ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών

Τίτλος Μαθήματος

Ενότητα 2: Η γλώσσα προγραμματισμού Prolog

Παναγιώτης Σταματόπουλος
Σχολή Θετικών Επιστημών
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Περιγραφή ενότητας

Αναλυτική παρουσίαση της γλώσσας προγραμματισμού Prolog μέσω της παράθεσης προγραμμάτων που επιδεικνύουν τις δυνατότητες της γλώσσας. Περιγραφή των συνηθέστερων ενσωματωμένων κατηγορημάτων της Prolog.

Η γλώσσα προγραμματισμού Prolog

Επέκταση προγράμματος (1/2)

```
female(pam).
female(liz).
female(pat).
female(ann).
male(tom).
male(bob).
male(jim).
```

- ή εναλλακτικά:

```
sex(pam, feminine).
sex(tom,
masculine).
sex(bob,
masculine).
```

- ....

```
offspring(Y, X) :- parent(X, Y).
        ?- offspring(liz, tom).
           yes
     mother(X, Y) :- parent(X, Y), female(X).
```

Επέκταση προγράμματος (2/2)

- προτάσεις (clauses)
  - γεγονότα (facts)
  - κανόνες (rules)
  - ερωτήσεις (questions)
- κατηγορήματα (predicates)
- κεφαλή (head) / σώμα (body)
- στόχοι (goals)
- άτομα (atoms) / μεταβλητές (variables)
- διαδικασία (procedure) / σχέση (relation)

Και άλλοι κανόνες (1/2)

```
grandparent(X, Z) :- parent(X, Y), parent(Y,Z).
sister(X, Y) :- parent(Z, X), parent(Z, Y),
female(X).

    ?- sister(ann, pat).
       yes
    ?- sister(X, pat).
       X = ann        -> ;
       X = pat                    ---------> ;;;
       yes
```

Και άλλοι κανόνες (2/2)

```
sister(X, Y) :- parent(Z, X), parent(Z, Y),
                female(X), different(X, Y).



hasachild(X) :- ...............


aunt(X, Y) :- .................


grandchild(X, Y) :- ...........
```

Αναδρομικοί κανόνες (1/6)

```
(1) predecessor(X, Z) :-
            parent(X, Z).
(2) predecessor(X, Z) :-
            parent(X, Y),
            parent(Y, Z).        --(1)-->   predecessor(Y, Z)
(3) predecessor(X, Z) :-
            parent(X, Y1),
            parent(Y1, Y2),
            parent(Y2, Z).       --(2)-->   predecessor(Y1, Z)
```

Αναδρομικοί κανόνες (2/6)

```
(4) predecessor(X, Z) :-
            parent(X, Y1),
            parent(Y1, Y2),
            parent(Y2, Y3),          (3)
            parent(Y3, Z).          ⟶      predecessor(Y1, Z)
    . . . . . .     ⇕
predecessor(X, Z) :-
            parent(X, Z).
predecessor(X, Z) :-
            parent(X, Y),
            predecessor(Y, Z).
```

Αναδρομικοί κανόνες (3/6)

```
?- predecessor(pam, X).
   X = bob      -> ;
   X = ann      -> ;
   X = pat      -> ;
   X = jim
   yes
predecessor(X, Z) :-
     parent(X, Y),
     predecessor(Y, Z).
```

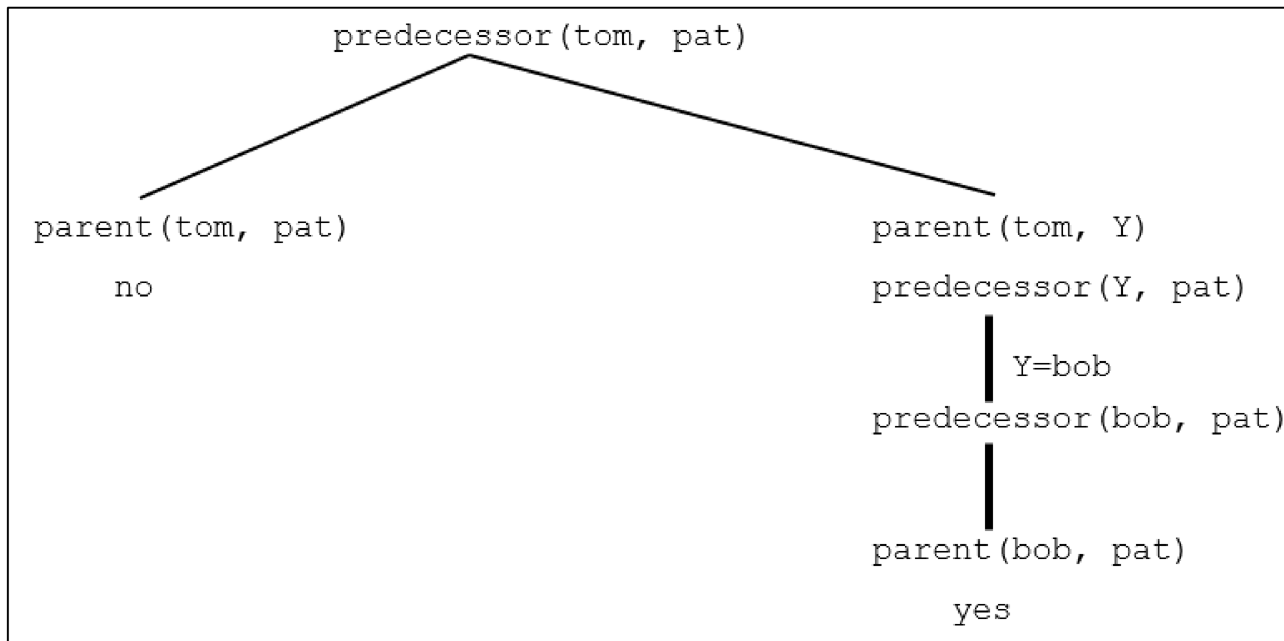
?⇔

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

Αναδρομικοί κανόνες (4/6)

- δηλωτική σημασία (declarative meaning)
- διαδικαστική σημασία (procedural meaning)
- εύρος δράσης μεταβλητών (scope of variables)

Αναδρομικοί κανόνες (5/6)




```
?- parent(pam, bob).
?- mother(pam, bob).
?- grandparent(pam, ann).
?- grandparent(bob, jim).
```

ΕΚΤΕΛΕΣΗ;

Αναδρομικοί κανόνες (6/6)

- οπισθοδρόμηση (backtracking)
- αποτίμηση (instantiation)
- επιτυχία (success) / αποτυχία (failure)
- ανάλυση (resolution)
- δέντρο ανάλυσης (resolution tree) / ΚΑΙ–Ή δέντρο (AND–OR tree)

Αλφάβητο της Prolog

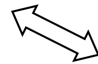
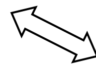
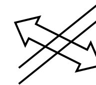
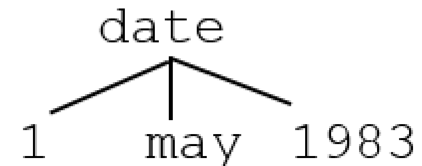
- A, B, C, ... Z
- a, b, c, ... z
- 0, 1, 2, ... 9
- +, -, *, /, <, >, =, :, ., &, _, ~, ...

Άτομα

- ```
  anna
  nil
  x25
  x_25
  x_25AB
  x_
  x___y
  alpha_beta_procedure
  miss_Jones
  sarah_jones
  ```

- ```
  <--->
  ======>
  ...
  .:.
  ::=
  //
  ```

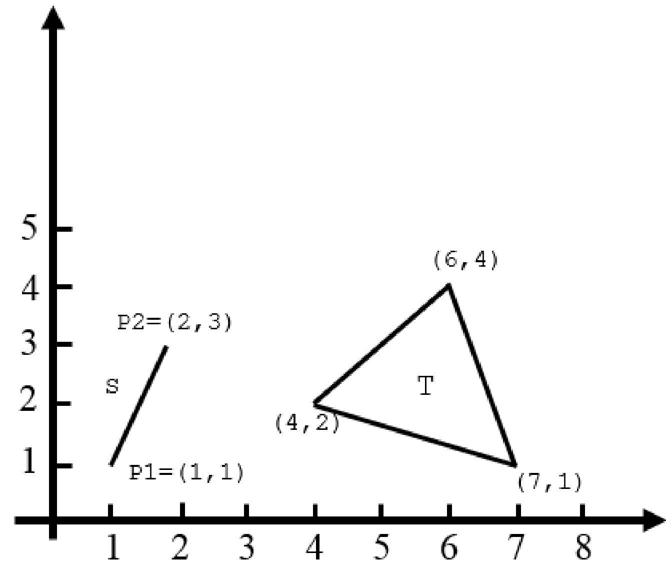
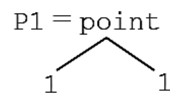
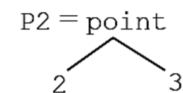
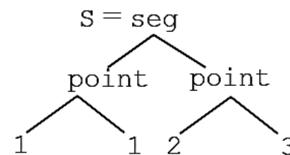
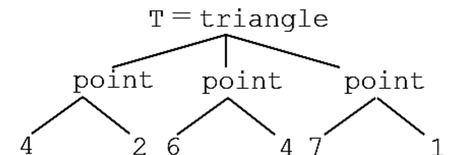
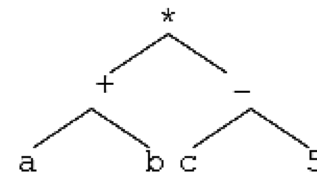
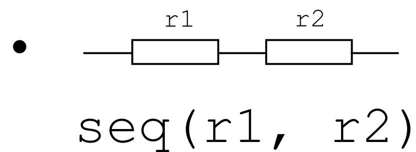
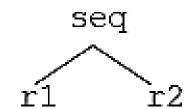
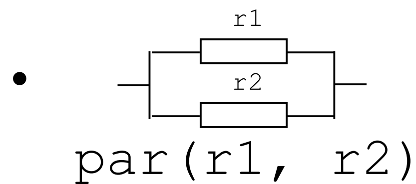
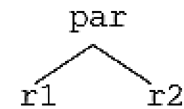
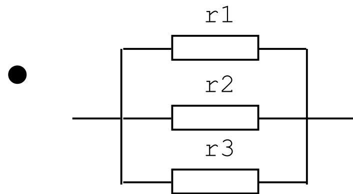
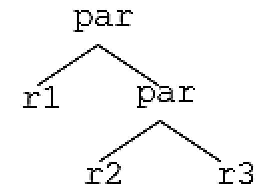
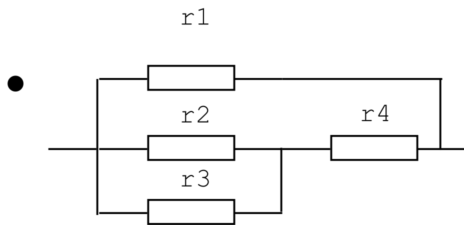
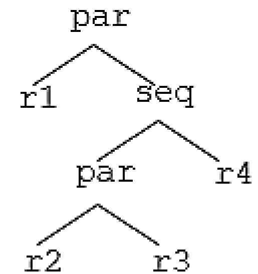
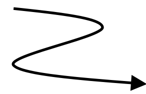
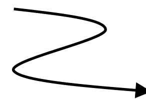
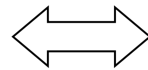
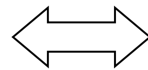
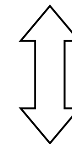
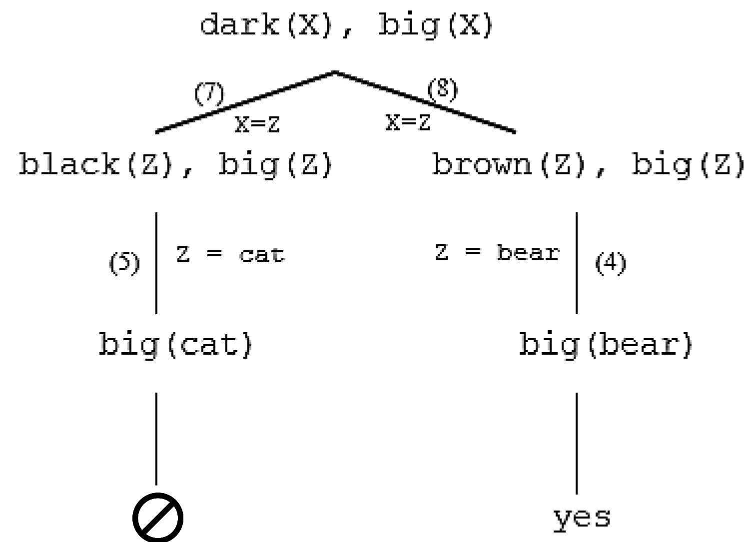
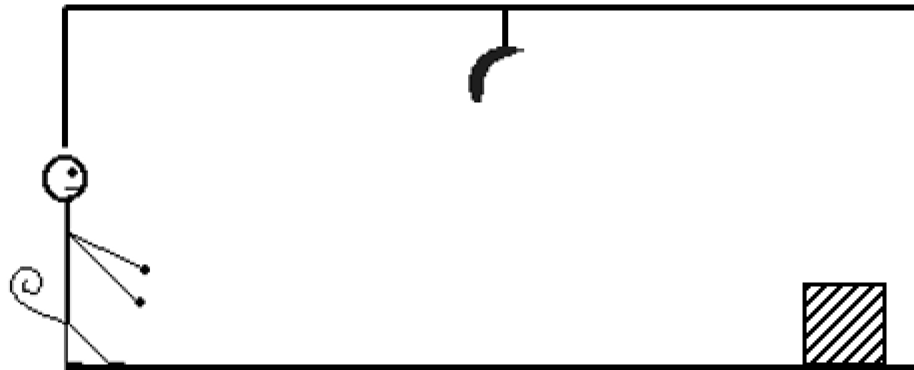
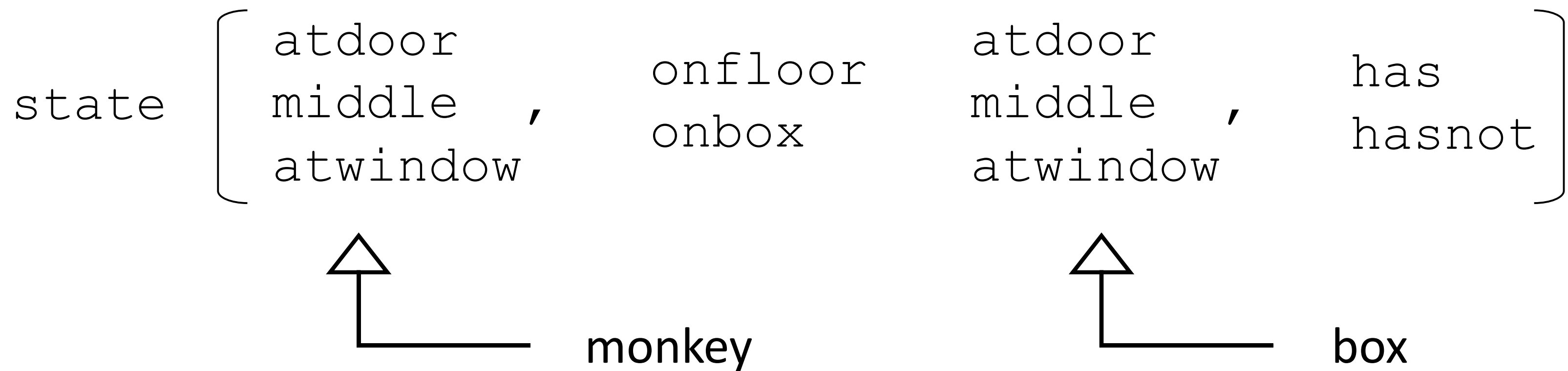
- ```
  ’Tom’
  ’South_America’
  ’Sarah Jones’
  ’,’
  ```

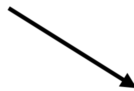
  ```

# Αριθμοί

- 1    1313    0    -97
- 3.14   -0.0035   100.2

# Μεταβλητές

```
X   ShoppingList
Result   _x23
Object2   _23
Participant_list   _
```

Ανώνυμη (anonymous) μεταβλητή (1/2)

```
hasachild(X) :- parent(X, Y).
```



```
hasachild(X) :- parent(X, _).
```

```
somebody_has_child :- parent(X, Y).
```



```
somebody_has_child :- parent(_, _).
```



```
somebody_has_child :-parent(X, X).
```

Ανώνυμη (anonymous) μεταβλητή (2/2)

```
?- parent(X, _).
      X = pam       -> ;
      X = tom       -> ;
      X = tom       -> ;
      X = bob       -> ;
      X = bob       -> ;
      X = pat
      yes
```

Δομές (Structures)

`date(1, may, 1983)`



functor (συναρτησιακό σύμβολο)

arguments (ορίσματα)

arity (βαθμός)

terms (όροι)

- άτομα
- αριθμοί
- μεταβλητές
- δομές (ή σύνθετοι όροι)

`date(Day, may, 1983)`

Αποδεκτοί όροι

```
Diana                          goes(Diana, south)
diana                          45
'Diana'                        5(X, Y)              ⃠
_diana                         +(north, west)
'Diana goes south'             three(Black(Cats))   ⃠
```

Αναπαράσταση (1/3)

```
P1 = point(1, 1)

