Περισσότερη Prolog

Tamara de Łempicka

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

Περιεχόμενα

- Ενοποίηση
- Μοντέλα εκτέλεσης της Prolog
  - Το διαδικαστικό μοντέλο
  - Το μοντέλο υλοποίησης
  - Το αφηρημένο μοντέλο
- Περισσότερη Prolog
  - Κατηγορήματα εισόδου και εξόδου
  - Κατηγορήματα διαχείρισης της βάσης της Prolog
  - Αριθμητικοί υπολογισμοί και συγκρίσεις στην Prolog
- Παραδείγματα προγραμματισμού σε Prolog
- Αποχαιρετισμός στην Prolog

Αντικαταστάσεις (Substitutions)

- Μια *αντικατάσταση* είναι μια συνάρτηση που απεικονίζει μεταβλητές σε όρους:

  σ = {**`X`** → **`a`**, **`Y`** → **`f(a,b)`**}

- Η αντικατάσταση σ αντιστοιχεί τη μεταβλητή **`X`** στο **`a`** και τη **`Y`** στο **`f(a,b)`**
- Το αποτέλεσμα της εφαρμογής μιας αντικατάστασης σε έναν όρο δημιουργεί ένα *στιγμιότυπο* του όρου
- Για παράδειγμα σ(**`g(X,Y)`**) = **`g(a,f(a,b))`**
- Ο όρος **`g(a,f(a,b))`** είναι στιγμιότυπο του **`g(X,Y)`**

Ενοποίηση

- Δύο όροι της Prolog t_1 και t_2 *ενοποιούνται* εάν υπάρχει κάποια αντικατάσταση σ (ο *ενοποιητής* τους) που κάνει τους δύο όρους ακριβώς τους ίδιους: σ(t_1) = σ(t_2)
  - Οι όροι **`a`** και **`b`** δεν ενοποιούνται
  - Οι όροι **`f(X,b)`** και **`f(a,Y)`** ενοποιούνται: ένας ενοποιητής είναι ο {**`X`** → **`a`**, **`Y`** → **`b`**}
  - Οι όροι **`f(X,b)`** και **`g(Y,b)`** δεν ενοποιούνται
  - Οι όροι **`a(X,X,b)`** και **`a(b,Y,Y)`** ενοποιούνται: ένας ενοποιητής είναι ο {**`X`** → **`b`**, **`Y`** → **`b`**}
  - Οι όροι **`a(X,X,b)`** και **`a(c,Y,Y)`** δεν ενοποιούνται
  - Οι όροι **`a(X,f)`** και **`a(Y,f)`** ενοποιούνται: ένας ενοποιητής είναι ο {**`X`** → **`42`**, **`Y`** → **`42`**}

Πολλαπλοί ενοποιητές

- `parent(X,Y)` και `parent(fred,Z)`:
  - Ένας ενοποιητής είναι ο $\sigma_1 = \{X \to fred,\ Y \to Z\}$
  - Άλλος ένας είναι ο $\sigma_2 = \{X \to fred, Y \to mary, Z \to mary\}$
  - Άλλος ένας είναι ο $\sigma_3 = \{X \to fred, Y \to foo(42), Z \to foo(42)\}$
- Η Prolog επιλέγει ενοποιητές όπως ο σ_1 οι οποίοι καθορίζουν ακριβώς τις αντικαταστάσεις που είναι αναγκαίες για την ενοποίηση των δύο όρων
- Με άλλα λόγια, η Prolog επιλέγει τον πιο γενικό ενοποιητή των δύο όρων (MGU – Most General Unifier)

Ο πιο γενικός ενοποιητής (ΠΓΕ)

- Ο όρος t_1 είναι πιο γενικός από τον όρο t_2 εάν ο t_2 είναι στιγμιότυπο του t_1 αλλά ο t_1 δεν είναι στιγμιότυπο του t_2
  - Παράδειγμα: ο όρος `parent(fred,Y)` είναι πιο γενικός από τον όρο `parent(fred,mary)`
- Ένας ενοποιητής σ_1 δύο όρων t_1 και t_2 είναι ο πιο γενικός ενοποιητής εάν δεν υπάρχει άλλος ενοποιητής σ_2 τέτοιος ώστε ο όρος $\sigma_2(t_1)$ να είναι πιο γενικός από τον όρο $\sigma_1(t_1)$
- Μπορεί να αποδειχθεί ότι ο πιο γενικός ενοποιητής είναι μοναδικός (αν αγνοήσουμε τα ονόματα των μεταβλητών)

Η Prolog χρησιμοποιεί ενοποίηση για τα πάντα

- Πέρασμα παραμέτρων
  - `reverse([1,2,3],X)`
- Δέσιμο μεταβλητών
  - `X = 0`
- Κατασκευή δεδομένων
  - `X = .(1,[2,3])`
- Επιλογή δεδομένων
  - `[1,2,3] = .(X,Y)`

Έλεγχος εμφάνισης (Occurs check)

- Μια μεταβλητή `X` και ένας όρος *t* ενοποιούνται με την αντικατάσταση $\{X \to t\}$:
  - `X` και `b` ενοποιούνται: ο ΠΓΕ είναι $\{X \to b\}$
  - `X` και `f(a,g(b))` ενοποιούνται: ο ΠΓΕ είναι $\{X \to f(a,g(b))\}$
  - `X` και `f(a,Y)` ενοποιούνται: ο ΠΓΕ είναι $\{X \to f(a,Y)\}$
- *Εκτός εάν η μεταβλητή* `X` *περιλαμβάνεται στον όρο t:*
  - Οι όροι `X` και `f(a,g(X))` δεν ενοποιούνται: η αντικατάσταση $\{X \to f(a,g(X))\}$ δεν είναι ενοποιητής
- Με άλλα λόγια, τουλάχιστον στη θεωρία, η ενοποίηση πρέπει να είναι καλά θεμελιωμένη (well-founded)

Ενοποίηση χωρίς έλεγχο εμφάνισης

- Η default ενοποίηση της Prolog δεν περιλαμβάνει έλεγχο εμφάνισης

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

```
?- append([], X, [a | X]).

X = [a, a, a, a, a, a, a, a, a|...] ;

No
```

- Αλλά το ISO Prolog standard περιλαμβάνει ένα κατηγόρημα με όνομα **`unify_with_occurs_check/2`**

