Εισαγωγή στη Γλώσσα Prolog

Tamara de Łempicka, *Jeune Fille Vert*, 1929

Κωστής Σαγώνας <kostis@cs.ntua.gr>
Νίκος Παπασπύρου <nickie@softlab.ntua.gr>

Περιεχόμενα

- Λογικός προγραμματισμός
- Όροι (terms)
- Χρήση ενός συστήματος Prolog
- Κανόνες (rules)
- Οι δύο όψεις της Prolog
- Τελεστές (operators)
- Λίστες (lists)
- Άρνηση ως αποτυχία (negation as failure)
- Σε τί είναι καλή η Prolog;

Λογικός προγραμματισμός

- Η χρήση της μαθηματικής λογικής στον προγραμματισμό
- Η λογική χρησιμοποιείται ως μια αγνή δηλωτική γλώσσα για την αναπαράσταση του προβλήματος
- Το έργο του προγραμματιστή είναι να γράφει προγράμματα που να είναι «αληθείς» αναπαραστάσεις του προβλήματος στη λογική
- Το σύστημα εκτέλεσης φροντίζει για την επίλυση του προβλήματος, με αποδοτικό τρόπο
  - αποδείκτες θεωρημάτων (theorem provers)
  - γεννήτριες μοντέλων (model generators)

Όροι (Terms)

- Κάθε τι στην Prolog κατασκευάζεται από όρους:
  - Τα προγράμματα σε Prolog
  - Τα δεδομένα που χειρίζονται τα προγράμματα σε Prolog
- Τρεις κατηγορίες όρων:
  - **Μεταβλητές**
  - **Σταθερές**: ακέραιοι (integers), πραγματικοί αριθμοί (reals), και άτομα (atoms)
  - **Σύνθετοι όροι**: λίστες (lists) και δομημένα δεδομένα (structures)

Άτομα (Atoms)

- Ένα άτομο είναι μια σταθερά που αποτελείται από αλφαριθμητικούς χαρακτήρες:

  `an`, **`atom`**, **`has`**, **`many_characters`**

- Εάν ο αρχικός χαρακτήρας δεν είναι ένα μικρό γράμμα ή υπάρχουν «περίεργοι» χαρακτήρες, το άτομο πρέπει να βρίσκεται μέσα σε αποστρόφους

  `'Kostis'`, **`'40 παλικάρια'`**, **`'$^%(@(&$#*&$'`**

- Κάθε άτομο έχει μια τιμή (τους χαρακτήρες του) και είναι ίσο μόνο με άλλα άτομα που έχουν ακριβώς τους ίδιους χαρακτήρες ως τιμή
- Σκεφτείτε τα άτομα κάτι σαν strings σε άλλες γλώσσες

Μεταβλητές

- Μεταβλητή είναι κάθε τι που αρχίζει από ένα κεφαλαίο γράμμα ή underscore, και ακολουθείται από γράμματα, αριθμούς ή underscores:

  `_`, `X`, `Variable`, `Kostis`, `_123`

- Οι μεταβλητές έχουν δύο καταστάσεις:
  - Μπορεί να μην έχουν ακόμα δεθεί με κάποια τιμή (unbound)
  - Μπορεί να έχουν κάποια τιμή, δηλαδή να είναι δεμένες (bound)
- Όταν όμως μια μεταβλητή είναι σε κατάσταση «με τιμή», η τιμή αυτή δεν αλλάζει

  (Όμως, όπως θα δούμε, η μεταβλητή μπορεί να επανέρθει σε κατάσταση unbound με χρήση οπισθοδρόμησης)

Σύνθετοι όροι (Compound terms)

- Αρχίζουν με ένα άτομο και ακολουθούνται από μια ακολουθία όρων που διαχωρίζονται με κόμματα και περικλείονται από παρενθέσεις:

```
university('Ε.Μ.Π.')
'+'(2,3.14)
couple(adam,eve)
tree(V,tree(42,Y,Z),R)
```

- Οι λίστες είναι και αυτές σύνθετοι όροι αλλά όπως θα δούμε έχουν ειδική σύνταξη
- Σκεφτείτε τους σύνθετους όρους σαν δομές δεδομένων

Σύνταξη των όρων

<term> **::=** *<constant>* | *<variable>* | *<compound-term>*
<constant> **::=** *<integer>* | *<real number>* | *<atom>*
<compound-term> **::=** *<atom>* **(** *<termlist>* **)**
<termlist> **::=** *<term>* | *<term>* **,** *<termlist>*

- Όλα τα προγράμματα και τα δεδομένα της Prolog κατασκευάζονται από τέτοιους όρους
- Σε λίγο θα δούμε ότι, για παράδειγμα, ο σύνθετος όρος `'+'(1,2)` συνήθως γράφεται ως `1+2`
- Αλλά αυτή είναι απλά μια συντομογραφία του ίδιου όρου

Ενοποίηση (Unification)

- Ταίριασμα προτύπων σε όρους της Prolog
- Δύο όροι *ενοποιούνται* εάν υπάρχει κάποιος τρόπος να βρεθούν τιμές για τις μεταβλητές τους που να τους κάνουν ακριβώς ίδιους
- Για παράδειγμα, οι όροι **`couple(adam,eve)`** και **`couple(Who,WithWhom)`** ενοποιούνται αν η μεταβλητή **`Who`** πάρει την τιμή **`adam`** και η μεταβλητή **`WithWhom`** πάρει την τιμή **`eve`**