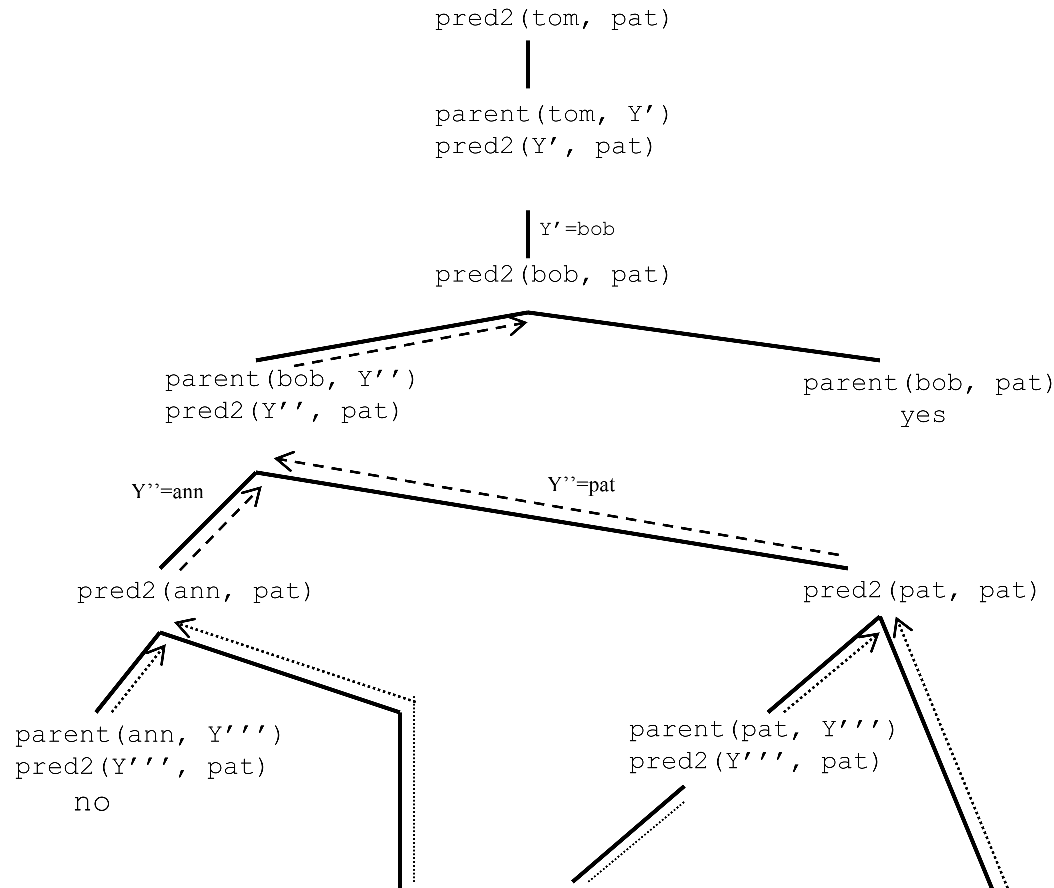
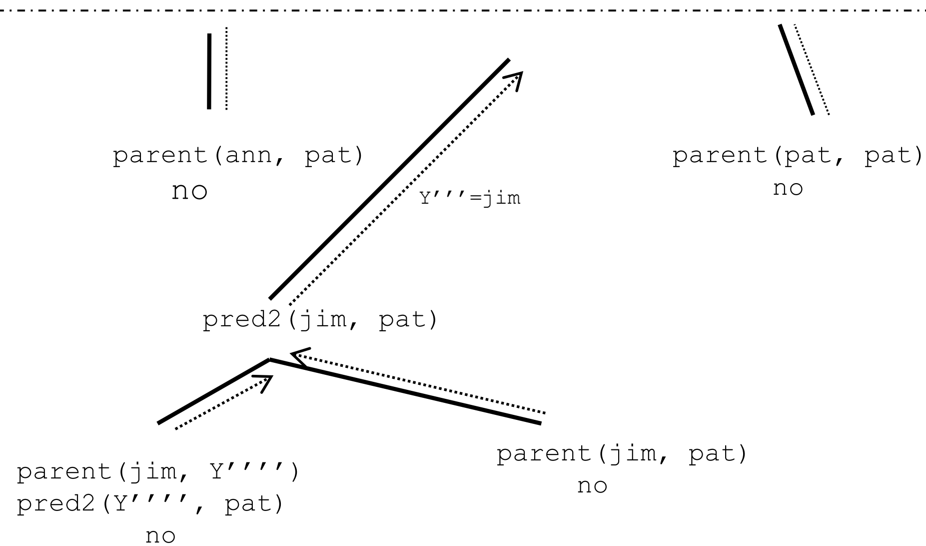
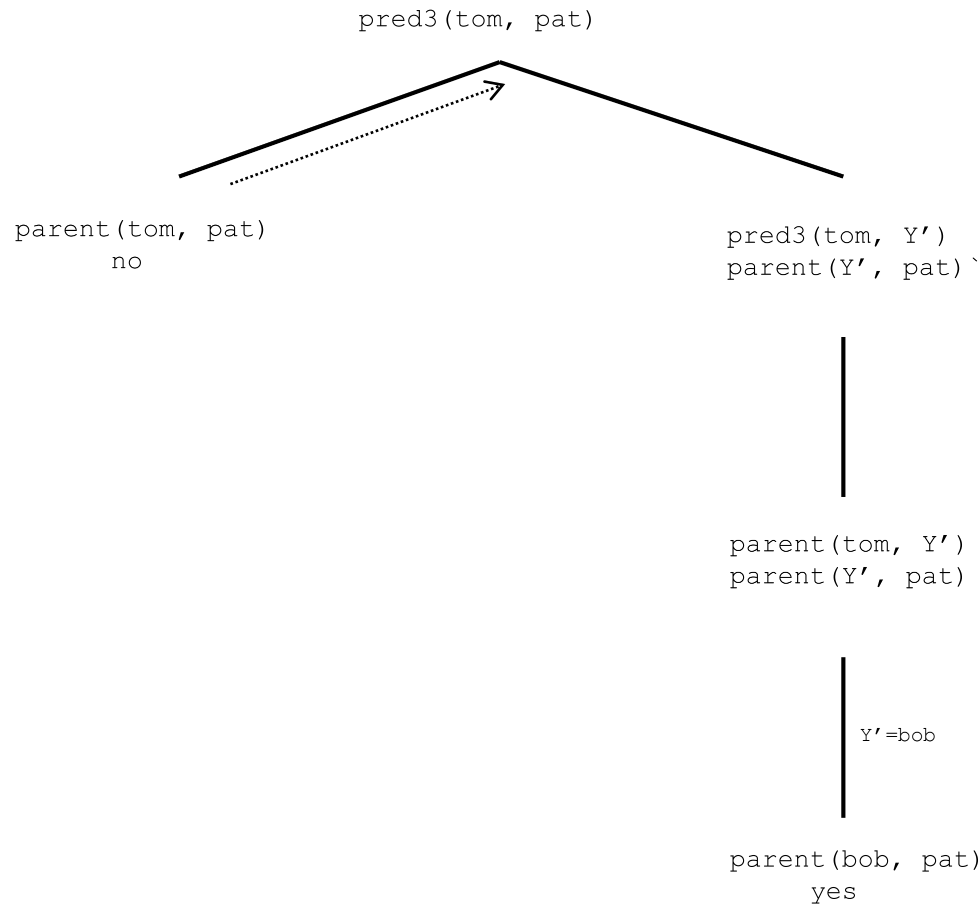
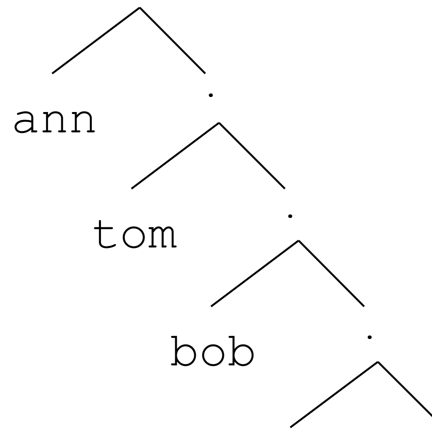
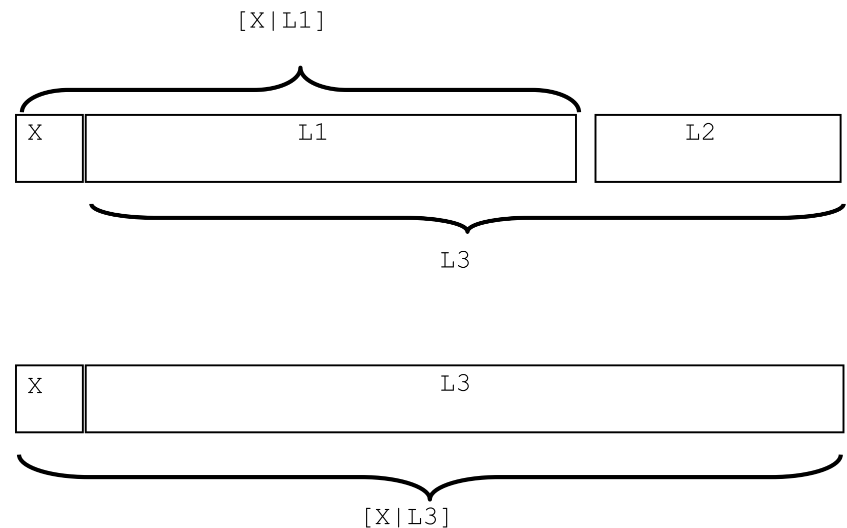
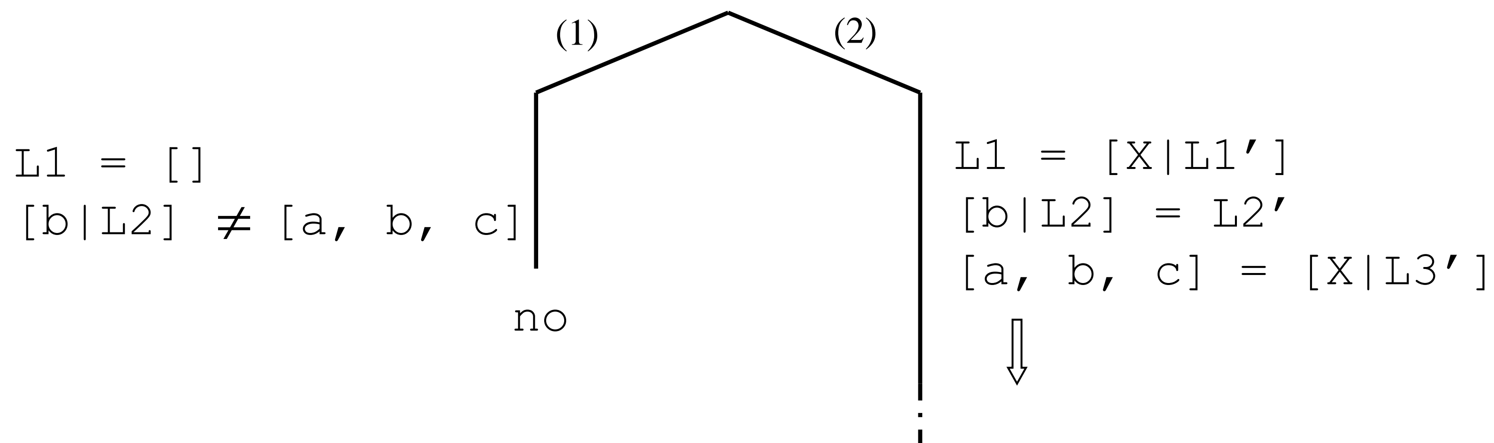
P2 = point(2, 3)

S = seg(P1, P2) =
seg(point(1, 1),
point(2, 3))

T = triangle(point(4,
2), point(6, 4),

point(7, 1))
```











3-D: `point3(X, Y, Z)` ή `point(X, Y, Z)`

Αναπαράσταση (2/3)

- (a+b)*(c−5)*(+(a, b), -(c, 5))



- 

seq(r1, r2)



- 

par(r1, r2)



Αναπαράσταση (3/3)

- 



par(r1, par(r2, r3))

- 



par(r1, seq(par(r2, r3), r4))

ορθογώνια παραλληλόγραμμα, τετράγωνα, κύκλοι (?)

Ταίριασμα (Matching)
Ενοποίηση (Unification)

```
?- date(D, M, 1983) =
date(D1, may, Y1).
      D = _0084
      D1 = _0084
      M = may
      Y1 = 1983
      yes
```

γενικότερη ενοποίηση

```
      D = 1
      D1 = 1
      M = may
      Y1 = 1983
```

λιγότερο γενική ενοποίηση

Κανόνες ενοποίησης (1/4)

- Δύο ίδια άτομα ενοποιούνται
- Δύο ίδιοι αριθμοί ενοποιούνται
- Μία μεταβλητή ενοποιείται με οτιδήποτε (παίρνοντάς το σαν τιμή)
- Δύο δομές ενοποιούνται αν έχουν το ίδιο συναρτησιακό σύμβολο και τα αντίστοιχα ορίσματά τους ενοποιούνται

Κανόνες ενοποίησης (2/4)

```
?- triangle(point(1, 1), A, point(2, 3)) =
        triangle(X, point(4, Y), point(2, Z)).
   X = point(1, 1)
   A = point(4, _0090)
   Z = 3
   Y = _0090
   yes
vertical(seg(point(X, Y), point(X, Y1))).
horizontal(seg(point(X, Y), point(X1, Y))).
?- vertical(seg(point(1, 1), point(1, 2))).
       yes
```

Κανόνες ενοποίησης (3/4)

```
?- vertical(seg(point(1, 1), point(2, Y))).
       no
?- horizontal(seg(point(1, 1), point(2, Y))).
      Y = 1
      yes
?- vertical(seg(point(2, 3), P)).
      P = point(2, _0082)
      yes
?- vertical(S), horizontal(S).
      S = seg(point(_0084, _0085),
      point(_0084, _0085))
      yes
```

Κανόνες ενοποίησης (4/4)

Αν ένα τετράπλευρο παριστάνεται από τον όρο `four_points(P1,P2,P3,P4)`, όπου τα `Pi` είναι οι κορυφές του κυκλικά, πώς πρέπει να οριστεί η σχέση `regular_rectangle(R)` έτσι ώστε να αληθεύει όταν το `R` είναι ορθογώνιο παραλληλόγραμμο με τις πλευρές του κατακόρυφες και οριζόντιες;

Δηλωτική και διαδικαστική σημασία

- P :- Q, R
- Το P είναι αληθές αν τα Q και R είναι αληθή
- Από τα Q και R έπεται το P

δηλωτική σημασία

- Για να λυθεί το πρόβλημα P πρώτα πρέπει να λυθεί το υποπρόβλημα Q και μετά το υποπρόβλημα R
- Για να ικανοποιηθεί το P πρώτα πρέπει να ικανοποιηθεί το Q και μετά το R

διαδικαστική σημασία

Παραλλαγή πρότασης Στιγμιότυπο πρότασης

Παραλλαγή πρότασης

```
hasachild(X) :- parent(X, Y).
```



```
hasachild(A) :- parent(A, B).
```

Στιγμιότυπο πρότασης

```
hasachild(X) :- parent(X, Y).
```



```
hasachild(peter) :- parent(peter, Y).
```

Δηλωτική σημασία

- Ένας στόχος G είναι αληθής αν για κάποια αποτίμηση των μεταβλητών του ταυτίζεται με την κεφαλή ενός στιγμιότυπου I κάποιας πρότασης C του προγράμματος και οι στόχοι του σώματος του I είναι επίσης αληθείς.
- Μία σύζευξη από στόχους είναι αληθής όταν κάθε ένας από αυτούς είναι αληθής, για την ίδια αποτίμηση των μεταβλητών τους.

Στόχοι υπό διάζευξη (1/3)

- P :- Q.
- P :- R.



P :- Q ; R.

- P :- Q, R.
- P :- S, T, U.



P :- (Q, R) ; (S, T, U).



P :- Q, R ; S, T, U.

Στόχοι υπό διάζευξη (2/3)

1. `f(1, one).`
```
   f(s(1), two).
   f(s(s(1)), three).
   f(s(s(s(X))), N) :- f(X, N).
     ?- f(s(1), A).
     ?- f(s(s(1)), two).
     ?- f(s(s(s(s(s(s(1)))))), C).
```
2. Να οριστεί η σχέση `relatives(X, Y)` η οποία αληθεύει όταν οι `X` και `Y` είναι συγγενείς

Στόχοι υπό διάζευξη (3/3)

3. Να αναδιατυπωθεί η πρόταση

```
translate(Number, Word) :-
  Number = 1, Word = one ;
  Number = 2, Word = two ;
  Number = 3, Word = three.
```

ώστε να μην περιέχει διαζεύξεις

Διαδικαστική σημασία (1/3)

- Έστω μία λίστα από στόχους προς επίλυση

  G1, G2, ……., Gm

- Έστω η πρώτη πρόταση C του προγράμματος της οποίας η κεφαλή H ενοποιείται με το G1

  H :- B1, B2, …….., Bn

- Έστω μια παραλλαγή της C , η C’

  H’ :- B1’, B2’, ……., Bn’

- τέτοια ώστε να μην έχει κοινές μεταβλητές με τα G1, G2, ……, Gm

Διαδικαστική σημασία (2/3)

- Στους προς επίλυση στόχους πρέπει να αντικατασταθεί το G1 με τα B1’, B2’, ………, Bn’ και στη συνέχεια πρέπει να εφαρμοστούν οι αποτιμήσεις μεταβλητών που προέκυψαν κατά την ενοποίηση των G1 και H’ στους νέους προς επίλυση στόχους, παίρνοντας τους

B1’’, B2’’, ….., Bn’’, G2’, ……., Gm’

- Αν κάποια στιγμή δεν βρεθεί πρόταση στο πρόγραμμα με κεφαλή ενοποιήσιμη με τον πρώτο προς επίλυση στόχο, γίνεται οπισθοδρόμηση στην τελευταία επιλογή που έχει γίνει και δοκιμάζεται η επόμενη πρόταση
- Όταν εξαντληθεί η λίστα των στόχων προς επίλυση, σημαίνει επιτυχία

Διαδικαστική σημασία (3/3)

```
1) big(bear).
2) big(elephant).
3) small(cat).
4) brown(bear).
5) black(cat).
6) gray(elephant).
7) dark(Z) :- black(Z).
8) dark(Z) :- brown(Z).
```

```
?- dark(X), big(X).
```



Ο πίθηκος και η μπανάνα (1/4)





Ο πίθηκος και η μπανάνα (2/4)

```
move(state(middle, onbox, middle, hasnot),
     grasp,
     state(middle, onbox, middle, has)).
move(state(P, onfloor, P, H),
     climb,
     state(P, onbox, P, H)).
move(state(P1, onfloor, P1, H),
     push(P1, P2),
     state(P2, onfloor, P2, H)).
```

Ο πίθηκος και η μπανάνα (3/4)

```
move(state(P1, onfloor, B, H),
     walk(P1, P2),
     state(P2, onfloor, B, H)).
canget(state(_, _, _, has)).
canget(State1) :-
      move(State1, Move, State2),
      canget(State2).