Μοντέλα Εκτέλεσης της Prolog

Το διαδικαστικό μοντέλο εκτέλεσης της Prolog

- Κάθε κατηγόρημα είναι μια διαδικασία για την απόδειξη στόχων
  - **`p :- q, r.`**
    - Για την απόδειξη του στόχου **`p`**, πρώτα ενοποίησε το στόχο με την κεφαλή του κανόνα **`p`**, μετά απόδειξε το **`q`**, και μετά απόδειξε το **`r`**
  - **`s.`**
    - Για την απόδειξη του στόχου **`s`**, ενοποίησε το στόχο με το **`s`**
- Η κάθε πρόταση (γεγονός ή κανόνας) αποτελεί ένα διαφορετικό τρόπο απόδειξης του στόχου
- Η απόδειξη μπορεί να περιλαμβάνει κλήσεις σε άλλα κατηγορήματα-διαδικασίες

Παραδείγματα

```
p :- q, r.        boolean p() {return q() && r();}
q :- s.           boolean q() {return s();}
r :- s.           boolean r() {return s();}
s.                boolean s() {return true;}
```

```
p :- p.           boolean p() {return p();}
```

Οπισθοδρόμηση (Backtracking)

- Μια περιπλοκή: οπισθοδρόμηση
- Η Prolog κρατάει μια λίστα με όλους τους δυνατούς τρόπους με τους οποίους μπορεί να ικανοποιηθεί κάποιος στόχος και τους δοκιμάζει με τη σειρά μέχρι να ικανοποιήσει πλήρως το στόχο ή μέχρι να εξαντλήσει όλες τις δυνατότητες ικανοποίησής του
- Έστω ο στόχος **`p`** στο παρακάτω πρόγραμμα: ο στόχος ικανοποιείται αλλά μόνο με χρήση οπισθοδρόμησης

```
p :- q, r.
q :- s.
q.
r.
s :- 0 = 1.
```


Αντικατάσταση

- Άλλη μια περιπλοκή: αντικατάσταση μεταβλητών
- Μια κρυμμένη ροή πληροφορίας

```
p(f(Y)) :- q(Y) , r(Y) .
```

Η αντικατάσταση σ_1 = MGU(**`p(f(Y))`**,*t*) εφαρμόζεται σε όλες τις μεθεπόμενες συνθήκες προς ικανοποίηση της πρότασης

Ο αρχικός όρος προς απόδειξη *t*

Το ίδιο συμβαίνει και με την αντικατάσταση σ_2 η οποία προκύπτει από την διαδικασία απόδειξης του στόχου σ_1(**`q(Y)`**)

Η αντικατάσταση σ_3 προκύπτει από την απόδειξη του όρου σ_2(σ_1(**`r(Y)`**))

Η σύνθεση των αντικαταστάσεων επιστρέφεται στον καλούντα

Ο όρος που αποδείχθηκε είναι ο $\sigma_3(\sigma_2(\sigma_1(t)))$

Το μοντέλο υλοποίησης: Επίλυση (Resolution)

- Το βασικό βήμα συμπερασμού
- Κάθε κανόνας αναπαριστάται με μια λίστα από όρους (κάθε γεγονός αναπαριστάται από λίστα ενός στοιχείου)
- Κάθε βήμα επίλυσης χρησιμοποιεί μια από αυτές τις λίστες, μια φορά, για να επιτύχει κάποια πρόοδο στην απόδειξη μιας λίστας από όρους που είναι στόχοι προς απόδειξη για την απάντηση κάποιας ερώτησης

function **`resolution`**(**`clause`**, **`goals`**):
  let **`sub`** = the MGU of head(**`clause`**) and head(**`goals`**)
  return **`sub`**(body(**`clause`**) concatenated with tail(**`goals`**))

Παράδειγμα επίλυσης

Για την παρακάτω λίστα όρων προς απόδειξη:

```
[p(X),s(X)]
```

και τον κανόνα:

```
p(f(Y)) :- q(Y), r(Y).
```

Ο ΠΓΕ είναι ο {**`X`**$\rightarrow$**`f(Y)`**}, και στο επόμενο βήμα προκύπτει η λίστα αποτελούμενη από τους όρους:

```
resolution([p(f(Y)),q(Y),r(Y)], [p(X),s(X)])
  = [q(Y),r(Y),s(f(Y))]
```

function **`resolution`**(**`clause`**, **`goals`**):
  let **`sub`** = the MGU of head(**`clause`**) and head(**`goals`**)
  return **`sub`**(body(**`clause`**) concatenated with tail(**`goals`**))

Ένας διερμηνέας της Prolog

```
function solve(goals)
    if goals is empty then succeed()
    else for each clause c in the program, in order
        if head(c) does not unify with head(goals) then do nothing
        else solve(resolution(c, goals))
```


Παράδειγμα

```
1. p(f(Y)) :-
         q(Y),r(Y).
2. q(g(Z)).
3. q(h(Z)).
4. r(h(a)).
```

Μια μερική επίλυση του **p(X)**:

```
solve([p(X)])
 1. solve([q(Y),r(Y)])
        …
 2. nothing
 3. nothing
 4. nothing
```

- Η συνάρτηση **solve** δοκιμάζει τις τέσσερις προτάσεις τη μία μετά την άλλη
  - Η πρώτη πρόταση ταιριάζει, και η συνάρτηση **solve** καλεί τον εαυτό της αναδρομικά με το αποτέλεσμα της διάλυσης
  - Οι άλλες τρεις προτάσεις δεν ταιριάζουν: οι κεφαλές τους δεν ενοποιούνται με τον όρο της λίστας

Παραδείγματος συνέχεια

```
1. p(f(Y)) :-
         q(Y),r(Y).
2. q(g(Z)).
3. q(h(Z)).
4. r(h(a)).
```

Μια μερική επίλυση του **p(X)**, επεκταμένη:

```
solve([p(X)])
 1. solve([q(Y),r(Y)])
     1. nothing
     2. solve([r(g(Z))])
            …
     3. solve([r(h(Z))])
            …
     4. nothing
 2. nothing
 3. nothing
 4. nothing
```


Το παράδειγμα ολοκληρωμένο

```
1. p(f(Y)) :-
         q(Y),r(Y).
2. q(g(Z)).
3. q(h(Z)).
4. r(h(a)).
```