- (Περισσότερες λεπτομέρειες στο επόμενο μάθημα…)

Η βάση δεδομένων της Prolog

- Η υλοποίηση της Prolog διατηρεί μια αναπαράσταση των προγραμμάτων που έχουν φορτωθεί στο σύστημα
- Ο κώδικας συμπεριφέρεται σαν μια βάση δεδομένων
- Ένα πρόγραμμα Prolog είναι απλώς ένα σύνολο από δεδομένα για αυτή τη βάση
- Το απλούστερο από τα δεδομένα αυτής της βάσης είναι ένα *γεγονός (fact)* : συντακτικά, ένα γεγονός είναι ένας όρος που ακολουθείται από μια τελεία

Παράδειγμα κατηγορήματος με γεγονότα

```
parent(kim, holly).
parent(margaret, kim).
parent(margaret, kent).
parent(esther, margaret).
parent(herbert, margaret).
parent(herbert, jean).
```

- Ένα πρόγραμμα Prolog με έξι γεγονότα
- Ορίζει ένα *κατηγόρημα (predicate)* **`parent/2`**
- Αποτελούμενο από γεγονότα που έχουν την «προφανή» ερμηνεία: ο **`kim`** είναι γονιός της **`holly`**, η **`margaret`** είναι γονιός του **`kim`**, κ.ο.κ.

Ο διερμηνέας της Prolog

```
% swipl

Welcome to SWI-Prolog (Multi-threaded, Version 5.10.4)
Copyright (c) 1990-2011 University of Amsterdam, ...
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. ...

Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

- Το `?-` είναι η προτροπή της Prolog για μια ερώτηση
- Ο διερμηνέας είναι διαδραστικός: δέχεται μια ερώτηση, προσπαθεί να την απαντήσει, τυπώνει την απάντησή της, και επαναλαμβάνει αυτή τη διαδικασία

Το κατηγόρημα `consult/1`

```
?- consult(relations).
% relations compiled 0.00 sec, XXXX bytes
true.

?-
```

- Κατηγόρημα του συστήματος με το οποίο φορτώνουμε ένα πρόγραμμα από ένα αρχείο στη βάση της Prolog
- Το αρχείο **`relations`** (ή **`relations.pl`**) περιλαμβάνει τα γεγονότα του κατηγορήματος **`parent/2`**

```
?- [relations].
% relations compiled 0.00 sec, XXXX bytes
true.
```

Απλές ερωτήσεις

```
?- parent(margaret, kent).
true.

?- parent(fred, pebbles).
false.

?-
```

- Μια ερώτηση προκαλεί την Prolog να αποδείξει κάτι
- Η απάντηση των απλών ερωτήσεων είναι **`true`** ή **`false`**

(Κάποιες ερωτήσεις, όπως η ερώτηση στο κατηγόρημα **`consult/1`**, εκτελούνται μόνο για τις παρενέργειές τους)

Η τελεία

```
?- parent(margaret, kent)
|     .
true.

?-
```

- Οι ερωτήσεις μπορούν να εκτείνονται σε πολλές γραμμές
- Εάν ξεχάσετε τη τελεία (το χαρακτήρα τέλους της ερώτησης), ο διερμηνέας της Prolog θα σας ζητήσει περισσότερη είσοδο με το χαρακτήρα |

Ερωτήσεις με μεταβλητές

- Οι ερωτήσεις μπορεί να είναι οποιοσδήποτε όρος Prolog, συμπεριλαμβανομένων όρων με μεταβλητές
- Ο διερμηνέας της Prolog επιστρέφει ως απάντηση τις τιμές των μεταβλητών για τις οποίες η ερώτηση μπορεί να απαντηθεί

```
?- parent(P, jean).
P = herbert.

?- parent(P, esther).
false.
```

Ευελιξία στις ερωτήσεις

- Οι μεταβλητές μπορούν φυσικά να βρίσκονται σε οποιαδήποτε θέση κάποιας ερώτησης:

```
parent(Parent, jean)
parent(esther, Child)
parent(Parent, Child)
parent(Person, Person)
```

Συζεύξεις (Conjunctions)

```
?- parent(margaret, X), parent(X, holly).
X = kim ;
false.

?-
```

- Μια σύζευξη ερωτήσεων είναι μια ακολουθία από όρους που διαχωρίζονται από κόμματα (and)
- Ο διερμηνέας της Prolog προσπαθεί να τις αποδείξει όλες (χρησιμοποιώντας το ίδιο σύνολο «δεσμάτων» για τις μεταβλητές της ερώτησης)

Πολλαπλές λύσεις

```
?- parent(margaret, Child).
Child = kim ;
Child = kent.

?-
```

- Μπορεί να υπάρχουν περισσότεροι από ένας τρόποι για να απαντηθεί μια ερώτηση
- Με το να πληκτρολογήσουμε `;` αντί για **`enter`**, μετά την πρώτη απάντηση, προτρέπουμε το σύστημα Prolog να βρει και να μας δώσει περισσότερες απαντήσεις

```
?- parent(Parent, kim), parent(GrandParent, Parent).
Parent = margaret
GrandParent = esther ;
Parent = margaret
GrandParent = herbert ;
false.


