];
L = [_,_,_,_,_,_,3|_]
```

... άπειρες λύσεις

Παραδείγματα (5)

- Λίστες και (*αφελής*) ταξινόμηση

```
sort(L,SL) :-
  permutation(L,SL), sorted(SL), !.
permutation(L,[H|T]) :-
  append(V,[H|U],L),
  append(V,U,W),
  permutation(W,T).
permutation([],[]).
sorted([]).
sorted([_]).
sorted([A,B|T]) :- A=<B, sorted([B|T]).
```

Σταματάει τη διαδικασία της αναζήτησης περισσότερων λύσεων

Παραδείγματα (6)

- Λίστες και (*αφελής*) ταξινόμηση (συνέχεια)

```
?- sort([42,13,77],L).
L = [13,42,77];
no
?- permutation([1,2,3],X).
X = [1,2,3];
X = [1,3,2];
X = [2,1,3];
X = [2,3,1];
X = [3,1,2];
X = [3,2,1];
no
```

Παραδείγματα (7)

- Λίστες και QuickSort

```
qsort([H|T],S) :-
   split(H,T,A,B),
   qsort(A,SA),
   qsort(B,SB),
   append(SA,[H|SB],S).
qsort([],[]).
split(H,[X|Xs],A,B) :-
   X=<H, !, A = [X|As], split(H,Xs,As,B).
split(H,[X|Xs],A,B) :-
   X>H, B = [X|Bs], split(H,Xs,A,Bs).
split(_,[],[],[]).
```

Σταματάει τη διαδικασία της αναζήτησης περισσότερων κανόνων