Η ολική επίλυση της ερώτησης **p(X)**:

```
solve([p(X)])
 1. solve([q(Y),r(Y)])
     1. nothing
     2. solve([r(g(Z))])
         1. nothing
         2. nothing
         3. nothing
         4. nothing
     3. solve([r(h(Z))])
         1. nothing
         2. nothing
         3. nothing
         4. solve([]) —success!
     4. nothing
 2. nothing
 3. nothing
 4. nothing
```

Συλλογή των αντικαταστάσεων

```
function resolution(clause, goals, query):
    let sub = the MGU of head(clause) and head(goals)
    return (sub(tail(clause) concatenated with tail(goals)), sub(query))

function solve(goals, query)
    if goals is empty then succeed(query)
    else for each clause c in the program, in order
        if head(c) does not unify with head(goals) then do nothing
        else solve(resolution(c, goals, query))
```

- Τροποποιημένη συνάρτηση που δέχεται ως όρισμα την αρχική ερώτηση και εφαρμόζει όλες τις αντικαταστάσεις πάνω της
- Στο τέλος της επίλυσης, το αποδειχθέν στιγμιότυπο περνιέται ως όρισμα στη συνάρτηση **succeed**


```
1. p(f(Y)) :-
       q(Y),r(Y).
2. q(g(Z)).
3. q(h(Z)).
4. r(h(a)).
```

Η ολική επίλυση της ερώτησης **p(X)**:

```
solve([p(X)],p(X))
 1. solve([q(Y),r(Y)],p(f(Y)))
     1. nothing
     2. solve([r(g(Z))],p(f(g(Z))))
         1. nothing
         2. nothing
         3. nothing
         4. nothing
     3. solve([r(h(Z))],p(f(h(Z))))
         1. nothing
         2. nothing
         3. nothing
         4. solve([],p(f(h(a))))
     4. nothing
 2. nothing
 3. nothing
 4. nothing
```

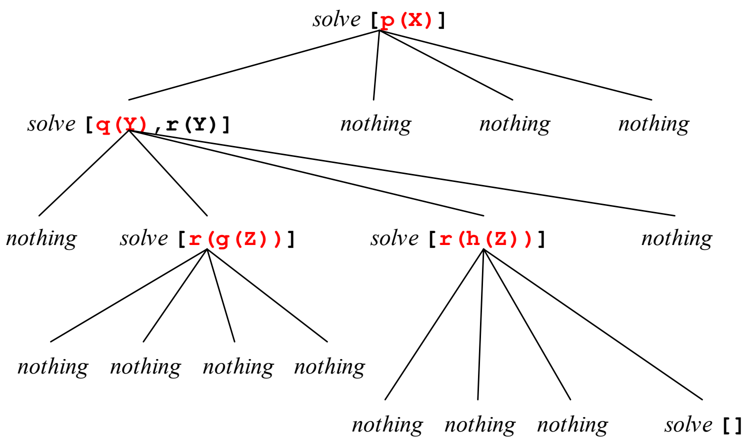

Διερμηνείς της Prolog

- Ο διερμηνέας που μόλις είδαμε είναι λειτουργικός αλλά δεν χρησιμοποιείται από τις υλοποιήσεις της Prolog
- Όλες οι υλοποιήσεις της Prolog πρέπει μεν να λειτουργήσουν με τον τρόπο που μόλις περιγράψαμε αλλά συνήθως χρησιμοποιούν εντελώς διαφορετικές τεχνικές υλοποίησης: μεταφράζουν τον κώδικα σε γλώσσα κάποιας αφηρημένης μηχανής
- Η πιο συνηθισμένη τέτοια μηχανή είναι η Αφηρημένη Μηχανή του Warren (Warren Abstract Machine)

Το αφηρημένο μοντέλο: Δένδρα απόδειξης

- Θέλουμε να αναπαραστήσουμε με κάποιο τρόπο τη σειρά των λειτουργιών της εκτέλεσης, όμως χωρίς να περιορίζεται η τεχνική της υλοποίησης
- Τα δένδρα απόδειξης αποτελούν έναν τέτοιο φορμαλισμό:
  - Η ρίζα είναι η αρχική ερώτηση
  - Οι κόμβοι του δένδρου είναι λίστες από όρους προς απόδειξη, και κάθε κόμβος έχει ένα παιδί για κάθε πρόταση του προγράμματος

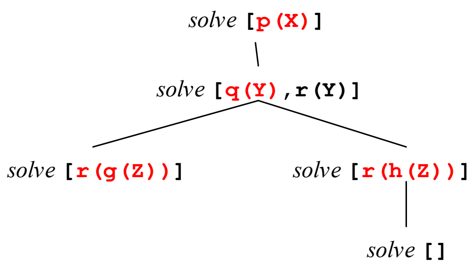
Παράδειγμα



Απλοποίηση των δένδρων απόδειξης

- Τα παιδιά κάθε κόμβου αντιστοιχούν στις προτάσεις του προγράμματος
- Και η σειρά τους είναι ίδια με τη σειρά εμφάνισής τους στο πρόγραμμα
- Μπορούμε να απλοποιήσουμε το δένδρο με το να απαλείψουμε όλους τους *nothing* κόμβους
- Οι κόμβοι αυτοί αντιστοιχούν σε προτάσεις οι οποίες δεν ενοποιούνται με το πρώτο στοιχείο της λίστας του κόμβου πατέρα

Παράδειγμα απλοποιημένου δένδρου απόδειξης



Σημασιολογία της Prolog

- Δοθέντος ενός προγράμματος και μιας ερώτησης, ένα σύστημα Prolog πρέπει να λειτουργήσει με τρόπο που καθορίζεται από μια πρώτα κατά βάθος, αριστερά προς τα δεξιά διάσχιση (depth-first, left-to-right traversal) του δένδρου απόδειξης
- Ένας τρόπος υλοποίησης είναι μέσω κάποιου διερμηνέα όπως αυτός ορίζεται από τη συνάρτηση **`solve`**
- Στην πράξη συνήθως χρησιμοποιούνται άλλοι, πιο αποδοτικοί, τρόποι υλοποίησης

Κατηγορήματα εισόδου και εξόδου

- Η είσοδος και η έξοδος όρων γίνεται σχετικά απλά

```
?- write('Hello world').
Hello world

Yes
?- read(X).
|: hello(world).

X = hello(world) ;

No
```

- Οποιοσδήποτε όρος της Prolog μπορεί να διαβαστεί
- Υπάρχει επίσης το κατηγόρημα **`nl/0`** το οποίο είναι ισοδύναμο με την κλήση **`write('\n')`**