?- parent(esther, Child),
|  parent(Child, GrandChild),
|  parent(Grandchild, GreatGrandChild).
Child = margaret
GrandChild = kim
GreatGrandChild = holly ;
false.


?-
```

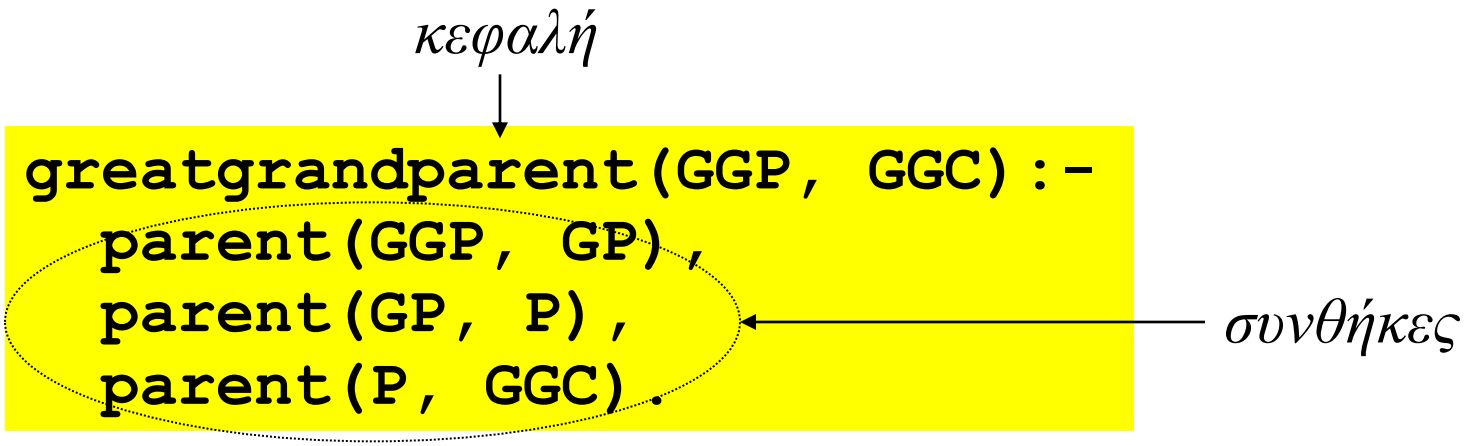
Ανάγκη για κανόνες

- Στο προηγούμενο παράδειγμα χρησιμοποιήσαμε μια μακροσκελή ερώτηση για τα δισέγγονα της **`esther`**
- Θα ήταν καλύτερο εάν μπορούσαμε να γράψουμε κατ' ευθείαν την ερώτηση:

```
greatgrandparent(esther, GGC)
```

- Αλλά δε θέλουμε να εισαγάγουμε ξεχωριστά γεγονότα αυτού του κατηγορήματος στη βάση των δεδομένων διότι η σχέση **`greatgrandparent/2`** προκύπτει (και πρέπει να προκύπτει) από την ήδη ορισμένη βασική σχέση **`parent/2`**

Ένα παράδειγμα κανόνα



- Ένας κανόνας καθορίζει πώς αποδεικνύουμε κάτι: για να αποδείξουμε την κεφαλή του κανόνα πρέπει να αποδείξουμε τις συνθήκες του
- Για να αποδείξουμε **greatgrandparent(GGP,GGC)**, πρέπει να βρούμε τιμές **GP** και **P** για τις οποίες μπορούμε να αποδείξουμε ότι **parent(GGP,GP)**, μετά ότι **parent(GP,P)** και τελικά ότι **parent(P,GGC)**

Πρόγραμμα με χρήση του κανόνα

```
parent(kim, holly).
parent(margaret, kim).
parent(margaret, kent).
parent(esther, margaret).
parent(herbert, margaret).
parent(herbert, jean).

greatgrandparent(GGP, GGC) :-
  parent(GGP, GP),
  parent(GP, P), parent(P, GGC).
```

- Ένα πρόγραμμα αποτελείται από μια ακολουθία από *προτάσεις (clauses)*
- Μια πρόταση είναι είτε ένα γεγονός ή ένας κανόνας

Παράδειγμα

```
?- greatgrandparent(esther, GreatGrandChild).
GreatGrandChild = holly ;
false.