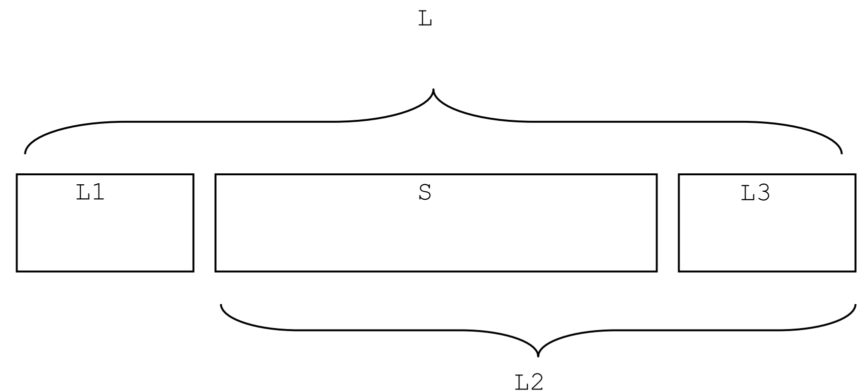

?-  canget(state(atdoor, onfloor, atwindow,
hasnot)).
    yes
```

Ο πίθηκος και η μπανάνα (4/4)

Ατέρμονες ανακυκλώσεις

- `p :- p.`

```
   ?- p.
```

- Πίθηκος και μπανάνα:

```
grasp, climb, push, walk
```



```
walk, grasp, climb, push
```

```
?- canget(state(atdoor, onfloor, atwindow,
hasnot)).
```

Συμπεράσματα

- Ένα πρόγραμμα Prolog δηλωτικά σωστό μπορεί να είναι διαδικαστικά λάθος.
- Μεθοδολογία προγραμματισμού:
- Η διαδικαστική ορθότητα ενός προγράμματος Prolog μπορεί να βασίζεται στη σειρά των προτάσεων που ορίζουν μια διαδικασία.
- Αυτό όμως είναι καλό να το αποφεύγει κανείς.

Ανακατατάξεις προτάσεων και στόχων (1/9)

```
predecessor(Parent, Child) :-
            parent(Parent, Child).
predecessor(Predecessor, Successor):-
            parent(Predecessor, Child),
            predecessor(Child,Successor).
```

Ανακατατάξεις προτάσεων και στόχων (2/9)

```
pred1(X, Z) :-
   parent(X, Z).
pred1(X, Z) :-
   parent(X, Y),
   pred1(Y, Z).
      (1)
```

```
pred2(X, Z) :-
   parent(X, Y),
   pred2(Y, Z).
pred2(X, Z) :-
   parent(X, Z).
      (2)
```

```
pred3(X, Z) :-
   parent(X, Z).
pred3(X, Z) :-
   pred3(X, Y),
   parent(Y, Z).
      (3)
```

```
pred4(X, Z) :-
   pred4(X, Y) ,
   parent(Y, Z).
pred4(X, Z) :-
   parent(X, Z).
      (4)
```

Ανακατατάξεις προτάσεων και στόχων (3/9)

```
?- pred1(tom, pat).        ?- pred2(tom, pat).
   yes                        yes

?- pred3(tom, pat).        ?- pred4(tom, pat).
   yes                        ..............
```

Ανακατατάξεις προτάσεων και στόχων (4/9)

Ανακατατάξεις προτάσεων και στόχων (5/9)



Ανακατατάξεις προτάσεων και στόχων (6/9)



Ανακατατάξεις προτάσεων και στόχων (7/9)



Ανακατατάξεις προτάσεων και στόχων (8/9)

```
pred4(tom, pat)
      |
pred4(tom, Y')
parent(Y', pat)
      |
pred4(tom, Y'')
parent(Y'', Y')
parent(Y', pat)
```

```
pred4(tom, Y''')
parent(Y''', Y'')
parent(Y'', Y')
parent(Y', pat)
      |
.................................
```

Ανακατατάξεις προτάσεων και στόχων (9/9)

- `pred1:`

  Δεν οδηγεί σε ατέρμονα ανακύκλωση
- `pred2:`

  Δεν οδηγεί σε ατέρμονα ανακύκλωση

  Κάνει περιττές εργασίες σε σχέση με το `pred1`
- `pred3:`

  Μερικές φορές δεν οδηγεί σε ατέρμονα ανακύκλωση (`pred3(tom, pat)`)

  Άλλες φορές οδηγεί σε ατέρμονα ανακύκλωση (`pred3(liz, jim)`)
- `pred4:`

  Πάντα οδηγεί σε ατέρμονα ανακύκλωση
- Κανόνας σωστού προγραμματισμού:

  Να δοκιμάζεις τα απλούστερα πράγματα πρώτα ⟶ `pred1`

Prolog και λογική

Κατηγορηματική λογική πρώτης τάξης

(First order predicate logic)

Προτασιακή μορφή

(Clause form)

Προτάσεις Horn

(Horn Clauses)

- Αρχή της ανάλυσης (Resolution principle)
- Ενοποίηση (Unification)

Ενοποίηση (Unification) ←?→ Ταίριασμα (Matching)

```
?- X = f(X).
      X =
f(f(f(f(f(..........
```

Λίστες στην Prolog

- Οι λίστες χρησιμοποιούνται στην αναπαράσταση διατεταγμένων ακολουθιών από αντικείμενα μη προκαθορισμένου πλήθους
- Μια μη κενή λίστα είναι ένας σύνθετος όρος με συναρτησιακό σύμβολο το . και αποτελείται από το στοιχείο κεφαλή (head) και από τη λίστα ουρά (tail)
- Η κενή λίστα συμβολίζεται με το άτομο []

Λίστες στην Prolog




```
.(ann, .(tom, .(bob, .(pat, [])))))
```

|||

```
[ann, tom, bob, pat]
```

```
.(Head, Tail)
```

|||

```
[Head|Tail]
```

```
.(ann, .(tom, .(bob, .(pat, [])))))
```

Ταυτόσημες λίστες

```
[ann, tom, bob, pat]
[ann|[tom, bob, pat]]
[ann|[tom|[bob, pat]]]
[ann|[tom|[bob|[pat]]]]
[ann|[tom|[bob|[pat|[]]]]]
[ann, tom|[bob, pat]]
[ann, tom, bob|[pat]]
[ann, tom, bob, pat|[]]
```

Παραδείγματα λιστών (1/7)

```
[a]
[X]
[X, Y]
[X|Y]
[[]]
[2, 3|L]
```

```
[p(1, 3)|R]
[[1, 2], [3, 4, 5],
[], [6]]
[A, B|[e, f, g]]
[q([2, 7]), r([[4]])]
[[a, b]|[c, d]]
[_|_]
```

Παραδείγματα λιστών (2/7)

- `member(X, L)`: Αληθεύει αν το `X` είναι μέλος της λίστας `L`

  π.χ. `member(b, [a, b, c])` : αληθές

  `member(b, [a, [b, c]])` : ψευδές

  `member([b, c], [a, [b, c]])` : αληθές

- Το `X` είναι μέλος της λίστας `L` είτε αν ταυτίζεται με την κεφαλή της `L` είτε αν είναι μέλος της ουράς της `L`.

```
member(X, [X|Tail]).
member(X, [Head|Tail]) :-
    member(X, Tail).
```

Παραδείγματα λιστών (3/7)

```
?- member(b, [a, b, c]).
   yes
?- member(b, [a, [b, c]]).
   no
?- member([b, c], [a, [b, c]]).
   yes
?- member(X, [1, 2, 3]).
   X = 1    ->  ;
   X = 2    ->  ;
   X = 3
   yes
```

Παραδείγματα λιστών (4/7)

`append(L1, L2, L3)`: Αληθεύει όταν η λίστα `L3` είναι η συνένωση των λιστών `L1` και `L2` (μερικές φορές ορίζεται και με το κατηγόρημα `conc`)

π.χ. `append([a, b], [c, d, e], [a, b, c, d, e])` : αληθές

`append([a], [[b]], [a, b])` : ψευδές

`append([1, 2], [], [1, 2])` : αληθές

Παραδείγματα λιστών (5/7)

i. Η συνένωση της κενής λίστας [] με μία τυχαία λίσταL είναι η λίστα L
ii. Η συνένωση της λίστας με κεφαλή X και ουρά L1 με τη λίστα L2 είναι μία λίστα με κεφαλή X και ουρά L3 αν η συνένωση των λιστών L1 και L2 είναι η λίστα L3




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

Παραδείγματα λιστών (6/7)

```
?- append([a, b], [c, d, e], L).
   L = [a, b, c, d, e]
   yes
?- append([a, [b, c], d], [a, [], b], L).
   L = [a, [b, c], d, a, [], b]
   yes
?- append(L1, L2, [a, b, c]).
   L1 = []
   L2 = [a, b, c]       -> ;
   L1 = [a]
   L2 = [b, c]          -> ;
   L1 = [a, b]
   L2 = [c]             -> ;
   L1 = [a, b, c]
   L2 = []
   yes
```

Παραδείγματα λιστών (7/7)

```
?- append(Before, [may|After],
            [jan, feb, mar, apr, may, jun, jul,
aug, sep, oct, nov, dec]).
   Before = [jan, feb, mar, apr]
   After = [jun, jul, aug, sep, oct, nov, dec]
   yes
?- append(_, [Month1, may, Month2|_],
            [jan, feb, mar, apr, may, jun, jul,
aug, sep, oct, nov, dec]).
   Month1 = apr
   Month2 = jun
   yes
```

Άλλος ορισμός της `member` (1/2)

```
member1(X, L) :-
    append(L1, [X|L2], L).
```

```
member1(X, L) :-
    append(_, [X|_ ], L).
```



Άλλος ορισμός της `member` (2/2)

```
⋮  X = a
   L3' = [b, c]

append(L1', [b|L2], [b, c])
(1)
   L1' = []
   [b|L2] = [b, c]
      ⇓
   L2 = [c]

yes
```

Εισαγωγή στοιχείου στην αρχή της λίστας

- Εισαγωγή στοιχείου στην αρχή της λίστας

```
add(X, L, [X|L]).
```

- Διαγραφή στοιχείου από λίστα

```
del(X, [X|Tail], Tail).
del(X, [Y|Tail], [Y|Tail1]) :-
    del(X, Tail, Tail1).
?- del(a, [a, b, a, a], L).
   L = [b, a, a]      ->   ;
   L = [a, b, a]      ->   ;
   L = [a, b, a]      ->   ;
   yes
```

Διαγραφή στοιχείου από λίστα (1/2)

```
?- del(a, L, [1, 2, 3]).
   L = [a, 1, 2, 3]   ->  ;
   L = [1, a, 2, 3]   ->  ;
   L = [1, 2, a, 3]   ->  ;
   L = [1, 2, 3, a]   ->  ;
   yes

  insert(X, List, BiggerList) :-
    del(X, BiggerList, List).

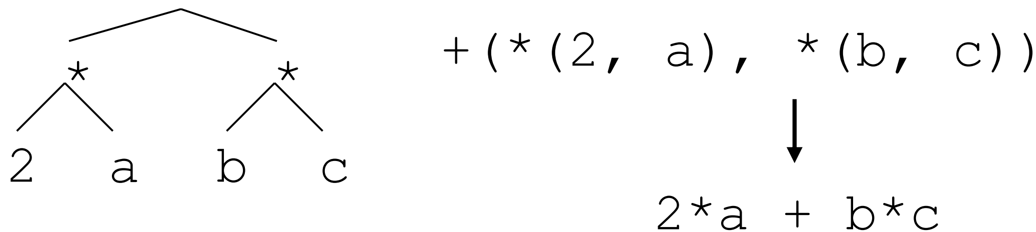
  member2(X, List) :-
    del(X, List, _ ).
```

Διαγραφή στοιχείου από λίστα (2/2)

1) Από μία λίστα L να διαγραφούν τα τρία τελευταία στοιχεία της ώστε να προκύψει η λίστα L1 (μέσω της append).
2) Από μία λίστα L να διαγραφούν τα τρία πρώτα και τα τρία τελευταία στοιχεία της ώστε να προκύψει η λίστα L2.
3) Να οριστεί η σχέση last (Item, List) έτσι ώστε το Item να είναι το τελευταίο στοιχείο της λίστας List
   a) μέσω της append
   b) χωρίς την append

Υπολίστα (1/2)

`sublist(S, L)`: Αληθεύει όταν η `S` είναι υπολίστα της `L`, δηλαδή ένα υποσύνολο διαδοχικών στοιχείων της `L`

π.χ.

`sublist([c, d, e], [a, b, c, d, e, f])` : αληθές

`sublist([c, e], [a, b, c, d, e, f])` : ψευδές



Υπολίστα (2/2)

```
sublist(S, L) :-
      append(L1, L2, L),
      append(S, L3, L2).

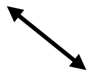
?- sublist(S, [a, b, c]).
   S = []            -> ;
   S = [a]           -> ;
   S = [a, b]        -> ;
   S = [a, b, c]     -> ;
   S = []            -> ;
   S = [b]           -> ;
   .........
```

Αναδιάταξη λίστας (1/2)

```
permutation([], []).
  permutation([X|L], P) :-
     permutation(L, L1),
     insert(X, L1, P).
```

ή

```
  permutation2([], []).
  permutation2(L, [X|P]) :-
     del(X, L, L1),
     permutation2(L1, P).
```

Αναδιάταξη λίστας (2/2)

```
?- permutation2([red, blue, green], P).
   P = [red, blue, green]         -> ;
   P = [red, green, blue]         -> ;
   P = [blue, red, green]         -> ;
   P = [blue, green, red]         -> ;
   P = [green, red, blue]         -> ;
   P = [green, blue, red]
   yes
```

```
?- permutation(L, [a, b, c]).
?- permutation2(L, [a, b, c]).
```

} ΑΠΑΝΤΗΣΕΙΣ;

Αντιστροφή λίστας

```
reverse([], []).
  reverse([First|Rest], Reversed) :-
     reverse(Rest, ReversedRest),
     append(ReversedRest, [First] , Reversed).

?- reverse([a, b, c, d], L).
   L = [d, c, b, a]
   yes
```

Άλλος ορισμός της `reverse`

```
reverse1(List1, List2) :-
     rev_app(List1, [], List2).

  rev_app([], List, List).
  rev_app([X|List1], List2, List3) :-
     rev_app(List1, [X|List2], List3).
```

Χρήση συσσωρευτών (1/2)

- Ο προγραμματισμός με χρήση συσσωρευτών είναι συνήθως λιγότερο δηλωτικός, αλλά επιβαρύνει λιγότερο την εκτέλεση (κυρίως από πλευράς χώρου, αλλά και από πλευράς χρόνου).

```
means(0, zero).
means(1, one).
means(2, two).
means(3, three).
means(4, four).
means(5, five).
means(6, six).
means(7, seven).
means(8, eight).
means(9, nine).
```

Χρήση συσσωρευτών (2/2)

```
translate([], []).
translate([X|L1], [Y|L2]) :-
    means(X, Y),
    translate(L1, L2).

?- translate([3, 5, 1, 3], List).
    List = [three, five, one, three]
    yes
```

Καταγραφή των κινήσεων στο πρόβλημα του πιθήκου και της μπανάνας

```
canget(state(_, _, _, has), []).
  canget(State, [Action|Actions]) :-
     move(State, Action, NewState),
     canget(NewState, Actions).