Το κατηγόρημα assert/1

- Προσθέτει ένα γεγονός ή έναν κανόνα στην εσωτερική βάση δεδομένων της Prolog (στο τέλος της βάσης)

```
?- dynamic parent/2.

Yes
?- parent(X,Y).

No
?- assert(parent(joe,mary)).

Yes
?- parent(X,Y).

X = joe
Y = mary ;

No
```


Το κατηγόρημα retract/1

- Απομακρύνει την πρώτη πρόταση (κανόνα ή γεγονός) που ενοποιείται με το όρισμά του από τη βάση των δεδομένων

```
?- parent(joe, mary).

Yes
?- retract(parent(joe, mary)).

Yes
?- parent(joe, mary).

No
```

- Υπάρχει επίσης και το κατηγόρημα **`retractall/1`** το οποίο απομακρύνει όλες τις προτάσεις που ενοποιούνται με το όρισμά του

Τα κατηγορήματα listing/0 και listing/1

- Τυπώνουν την εσωτερική βάση δεδομένων της Prolog:
  - όλη (**`listing/0`**) ή
  - μόνο κάποιο συγκεκριμένο κατηγόρημα (**`listing/1`**)

```
?- dynamic parent/2.

Yes
?- assert(parent(joe,mary)).

Yes
?- listing(parent/2).
:- dynamic parent/2.

parent(joe, mary).

Yes
```

Μη αποτιμημένοι όροι

- Οι τελεστές της Prolog επιτρέπουν πιο συμπυκνωμένη γραφή των όρων, αλλά οι όροι δεν αποτιμώνται
- Όλες οι παρακάτω γραφές είναι ο ίδιος όρος Prolog:

```
1+ *(2,3)
+(1,2*3)
(1+(2*3))
1+2*3
```

- Ο παραπάνω όρος *δεν* ενοποιείται με τον όρο **7**

Αποτίμηση αριθμητικών εκφράσεων

```
?- X is 1+2*3.

X = 7 ;

No
```

- Το προκαθορισμένο κατηγόρημα **is/2** μπορεί να χρησιμοποιηθεί για την αποτίμηση ενός όρου που είναι μια αριθμητική έκφραση
- **is(X,Y)** αποτιμά τον όρο **Y** και ενοποιεί τον όρο **X** με το αποτέλεσμα της αποτίμησης
- Συνήθως χρησιμοποιείται ως δυαδικός τελεστής

Η αποτίμηση προϋποθέτει τιμές…

```
?- Y = X+2, X = 1.

Y = 1+2
X = 1 ;

No
?- X = 1, Y is X+2.

X = 1
Y = 3 ;

No
?- Y is X+2, X = 1.

ERROR: Arguments are not sufficiently instantiated
```


Αριθμητικές εκφράσεις και συναρτήσεις

- Σε έναν όρο **X is Y**, οι υπό-όροι του **Y** πρέπει να είναι αριθμοί ή *αποτιμώμενες συναρτήσεις*
- Οι συναρτήσεις αυτές περιλαμβάνουν τους προκαθορισμένους αριθμητικούς τελεστές **+**, **-**, ***** και **/**
- Όπως και προκαθορισμένες αριθμητικές συναρτήσεις, π.χ. **abs(X)**, **sqrt(X)**, **floor(X)**, …

Πραγματικές και ακέραιες τιμές

```
?- X is 1/2.
X = 0.5 ;
No
?- X is 4.0/2.0.
X = 2.0 ;
No
?- X is 5 // 2.
X = 2 ;
No
?- X is 4.0/2.
X = 2.0 ;
No
```

Δύο αριθμητικοί τύποι: ακέραιοι και πραγματικοί.

Οι περισσότερες αριθμητικές συναρτήσεις είναι υπερφορτωμένες για όλους τους συνδυασμούς ορισμάτων.

Η Prolog έχει δυναμικούς τύπους – οι τύποι χρησιμοποιούνται κατά το χρόνο εκτέλεσης για την επίλυση της υπερφόρτωσης.

Παρατηρείστε όμως ότι ο στόχος **2 = 2.0** αποτυγχάνει.

Αριθμητικές συγκρίσεις

- Τελεστές αριθμητικής σύγκρισης:
  <, >, =<, >=, =:=, =\=
- Σε μια αριθμητική σύγκριση, η Prolog αποτιμά και τα δύο ορίσματα του τελεστή και συγκρίνει τις τιμές που προκύπτουν
- Άρα και τα δύο ορίσματα πρέπει να έχουν τιμές για όλες τις μεταβλητές τους

Αριθμητικές συγκρίσεις

```
?- 1+2 < 1*2.

No
?- 1 < 2.0.

Yes
?- 1+2 >= 1+3.

No
?- X is 1-3, Y is 0-2.0, X =:= Y.

X = -2
Y = -2.0 ;

No
```


Συγκρίσεις ισότητας στην Prolog

- Μέχρι στιγμής έχουμε χρησιμοποιήσει τρεις διαφορετικούς τελεστές σύγκρισης ισότητας:
  - **X is Y** αποτιμά τον όρο **Y** και ενοποιεί το αποτέλεσμα με το **X**
    π.χ. **3 is 1+2** επιτυγχάνει, αλλά **1+2 is 3** αποτυγχάνει
  - **X = Y** ενοποιεί τους όρους **X** και **Y**, αλλά δεν τους αποτιμά
    π.χ. τόσο ο στόχος **3 = 1+2** όσο και ο **1+2 = 3** αποτυγχάνουν
  - **X =:= Y** αποτιμά τους δύο όρους και τους συγκρίνει αριθμητικά
    π.χ. τόσο ο στόχος **3 =:= 1+2** και ο **1+2 =:= 3** επιτυγχάνουν

Παραδείγματα

```
mylength1([],0).
mylength1([_|T], Len) :-
  mylength1(T, LenT),
  Len = LenT + 1.
```

```
?- mylength1([a,b,c],X).

X = 0+1+1+1 ;

No
```

```
mylength2([],0).
mylength2([_|T], Len) :-
  mylength2(T, LenT),
  Len is LenT + 1.
```

```
?- mylength2([a,b,c],X).

X = 3 ;

No
?- mylength2(X,3).

X = [_, _, _] ;

No
```


Παράδειγμα: sum

```
sum([], 0).
sum([Head|Tail],Sum) :-
  sum(Tail,TailSum),
  Sum is Head + TailSum.
```

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

X = 6 ;

No
?- sum([1, 2.5, 3], X).