?-
```

- Για την απάντηση της παραπάνω ερώτησης, εσωτερικά εξετάζονται διάφοροι ενδιάμεσοι *στόχοι (goals):*
  - Ο πρώτος (υπο)στόχος είναι η αρχική ερώτηση
  - Ο επόμενος είναι ό,τι απομένει προς απόδειξη μετά την απόδειξη του πρώτου (υπο)στόχου με χρήση κάποιας πρότασης του προγράμματος (στη συγκεκριμένη περίπτωση, με χρήση του κανόνα για το κατηγόρημα **`greatgrandparent/2`**)
  - Κοκ., μέχρι να μη μείνει τίποτα προς απόδειξη

```
1. parent(kim, holly).
2. parent(margaret, kim).
3. parent(margaret, kent).
4. parent(esther, margaret).
5. parent(herbert, margaret).
6. parent(herbert, jean).
7. greatgrandparent(GGP, GGC) :-
      parent(GGP, GP), parent(GP, P), parent(P, GGC).
```

```
greatgrandparent(esther, GreatGrandChild)
```

⇩ Clause 7, binding **GGP** to **esther** and **GGC** to **GreatGrandChild**

```
parent(esther,GP), parent(GP,P), parent(P,GreatGrandChild)
```

⇩ Clause 4, binding **GP** to **margaret**

```
parent(margaret, P), parent(P, GreatGrandChild)
```

⇩ Clause 2, binding **P** to **kim**

```
parent(kim, GreatGrandChild)
```

⇩ Clause 1, binding **GreatGrandChild** to **holly**

Κανόνες με χρήση άλλων κανόνων

```
grandparent(GP, GC) :-
  parent(GP, P), parent(P, GC).

greatgrandparent(GGP, GGC) :-
  grandparent(GGP, P), parent(P, GGC).
```

- Η ίδια σχέση με την προηγούμενη, ορισμένη σε στρώσεις
- Παρατηρήστε ότι και οι δύο κανόνες χρησιμοποιούν μεταβλητές με όνομα `P`
- Η εμβέλεια κάποιας μεταβλητής είναι μόνο ο κανόνας που την περιέχει
- Οι δύο μεταβλητές με όνομα `P` είναι ανεξάρτητες

Αναδρομικοί κανόνες

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

- Ένας **X** είναι πρόγονος κάποιου **Y**:
  - Εάν ο **X** είναι γονιός του **Y**, ή
  - Εάν υπάρχει κάποιος **Z** τέτοιος ώστε ο **Z** να είναι ο γονιός του **Y**, και ο **X** να είναι πρόγονος του **Z**
- Η Prolog δοκιμάζει τους κανόνες με τη σειρά που αυτοί εμφανίζονται, οπότε είναι καλό (και μάλιστα πολλές φορές αναγκαίο) τα γεγονότα και οι μη αναδρομικοί κανόνες να εμφανίζονται πριν από τους αναδρομικούς

```
?- ancestor(jean, jean).
false.

?- ancestor(kim, holly).
true.

?- ancestor(A, holly).
A = kim ;
A = margaret ;
A = esther ;
A = herbert ;
false.
```

Ο πυρήνας της σύνταξης της Prolog

- Ολόκληρη η σύνταξη του πυρήνα της Prolog:

```
<clause>   ::=   <fact>  |  <rule>
<fact>     ::=   <term>  .
<rule>     ::=   <term>  :-  <termlist>  .
<termlist> ::=   <term>  |  <term>  ,  <termlist>
```

- Εάν εξαιρεθεί η δυνατότητα ορισμού νέων τελεστών από το χρήστη, συντακτικά η Prolog είναι πολύ απλή γλώσσα
- Αυτό ήταν όλο!

Η διαδικαστική όψη της Prolog