?- canget(state(atdoor, onfloor, atwindow,
hasnot), Actions).
   Actions = [walk(atdoor, atwindow),
              push(atwindow, middle),
              climb,
              grasp]
   yes
```

Παραδείγματα (1/3)

- `evenlength(List) oddlength(List)`
- `palindrome(List)`
- `shift(List1, List2)`

π.χ.
```
?- shift([1, 2, 3, 4, 5], L1), shift(L1, L2).
   L1 = [2, 3, 4, 5, 1]
   L2 = [3, 4, 5, 1, 2]
   yes
```

Παραδείγματα (2/3)

- `subset(Set, SubSet)`

  π.χ. `?- subset([a, b, c], S).`

```
S = [a, b, c]   -> ;
S = [a, b]      -> ;
S = [a, c]      -> ;
S = [a]         -> ;
S = [b, c]      -> ;
S = [b]         -> ;
S = [c]         -> ;
S = []
yes
```

Παραδείγματα (3/3)

- `dividelist(List, List1, List2)`

  π.χ. `?- dividelist([a, b, c, d, e], L1, L2).`

  ```
  L1 = [a, c, e]
  L2 = [b, d]
  yes
  ```

- `flatten(List, FlatList)`

  π.χ. `?- flatten([a, b, [c, d], [], [[[e]]], f], L).`

  ```
  L = [a, b, c, d, e, f]
  yes
  ```

Τελεστές (Operators)



- προκαθορισμένοι (+, *, ……)
- οριζόμενοι από τον προγραμματιστή με οδηγίες (directives)

```
:- op(600, xfx, has).
has(peter, information).  ⟺  peter has information.
:- op(650, xfx, supports).
supports(floor, table).   ⟺  floor supports table.
```

Προτεραιότητα

Σε μια δομή με τελεστές, εκείνος με τη μεγαλύτερη προτεραιότητα είναι το βασικό συναρτησιακό της σύμβολο.

Είδη τελεστών

- ενδοσημασμένοι (infix) `xfx`, `yfx`, `xfy`
- προσημασμένοι (prefix) `fx`, `fy`
- μετασημασμένοι (postfix) `xf`, `yf`

Οι τελεστές, παρά την ονομασία τους, ΔΕΝ τελούν (προκαλούν) κάτι στα ορίσματα τους

Προτεραιότητα ορίσματος

- Αν ένα όρισμα κάποιου τελεστή είναι μέσα σε παρενθέσεις ή αν δεν είναι σύνθετος όρος στον οποίο συμμετέχουν τελεστές, τότε έχει προτεραιότητα μηδέν.
- Σε αντίθετη περίπτωση, η προτεραιότητα του είναι αυτή του τελεστή που είναι το βασικό συναρτησιακό του σύμβολο.
- Στις εκφράσεις xfy, fx, yf κ.λ.π. το x παριστάνει ένα όρισμα του τελεστή f με προτεραιότητα μικρότερη αυτής του f, ενώ το y είναι όρισμα με προτεραιότητα μικρότερη ή ίση αυτής του τελεστή.
- Κάθε είδος τελεστή έχει καθορισμένη προσεταιριστικότητα (associativity).

Μερικοί προκαθορισμένοι τελεστές (1/7)

1200	xfx	:-
1200	fx	:-, ?-
1100	xfy	;
1000	xfy	,
900	fy	not
700	xfx	=, is, <, >, =<, >=, ==, =\=, \==, =:=
500	yfx	+, -
400	yfx	*, /, div
300	xfx	mod
200	fy	+, -

Μερικοί προκαθορισμένοι τελεστές (2/7)

```
a+b*c   → +(a, *(b, c))
        ⊘→ *(+(a, b), c)  ↔  (a+b)*c

a-b-c   → (a-b)-c
        ⊘→ a-(b-c)
```

- `xfx` : Μη προσεταιριστικοί ενδοσημασμένοι τελεστές
- `yfx` : Αριστερά προσεταιριστικοί ενδοσημασμένοι τελεστές
- `xfy` : Δεξιά προσεταιριστικοί ενδοσημασμένοι τελεστές
- `fx` : Προσημασμένοι τελεστές χωρίς δυνατότητα επανάληψης
- `fy` : Προσημασμένοι τελεστές με δυνατότητα επανάληψης
- `xf` : Μετασημασμένοι τελεστές χωρίς δυνατότητα επανάληψης
- `yf` : Μετασημασμένοι τελεστές με δυνατότητα επανάληψης

Μερικοί προκαθορισμένοι τελεστές (3/7)

```
:- op(800, xfx,
<===>).
:- op(700, xfy, v).
:- op(600, xfy, &).
:- op(500, fy, ~).
~(A&B) <===> ~A v ~B.
```



```
<===>(~(&(A, B)),
v(~(A), ~(B))).
```

```
:- op(100, xfx, in).
:- op(300, fx,
appending).
:- op(200, xfx,
gives).
:- op(100, xfx, and).
:- op(300, fx,
deleting).
:- op(100, xfx, from).
```

Μερικοί προκαθορισμένοι τελεστές (4/7)

```
:- op(100, xfx, in).
:- op(300, fx, appending).
:- op(200, xfx, gives).
:- op(100, xfx, and).
:- op(300, fx, deleting).
:- op(100, xfx, from).

Item in [Item|List].
Item in [First|Rest] :-
   Item in Rest
```

Μερικοί προκαθορισμένοι τελεστές (5/7)

```
appending [] and List gives List.
appending [X|L1] and L2 gives [X|L3] :-
   appending L1 and L2 gives L3.

deleting Item from [Item|Rest] gives Rest.
deleting Item from [First|Rest] gives
[First|NewRest] :-
   deleting Item from Rest gives NewRest.
```

Μερικοί προκαθορισμένοι τελεστές (6/7)

1)
```
:- op(300, xfx, plays).
:- op(200, xfy, and).
jimmy plays football and squash ?
susan plays tennis and basketball and volleyball ?
```

2)
```
:- op(..., ..., was).
:- op(..., ..., of).
:- op(..., ..., the).
diana was the secretary of the department.
?- Who was the secretary of the department.
   Who = diana
   yes
```

Μερικοί προκαθορισμένοι τελεστές (7/7)

```
?- diana was What.
      What = the secretary of the department
      yes
```

3)

```
t(0+1, 1+0).
t(X+0+1, X+1+0).
t(X+1+1, Z) :- t(X+1, X1), t(X1+1, Z).
?- t(0+1, A).
?- t(0+1+1, B).
?- t(1+0+1+1+1, C).
?- t(D, 1+1+1+0).
```

ΑΠΑΝΤΗΣΕΙΣ;

Αριθμητική στην Prolog (1/2)

```
?- X = 1+2.
X = 1+2
yes
```

```
?- X is 1+2.
X = 3
yes
```

- Αριθμητικοί τελεστές: `+`, `-`, `*`, `/`, `mod`
- Ενσωματωμένο κατηγόρημα που προκαλεί τον υπολογισμό αριθμητικών εκφράσεων: `is`
- Τελεστές σύγκρισης: `>`, `<`, `>=`, `=<`, `=:=`, `=\=`

Αριθμητική στην Prolog (2/2)

```
?- 12*7 > 50.
    yes
  ?- 1+2 =:= 2+1.
    yes
  ?- 1+2 = 2+1.
    no
  ?- 1+A = B+2.
    A = 2
    B = 1
    yes
  ?- born(Name, Year),
Year >= 1950, Year =<
1960.

...............
```

```
gcd(X, X, X).
gcd(X, Y, D) :- X < Y, Y1
is Y-X, gcd(X, Y1, D).
gcd(X, Y, D) :- Y < X,
gcd(Y, X, D).
?- gcd(20, 25, D).
   D = 5
   yes
```

Μήκος λίστας

```
length([], 0).
length([_|Tail], N) :-
    length(Tail, N1),
    N is 1+N1.

?- length([a, b, [c, d], e], N).
    N = 4
    yes
```

Άλλος ορισμός της `length` (με χρήση συσσωρευτή)

```
length1(List, N) :-
    length2(List, 0, N).

length2([], N, N).
length2([_|Tail], N1, N) :-
    N2 is 1+N1,
    length2(Tail, N2, N).
```

Αριθμητική έκφραση μήκους λίστας (1/2)

```
length3([], 0).
        length3([_|Tail], N) :-
          length3(Tail, N1),
          N = 1+N1.
        length3([_|Tail], N) :-
          N = 1+N1,
          length3(Tail, N1).
        length3([_|Tail], 1+N) :-
          length3(Tail, N).
```

Ή (first two clauses)

Ή (all three clauses)

Αριθμητική έκφραση μήκους λίστας (2/2)

```
?- length3([a, b, c], N).
        N = 1+(1+(1+0))
        yes

?- length3([a, b, c], N), Length is N.
        N = 1+(1+(1+0))
        Length = 3
        yes
```

Χρήση τελεστών σε if-then κανόνες εμπείρων συστημάτων (1/4)

```
:- op(100, xfx, [has, gives, eats, lays, isa]).
:- op(100, xf, [flies]).

rule1:   if
               Animal has hair
               or
               Animal gives milk
         then
               Animal isa mammal.
```

Χρήση τελεστών σε if-then κανόνες εμπείρων συστημάτων (2/4)

```
rule2:   if
               Animal has feathers
               or
               Animal flies and
               Animal lays eggs
         then
               Animal isa bird.
```

Χρήση τελεστών σε if-then κανόνες εμπείρων συστημάτων (3/4)

```
rule3:   if
              Animal isa mammal and
              (Animal eats meat
              or
              Animal has ’pointed teeth’ and
              Animal has claws and
              Animal has ’forward pointing eyes’)
         then
              Animal isa carnivore.
```

Χρήση τελεστών σε if-then κανόνες εμπείρων συστημάτων (4/4)

1) `max(X, Y, Max)`

2) `maxlist(List, Max)`

3) `sumlist(List, Sum)`

4) `ordered(List)`

5) `subsum(Set, Sum, SubSet)`

π.χ. `?- subsum([1, 2, 5, 3, 2], 5, Sub).`

```
Sub = [1, 2, 2]   -> ;
Sub = [2, 3]      -> ;
Sub = [5]         -> ;
..............
```

6) `between(N1, N2, X)`

π.χ. `?- between(3, 6, X).`

```
X = 3        -> ;
X = 4        -> ;
X = 5        -> ;
X = 6
yes
```

Μερικά ενσωματωμένα κατηγορήματα (1/6)

- `write/1`
- `nl/0`

```
?- write('Hello there!'), nl.
   Hello there!
   yes
```

```
writelist([]).
writelist([X|L]) :- write(X), nl, writelist(L).
?- writelist([f(a, b), 8, haha]).
     f(a, b)
     8
     haha
     yes
```

Μερικά ενσωματωμένα κατηγορήματα (2/6)

```
bars([]).
 bars([N|L]) :- stars(N), nl, bars(L).
 stars(N) :- N > 0, write(*), N1 is N-1, stars(N1).
 stars(N) :- N =< 0.
 ?- bars([3, 4, 6, 5]).
    * * *
    * * * *
    * * * * * *
    * * * * *
    yes
```

Μερικά ενσωματωμένα κατηγορήματα (3/6)

- `fail/0`

```
?- predecessor(pam, X).
     X = bob      -> ;
     X = ann      -> ;
     X = pat      -> ;
     X = jim
     yes
?- predecessor(pam, X), write(X), nl, fail.
     bob
     ann
     pat
     jim
```

Μερικά ενσωματωμένα κατηγορήματα (4/6)

- findall/3

```
?- findall(X, predecessor(pam, X), L).
   X = _0084
   L = [bob, ann, pat, jim]
   yes
?- findall(Y, parent(Y, bla), L).
   Y = _0085
   L = []
   yes
```

Μερικά ενσωματωμένα κατηγορήματα (5/6)

- `var/1`

```
?- var(X).
   X = _0084
   yes
?- var(foo).
   no
?- var(Z), Z = 2.
   Z = 2
   yes
?- Z = 2, var(Z).
   no
```

Μερικά ενσωματωμένα κατηγορήματα (6/6)

- `\=/2`

```
?- f(a) \= g(b).
   yes
?- f(X) \= f(a).
   no
```

- `name/2`

```
?- name(abc, L).
   L = [97, 98, 99]
   yes
?- name(X, [65, 66, 50, 51]).
   X = 'AB23'
   yes
```

Δομημένη πληροφορία σε πρόγραμμα (1/7)

```
family_data(
    person(tom, fox, date(7, may, 1950), works(bbc, 15200)),
    person(ann, fox, date(9, may, 1951), unemployed),
   [person(pat, fox, date(5, may, 1973), unemployed),
    person(jim, fox, date(5, may, 1973), unemployed)]) .


family_data(
    ...............
...
```

Δομημένη πληροφορία σε πρόγραμμα (2/7)

```
husband(X) :- family_data(X, _, _ ).

wife(X) :- family_data(_, X, _).

child(X) :- family_data(_, _, Children),

            member(X, Children).

exists(Person) :- husband(Person) ; wife(Person) ;
child(Person).

dateofbirth(person(_, _, Date, _), Date).

salary(person(_, _, _, works(_, S)), S).

salary(person(_, _, _, unemployed), 0).

total([], 0).
```

Δομημένη πληροφορία σε πρόγραμμα (3/7)

```
total([Person|List], Sum) :- salary(Person, S),
                             total(List, Rest),
                             Sum is S+Rest.
member(.........
...
...
...
husband(X) :- family_data(X, _, _ ).
wife(X) :- family_data(_, X, _).
child(X) :- family_data(_, _, Children),
            member(X, Children).
```

Δομημένη πληροφορία σε πρόγραμμα (4/7)

```
exists(Person) :- husband(Person) ; wife(Person) ;
child(Person).

dateofbirth(person(_, _, Date, _), Date).

salary(person(_, _, _, works(_, S)), S).

salary(person(_, _, _, unemployed), 0).

total([], 0).

total([Person|List], Sum) :- salary(Person, S),
                             total(List, Rest),
                             Sum is S+Rest.

member(.........
```

Δομημένη πληροφορία σε πρόγραμμα (5/7)

```
?- family_data(person(_, fox, _, _), _, _).

?- family_data(_, _, [_, _, _]).

?- family_data(_, person(N, S, _, _), [_, _, _|_]).

?- exists(person(N, S, _, _)).

?- child(X), dateofbirth(X, date(_, _, 1981)).

?- wife(person(N, S, _, works(_, _))).

?- exists(person(N, S, date(_, _, Y), unemployed)),
Y < 1963.
```

Δομημένη πληροφορία σε πρόγραμμα (6/7)

```
?- exists(Person), dateofbirth(Person, date(_, _, Year)),
       Year < 1950, salary(Person, Salary), Salary < 8000.

?- family_data(Husband, Wife, Children),
       total([Husband, Wife|Children], Income).

?- family_data(Husband, Wife, Children),
       total([Husband, Wife|Children], Income),
       length([Husband, Wife|Children], N),  ....
       Income/N < 20000.
```

Δομημένη πληροφορία σε πρόγραμμα (7/7)

- Ονόματα οικογενειών χωρίς παιδιά;
- Όλα τα εργαζόμενα παιδιά;
- Ονόματα οικογενειών με εργαζόμενες συζύγους και άνεργους συζύγους;
- Όλα τα παιδιά με γονείς που διαφέρουν οι ηλικίες τους τουλάχιστον 15 χρόνια;
- `twins(Child1, Child2) :- ............`

Δομημένη πληροφορία σε σύνθετους όρους (1/6)

```
husband(family_str(Husband, _, _), Husband).

wife(family_str(_, Wife, _), Wife).

children(family_str(_, _, Children), Children).

firstchild(Family, First) :- children(Family,
[First|_]).

secondchild(Family, Second) :-

         children(Family, [_, Second|_]).

nthchild(N, Family, Child) :-

         children(Family, Children),

         nth_member(N, Children, Child). ----->  ???
```

Δομημένη πληροφορία σε σύνθετους όρους (2/6)

```
firstname(person(N, _, _, _), N).
surname(person(_, S, _, _), S).
born(person(_, _, D, _), D).

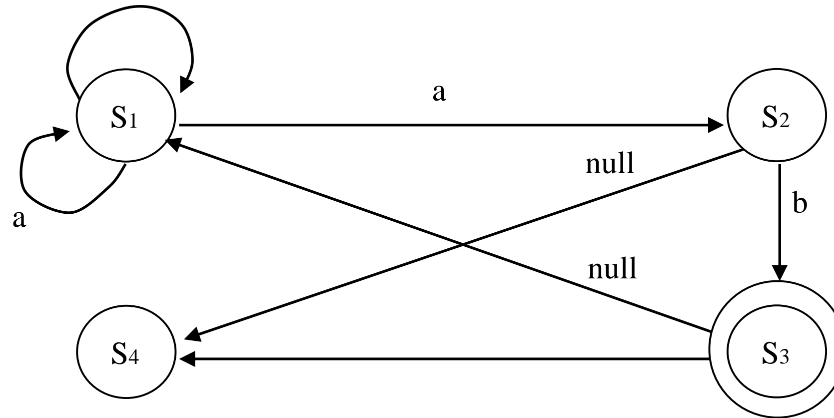

?- firstname(Person1, tom), surname(Person1,
fox),
   firstname(Person2, jim), surname(Person2,
fox),
   husband(Family, Person1),
   secondchild(Family, Person2).
```

Δομημένη πληροφορία σε σύνθετους όρους (3/6)

```
writefamily(family_str(H, W, C)) :-
      nl, write(parents), nl, writeperson(H), nl,
      writeperson(W), nl, write(children), nl,
      writepersonlist(C).

writeperson(person(N, S, date(D, M, Y), W)) :-
      write('   '), write(N), write(' '), write(S),
      write(', born '), write(D), write(' '),
      write(M), write(' '), write(Y), write(', '),
      writework(W).
```

Δομημένη πληροφορία σε σύνθετους όρους (4/6)

```
writepersonlist([]).

writepersonlist([P|L]) :- writeperson(P), nl,
writepersonlist(L).


writework(unemployed) :- write(unemployed).

writework(works(C, S)) :- write('works '), write(C),

      write(', salary '), write(S).
```

Δομημένη πληροφορία σε σύνθετους όρους (5/6)

```
?- writefamily(family_str(
            person(tom, fox, date(7, may, 1950), works(bbc, 15200)),
            person(ann, fox, date(9, may, 1951), unemployed),
           [person(pat, fox, date(5, may, 1973), unemployed),
            person(jim, fox, date(5, may, 1973), unemployed)])).
```

Δομημένη πληροφορία σε σύνθετους όρους (6/6)

```
parents
    tom fox, born 7 may 1950, works bbc, salary 1500
    ann fox, born 9 may 1951, unemployed
children
    pat fox, born 5 may 1973, unemployed
    jim fox, born 5 may 1973, unemployed
```

Προσομοίωση μη ντετερμινιστικού αυτομάτου (1/3)