X = 6.5 ;

No
```


Παράδειγμα: gcd

```
gcd(X,Y,GCD) :-
  X =:= Y,
  GCD is X.
gcd(X,Y,GCD) :-
  X < Y,
  NewY is Y - X,
  gcd(X,NewY,GCD).
gcd(X,Y,GCD) :-
  X > Y,
  NewX is X - Y,
  gcd(NewX,Y,GCD).
```

Προσοχή: όχι απλά

gcd(X,X,X)

Παράδειγμα: χρήση του κατηγορήματος gcd

```
?- gcd(5,5,X).
X = 5 ;
No
?- gcd(12,21,X).
X = 3 ;
No
?- gcd(91,105,X).
X = 7 ;
No
?- gcd(91,21*5,7).
Yes
?- gcd(91,X,7).
ERROR: Arguments are not sufficiently instantiated
```

Παράδειγμα: factorial

```
factorial(X,Fact) :-
  X =:= 1, Fact is 1.
factorial(X,Fact) :-
  X > 1,
  NewX is X - 1,
  factorial(NewX,NF),
  Fact is X * NF.
```

```
?- factorial(5,F).
F = 120 ;
No
?- factorial(2*5,F).
F = 3628800 ;
No
?- factorial(-2,F).
No
```


Διάζευξη και if-then-else στην Prolog

- Ο δυαδικός τελεστής `;/2` υλοποιεί τη διάζευξη (or)
- Ο ορισμός του είναι:

```
';'(X, _) :- X.
';'(_, Y) :- Y.
```

- Το if-then-else γράφεται με χρήση διάζευξης και του δυαδικού τελεστή `->/2`
- Ο ορισμός του είναι:

```
(Cond -> Then ; _) :- Cond, Then.
(Cond -> _ ; Else) :- \+ Cond, Else.
```


Παράδειγμα χρήσης διάζευξης και if-then-else

```
factorial(X,Fact) :-
  X =:= 1, Fact is 1.
factorial(X,Fact) :-
  X > 1,
  NewX is X - 1,
  factorial(NewX,NF),
  Fact is X * NF.
```

⇔

```
factorial(X,Fact) :-
  ( X =:= 1, Fact is 1
  ; X > 1,
    NewX is X - 1,
    factorial(NewX,NF),
    Fact is X * NF
  ).
```

⇕

```
factorial(X,Fact) :-
  ( X =:= 1 -> Fact is 1
  ; X > 1,
    NewX is X - 1,
    factorial(NewX,NF),
    Fact is X * NF
  ).
```


Παράδειγμα χρήσης if-then-else

```
gcd(X,Y,GCD) :-
  X =:= Y,
  GCD is X.
gcd(X,Y,GCD) :-
  X < Y,
  NewY is Y - X,
  gcd(X,NewY,GCD).
gcd(X,Y,GCD) :-
  X > Y,
  NewX is X - Y,
  gcd(NewX,Y,GCD).
```

⇔

```
gcd(X,Y,GCD) :-
  ( X =:= Y ->
      GCD is X
  ; X < Y ->
      NewY is Y - X,
      gcd(X,NewY,GCD)
  ; X > Y ->
      NewX is X - Y,
      gcd(NewX,Y,GCD)
  ).
```

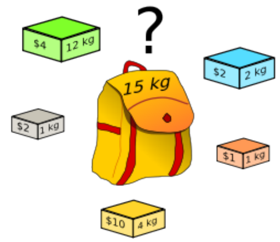
Από τον παραπάνω κώδικα μπορούμε να παραλείψουμε τον έλεγχο της συνθήκης **`X > Y`** (και φυσικά το **`->`**)

Συμπεριφορά του if-then χωρίς το else

Προσοχή: Ένα if-then χωρίς το else κομμάτι θα αποτύχει εάν η συνθήκη του if δεν είναι αληθής

- Αν θέλουμε να συνεχιστεί η εκτέλεση με τους στόχους μετά το if-then, πρέπει να χρησιμοποιήσουμε στο else κομμάτι το κατηγόρημα `true/0`

Δύο Παραδείγματα Προγραμμάτων



Προβλήματα αναζήτησης λύσης

- Η Prolog δε σχεδιάστηκε για προγραμματισμό αριθμητικών εφαρμογών (προφανώς)
- Τα προβλήματα στα οποία η Prolog δείχνει τις ικανότητές της είναι προβλήματα που χρησιμοποιούν αναζήτηση σε ένα χώρο λύσεων και όπου
  - Προσδιορίζουμε έναν ορισμό της λύσης με χρήση λογικής και
  - Αφήνουμε την Prolog να βρει αυτή τη λύση
- Θα εξετάσουμε τις λύσεις δύο προβλημάτων:
  - Το πρόβλημα του σακιδίου
  - Το πρόβλημα των οκτώ βασιλισσών

Το πρόβλημα του σακιδίου (Knapsack problem)

- Ετοιμάζουμε το σακίδιό μας για μια εκδρομή
- Στα ντουλάπια υπάρχουν τα εξής τρόφιμα:

Item	Weight in kilograms	Calories
bread	4	9200
pasta	2	4600
peanut butter	1	6700
baby food	3	6900

- Το σακίδιο χωράει πράγματα συνολικού βάρους 4 kg.
- Ποια επιλογή πραγμάτων βάρους ≤ 4 kg. μεγιστοποιεί το ποσό των θερμίδων;

Οι άπληστες μέθοδοι δε δουλεύουν

Item	Weight in kilograms	Calories
bread	4	9200
pasta	2	4600
peanut butter	1	6700
baby food	3	6900

- Πρώτα τις περισσότερες θερμίδες: bread = 9200
- Πρώτα τα πιο ελαφριά: peanut butter + pasta = 11300
- (Η βέλτιστη επιλογή: peanut butter + baby food = 13600)

Αναζήτηση

- Δεν υπάρχει γνωστός αλγόριθμος για το συγκεκριμένο πρόβλημα που
  - Πάντα δίνει τη βέλτιστη λύση για το πρόβλημα, και
  - Χρειάζεται λιγότερο από εκθετικό χρόνο για να τη βρει
- Κατά συνέπεια, δε θα πρέπει να ντρεπόμαστε αν χρησιμοποιήσουμε εξαντλητική αναζήτηση
- Το αντίθετο μάλιστα, επειδή η αναζήτηση είναι κάτι που η Prolog κάνει καλά

Αναπαράσταση

- Θα αναπαραστήσουμε κάθε είδος φαγητού με τον όρο **`food(N,W,C)`**
- Το ντουλάπι του παραδείγματός μας αναπαριστάται ως μια λίστα από τέτοιους όρους:

```
[food(bread,4,9200),
 food(pasta,2,4500),
 food(peanutButter,1,6700),
 food(babyFood,3,6900)]
```

- Θα χρησιμοποιήσουμε την ίδια αναπαράσταση για τα περιεχόμενα του σακιδίου

Ακολουθίες και υπακολουθίες

```
/*
  subseq(X, Y) succeeds when list X is the same as
  list Y, but with zero or more elements omitted.
  This can be used with any pattern of instantiations.
*/
subseq([], []).
subseq([Item | RestX], [Item | RestY]) :-
  subseq(RestX, RestY).
subseq(X, [_ | RestY]) :-
  subseq(X, RestY).
```

- Μια υπακολουθία μιας λίστας είναι ένα αντίγραφο της λίστας όπου κάποια στοιχεία της λίστας έχουν παραλειφθεί - στο παράδειγμά μας τα περιεχόμενα του σακιδίου είναι μια υπακολουθία του ντουλαπιού

Κάποια παραδείγματα

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

Yes
?- subseq(X,[1,2,3]).

X = [1, 2, 3] ;

X = [1, 2] ;

X = [1, 3] ;

X = [1] ;

X = [2, 3] ;

X = [2] ;

X = [3] ;

X = [] ;

No
```

Παρατηρείστε ότι το κατηγόρημα **subseq/2** *όχι μόνο μπορεί να ελέγξει κατά πόσο μια λίστα είναι μια υπακολουθία κάποιας άλλης αλλά μπορεί επίσης να παράγει υπακολουθίες, τις οποίες και θα χρησιμοποιήσουμε για τη λύση του προβλήματος του σακιδίου.*


```
/*
  knapsackDecision(Items, Capacity, Goal, Knapsack)
  takes a list Items of food terms, a positive number
  Capacity, and a positive number Goal.  We unify
  Knapsack with a subsequence of Items representing
  a knapsack with total calories >= Goal, subject to
  the constraint that the total weight is =< Capacity.
*/
knapsackDecision(Items, Capacity, Goal, Knapsack) :-
  subseq(Knapsack, Items),
  weight(Knapsack, Weight),
  Weight =< Capacity,
  calories(Knapsack, Calories),
  Calories >= Goal.
```


Για να δούμε που βρισκόμαστε…

```
?- knapsackDecision(
|    [food(bread,4,9200),
|     food(pasta,2,4500),
|     food(peanutButter,1,6700),
|     food(babyFood,3,6900)],
|    4,
|    10000,
|    X).

X = [food(pasta,2,4500), food(peanutButter,1,6700)]

Yes
```

- Η παραπάνω κλήση αποφασίζει εάν υπάρχει λύση που να ικανοποιεί το στόχο των ελάχιστων θερμίδων (**10000**)
- Όχι όμως υποχρεωτικά τη λύση που ζητάμε εμείς…

Αποφασισιμότητα και βελτιστοποίηση

- Μέχρι στιγμής λύσαμε το *πρόβλημα της απόφασης* για το πρόβλημα του σακιδίου
- Αυτό που θέλουμε να λύσουμε είναι το *πρόβλημα της βελτιστοποίησης* της λύσης του συγκεκριμένου προβλήματος
- Για να το επιτύχουμε, θα χρησιμοποιήσουμε ένα άλλο προκαθορισμένο κατηγόρημα της Prolog: **findall/3**

Το κατηγόρημα findall/3

- **findall(X, Goal, L)**
  - Βρίσκει όλους τους τρόπους με τους οποίους ο στόχος **Goal** ικανοποιείται
  - Για τον καθένα από αυτούς, εφαρμόζει στον όρο **X** την ίδια αντικατάσταση με την οποία ο στόχος **Goal** ικανοποιήθηκε
  - Ενοποιεί τη λίστα **L** με τη λίστα όλων αυτών των **X**

```
?- findall(42, subseq(_,[1,2]), L).

L = [42, 42, 42, 42] ;

No
```

- Υπάρχουν τέσσερις λύσεις για το **subseq(_,[1,2])**
- Συλλέξαμε μια λίστα από **42**, ένα για κάθε λύση

Συλλογή των απαντήσεων που θέλουμε

- Συνήθως, η πρώτη παράμετρος του **findall/3** είναι ένας όρος με τις μεταβλητές που υπάρχουν στο στόχο προς επίλυση

```
?- findall(X, subseq(X,[1,2]), L).

X = _G396
L = [[1, 2], [1], [2], []] ;

No
```

- Η παραπάνω ερώτηση συλλέγει όλες τις απαντήσεις της ερώτησης-στόχου **subseq(X,[1,2])**


```
/*
  knapsackOptimization(Items,Capacity,Knapsack) takes
  a list Items of food items and a positive integer
  Capacity.  We unify Knapsack with a subsequence of
  Items representing a knapsack of maximum total
  calories, subject to the constraint that the total
  weight is =< Capacity.
*/
knapsackOptimization(Items, Capacity, Knapsack) :-
  findall(K, legalKnapsack(Items,Capacity,K), L),
  maxCalories(L, Knapsack).
```



```
/*
  legalKnapsack(Items,Capacity,Knapsack) takes a list
  Items of food terms and a positive number Capacity.
  We unify Knapsack with a subsequence of Items whose
  total weight is =< Capacity.
*/
legalKnapsack(Items, Capacity, Knapsack):-
  subseq(Knapsack, Items),
  weight(Knapsack, W),
  W =< Capacity.
```

```
/*
  maxCalories(List, Result) takes a non-empty List of
  lists of food terms.  We unify Result with an element
  from the list that maximizes the total calories.
  We use a helper predicate maxC that takes four
  parameters: the remaining list of lists of food
  terms, the best list of food terms seen so far, its
  total calories, and the final result.
*/
maxCalories([First | Rest], Result) :-
  calories(First, FirstC),
  maxC(Rest, First, FirstC, Result).

maxC([], Sofar, _, Sofar).
maxC([First | Rest], _, MC, Result) :-
  calories(First, FirstC),
  MC =< FirstC,
  maxC(Rest, First, FirstC, Result).
maxC([First | Rest], Sofar, MC, Result) :-
  calories(First, FirstC),
  MC > FirstC,
  maxC(Rest, Sofar, MC, Result).
```



```
/*
  maxCalories(List, Result) takes a non-empty List of
  lists of food terms.  We unify Result with an element
  from the list that maximizes the total calories.
  We use a helper predicate maxC that takes four
  parameters: the remaining list of lists of food
  terms, the best list of food terms seen so far, its
  total calories, and the final result.
*/
maxCalories([First | Rest], Result) :-
  calories(First, FirstC),
  maxC(Rest, First, FirstC, Result).

maxC([], Sofar, _, Sofar).
maxC([First | Rest], Sofar, MC, Result) :-
  calories(First, FirstC),
  ( MC =< FirstC ->
      NSofar = First, NMC = FirstC
  ;
      NSofar = Sofar, NMC = MC
  ),
  maxC(Rest, NSofar, NMC, Result).
```