```
greatgrandparent(GGP, GGC) :-
  parent(GGP, GP),
  parent(GP, P),
  parent(P, GGC).
```

- Ένας κανόνας λέει πώς μπορούμε να αποδείξουμε κάτι:
  - Για να αποδείξουμε **`greatgrandparent(GGP,GGC)`**, πρέπει να βρούμε τιμές **`GP`** και **`P`** για τις οποίες μπορούμε να αποδείξουμε ότι **`parent(GGP,GP)`**, μετά ότι **`parent(GP,P)`** και τελικά ότι **`parent(P,GGC)`**
- Ένα πρόγραμμα Prolog προσδιορίζει διαδικασίες εύρεσης απαντήσεων σε ερωτήσεις

Η δηλωτική όψη της Prolog

- Ένας κανόνας είναι μια λογική δήλωση:
  - Για όλες τις τιμές των **`GGP`**, **`GP`**, **`P`**, και **`GGC`**, αν ισχύει ότι **`parent(GGP,GP)`** και **`parent(GP,P)`** και **`parent(P,GGC)`**, τότε επίσης ισχύει και **`greatgrandparent(GGP,GGC)`**
- Με άλλα λόγια, ένας κανόνας είναι απλώς μια φόρμουλα
  - δε λέει πώς θα γίνει κάτι αλλά απλώς ορίζει τις προϋποθέσεις κάτω από τις οποίες μια πρόταση είναι αληθής:

$$\forall GGP, GP, P, GGC \;.\; \text{parent}(GGP, GP) \wedge \text{parent}(GP, P) \wedge \text{parent}(P, GGC) \Rightarrow \text{greatgrandparent}(GGP, GGC)$$

Δηλωτικές γλώσσες προγραμματισμού

- Κάθε κομμάτι του προγράμματος αντιστοιχεί σε μια απλή μαθηματική έννοια:
  - Προτάσεις της Prolog – σε φόρμουλες της κατηγορηματικής λογικής πρώτης τάξης (first-order predicate logic)
  - Οι ορισμοί συναρτήσεων στην ML – σε μαθηματικές συναρτήσεις
- Πολλές φορές ο όρος *δηλωτικές (declarative) γλώσσες προγραμματισμού* χρησιμοποιείται για να δείξει την αντιδιαστολή των γλωσσών αυτών σε σχέση με τις *προστακτικές (imperative) γλώσσες*
- Ως δηλωτικές γλώσσες συνήθως θεωρούμε τις λογικές και τις συναρτησιακές

Πλεονεκτήματα των δηλωτικών γλωσσών

- Τα προγράμματα σε προστακτικές γλώσσες πολλές φορές έχουν προβλήματα λόγω παρενεργειών και αλληλεξαρτήσεων
- Επειδή οι δηλωτικές γλώσσες έχουν σχετικά απλή και «καθαρότερη» σημασιολογία, αποφεύγονται κάποιου είδους προγραμματιστικά λάθη με συνέπεια η ανάπτυξη και η συντήρηση προγραμμάτων να γίνεται πιο εύκολα
- Ο προγραμματισμός είναι ανώτερου επιπέδου και πιο αυτόματος: ο προγραμματιστής απλώς περιγράφει την προδιαγραφή του προβλήματος και κάποια κομμάτια της εκτέλεσής του παρέχονται από το σύστημα υλοποίησης

Η Prolog έχει και τα δύο χαρακτηριστικά

- Έχει **δηλωτική σημασιολογία**
  - Ένα πρόγραμμα Prolog έχει ένα λογικό περιεχόμενο
- Έχει **διαδικαστική σημασιολογία**
  - Ένα πρόγραμμα Prolog έχει και καθαρά διαδικαστικά χαρακτηριστικά: τη σειρά με την οποία εμφανίζονται οι κανόνες, τη σειρά εμφάνισης των υποστόχων σ' ένα κανόνα, κατηγορήματα με παρενέργειες (π.χ. I/O), κ.λπ.
- Όμως είναι σημαντικό ο προγραμματιστής να είναι ενήμερος και να παίρνει υπόψη του και τις δύο όψεις της γλώσσας κατά τον προγραμματισμό

Τελεστές της Prolog

- Η Prolog έχει κάποιους προκαθορισμένους τελεστές (και επιτρέπει τον ορισμό νέων τελεστών από το χρήστη)
- Ένας τελεστής είναι απλώς ένα κατηγόρημα για το οποίο είναι δυνατή η χρήση κάποιας ειδικής συντακτικής συντομογραφίας στις χρήσεις του

Το κατηγόρημα =/2

- Ο στόχος **=(X,Y)** επιτυγχάνει αν και μόνον αν οι όροι **X** και **Y** μπορούν να ενοποιηθούν:

```
?- =(parent(adam,seth), parent(adam,X)).
X = seth.

?-
```

- Επειδή το άτομο **'='** είναι ορισμένο ως δυαδικός τελεστής, συνήθως γράφουμε το παραπάνω ως:

```
?- parent(adam,seth) = parent(adam,X).
X = seth.

?-
```

Αριθμητικοί τελεστές

- Τα άτομα `'+'`, `'-'`, `'*'` και `'/'` είναι ορισμένα ως τελεστές, με τη συνήθη προτεραιότητα και προσεταιριστικότητα

```
?- X = +(1,*(2,3)).
X = 1+2*3.

?- X = 1+2*3.
X = 1+2*3.

?-
```

Η Prolog μας επιτρέπει να χρησιμοποιήσουμε σύνταξη τελεστών στην είσοδο, και χρησιμοποιεί αυτή τη μορφή στην έξοδο των όρων, αλλά και στις δύο περιπτώσεις ο υποκείμενος όρος είναι ο `+(1,*(2,3))`

Μη αποτιμημένοι όροι

```
?- +(X,Y) = 1+2*3.
X = 1
Y = 2*3.

?- 7 = 1+2*3.
false.

?-
```

- Μετά την ενοποίηση, ο όρος παραμένει **+(1,*(2,3))**
- Με άλλα λόγια, ο όρος είναι μη αποτιμημένος

(Σε επόμενη διάλεξη θα δούμε ότι υπάρχει τρόπος στην Prolog να αποτιμήσουμε τέτοιους όρους)

Λίστες στην Prolog

- Συντακτικά είναι περίπου σαν τις λίστες στην ML
- Το άτομο `[]` αναπαριστά την κενή λίστα
- Το συναρτησιακό σύμβολο `'.'/2` αντιστοιχεί στον τελεστή `::` της ML

Έκφραση στην ML	Όρος στην Prolog
`[]`	`[]`
`1::[]`	`.(1,[])`
`1::2::3::[]`	`.(1,.(2,.(3,[])))`
Δεν υπάρχει αντίστοιχο.	`.(1,.(2,.(3.14,[])))`

Οι αναπαραστάσεις των λιστών στην Prolog

Λίστα	Όρος στον οποίο αντιστοιχεί
`[]`	`[]`
`[1]`	`.(1,[])`
`[1,2,3]`	`.(1,.(2,.(3,[])))`
`[1,2,3.14]`	`.(1,.(2,.(3.14,[])))`

- Η μορφή που συνήθως χρησιμοποιείται είναι η αριστερή
- Αλλά είναι απλώς μια συντομογραφία της μορφής που φαίνεται δεξιά στον παραπάνω πίνακα
- Η Prolog τυπώνει τις λίστες με χρήση της μορφής στο αριστερό μέρος του πίνακα

Παραδείγματα

```
?- X = .(1,.(2,.(3,[]))).
X = [1, 2, 3].