```
final(s3).

trans(s1, a, s1).
trans(s1, a, s2).
trans(s1, b, s1).
trans(s2, b, s3).
trans(s3, b, s4).

silent(s2, s4).
silent(s3, s1).
```

```
accepts(State, []) :-
    final(State).

accepts(State, [X|Rest]) :-
    trans(State, X, State1),
    accepts(State1, Rest).

accepts(State, List) :-
    silent(State, State1),
    accepts(State1, List).
```

Προσομοίωση μη–ντετερμινιστικού αυτομάτου (2/3)

```
?- accepts(s1, [a, a, a,
b]).
   yes

?- accepts(S, [a, b]).
   S = s1       -> ;
   S = s3
   yes

?- accepts(s1, [X1, X2,
X3]).
   X1 = a
   X2 = a
   X3 = b     -> ;
.
.
.
.
   X1 = b
   X2 = a
   X3 = b
   yes

?- String = [_, _, _],
accepts(s1, String).
   String = [a, a, b]
-> ;
   String = [b, a, b]
   yes
```

Προσομοίωση μη–ντετερμινιστικού αυτομάτου (3/3)

- Αν προστεθεί στο πρόγραμμα το:

```
silent(s1, s3).
```

- Ποια είναι η απάντηση του:

```
?- accepts(s1, [a]).
```

```
accepts(State, String, MaxMoves) :-
    ....................
```

Προγραμματισμός ταξιδιού (1/7)

```
:- op(200, xfx, :).


timetable(edinburgh, london,
      [9:40/10:50/ba4733/alldays,13:40/14:50/ba4773/allday
      s,19:40/20:50/ba4833/[mo,tu,we,th,fr,su]]).

timetable(london, edinburgh,[9:40/10:50/ba4732/alldays,
      11:40/12:50/ba4752/alldays,18:40/19:50/ba4822/[mo,tu
      ,we,th,fr]]).

timetable(london, ljubljana,
      [13:20/16:20/yu201/[fr],13:20/16:20/yu213/[su]]).

timetable(london, zurich,[9:10/11:45/ba614/alldays,
      14:45/17:20/sr805/alldays]).
```

Προγραμματισμός ταξιδιού (2/7)

```
timetable(london, milan,
      [8:30/11:20/ba510/alldays,11:00/13:50/az45
      9/alldays]).

timetable(ljubljana, zurich,
      [11:30/12:40/yu322/[tu,th]]).

timetable(ljubljana, london,    [11:10/12:20/yu200/[fr],
      11:25/12:20/yu212/[su]]).

timetable(milan, london, [9:10/10:00/az458/alldays,
      12:20/13:10/ba511/alldays]).

timetable(milan, zurich, [9:25/10:15/sr621/alldays,
      12:45/13:35/sr623/alldays]).

timetable(zurich, ljubljana,
      [13:30/14:40/yu323/[tu,th]]).
```

Προγραμματισμός ταξιδιού (3/7)

```
timetable(zurich, london,
      [9:00/9:40/ba613/[mo,tu,we,th,fr,sa],
      16:10/16:55/sr806/[mo,tu,we,th,fr,su]]).

timetable(zurich, milan,[7:55/8:45/sr620/alldays]).

route(P1, P2, Day, [P1:P2-Fnum-Deptime]) :-

   flight(P1, P2, Day, Fnum, Deptime, _).

route(P1, P2, Day, [P1:P3-Fnum1-Dep1|Route]) :-

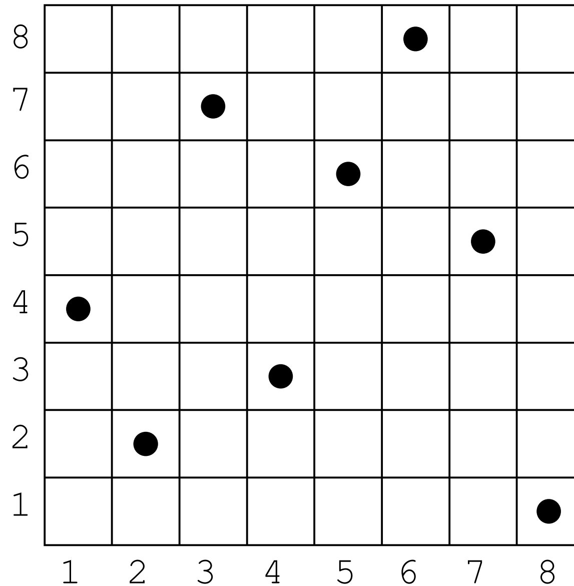
   route(P3, P2, Day, Route),

   flight(P1, P3, Day, Fnum1, Dep1, Arr1),

   deptime(Route, Dep2),

   transfer(Arr1, Dep2).
```

Προγραμματισμός ταξιδιού (4/7)

```
flight(Place1, Place2, Day, Fnum, Deptime, Arrtime) :-
   timetable(Place1, Place2, Flightlist),
   member(Deptime/Arrtime/Fnum/Daylist, Flightlist),
   flyday(Day, Daylist).

flyday(Day, Daylist) :-
   member(Day, Daylist).
flyday(Day, alldays) :-
   member(Day, [mo, tu, we, th, fr, sa, su]).

deptime([P1:P2-Fnum-Dep|_], Dep).
```

Προγραμματισμός ταξιδιού (5/7)

```
?- flight(london, ljubljana, Day, _, _, _).
   Day = fr      -> ;
   Day = su
   yes
?- route(ljubljana, edinburgh, th, R).
   R = [ljubljana:zurich-yu322-11:30,
        zurich:london-sr806-16:10,
        london:edinburgh-ba4822-18:40]
   yes
```

Προγραμματισμός ταξιδιού (6/7)

```
?- permutation([milan, ljubljuana, zurich],
               [C1, C2, C3]),
   flight(london, C1, tu, FN1, Dep1, Arr1),
   flight(C1, C2, we, FN2, Dep2, Arr2),
   flight(C2, C3, th, FN3, Dep3, Arr3),
   flight(C3, london, fr, FN4, Dep4, Arr4).
   C1 = milan
   C2 = zurich
   C3 = ljubljana
```

Προγραμματισμός ταξιδιού (7/7)

```
FN1 = ba510
  Dep1 = 8:30
  Arr1 = 11:20
  FN2 = sr621
  Dep2 = 9:25
  Arr2 = 10:15
  yes
```

```
FN3 = yu323
Dep3 = 13:30
Arr3 = 14:40
FN4 = yu200
Dep4 = 11:10
Arr4 = 12:20
```

Το πρόβλημα των 8 βασιλισσών, 1ος τρόπος (1/2)




```
[X1/Y1, X2/Y2, X3/Y3, ..., X8/Y8]

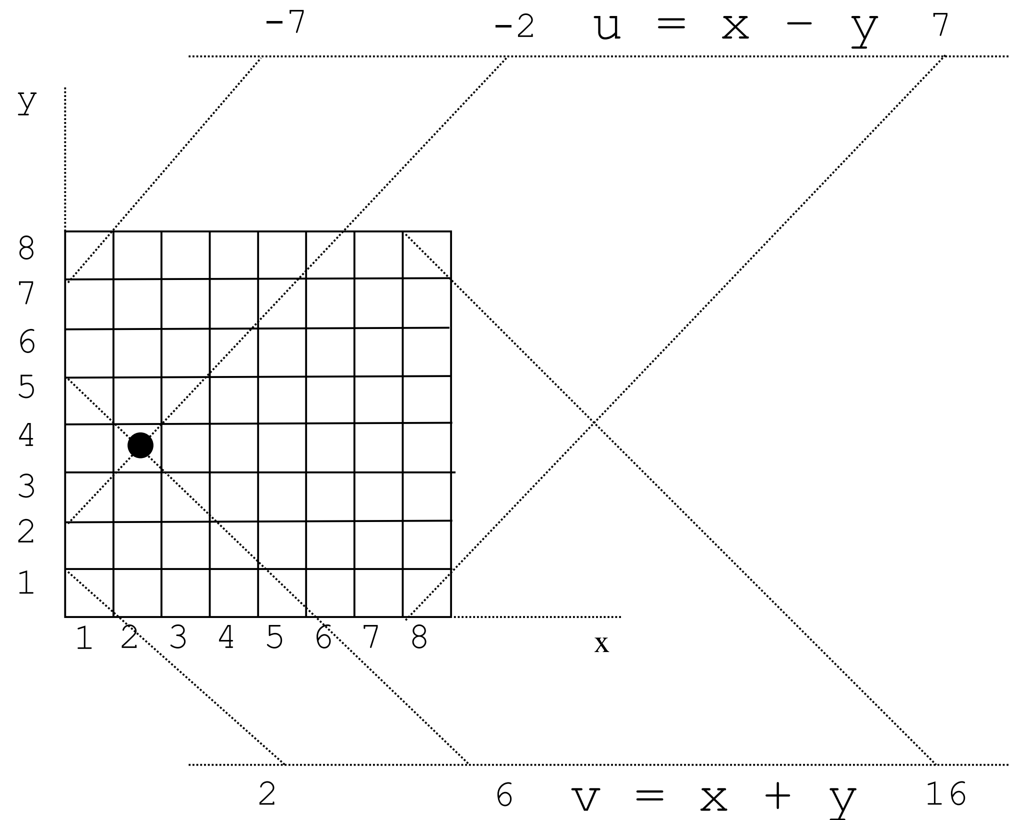
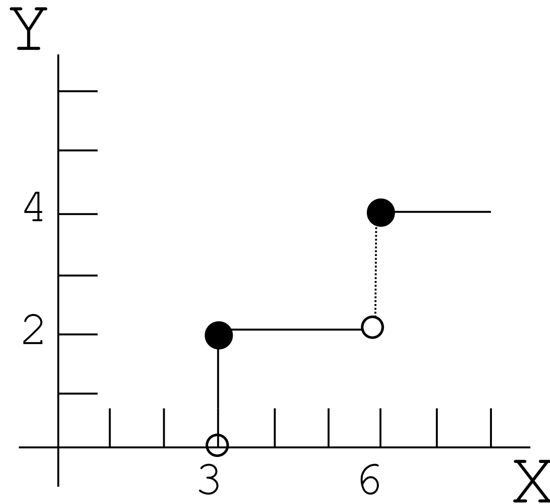
[1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]

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

```
solution([]).
solution([X/Y|Others]) :-
    solution(Others), member(Y, [1,2,3,4,5,6,7,8]),
    noattack(X/Y, Others).
```

Το πρόβλημα των 8 βασιλισσών, 1ος τρόπος (2/2)

```
noattack(_, []).
noattack(X/Y, [X1/Y1|Others]) :-
    Y =\= Y1, Y1-Y =\= X1-X, Y1-Y =\= X-X1,
    noattack(X/Y, Others).
member(............
template([1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7,
8/Y8]).
?- template(S), solution(S).
    S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1]      -> ;
    S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1]      -> ;
    S = [1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1]      -> ;
```

Το πρόβλημα των 8 βασιλισσών, 2ος τρόπος (1/3)

```
[Y1, Y2, Y3,... ,Y8]

solution(Queens) :-
   permutation([1,2,3,4,5,6,7,8], Queens),
   safe(Queens).

permutation([], []).
permutation([Head|Tail], PermList) :-
   permutation(Tail, PermTail),
   del(Head, PermList, PermTail).
```

Το πρόβλημα των 8 βασιλισσών, 2ος τρόπος (2/3)

```
del(Item, [Item|List], List).
del(Item, [First|List], [First|List1]) :-
    del(Item, List, List1).

safe([]).
safe([Queen|Others]) :-
    safe(Others),
    noattack(Queen, Others, 1).
```

Το πρόβλημα των 8 βασιλισσών, 2ος τρόπος (3/3)

```
noattack(_, [], _ ).
noattack(Y, [Y1|Ylist], Xdist) :-
    Y1-Y =\= Xdist, Y-Y1 =\= Xdist,
    Dist1 is Xdist+1,
    noattack(Y, Ylist, Dist1).


?- solution(S).
```

Το πρόβλημα των 8 βασιλισσών, 3ος τρόπος (1/2)

x: στήλη

y: γραμμή

u: ανιούσα διαγώνιος

v: κατιούσα διαγώνιος

u = x – y

v = x + y



Το πρόβλημα των 8 βασιλισσών, 3ος τρόπος (2/2)

```
solution(Ylist) :- sol(Ylist, [1,2,3,4,5,6,7,8],
     [1,2,3,4,5,6,7,8], [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],
     [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).
sol([], [], Dy, Du, Dv).
sol([Y|Ylist], [X|Dx1], Dy, Du, Dv) :-
   del(Y, Dy, Dy1),
   U is X-Y,
  del(U, Du, Du1),
   V is X+Y,
   del(V, Dv, Dv1),
   sol(Ylist, Dx1, Dy1, Du1, Dv1).
del(Item, [Item|List], List).
del(Item, [First|List], [First|List1]) :-
   del(Item, List, List1).
```

Γενίκευση για N βασίλισσες (1/2)

```
gen(N, N, [N]).
gen(N1, N2, [N1|List]) :-
   N1 < N2,
   M is N1+1,
   gen(M, N2, List).
solution(N, S) :-
   gen(1, N, Dxy),
   Nu1 is 1-N,
   Nu2 is N-1,
   gen(Nu1, Nu2, Du),
   Nv2 is N+N,
   gen(2, Nv2, Dv),
   sol(S, Dxy, Dxy, Du, Dv).
```

Γενίκευση για Ν βασίλισσες (2/2)

```
?- solution(12, S).
   S = [1,3,5,8,10,12,6,11,2,7,9,4] -> ;
   ......
```

- `jump(Square1, Square2) :- ......` (κίνηση αλόγου στο σκάκι)
- `knightpath(Path) :- ......` (νόμιμη διαδρομή αλόγου σε άδεια σκακιέρα)
- Να βρεθούν οι δυνατές διαδρομές 4 κινήσεων ενός αλόγου σε άδεια σκακιέρα που ξεκινούν από το τετράγωνο 2/1, καταλήγουν στην απέναντι πλευρά της σκακιέρας (Y = 8) και στη δεύτερη κίνηση περνούν από το τετράγωνο 5/4.

Έλεγχος στην οπισθοδρόμηση (1/4)




```
f(X, 0) :- X < 3.
f(X, 2) :- 3 =< X, X < 6.
f(X, 4) :- 6 =< X.
?- f(1, Y), 2 < Y.
```

- Περιττή οπισθοδρόμηση
- Ενσωματωμένο κατηγόρημα `!/0` (cut)

Έλεγχος στην οπισθοδρόμηση (2/4)

- 2ος ορισμός του `f/2`

```
f(X, 0) :- X < 3, !.
f(X, 2) :- 3 =< X, X < 6, !.
f(X, 4) :- 6 =< X.
```

```
?- f(1, Y), 2 < Y.
?- f(7, Y).
```

- 3ος ορισμός του `f/2`

```
f(X, 0) :- X < 3, !.
f(X, 2) :- X < 6, !.
f(X, 4).
```

```
?- f(7, Y).
```

- Στο 2ο ορισμό τα ! βοηθούν στην απόδοση. Μπορούν να διαγραφούν χωρίς να αλλάξει η σημασία του προγράμματος.
- Στον 3ο ορισμό τα ! είναι απαραίτητα για την ορθότητα του προγράμματος.

Έλεγχος στην οπισθοδρόμηση (3/4)

- Το `!` επιτυγχάνει πάντοτε κατά την εμπρόσθια φορά του ελέγχου. Σε οπισθοδρόμηση θεωρείται ότι αποτυγχάνει ο στόχος που ενεργοποίησε τον κανόνα στο σώμα του οποίου βρίσκεται το `!`.

```
C :- P, Q, R, !, S, T, U.

C :- V.

A :- B, C, D.

          ?- A.
```

- Κατά τη διάρκεια ικανοποίησης του στόχου `C`, οπισθοδρόμηση επιτρέπεται στην ομάδα στόχων `P`, `Q`, `R`. Όταν ο έλεγχος περάσει μετά το `!`, αγνοούνται τυχόν εναλλακτικοί τρόποι ικανοποίησης των `P` , `Q` , `R` αλλά και άλλες προτάσεις για το `C` (`C :- V`). Οπισθοδρόμηση μπορεί να γίνει μόνο στην ομάδα στόχων `S`, `T`, `U`.

Έλεγχος στην οπισθοδρόμηση (4/4)

- Για να ικανοποιηθεί το C αρκεί να ικανοποιηθούν τα P, Q, R (και αν συμβεί αυτό, τότε αυτός είναι ο μοναδικός τρόπος ικανοποίησης του C) και στη συνέχεια τα S, T, U. Αν δεν ικανοποιούνται τα P, Q, R, τότε αρκεί να ικανοποιηθεί το V για να θεωρηθεί ότι ικανοποιείται το C.