Το ίδιο με χρήση if-then-else

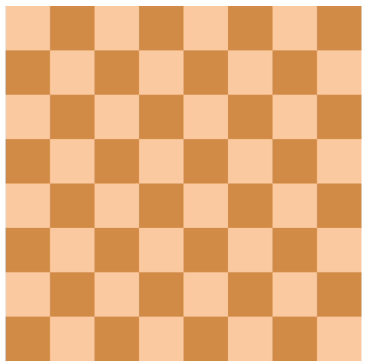

```
?- knapsackOptimization(
|    [food(bread,4,9200),
|     food(pasta,2,4500),
|     food(peanutButter,1,6700),
|     food(babyFood,3,6900)],
|    4,
|    Knapsack).

Knapsack = [food(peanutButter,1,6700),
            food(babyFood,3,6900)] ;

No
```


Το πρόβλημα των οκτώ βασιλισσών (8-queens)

- Κάποιες πληροφορίες για το σκάκι:
  - Παίζεται σ' ένα τετράγωνο 8×8
  - Μια βασίλισσα μπορεί να κινηθεί οριζόντια, κάθετα ή διαγώνια για όσα τετράγωνα θελήσει
  - Δύο βασίλισσες απειλούν η μία την άλλη εάν είναι στην ίδια γραμμή, στήλη ή διαγώνιο (και η μία μπορεί να κινηθεί άμεσα στο τετράγωνο της άλλης)
- Το πρόβλημα: βρες ένα τρόπο να τοποθετηθούν οκτώ βασίλισσες σε μια κενή σκακιέρα έτσι ώστε καμία βασίλισσα να μην απειλείται από κάποια άλλη



Αναπαράσταση

- Θα μπορούσαμε να αναπαραστήσουμε τη βασίλισσα στη στήλη 2, σειρά 5 με τον όρο **`queen(2,5)`**
- Αλλά αφού δεν υπάρχουν άλλα κομμάτια στη σκακιέρα–π.χ. δε θα έχουμε κάποιο **`pawn(X,Y)`** ή **`king(X,Y)`**–θα χρησιμοποιήσουμε απλώς έναν όρο της μορφής **`X/Y`**
- (Δε θα τον αποτιμήσουμε ως διαίρεση)

Παράδειγμα

- Ένας σχηματισμός σκακιέρας είναι μια λίστα από βασίλισσες
- Ο παρακάτω είναι για τις βασίλισσες **`[2/5,3/7,6/1]`**

8								
7			Q					
6								
5		Q						
4								
3								
2								
1						Q		
	1	2	3	4	5	6	7	8


```
/*
  nocheck(L, X/Y) takes a queen X/Y and a list
  of queens. It succeeds if and only if the X/Y
  queen holds none of the others in check.
*/
nocheck([], _).
nocheck([X1/Y1 | Rest], X/Y) :-
  X =\= X1,
  Y =\= Y1,
  abs(Y1-Y) =\= abs(X1-X),
  nocheck(Rest, X/Y).

/*
  legal(L) succeeds if L is a legal placement of
  queens: all coordinates in range and no queen
  in check.
*/
legal([]).
legal([X/Y | Rest]) :-
  legal(Rest),
  member(X, [1,2,3,4,5,6,7,8]),
  member(Y, [1,2,3,4,5,6,7,8]),
  nocheck(Rest, X/Y).
```


Επάρκεια

- Έχουμε ήδη αρκετά συστατικά για να λύσουμε το πρόβλημα: η ερώτηση **`legal(X)`** βρίσκει όλους τους επιτρεπτούς σχηματισμούς:

```
?- legal(X).

X = [] ;

X = [1/1] ;

X = [1/2] ;

X = [1/3]
```

Λύση για τις οκτώ βασίλισσες

- Φυσικά, η παραπάνω «λύση» θα πάρει πολύ χρόνο: θα αρχίσει με το να βρει όλες τις 64 λύσεις με μια βασίλισσα, και μετά θα αρχίσει με όλες τις λύσεις με δύο, κοκ.
- Μπορούμε να «εκβιάσουμε» μια λύση με 8 βασίλισσες με την ερώτηση:

```
?- X = [_,_,_,_,_,_,_,_],
|  legal(X).

X = [8/4, 7/2, 6/7, 5/3,
     4/6, 3/8, 2/5, 1/1]

Yes
```

8			Q					
7						Q		
6				Q				
5		Q						
4								Q
3					Q			
2							Q	
1	Q							
	1	2	3	4	5	6	7	8

Υπάρχει δυνατότητα για βελτίωση;

- Η εύρεση της λύσης με αυτόν τον τρόπο παίρνει πολύ χρόνο
- Στη συνέχεια, η Prolog βρίσκει απλές αναδιατάξεις της πρώτης λύσης:

```
?- X = [_,_,_,_,_,_,_,_], legal(X).

X = [8/4, 7/2, 6/7, 5/3, 4/6, 3/8, 2/5, 1/1] ;

X = [7/2, 8/4, 6/7, 5/3, 4/6, 3/8, 2/5, 1/1] ;

X = [8/4, 6/7, 7/2, 5/3, 4/6, 3/8, 2/5, 1/1] ;

X = [6/7, 8/4, 7/2, 5/3, 4/6, 3/8, 2/5, 1/1]
```