?- .(X,Y) = [1,2,3].
X = 1
Y = [2, 3].

?-
```

Αναπαραστάσεις λιστών με ουρά

Λίστα	Όρος στον οποίο αντιστοιχεί
`[1|X]`	`.(1,X)`
`[1,2|X]`	`.(1,.(2,X))`
`[1,2|[3,4]]`	Το ίδιο με `[1,2,3,4]`

- Πολλές φορές το τέλος της λίστας είναι το σύμβολο `|` ακολουθούμενο από την ουρά της λίστας
- Χρησιμοποιείται σε πρότυπα: ο όρος `[1,2|X]` ενοποιείται με κάθε λίστα που αρχίζει με `1,2` και έχει έναν όρο `X` ως ουρά

```
?- [1,2|X] = [1,2,3,4,5].
X = [3, 4, 5].
```

Το κατηγόρημα append/3

- Το προκαθορισμένο κατηγόρημα **append(X,Y,Z)** επιτυγχάνει αν και μόνο αν η λίστα **Z** είναι το αποτέλεσμα της συνένωσης της λίστας **Y** στο τέλος της λίστας **X**

```
?- append([1,2], [3,4], Z).
Z = [1, 2, 3, 4].

?-
```

Όχι απλώς μια συνάρτηση!

```
?- append(X, [3,4], [1,2,3,4]).
X = [1, 2] ;
false.

?-
```

- Το κατηγόρημα **append** μπορεί να χρησιμοποιηθεί με κάθε όρο της Prolog (με άλλα λόγια, μπορεί να κληθεί και με μεταβλητές σε κάθε όρισμα)

Όχι απλώς μια συνάρτηση!

```
?- append(X, Y, [1,2,3]).
X = []
Y = [1, 2, 3] ;
X = [1]
Y = [2, 3] ;
X = [1, 2]
Y = [3] ;
X = [1, 2, 3]
Y = [] ;
false.

?-
```

Η υλοποίηση του κατηγορήματος append/3

```
append([], L, L).
append([H|L1], L2, [H|L3]) :-
  append(L1, L2, L3).
```

Άλλα κατηγορήματα για λίστες

Κατηγόρημα	Περιγραφή
`member(X,Y)`	Επιτυγχάνει εάν η λίστα **Y** έχει ως μέλος τον όρο **X**.
`select(X,Y,Z)`	Επιτυγχάνει εάν η λίστα **Y** περιέχει το **X** ως μέλος, και ο όρος **Z** είναι ο ίδιος με τη λίστα **Y** αλλά χωρίς ένα στιγμιότυπο του όρου **X**.
`nth0(N,L,X)`	Επιτυγχάνει εάν ο **N** είναι ένας μη αρνητικός ακέραιος, **L** μια λίστα, και το **X** είναι το **N**-οστό στοιχείο της λίστας **L**, αρχίζοντας την αρίθμηση από το 0.
`length(L,N)`	Επιτυγχάνει εάν ο όρος **L** είναι μια λίστα με μήκος **N**.

- Όλα τα παραπάνω είναι ευέλικτα, σαν το **`append/3`**
- Με άλλα λόγια οι ερωτήσεις στα κατηγορήματα μπορούν να περιέχουν μεταβλητές σε οποιαδήποτε όρισμά τους

Χρήση του κατηγορήματος `select/3`

```
?- select(2, [1,2,3], Z).
Z = [1, 3] ;
false.

?-
```

```
?- select(2, Y, [1,3]).
Y = [2, 1, 3] ;
Y = [1, 2, 3] ;
Y = [1, 3, 2] ;
false.

?-
```

```
?- select(2, [1,2,3,2], Z).
Z = [1, 3, 2] ;
Z = [1, 2, 3] ;
false.

?-
```

```
?- select(2, [X,Y], [1]).
X = 2
Y = 1 ;
X = 1
Y = 2 ;
false.

?-
```

Το κατηγόρημα `reverse/2`

- **`reverse(X,Y)`** είναι αληθές εάν ο όρος **`Y`** ενοποιείται με την αναστροφή της λίστας **`X`**

```
?- reverse([1,2,3,4], Y).
Y = [4, 3, 2, 1] ;
false.

?-
```

Η υλοποίηση του κατηγορήματος `reverse/2`

```
reverse([], []).
reverse([Head|Tail], Reversed) :-
  reverse(Tail, RevTail),
  append(RevTail, [Head], Reversed).
```

- Δεν είναι ένας αποδοτικός τρόπος να αντιστρέψουμε μια λίστα
- Υπάρχουν πιο αποδοτικοί τρόποι αντιστροφής λιστών (όπως ακριβώς είδαμε και στην ML)

Όταν δεν πάνε όλα καλά…

```
?- reverse(L, [1,2,3,4]).