Μέγιστος δύο αριθμών

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.
```

ή με χρήση του !

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

Μέλος λίστας (ντετερμινιστικός ορισμός)

```
member(X, [X|L]) :- ! .
member(X, [Y|L]) :- member(X, L).
   ?- member(f(Z), [g(a), f(b), g(c), f(d)]).
      Z = b
      yes
```

Προσθήκη στοιχείου σε λίστα (χωρίς διπλή εμφάνιση) (1/5)

```
add(X, L, L) :- member(X, L), !.
add(X, L, [X|L]).
  ?- add(a, [b, c], L).
     L = [a, b, c]
     yes
  ?- add(b, [a, b, c], L).
     L = [a, b, c]
     yes
beat(tom, jim).
beat(ann, tom).
beat(pat, jim).
```

Προσθήκη στοιχείου σε λίστα (χωρίς διπλή εμφάνιση) (2/5)

- Θέλουμε να κατατάξουμε τους παίκτες σε κατηγορίες.

Ένας παίκτης είναι:

`winner`: αν κέρδισε κάθε παιχνίδι που έπαιξε

`fighter`: αν άλλα παιχνίδια κέρδισε και άλλα έχασε

`sportsman`: αν έχασε κάθε παιχνίδι που έπαιξε

```
class(X, fighter) :- beat(X, _),
                     beat(_, X),
                     !.
class(X, winner) :- beat(X, _),
                    !.
```

Προσθήκη στοιχείου σε λίστα (χωρίς διπλή εμφάνιση) (3/5)

```
class(X, sportsman) :- beat(_, X).
  ?- class(tom, C).
     C = fighter
     yes
  ?- class(jim, C).
     C = sportsman
     yes
  ?- class(tom, sportsman).
     yes                              ???
```

- Συνήθως, προγράμματα με `!`, λόγω της κατασκευής τους, προϋποθέτουν ένα συγκεκριμένο τρόπο χρήσης τους. π.χ. `class(`<Atom>`, `<Variable>`)`

Προσθήκη στοιχείου σε λίστα (χωρίς διπλή εμφάνιση) (4/5)

(1)
```
p(1).
p(2) :- !.
p(3).
  a) ?- p(X).
  b) ?- p(X), p(Y).
  c) ?- p(X), !, p(Y).
```

(2)
```
class(Number, positive) :- Number > 0.
class(0, zero).
class(Number, negative) :- Number < 0.
```

- Να οριστεί το `class/2` με χρήση `!`.

Προσθήκη στοιχείου σε λίστα (χωρίς διπλή εμφάνιση) (5/5)

(3) `split(Numbers, Positives, Negatives)`

```
?- split([3, -1, 0, 5, -2], P, N).
        P = [3, 0, 5]
        N = [-1, -2]
        yes
```

- Να οριστεί το `split/3` με δύο τρόπους, με `!` και χωρίς `!`.

Άρνηση στην Prolog (μέσω αποτυχίας) (1/3)

```
likes(mary, X) :-
    snake(X), !, fail.
likes(mary, X) :-
    animal(X).
```

```
likes(mary, X) :-
    snake(X), !, fail
    ;
    animal(X).
```

```
different(X, X) :-
    !,
    fail.
different(X, Y).
```

```
different(X, Y) :-
    X = Y, !, fail
    ;
    true.
```

(ενσωματωμένο `true/0`)

Άρνηση στην Prolog (μέσω αποτυχίας) (2/3)

```
not(Goal) :-
    call(Goal), !, fail
    ;
    true.
```

(ενσωματωμένο `call/1`)

`not(snake(X))` ⟷ `not snake(X)`

```
likes(mary, X) :-
    animal(X),
    not snake(X).

different(X, Y) :-
    not (X = Y).
```

Άρνηση στην Prolog (μέσω αποτυχίας) (3/3)

```
class(X, fighter) :- beat(X, _),
                     beat(_, X).
class(X, winner) :-  beat(X, _),
                     not beat(_, X).
class(X, sportsman) :- beat(_, X),
                       not beat(X, _).
```

Άλλος τρόπος για την πρώτη εκδοχή του προβλήματος των 8 βασιλισσών

```
solution([]).
solution([X/Y|Others]) :-
   solution(Others),
   member(Y, [1,2,3,4,5,6,7,8]),
   not attacks(X/Y, Others).

attacks(X/Y, Others) :-
   member(X1/Y1, Others),
   (Y1 = Y ;
    Y1 is Y+X1-X ;
    Y1 is Y-X1+X).
member(........
template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

Συμπέρασμα (1/4)

Η άρνηση στην Prolog ορίζεται μέσω του συνδυασμού `!` και `fail`

1. Να διατυπωθεί σε Prolog η ερώτηση:

   Ποια στοιχεία της λίστας `Candidates` δεν ανήκουν στη λίστα `RuledOut`; Να βρεθούν όλα μέσω οπισθοδρόμησης.

2. Να οριστεί η διαφορά συνόλων

```
difference(Set1, Set2, SetDifference).
  ?- difference([a,b,c,d], [b,d,e,f], D).
     D = [a,c]
     yes
```

Συμπέρασμα (2/4)

3. Να οριστεί το

   `unifiable(List1, Term, List2)`

   έτσι ώστε η λίστα `List2` να περιέχει εκείνα τα στοιχεία της λίστας `List1` που είναι ενοποιήσιμα με τον όρο `Term`, χωρίς όμως να γίνουν οι σχετικές ενοποιήσεις.

```
?- unifiable([X, b, t(Y)], t(a), List).
     X = _0084
     Y = _0086
     List = [_0084, t(_0086)]
     yes
```

Συμπέρασμα (3/4)

- Ένα ! που η διαγραφή του από το πρόγραμμα δεν μεταβάλλει τη δηλωτική σημασία του προγράμματος είναι πράσινο !.
- Ένα ! που η διαγραφή του από το πρόγραμμα επηρεάζει τη δηλωτική σημασία του προγράμματος είναι κόκκινο !.
- Τα κόκκινα ! πρέπει να χρησιμοποιούνται με προσοχή, ενώ τα πράσινα ! είναι πιο ακίνδυνα.
- Ο ορισμός της άρνησης στην Prolog βασίζεται στην υπόθεση του κλειστού κόσμου: ισχύει μόνο ό,τι έχει δηλωθεί . Γι’ αυτό χρειάζεται προσοχή στη χρήση της.

Συμπέρασμα (4/4)

```
?- not human(mary).
   yes

  good_standard(jeanluis).
  expensive(jeanluis).
  good_standard(francesco).
  reasonable(Restaurant) :- not
      expensive(Restaurant).
  ?- good_standard(X), reasonable(X).
     X = francesco
     yes
  ?- reasonable(X), good_standard(X).
     no
```

Κι άλλα ενσωματωμένα κατηγορήματα εισόδου/ εξόδου (1/8)

- `read/1`

```
?- read(X).
   p(bla).
   X = p(bla)
   yes
```

```
cube(N, C) :- C is N*N*N.
?- cube(2, X).
   X = 8
   yes
```

```
?- cube(12, Z).
   Z = 1728
   yes
```

Κι άλλα ενσωματωμένα κατηγορήματα εισόδου/ εξόδου (2/8)

```
cube :- write('Next item, please: '),
         read(X), process(X).

process(stop) :- !.

process(N) :- C is N*N*N, write('Cube of '),
              write(N), write(' is '),
              write(C), nl, cube.

?- cube.
   Next item, please: 5.
   Cube of 5 is 125
   Next item, please: 10
   Cube of 10 is 1000
   Next item, please: stop
   yes
```

Κι άλλα ενσωματωμένα κατηγορήματα εισόδου/ εξόδου (3/8)

- put/1

```
?- put(65), put(66), put(67).
    ABC
    yes
```

get0/1, get/1

- ```
  ?- get0(C1), get0(C2), get0(C3).
      a  b
      C1 = 97
      C2 = 32
      C3 = 98
      yes
  ```

...

...

```
?- get(C1), get(C2), get(C3).
    a      b     c
    C1 = 97
    C2 = 98
    C3 = 99
    yes
```

```

# Κι άλλα ενσωματωμένα κατηγορήματα εισόδου/ εξόδου (4/8)

- Τα ενσωματωμένα κατηγορήματα εισόδου/εξόδου είναι πολύ εξειδικευμένα και ποικίλλουν αρκετά μεταξύ των διαφορετικών συστημάτων Prolog. Ακριβείς προδιαγραφές τους δίνονται στο εγχειρίδιο της γλώσσας που χρησιμοποιείται.
- Τα παραπάνω ισχύουν πολύ περισσότερο για το χειρισμό αρχείων.

```
consult/1, reconsult/1
```

Κι άλλα ενσωματωμένα κατηγορήματα εισόδου/ εξόδου (5/8)

```
getsentence(Wordlist) :-
    get0(Char),
    getrest(Char, Wordlist).

getrest(46, []) :- !.
getrest(32, Wordlist) :-
    !,
    getsentence(Wordlist).
getrest(Letter, [Word|Wordlist]) :-
    getletters(Letter, Letters, Nextchar),
    name(Word, Letters),
    getrest(Nextchar, Wordlist).
```

Κι άλλα ενσωματωμένα κατηγορήματα εισόδου/ εξόδου (6/8)

```
getletters(46, [], 46) :- !.
getletters(32, [], 32) :- !.
getletters(Let, [Let|Letters], Nextchar) :-
   get0(Char),
   getletters(Char, Letters, Nextchar).

?- getsentence(Wordlist).
   Mary was pleased to see the robot fail.
   Wordlist = ['Mary', was, pleased, to, see, the,
               robot, fail]
   yes
```

Κι άλλα ενσωματωμένα κατηγορήματα εισόδου/ εξόδου (7/8)

1. Να οριστεί το `starts(Atom, Character)` έτσι ώστε το `Atom` να ξεκινά με τον `Character`
2. Να οριστεί το `plural(Singular, Plural)` έτσι ώστε το `Plural` να είναι το ουσιαστικό `Singular` σε πληθυντικό αριθμό, π.χ.

```
?- plural(table, X).
     X = tables
     yes
```

Κι άλλα ενσωματωμένα κατηγορήματα εισόδου/ εξόδου (8/8)

- Πόσο εύκολο είναι να γίνει πλήρης ο ορισμός αυτός;
- Για παράδειγμα, να παίρνουμε:

```
 ?- plural(country, X).
    X = countries
    yes
?- plural(child, X).
    X = children
    yes
```

Κι άλλα ενσωματωμένα κατηγορήματα ελέγχου τύπων (1/7)

- nonvar/1
- integer/1
- atom/1

```
?- nonvar(X).
   no
?- nonvar(bbb).
   yes
?- Z = 2, integer(Z),
nonvar(Z).
   Z = 2
   yes
```

```
?- integer(47).
   yes
?- integer(c(8)).
   no
```

Κι άλλα ενσωματωμένα κατηγορήματα ελέγχου τύπων (2/7)

```
?- atom(22).
   no
?- atom(==>).
   yes
?- atom(X), X =
costas.
   no
?- X = costas,
atom(X).
   X = costas
   yes
```

```
?- atom(p(1)).
   no
?- atom(foo).
   yes
```

Κι άλλα ενσωματωμένα κατηγορήματα ελέγχου τύπων (3/7)

```
 DONALD
 GERALD +
--------
 ROBERT
```

```
?- sum([D,O,N,A,L,D],
       [G,E,R,A,L,D],
       [R,O,B,E,R,T]).
```

```
sum(N1, N2, N) :-
   sum1(N1, N2, N, 0, 0, [0,1,2,3,4,5,6,7,8,9], _).
sum1([], [], [], C, C, Digits, Digits).
sum1([D1|N1], [D2|N2], [D|N], C1, C, Digs1, Digs) :-
   sum1(N1, N2, N, C1, C2, Digs1, Digs2),
   digitsum(D1, D2, C2, D, C, Digs2, Digs).
```

Κι άλλα ενσωματωμένα κατηγορήματα ελέγχου τύπων (4/7)

```
digitsum(D1, D2, C1, D, C, Digs1, Digs) :-
    del(D1, Digs1, Digs2),
    del(D2, Digs2, Digs3),
    del(D, Digs3, Digs),
    S is D1+D2+C1,
    D is S mod 10,
    C is S//10.
del(A, L, L) :-  nonvar(A), !.
del(A, [A|L], L).
del(A, [B|L], [B|L1]) :- del(A, L, L1).
puzzle1([D,O,N,A,L,D], [G,E,R,A,L,D], [R,O,B,E,R,T]).
puzzle2([0,S,E,N,D], [0,M,O,R,E], [M,O,N,E,Y]).
```

Κι άλλα ενσωματωμένα κατηγορήματα ελέγχου τύπων (5/7)

```
?- puzzle1(N1, N2, N), sum(N1, N2, N).
   N1 = [5,2,6,4,8,5]
   N2 = [1,9,7,4,8,5]
   N = [7,2,3,9,7,0]
   yes
```

1. Να οριστεί το `simplify/2` έτσι ώστε να επιτελεί απλοποιήσεις αριθμητικών εκφράσεων που είναι αθροίσματα ακεραίων αριθμών και σταθερών, π.χ.

Κι άλλα ενσωματωμένα κατηγορήματα ελέγχου τύπων (6/7)

```
?- simplify(1+1+a, E).
       E = a+2
       yes
   ?- simplify(1+a+4+2+b+c, E).
       E = a+b+c+7
       yes
   ?- simplify(3+x+x, E).
       E = 2*x+3
       yes
```

Κι άλλα ενσωματωμένα κατηγορήματα ελέγχου τύπων (7/7)

2. Να οριστεί το `add(Item, List)` το οποίο να εισάγει το στοιχείο `Item`, που είναι άτομο, στη λίστα `List`. Η `List` είναι μια λίστα από άτομα με ουρά που είναι μεταβλητή. Το `add/2` πρέπει να κάνει το `Item` πρώτο στοιχείο της ουράς της `List`, η οποία να αποκτά μια νέα ουρά–μεταβλητή. Π.χ.

```
?- List = [a, b, c|Tail], add(d, List).
      List = [a, b, c, d|_0086]
      Tail = [d|_0086]
      yes
```

Ενσωματωμένα κατηγορήματα χειρισμού όρων (1/8)

- =../2

```
?- f(a, b) =.. L.
   L = [f, a, b]
   yes
?- T =.. [rectangle, 3, 5].
   T = rectangle(3, 5)
   yes
?- Z =.. [p, X, f(X, Y)].
   Z = p(_0082, f(_0082, _0083))
   X = _0082
   Y = _0083
   yes
```

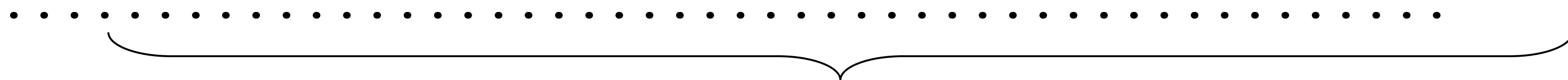
Ενσωματωμένα κατηγορήματα χειρισμού όρων (2/8)

- `functor/3, arg/3`

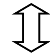
```
?- functor(t(f(X), X, a), Fun, Arity).
   Fun = t
   Arity = 3
   X = _0087
   yes
?- arg(2, f(X, t(a), t(b)), Y).
   X = _0083
   Y = t(a)
   yes
```

Ενσωματωμένα κατηγορήματα χειρισμού όρων (3/8)

```
 ?- functor(D, date, 3), arg(1, D, 29),
     arg(2, D, june), arg(3, D, 1982).
     D = date(29, june, 1982)
     yes
enlarge(square(A), F, square(A1)) :-
   A1 is F*A.
enlarge(circle(R), F, circle(R1)) :-
   R1 is F*R.
enlarge(rectangle(A, B), F, rectangle(A1, B1)) :-
   A1 is F*A, B1 is F*B.
................................................
```



Ενσωματωμένα κατηγορήματα χειρισμού όρων (4/8)



```
enlarge(Fig, F, Fig1) :-
    Fig =.. [Type|Parameters],
    multiplylist(Parameters, F, Parameters1),
    Fig1 =.. [Type|Parameters1].
multiplylist([], _, []).
multiplylist([X|L], F, [X1|L1]) :-
    X1 is F*X,
    multiplylist(L, F, L1).
```

Ενσωματωμένα κατηγορήματα χειρισμού όρων (5/8)

```
?- enlarge(circle(5), 3, Fig).
   Fig = circle(15)
   yes
?- enlarge(triangle(4, 5, 8), 2, Fig).
   Fig = triangle(8, 10, 16)
   yes
substitute(Term, Term, Term1, Term1) :- !.
substitute(_, Term, _, Term) :-
    (atom(Term) ;
     integer(Term)),
    !.
```

Ενσωματωμένα κατηγορήματα χειρισμού όρων (6/8)

```
substitute(Sub, Term, Sub1, Term1) :-
   Term =.. [F|Args],
   substlist(Sub, Args, Sub1, Args1),
   Term1 =.. [F|Args1].
substlist(_, [], _, []).
substlist(Sub, [Term|Terms], Sub1, [Term1|Terms1])
:-
   substitute(Sub, Term, Sub1, Term1),
   substlist(Sub, Terms, Sub1, Terms1).
```

Ενσωματωμένα κατηγορήματα χειρισμού όρων (7/8)

```
?- substitute(sin(x), 2*sin(x)*f(sin(x)), t, F).
   F = 2*t*f(t)
   yes
?- substitute(a+b, f(a, A+B), v, F).
   A = a
   B = b
   F = f(a, v)
   yes
```

Ενσωματωμένα κατηγορήματα χειρισμού όρων (8/8)

- Να οριστεί το `ground(Term)` έτσι ώστε να είναι αληθές όταν ο όρος `Term` δεν περιέχει μεταβλητές χωρίς τιμή.