Βελτίωση του προγράμματος

- Προφανώς κάθε λύση έχει μία βασίλισσα σε κάθε στήλη
- Άρα, κάθε λύση μπορεί να γραφεί με τη μορφή:

```
X = [1/_,2/_,3/_,4/_,5/_,6/_,7/_,8/_]
```

- Δίνοντας έναν όρο αυτής της μορφής περιορίζουμε την αναζήτηση (επιταχύνοντάς την) και αποφεύγουμε την εύρεση απλών αναδιατάξεων

```
/* eightqueens(X) succeeds if X is a legal
   placement of eight queens, listed in order
   of their X coordinates.
*/
eightqueens(X) :-
  X = [1/_,2/_,3/_,4/_,5/_,6/_,7/_,8/_],
  legal(X).
```


Περισσότερες βελτιώσεις

- Επειδή ξέρουμε ότι όλες οι X-συντεταγμένες είναι εντός του διαστήματος και διαφορετικές μεταξύ τους, το πρόγραμμα μπορεί λιγάκι να βελτιωθεί

```
nocheck([], _).
nocheck([X1/Y1 | Rest], X/Y) :-
  % X =\= X1, assume the X's are distinct
  Y =\= Y1,
  abs(Y1-Y) =\= abs(X1-X),
  nocheck(Rest, X/Y).

legal([]).
legal([X/Y | Rest]) :-
  legal(Rest),
  % member(X, [1,2,3,4,5,6,7,8]), assume X in range
  member(Y, [1,2,3,4,5,6,7,8]),
  nocheck(Rest, X/Y).
```

Βελτιωμένη λύση στο πρόβλημα

- Το πρόγραμμα τώρα τρέχει αρκετά πιο γρήγορα
- Για παράδειγμα, δε βρίσκει πλέον όλες τις αναδιατάξεις

```
?- eightqueens(X).

X = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1] ;

X = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1]
```


Ένα πείραμα

```
legal([]).
legal([X/Y | Rest]) :-
  legal(Rest),
  % member(X, [1,2,3,4,5,6,7,8]), assume X in range
  1 =< Y, Y =< 8, % was member(Y,[1,2,3,4,5,6,7,8]),
  nocheck(Rest, X/Y).
```

- Το παραπάνω κατηγόρημα δε δουλεύει
- Εγείρει εξαίρεση: “arguments not sufficiently instantiated”
- Η κλήση του κατηγορήματος **`member/2`** δεν είναι απλώς κάποιος *έλεγχος* ότι οι συντεταγμένες είναι εντός ορίων αλλά είναι *παραγωγή* των συντεταγμένων

Άλλο ένα πείραμα

```
legal([]).
legal([X/Y | Rest]) :-
  % member(X, [1,2,3,4,5,6,7,8]), assume X in range
  member(Y, [1,2,3,4,5,6,7,8]),
  nocheck(Rest, X/Y),
  legal(Rest). % formerly the first condition
```

- Εγείρει εξαίρεση: “arguments not sufficiently instantiated”
- Η κλήση **`legal(Rest)`** πρέπει να προηγείται διότι παράγει την μερική λύση που χρειάζεται η αναδρομική κλήση του **`nocheck`**

Εύρεση μίας μόνο λύσης

- Αν θέλαμε να βρούμε μία μόνο λύση του προβλήματος θα μπορούσαμε να χρησιμοποιήσουμε οποιοδήποτε από τους δύο παρακάτω ορισμούς του σχετικού κατηγορήματος

```
/*
   eightqueens1(X) finds
   one solution to the
   eight queens problem.
*/
eightqueens1(X) :-
  once(eightqueens(X)).
```

```
/*
   eightqueens1(X) finds
   one solution to the
   eight queens problem.
*/
eightqueens1(X) :-
  eightqueens(X),
  !.
```

Το cut (!) της Prolog

- Το προκαθορισμένο κατηγόρημα `!/0` (διαβάζεται cut) χρησιμοποιείται για να περιορίσει την οπισθοδρόμηση
- Η εκτέλεσή του πάντα επιτυγχάνει και «κλαδεύει»:
  - Όλους τους εναλλακτικούς τρόπους ικανοποίησης των υποστόχων που πιθανώς να υπάρχουν μεταξύ της κεφαλής ενός κανόνα και του σημείου που το `!` εμφανίζεται στον κανόνα, και
  - Όλους τους κανόνες που μπορεί να έπονται του κανόνα που περιέχει το `!`
- Η χρήση του πρέπει να γίνεται με φειδώ και ύστερα από αρκετή σκέψη

Παράδειγμα: Οπισθοδρόμηση χωρίς και με cut

```
p(X,Y) :-
   x(X),
   y(Y).
p(X,Y) :-
   z(X,Y).
p(4,d).

x(1).
x(2).

y(a).
y(b).

z(3,c).
```

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

X = 1
Y = a ;

X = 1
Y = b ;

X = 2
Y = a ;

X = 2
Y = b ;

X = 3
Y = c ;

X = 4
Y = d ;

No
```

```
p(X,Y) :-
   x(X),
   !,
   y(Y).
p(X,Y) :-
   z(X,Y).
p(4,d).

x(1).
x(2).

y(a).
y(b).

z(3,c).
```

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

X = 1
Y = a ;

X = 1
Y = b ;

No
```


Μέρη της Prolog που δεν εξετάσαμε

- Τη δυνατότητα ορισμού νέων τελεστών
- Το χειρισμό εξαιρέσεων
  - Εξαιρέσεις που εγείρονται από το σύστημα και από το χρήστη
  - Τα σχετικά κατηγορήματα `throw` και `catch`
- Τις βιβλιοθήκες και πολλά από τα προκαθορισμένα κατηγορήματα της γλώσσας

Αποχαιρετισμός στην Prolog

- Δε χρειάστηκε να παραλείψουμε τόσο ποσοστό της Prolog όσο παραλείψαμε στην ML και στη Java
- Η Prolog είναι μια μικρή γλώσσα
- Επίσης, είναι ένα αρκετά ισχυρό εργαλείο επίλυσης (κάποιου είδους) προβλημάτων αλλά ο επιδέξιος χειρισμός της από τον προγραμματιστή δεν είναι πολύ εύκολος
- Ο επιδέξιος χειρισμός της Prolog προϋποθέτει την υιοθέτηση και εξοικείωση με μια διαφορετική φιλοσοφία προγραμματισμού