L = [4, 3, 2, 1] ;

Action (h for help) ? a
% Execution Aborted
?-
```

- Για το (naive) **reverse/2** της προηγούμενης διαφάνειας, ζητήσαμε μια ακόμα λύση και βρεθήκαμε σ' έναν ατέρμονο βρόχο
- Μπορούμε να παύσουμε την εκτέλεση πατώντας **Ctrl-C**, και στη συνέχεια πατάμε **a** για το οριστικό σταμάτημά της
- Το **reverse/2** δεν είναι τόσο ευέλικτο όσο το **append/3**

Αναστρέψιμα και μη αναστρέψιμα κατηγορήματα

- Σ' έναν ιδεατό κόσμο, τα κατηγορήματα είναι όλα τόσο ευέλικτα όσο το **`append/3`**
- Είναι πιο δηλωτικά και με λιγότερους περιορισμούς στη χρήση τους
- Αλλά πολλές φορές μη ευέλικτες υλοποιήσεις είναι προτιμότερες, για λόγους απόδοσης ή/και απλότητας
- Ένα παράδειγμα του τι σημαίνει η παραπάνω φράση μας δείχνει το ενσωματωμένο κατηγόρημα **`sort/2`**

Το κατηγόρημα `sort/2`

```
?- sort([2,3,1,4], X).
X = [1, 2, 3, 4].


?- sort(X, [1,2,3,4]).
ERROR: Arguments are not sufficiently instantiated
```

- Ένα αναστρέψιμο κατηγόρημα **`sort/2`** θα έπρεπε να μπορεί να «αντι-ταξινομήσει» μια ήδη ταξινομημένη λίστα – με άλλα λόγια θα έπρεπε να επιστρέφει όλες τις αναδιατάξεις των στοιχείων της – που είναι εκθετικό
- Θέλουμε κάτι σαν το παραπάνω να είναι ενσωματωμένο σε ένα βασικό κατηγόρημα της γλώσσας;

Ανώνυμες μεταβλητές

- Η μεταβλητή _ είναι μια μεταβλητή χωρίς όνομα
- Κάθε εμφάνισή της είναι ανεξάρτητη από όλες τις υπόλοιπες
- Ενοποιείται με κάθε όρο
- Με άλλα λόγια, συμπεριφέρεται σαν τις μεταβλητές _ της ML

Το κατηγόρημα member/2

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

```
?- member(b, [a,b,c]).
true ;
false.

?- member(d, [a,b,c]).
false.

?- member(X, [a,b]).
X = a ;
X = b.

?-
```

Προσοχή στις προειδοποιήσεις!

```
append([], L, L).
append([H|L1], L2, [H|L3]) :-
  append(LI, L2, L3).
```

```
Warning: Singleton variables [L1,LI]
```

- Μην αγνοείτε τα προειδοποιητικά μηνύματα ότι μεταβλητές εμφανίζονται μόνο μια φορά
- Όπως και στην ML, είναι κακό στυλ προγραμματισμού να ονομάζουμε μεταβλητές που δεν χρησιμοποιούμε - ειδικά επειδή είναι το μόνο μήνυμα που ο compiler θα βγάλει

Το κατηγόρημα \+/1

```
?- member(1, [1,2,3]).
true ;
false.

?- \+ member(4, [1,2,3]).
true.

?-
```

- Σε απλές εφαρμογές, συνήθως λειτουργεί σαν τον τελεστή λογικής άρνησης
- Αλλά έχει και μια σημαντική διαδικαστική πλευρά…

Άρνηση ως αποτυχία (negation as failure)

- Για να αποδείξει το `\+ X`, η Prolog προσπαθεί να αποδείξει το **X**
- `\+ X` επιτυγχάνει εάν ο στόχος **X** αποτυγχάνει
- Οι δύο όψεις ξανά:
  - **Δηλωτική**: `\+ X` = ¬**X**
  - **Διαδικαστική**: `\+ X` επιτυγχάνει εάν το **X** αποτυγχάνει, και αποτυγχάνει εάν το **X** επιτυγχάνει, και δεν τερματίζει εάν το **X** δεν τερματίζει

Παράδειγμα

```
sibling(X, Y) :-
  \+ (X = Y),
  parent(P, X),
  parent(P, Y).
```

```
sibling(X, Y) :-
  parent(P, X),
  parent(P, Y),
  \+ (X = Y).
```

```
?- sibling(kim, kent).

true.

?- sibling(kim, kim).

false.

?- sibling(X, Y).

false.
```

```
?- sibling(X, Y).

X = kim
Y = kent ;

X = kent
Y = kim ;

X = margaret
Y = jean ;

X = jean
Y = margaret ;