```
ground(Term) :-
    nonvar(Term),
    Term =.. [_|Args],
    ground_list(Args).
ground_list([]).
ground_list([Arg|Args]) :-
    ground(Arg),
    ground_list(Args).
```

- Πως ελέγχεται η περίπτωση ατόμων ή αριθμών;

Ισότητες (1/2)

- “Ισότητα” ενοποίησης: `X = Y` (`X \= Y`)
- “Ισότητα” ανάθεσης τιμής αριθμητικής έκφρασης: `X is E`
- “Ισότητα” τιμών αριθμητικών εκφράσεων: `E1 =:= E2` (`E1 =\= E2`)
- “Ισότητα” κατά λέξη: `T1 == T2` (`T1 \== T2`)

```
?- f(a, b) == f(a, b).

   yes

?- f(a, X) == f(a, Y)

   no
```

```
?- f(a, b) == f(a, X)
   no

?- X \== Y
   X = _0084
   Y = _0098
   yes
```

Ισότητες (2/2)

```
?- t(X, f(a, Y)) == t(X,
f(a, Y)).
     X = _0084
     Y = _0098
     yes
```

```
?- count(a, [a, b, X, a],
N).
   X = _0090
   N = 2
   yes
```

```
count(_, [], 0).
count(Term, [Head|Tail],
N) :-
   Term == Head, !,
   count(Term, Tail, N1),
   N is N1+1
   ;
   count(Term, Tail, N).
```

Ενσωματωμένα κατηγορήματα χειρισμού προγράμματος (1/2)

- assert/1, retract/1

```
?- assert(p(1)).
   yes
?- assert(p(2)).
   yes
?- p(X).
   X = 1        -> ;
   X = 2
   yes
```

```
?- retract(p(X)).
   X = 1        -> ;
   X = 2
   yes
?- p(X).
   no
```

Ενσωματωμένα κατηγορήματα χειρισμού προγράμματος (2/2)

Τα ενσωματωμένα κατηγορήματα χειρισμού προγράμματος:

- πρέπει να χρησιμοποιούνται με φειδώ
- είναι χρήσιμα σε πολύ εξειδικευμένες περιπτώσεις
- είναι κάτι σαν το GO TO στο διαδικαστικό προγραμματισμό
- **ΒΛΑΠΤΟΥΝ ΣΟΒΑΡΑ ΤΟ ΣΩΣΤΟ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ ΣΕ PROLOG**

Συνδυασμός repeat-fail

- repeat/0

```
repeat.
repeat :- repeat.
?- repeat.
        -> ;
        -> ;
        -> ;
        -> ;
     ............
dosquares :-
    repeat, read(X),
    (X=stop, !
        ;
Y is X*X,
     write(Y), nl,
fail).

?- dosquares.
     7.
     49
     10.
     100
     3.
     9
     stop.
     yes
```

Κι άλλα κατηγορήματα συλλογής λύσεων (1/5)

```
age(peter, 7).
age(ann, 5).
age(pat, 8).
age(tom, 5).
?- findall(Child, age(Child, Age), List).
   Child = _0084
   Age = _0098
   List = [peter, ann, pat, tom]
   yes
```

Κι άλλα κατηγορήματα συλλογής λύσεων (2/5)

- bagof/3, setof/3

```
?- bagof(Child,
age(Child, 5), List).
   Child = _0084
   List = [ann, tom]
   yes
?- bagof(Child,
age(Child, Age), List).
```

```
Child = _0084
   Age = 5
   List = [ann, tom]
-> ;
   Child = _0084
   Age = 7
   List = [peter]
-> ;
   Child = _0084
   Age = 8
   List = [pat]
   yes
```

Κι άλλα κατηγορήματα συλλογής λύσεων (3/5)

```
?- bagof(Child, Age^age(Child, Age), List).
   Child = _0084
   Age = _0098
   List = [peter, ann, pat, tom]
   yes
?- bagof(Age, Child^age(Child, Age), List).
   Age = _0084
   Child = _0098
   List = [7, 5, 8, 5]
   yes
```

Κι άλλα κατηγορήματα συλλογής λύσεων (4/5)

```
?- setof(Child, Age^age(Child, Age), ChildList),
   setof(Age, Child^age(Child, Age), AgeList).
   Child = _0084
   Age = _0098
   ChildList = [ann, pat, peter, tom]
   AgeList = [5, 7, 8]
   yes
?- setof(Age/Child, age(Child, Age), List).
   Age = _0084
   Child = _0098
   List = [5/ann, 5/tom, 7/peter, 8/pat]
   yes
```

Κι άλλα κατηγορήματα συλλογής λύσεων (5/5)

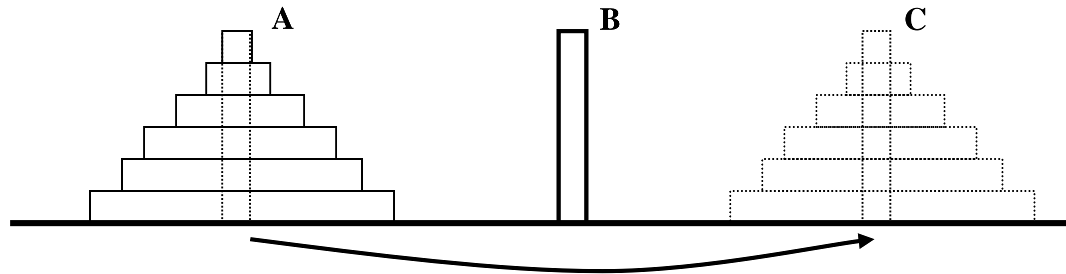
```
?- findall(X, bla(X), L).
   X = _0084
   L = []
   yes
?- bagof(X, bla(X), L).
   no
?- setof(X, bla(X), L).
   no
```

Ορισμός findall/3 μέσω assert/1 και retract/1

```
findall(X, Goal,
Xlist) :-
    call(Goal),
    assert(queue(X)),
    fail
      ;

assert(queue(bottom)),
    collect(Xlist).
```

```
collect(L) :-
    retract(queue(X)),
    !,
    (X == bottom,
     !,
     L=[]
       ;
     L=[X|Rest],
     collect(Rest)).
```

Το πρόβλημα των πύργων του Ανόι




```
hanoi(0, _, _, _).
hanoi(N, Pile1, Pile2, Pile3) :-
   N > 0,
   N1 is N-1,
   hanoi(N1, Pile1, Pile3, Pile2), write('Move a
disk from pile '),
   write(Pile1),
   write(' to pile '),
```

Το πρόβλημα των πύργων του Ανόι, συνέχεια

```
 write(Pile2),
   nl,
   hanoi(N1, Pile3, Pile2, Pile1).
?- hanoi(5, 'A', 'C', 'B').
   Move a disk from pile A to pile C
   Move a disk from pile A to pile B
   Move a disk from pile C to pile B
   Move a disk from pile A to pile C
   Move a disk from pile B to pile A
   Move a disk from pile B to pile C
   Move a disk from pile A to pile C
   .................................
```

Γενικές αρχές καλού προγραμματισμού (1/4)

- Ένα καλό πρόγραμμα (ανεξάρτητα από τη συγκεκριμένη γλώσσα υλοποίησης) πρέπει να είναι:
  – Σωστό
  – Αποδοτικό
  – Ευανάγνωστο
  – Εύκολα τροποποιήσιμο
  – Εύρωστο
  – Τεκμηριωμένο
- Τα παραπάνω ισχύουν, φυσικά, και για τα προγράμματα Prolog.

Γενικές αρχές καλού προγραμματισμού (2/4)

- Η μεθοδολογία της σταδιακής εκλέπτυνσης (stepwise refinement) είναι εν γένει χρήσιμη στον προγραμματισμό. Στην περίπτωση των προγραμμάτων Prolog, όμως, αποτελεί το βασικό εργαλείο ανάπτυξής τους.
- Η χρήση αναδρομής είναι αρκετά συνηθισμένη στην επίλυση προβλημάτων με Prolog. Με τη μέθοδο αυτή ορίζονται οι οριακές περιπτώσεις του προβλήματος καθώς επίσης και οι γενικές περιπτώσεις.

Γενικές αρχές καλού προγραμματισμού (3/4)

```
maplist([], _, []).
    maplist([X|Tail], F, [NewX|NewTail]) :-
        G =.. [F, X, NewX],
        call(G),
        maplist(Tail, F, NewTail).
```

- Μερικές φορές, αναδρομικές διαδικασίες είναι πολύ απαιτητικές σε χώρο. Ο λόγος είναι ότι τα συστήματα Prolog πρέπει να πάρουν υπόψη τους την πιθανότητα οπισθοδρόμησης και συνεπώς χρειάζονται να καταχωρήσουν κάποιες πληροφορίες ελέγχου.
- Γνώση που έχει ο προγραμματιστής για πιθανή ντετερμινιστική συμπεριφορά τμημάτων ενός προγράμματος καλό είναι να την παρέχει μέσω της χρήσης πράσινων `!` (τα κόκκινα `!` πρέπει να χρησιμοποιούνται με πολλή φειδώ).

Γενικές αρχές καλού προγραμματισμού (4/4)

- Δεν είναι σπάνιες οι περιπτώσεις που η χρήση αναδρομής μπορεί να υποκατασταθεί από ένα σχήμα `repeat-fail`. Όταν μπορεί να γίνει αυτό είναι οπωσδήποτε προτιμότερο.
- Η γενίκευση ενός συγκεκριμένου προβλήματος ίσως να είναι πιο εύκολο να επιλυθεί από το αρχικό πρόβλημα. Για παράδειγμα:

```
eightqueens(Pos)
```

ή

```
nqueens(Pos, N)

eightqueens(Pos) :- nqueens(Pos, 8).
```

Μερικοί κανόνες καλού προγραμματισμού σε Prolog

- Ορισμός μικρών προτάσεων (με λίγους στόχους).
- Μνημονικά ονόματα για κατηγορήματα και μεταβλητές.
- Διαρρύθμιση προγράμματος (στοίχιση, κενές γραμμές).
- Ομαδοποίηση προτάσεων με το ίδιο κατηγόρημα.
- Προσοχή στη χρήση του `!`.
- Συνειδητοποίηση της λειτουργίας του `not`.
- Αποφυγή χρήσης των `assert`/`retract` εκτός από πολύ εξειδικευμένες περιπτώσεις.
- Όχι κατάχρηση της διάζευξης μεταξύ στόχων (`;`).
- Ικανοποιητική τεκμηρίωση.

Τι γίνεται αν δεν “δουλεύει” το πρόγραμμά μας;

- Η καλύτερη μέθοδος διόρθωσης των λαθών είναι
  - ο σταδιακός (από επάνω-προς-τα-κάτω ή από κάτω-προς-τα-επάνω) έλεγχος των σχέσεων που έχουν ορισθεί.
  - Τα περισσότερα συστήματα Prolog παρέχουν δυνατότητες ιχνηλάτησης (tracing)

Συγχώνευση ταξινομημένων λιστών

```
?- merge([2,4,7], [1,3,4,8], X).
   X = [1,2,3,4,4,7,8]
   yes
```

Ποιο πρόγραμμα είναι το καλύτερο;

```
merge(List1, List2, List3) :-
    List1 = [], !, List3 = List2 ;
    List2 = [], !, List3 = List1 ;
    List1 = [X|Rest1],
    List2 = [Y|Rest2],
    (X < Y, !,
     Z = X,
     merge(Rest1, List2, Rest3) ;
     Z = Y,
     merge(List1, Rest2, Rest3)),
    List3 = [Z|Rest3].
-------------------------------------------------------
```

Ποιο πρόγραμμα είναι το καλύτερο; (συνέχεια)

```
---------------------------------------------------
merge([], List, List) :- !.
merge(List, [], List) :- !.
merge([X|Rest1], [Y|Rest2], [X|Rest3]) :-
   X < Y, !,
   merge(Rest1, [Y|Rest2], Rest3).
merge(List1, [Y|Rest2], [Y|Rest3]) :-
   merge(List1, Rest2, Rest3).
---------------------------------------------------
```

Το πρόβλημα του χρωματισμού χάρτη

- Δίνεται ένα σύνολο από χώρες καθώς και η πληροφορία “ποιες χώρες συνορεύουν με ποιες”. Δίνονται επίσης 4 χρώματα.
- Να χρωματισθεί κάθε χώρα με ένα από τα διαθέσιμα χρώματα έτσι ώστε γειτονικές χώρες να μην έχουν το ίδιο χρώμα.
- Αποδεικνύεται μαθηματικά ότι για οποιοδήποτε χάρτη το πρόβλημα έχει πάντοτε λύση.
- Μπορείτε να εφαρμόσετε τη μέθοδό σας σε μη τετριμμένες περιπτώσεις (π.χ. για την Ευρώπη);

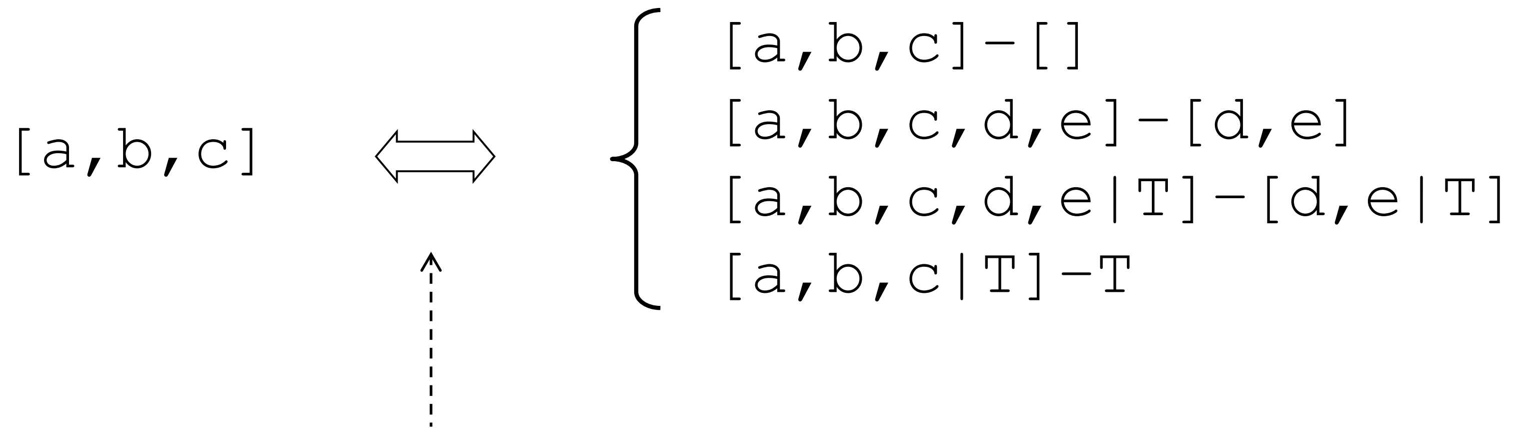
Το πρόβλημα του χρωματισμού γράφου

- Παρόμοιο πρόβλημα με το προηγούμενο (γιατί δεν είναι το ίδιο;) είναι το πρόβλημα του χρωματισμού των κορυφών ενός γράφου, έτσι ώστε γειτονικές (συνδεόμενες με κάποια ακμή) κορυφές να μην έχουν το ίδιο χρώμα.
- Να λυθεί το πρόβλημα για δεδομένο γράφο και δεδομένο αριθμό διαθέσιμων χρωμάτων. Υπάρχει πάντοτε λύση;

Εύρεση χρωματικού αριθμού γράφου

- Χρωματικός αριθμός ενός γράφου είναι ο ελάχιστος αριθμός χρωμάτων που απαιτούνται για το χρωματισμό του (σεβόμενοι τον προαναφερθέντα κανόνα).
- Να βρεθεί ο χρωματικός αριθμός δεδομένου γράφου. Για πόσο μεγάλους και πόσο πυκνούς γράφους παίρνετε απάντηση σε πεπερασμένο χρόνο;

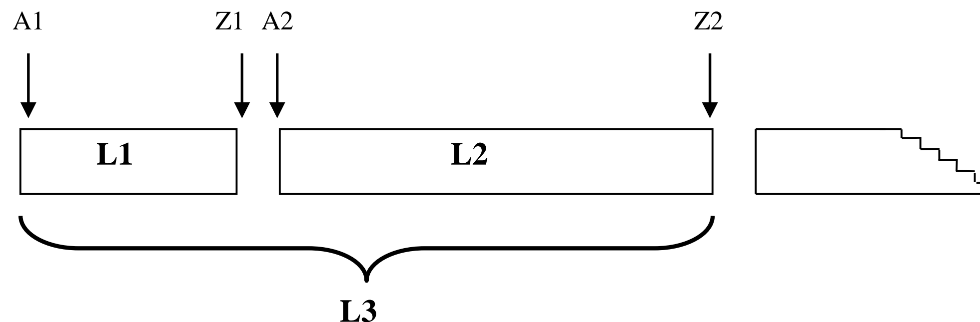
Διαφορές Λιστών



ΠΡΟΣΟΧΗ ΣΤΗ ΣΗΜΑΣΙΑ ΤΟΥ

Νέος ορισμός της συνένωσης λιστών

```
new_append(A1-Z1, Z1-Z2, A1-Z2).

?- new_append([a, b, c|T1]-T1, [d, e|T2]-T2, L).
   T1 = [d, e|_0084]
   T2 = _0084
   L = [a, b, c, d, e|_0084]-_0084
   yes
```



Ακολουθία Fibonacci

X1 = 1
X2 = 1
Xν = Xν-1 + Xν-2 , ν ≥ 3

1, 1, 2, 3, 5, 8, 13, …………

```
fib(1, 1).
fib(2, 1).
fib(N, F) :- N > 2,
             N1 is N-1,
             fib(N1, F1),
             N2 is N-2,
             fib(N2, F2),
             F is F1+F2.

?- fib(7, F).
   F = 13
   yes
```

Πόσο αποδοτικός ορισμός είναι;

Σχόλιο;

```
newfib(N, F) :- fib(2, N, 1, 1, F).
fib(M, N, F1, F2, F2) :- M >= N.
fib(M, N, F1, F2, F) :- M < N,
     NextM is M+1,
     NextF2 is F1+F2,
     fib(NextM, N, F2, NextF2, F).
```

Ερωτήσεις (1)

1. Να συγκριθούν οι αποδόσεις των τριών παρακάτω ορισμών υπολίστας `sub1`, `sub2` και `sub3`.