false.
```

Ένα μεγαλύτερο παράδειγμα σε Prolog

Μια κλασική σπαζοκεφαλιά

- Ένας άνθρωπος (man) ταξιδεύει μαζί μ' ένα λύκο (wolf), μια κατσίκα (goat) και ένα λάχανο (cabbage)
- Ο άνθρωπος θέλει να διασχίσει ένα ποτάμι από τη δυτική όχθη στην ανατολική
- Στην αρχική όχθη υπάρχει μια βάρκα, η οποία όμως χωράει τον άνθρωπο και το πολύ ένα άλλο αντικείμενο
- Ο λύκος τρώει την κατσίκα εάν βρεθεί μόνος μαζί της
- Η κατσίκα τρώει το λάχανο εάν βρεθεί μόνη μαζί του
- Με ποιο τρόπο μπορεί ο άνθρωπος να περάσει στην απέναντι όχθη με όλα τα υπάρχοντά του;

Δομές δεδομένων της λύσης

- Θα αναπαραστήσουμε τους σχηματισμούς του συστήματος με μια λίστα που δείχνει την όχθη στην οποία είναι το κάθε τι με την εξής σειρά:

  man, wolf, goat, cabbage

- Ο αρχικός σχηματισμός: **`[w,w,w,w]`**
- Εάν για παράδειγμα ο άνθρωπος με το λύκο μπουν στη βάρκα, ο νέος σχηματισμός είναι ο **`[e,e,w,w]`** - αλλά τότε η κατσίκα θα φάει το λάχανο,οπότε ο παραπάνω σχηματισμός δεν είναι επιτρεπτός ως ενδιάμεσος
- Ο επιθυμητός τελικός σχηματισμός: **`[e,e,e,e]`**

Κινήσεις

- Σε κάθε κίνηση, ο άνθρωπος διασχίζει το ποτάμι με το πολύ ένα από τα υπάρχοντά του
- Θα αναπαραστήσουμε τις τέσσερις επιτρεπτές κινήσεις με τέσσερα άτομα: **`wolf`**, **`goat`**, **`cabbage`**, **`nothing`**
- (Όπου, **`nothing`** σημαίνει ότι ο άνθρωπος διασχίζει το ποτάμι μόνος του μέσα στη βάρκα)

Οι κινήσεις τροποποιούν τον σχηματισμό

- Κάθε κίνηση μεταμορφώνει ένα σχηματισμό σ' έναν άλλο
- Στην Prolog, θα προγραμματίσουμε τη λύση με ένα κατηγόρημα: **`move(Config, Move, NextConfig)`**
  - **`Config`** είναι ένας σχηματισμός (π.χ. **`[w,w,w,w]`**)
  - **`Move`** είναι μια κίνηση (π.χ. **`wolf`**)
  - **`NextConfig`** είναι ο σχηματισμός-αποτέλεσμα (στο συγκεκριμένο παράδειγμα ο **`[e,e,w,w]`**)

Το κατηγόρημα move/3

```
change(e, w).
change(w, e).

move([X,X,Goat,Cabbage], wolf, [Y,Y,Goat,Cabbage]) :-
  change(X, Y).
move([X,Wolf,X,Cabbage], goat, [Y,Wolf,Y,Cabbage]) :-
  change(X, Y).
move([X,Wolf,Goat,X], cabbage, [Y,Wolf,Goat,Y]) :-
  change(X, Y).
move([X,Wolf,Goat,Cab], nothing, [Y,Wolf,Goat,Cab]) :-
  change(X, Y).
```

Ασφαλείς σχηματισμοί

- Ένας σχηματισμός είναι ασφαλής εάν
  - Ο λύκος και ή κατσίκα είτε φυλάσσονται από τον άνθρωπο ή είναι σε διαφορετικές όχθες, και
  - Η κατσίκα και το λάχανο είτε φυλάσσονται από τον άνθρωπο ή είναι σε διαφορετικές όχθες.

```
guarded_or_separated(X, X, X).
guarded_or_separated(_, Y, Z) :- Y \= Z.

safe([Man, Wolf, Goat, Cabbage]) :-
  guarded_or_separated(Man, Goat, Wolf),
  guarded_or_separated(Man, Goat, Cabbage).
```

Λύσεις

- Μια λύση αρχίζει από τον αρχικό σχηματισμό και δημιουργεί μια λίστα από κινήσεις που οδηγούν στο σχηματισμό `[e,e,e,e]`, έτσι ώστε όλοι οι ενδιάμεσοι σχηματισμοί να είναι ασφαλείς

```
solution([e,e,e,e], []).
solution(Config, [Move|Moves]) :-
  move(Config, Move, NextConfig),
  safe(NextConfig),
  solution(NextConfig, Moves).
```

Το κατηγόρημα `length/2`

```
?- length(L, N).
L = []
N = 0 ;

L = [_]
N = 1 ;

L = [_, _]
N = 2 ;

L = [_, _, _]
N = 3 ;

L = [_, _, _, _]
N = 4 ;

... άπειρες λύσεις ...
```

Η Prolog βρίσκει τη λύση

```
?- length(L, N), solution([w,w,w,w], L).

L = [goat, nothing, wolf, goat, cabbage, nothing, goat]
N = 7 .

?-
```

Σημείωση: χωρίς τη χρήση του **`length/2`**, η Prolog δε θα έβρισκε κάποια λύση (με το συγκεκριμένο πρόγραμμα). Θα μπερδευόταν ψάχνοντας για λύσεις της μορφής:

`[goat,goat,goat,goat,goat…]`

Κάποια καλά χαρακτηριστικά της Prolog

- Το πρόγραμμά μας απλώς κατέγραψε την εκφώνηση του προβλήματος προς επίλυση
- Δεν προσδιορίσαμε κάποιο τρόπο αναζήτησης της λύσης του προβλήματος - η ίδια η Prolog έκανε την αναζήτηση
- Αυτού του είδους τα προβλήματα είναι ακριβώς τα προβλήματα που είναι κατάλληλα προς επίλυση σε Prolog

- Περισσότερα παραδείγματα σε επόμενες διαλέξεις…