```
sub1(List, Sublist) :-
    prefix(List, Sublist).
sub1([_|Tail], Sublist) :-
    sub1(Tail, Sublist).

prefix(_, []).
prefix([X|List], [X|List2]) :-
    prefix(List1, List2).
```

Ερωτήσεις (2)

```
sub2(List, Sublist) :-
      append(List1, List2, List),
      append(List3, Sublist, List1).


sub3(List, Sublist) :-
      append(List1, List2, List),
      append(Sublist, List3, List2).
```

Ερωτήσεις (3)

2. Να ορισθεί η προσθήκη στοιχείου στο τέλος λίστας `add_at_end(List, Item, NewList)` χρησιμοποιώντας λίστες διαφορών.
3. Να ορισθεί η αντιστροφή λίστας `reverse(List, RevList)` χρησιμοποιώντας λίστες διαφορών

Εναλλακτικές αναπαραστάσεις λιστών (1/4)

- Για την αναπαράσταση μίας λίστας χρειάζεται ένα σύμβολο για την κενή λίστα ([]) και ένα άλλο (.) για τη δόμηση μίας κεφαλής με μία ουρά.
- Στην Prolog τα σύμβολα [] και . είναι προκαθορισμένα με την έννοια ότι οι λίστες που δομούνται με αυτά μπορούν να έχουν και μία πιο φιλική εξωτερική μορφή. Δηλαδή:

```
.(a, .(b, .(c, [])))        [a, b, c]
```

Εναλλακτικές αναπαραστάσεις λιστών (2/4)

- Οποιαδήποτε σύμβολα μπορούν να χρησιμοποιηθούν για τη δόμηση λιστών. Για παράδειγμα, αν ορισθεί το

```
:- op(500, xfy, then).
```

τότε η λίστα από τα στοιχεία `enter`, `sit` και `eat` μπορεί να παρασταθεί σαν `enter then sit then eat then donothing`, όπου το `donothing` παριστάνει την κενή λίστα. Ο ορισμός της συνένωσης μπορεί τότε να γραφτεί:

```
append1(donothing, L, L).
append1(X then L1, L2, X then L3) :-
          append1(L1, L2, L3).
```

- Με σύμβολο της κενής λίστας το `true` και με σύμβολο δόμησης λιστών το `&` (έχοντας ορίσει όμως

  `:- op(300, xfy, &)`) μπορούμε να έχουμε λίστες της μορφής `a & b & c & true`.

Εναλλακτικές αναπαραστάσεις λιστών (3/4)

1. Να ορισθεί το `list(Object)` έτσι ώστε να είναι αληθές όταν το `Object` είναι λίστα με τη γνωστή δομή που έχουν οι λίστες στην Prolog.
2. Να ορισθεί πότε ένα στοιχείο είναι μέλος μίας λίστας αν αυτή δομείται με τα σύμβολα `then` και `donothing`
3. Να ορισθεί το `convert(StandardList, List)` έτσι ώστε να μετασχηματίζει λίστες από τη γνωστή δομή της Prolog στη δομή `then/donothing` (και αντίστροφα). Δηλαδή, να είναι αληθές το `convert([a, b], a then b then donothing)`

Εναλλακτικές αναπαραστάσεις λιστών (4/4)

4. Να γενικευθεί ο προηγούμενος ορισμός του `convert` έτσι ώστε να παίρνει σαν ορίσματα τα σύμβολα δόμησης λιστών και κενής λίστας. Δηλαδή να ορισθεί το
`convert(StandardList, List, Functor, Empty)`
έτσι ώστε να χρησιμοποιείται ως εξής:

```
?- convert([a, b], L, then, donothing).
   L = a then b then donothing
   yes
?- convert([a, b, c], L, +, 0).
   L = a+(b+(c+0))
   yes
```

Ταξινόμηση λιστών

Θεωρείται ότι υπάρχει ορισμένη μία σχέση διάταξης `gt(X, Y)` μεταξύ στοιχείων. Π.χ. για αριθμούς:

```
gt(X, Y) :- X > Y.
```

Μέθοδος φυσαλίδας (bubblesort)

```
bubblesort(List, Sorted) :-
       swap(List, List1),
       !,
       bubblesort(List1, Sorted).
    bubblesort(Sorted, Sorted).

    swap([X, Y|Rest], [Y, X|Rest]) :-
       gt(X, Y).
    swap([Z|Rest], [Z|Rest1]) :-
       swap(Rest, Rest1).
```

Μέθοδος φυσαλίδας (bubblesort), συνέχεια

- Βρες δύο συνεχόμενα στοιχεία της προς ταξινόμηση λίστας τα οποία είναι σε λάθος διάταξη και αντιμετάθεσέ τα. Αυτή η διαδικασία να επαναλαμβάνεται μέχρις ότου δεν μπορεί να βρεθεί τέτοιο ζευγάρι στοιχείων.

Μέθοδος εισαγωγής (insertsort)

```
insertsort([], []).
    insertsort([X|Tail], Sorted) :-
        insertsort(Tail, SortedTail),
        insert(X, SortedTail, Sorted).

    insert(X, [Y|Sorted], [Y|Sorted1]) :-
        gt(X, Y),
        !,
        insert(X, Sorted, Sorted1).
    insert(X, Sorted, [X|Sorted]).
```

Μέθοδος εισαγωγής (insertsort), συνέχεια

- Για να ταξινομηθεί μια λίστα αρκεί να ταξινομηθεί η ουρά της και στη συνέχεια να εισαχθεί η κεφαλή στην κατάλληλη θέση της ταξινομημένης ουράς.

1. Να συγκριθούν οι αποδόσεις της bubblesort και της insertsort.
2. Για πόσο μεγάλες λίστες μπορούν να χρησιμοποιηθούν οι bubblesort και insertsort χωρίς να έχει πρόβλημα το σύστημα Prolog που χρησιμοποιείτε;

Γρήγορη μέθοδος (quicksort)

```
quicksort([], []).
   quicksort([X|Tail],
Sorted) :-
      split(X, Tail,
Small, Big),
      quicksort(Small,
SortedSmall),
      quicksort(Big,
SortedBig),

append(SortedSmall,
[X|SortedBig],
Sorted).
```

```
split(X, [], [], []).
   split(X, [Y|Tail],
[Y|Small], Big) :-
      gt(X, Y),
      !,
      split(X, Tail,
Small, Big).
   split(X, [Y|Tail],
Small, [Y|Big]) :-
      split(X, Tail,
Small, Big).

   append(............
```

Γρήγορη μέθοδος (quicksort), συνέχεια

- Για να τα ταξινομηθεί μία λίστα αρκεί να διαχωριστεί η ουρά της σε δύο λίστες, μία να περιέχει τα στοιχεία εκείνα που είναι μικρότερα από την κεφαλή και η άλλη εκείνα που είναι μεγαλύτερα ή ίσα από την κεφαλή, στη συνέχεια να ταξινομηθούν οι δύο αυτές λίστες και τέλος να συνενωθούν παρεμβάλλοντας μεταξύ τους την κεφαλή της αρχικής λίστας.

Η `quicksort` με λίστες διαφορών

[5, 3, 7, 8, 1, 4, 7, 6]

οδηγός: 5

[3, 7, 8, 1, 4, 7, 6]

< 5

>= 5

[3, 1, 4]

[7, 8, 7, 6]

ταξινόμηση

ταξινόμηση

[1, 3, 4]

[6, 7, 7, 8]

οδηγός

[1, 3, 4, 5, 6, 7, 7, 8]

Η `quicksort` με λίστες διαφορών, συνέχεια

```
quicksort(List, Sorted) :-
      quicksort2(List, Sorted-[]).


   quicksort2([], Z-Z).
   quicksort2([X|Tail], A1-Z2) :-
      split(X, Tail, Small, Big),
      quicksort2(Small, A1-[X|A2]),
      quicksort2(Big, A2-Z2).
```

1. Ποια είναι η σχετική απόδοση της `quicksort`, τόσο από πλευράς χώρου όσο και από πλευράς χρόνου, αν συγκριθεί με τις `bubblesort` και `insertsort`;

Μία άλλη μέθοδος ταξινόμησης είναι η `mergesort`

- Σύμφωνα με αυτήν:

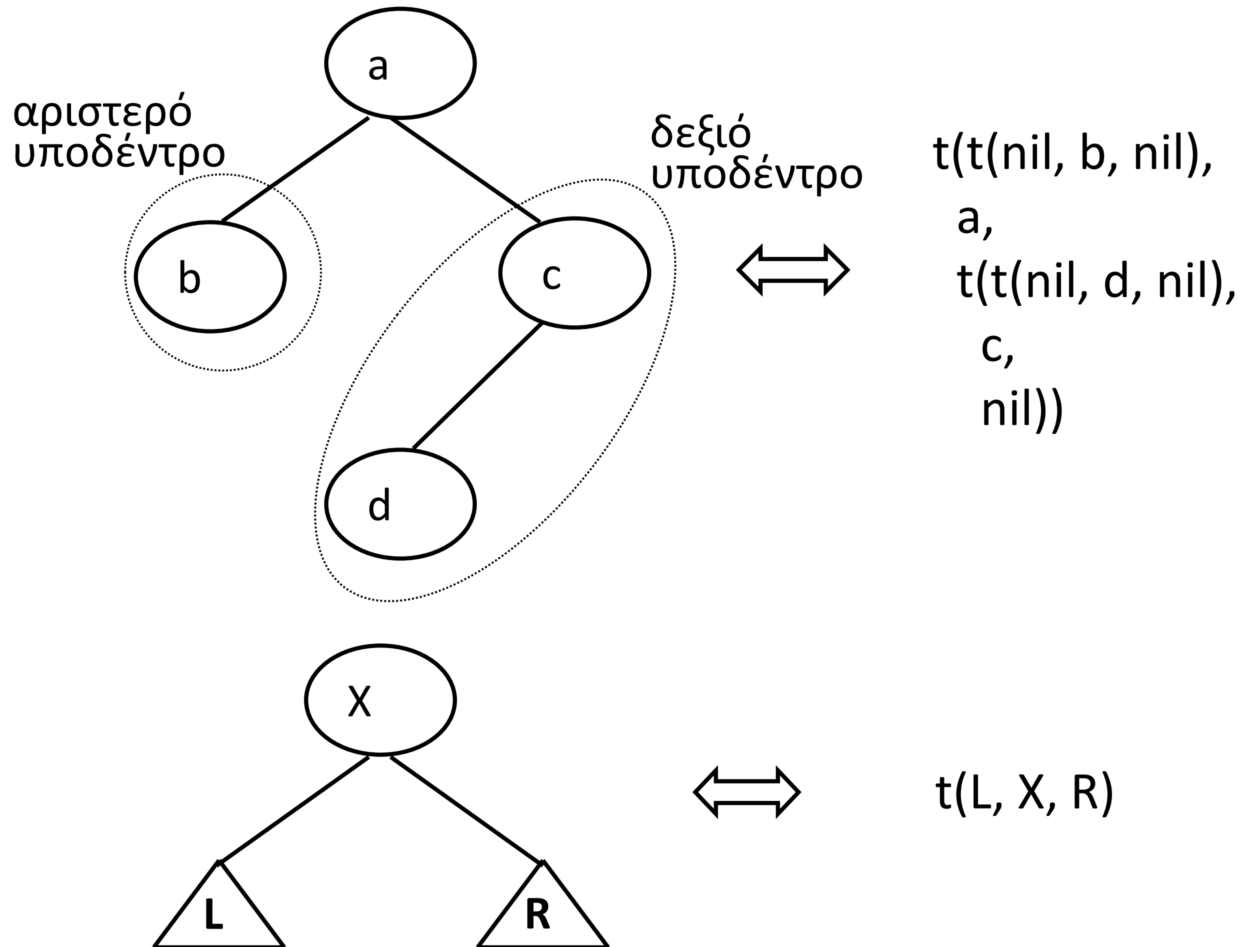
  Για να ταξινομηθεί μία λίστα αρκεί να διαιρεθεί σε δύο περίπου ισομήκεις λίστες (διαφορά μηκών 0 ή 1), αυτές οι λίστες μετά να ταξινομηθούν και στο τέλος να συγχωνευθούν.

- Να υλοποιηθεί σε Prolog αυτή η μέθοδος και να συγκριθεί η απόδοσή της με αυτή της `quicksort`.

Δυαδικά δέντρα (Binary trees)

- Μια άλλη οργάνωση για σύνολα στοιχείων αντί για τις λίστες.
- Ένα δυαδικό δέντρο είτε είναι κενό είτε αποτελείται από τη ρίζα του, ένα αριστερό υποδέντρο και ένα δεξιό υποδέντρο.
- Για την αναπαράσταση δυαδικών δέντρων στηνProlog απαιτούνται ένα σύμβολο για το κενό δέντρο (έστω `nil`) και ένα σύμβολο δόμησης της ρίζας με τα δύο υποδέντρα (έστω `t`)

Δυαδικά δέντρα (Binary trees), συνέχεια



Στοιχείο μέλος ενός δυαδικού δέντρου (1/5)

```
in(X, t(_, X, _)).
in(X, t(L, _, _)) :-
    in(X, L).
in(X, t(_, _, R)) :-
    in(X, R).


?- in(X, nil).
    no
?- T = t(t(nil, b, nil), a, t(t(nil, d, nil), c, 
nil)),
    in(c, T).
    T = .........
    yes
```

Στοιχείο μέλος ενός δυαδικού δέντρου (2/5)

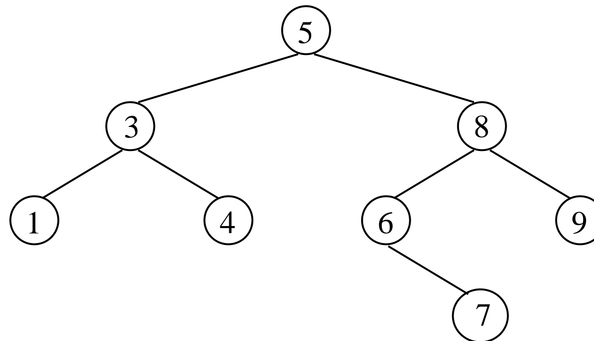
```
?- T = t(t(nil, b, nil), a, t(t(nil, d, nil), c,
nil)),
   in(e, T).
   no
?- T = t(t(nil, b, nil), a, t(t(nil, d, nil), c,
nil)),
   in(X, T).
   X = a
   T = .........        -> ;
   X = b
   T = .........        -> ;
   X = c
```

Στοιχείο μέλος ενός δυαδικού δέντρου (3/5)

```
T = .........          -> ;
  X = d
  T = .........          -> ;
  yes
```

Στοιχείο μέλος ενός δυαδικού δέντρου (4/5)

- Ένα μη κενό δυαδικό δέντρο `t(Left, X, Right)` λέγεται ότι είναι ταξινομημένο από τα αριστερά προς τα δεξιά αν όλα τα στοιχεία του υποδέντρου `Left` είναι μικρότερα από το `X`, όλα τα στοιχεία του υποδέντρου `Right` είναι μεγαλύτερα από το `X` και τα υποδέντρα `Left` και `Right` είναι επίσης ταξινομημένα από τα αριστερά προς τα δεξιά.(Θεωρείται ότι δεν υπάρχει κάποιο στοιχείο στο αρχικό δέντρο δύο φορές). Ένα τέτοιο δυαδικό δέντρο λέγεται δυαδικό λεξικό (binary dictionary).
- Π.χ.



Στοιχείο μέλος ενός δυαδικού δέντρου (5/5)

- Ένα δυαδικό λεξικό είναι πιο αποδοτική δομή από ένα απλό δυαδικό δέντρο, αφού για να βρεθεί αν κάποιο στοιχείο βρίσκεται στο δέντρο δεν χρειάζεται να διασχισθεί ολόκληρο (στη χειρότερη περίπτωση).
- Αυτό είναι τόσο πιο πολύ σωστό όσο πιο ισοζυγισμένο (balanced) είναι το δέντρο.
- Εν γένει, η αναζήτηση είναι πιο μεγάλη σε δυαδικά λεξικά μεγαλύτερου ύψους (height).

Στοιχείο μέλος ενός δυαδικού λεξικού (1/4)

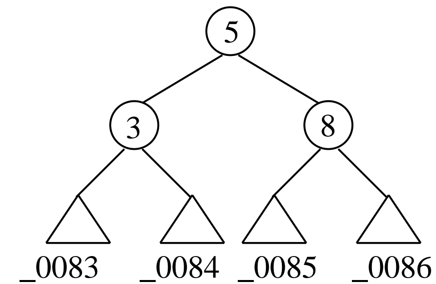
```
in(X, t(_, X, _)).
in(X, t(Left, Root, Right)) :-
   gt(Root, X),
   in(X, Left).
in(X, t(Left, Root, Right)) :-
   gt(X, Root),
   in(X, Right).
```

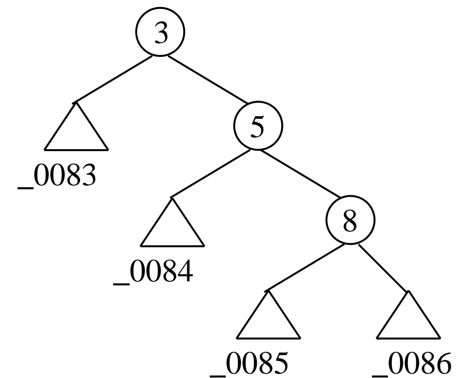
Στοιχείο μέλος ενός δυαδικού λεξικού (2/4)

```
?- in(6, t(t(t(nil, 1, nil), 3, t(nil, 4, nil)),
            5,
            t(t(nil, 6, t(nil, 7, nil)),
              8,
              t(nil, 9, nil)))).
   yes
?- in(2, t(t(t(nil, 1, nil), 3, t(nil, 4, nil)),
            5,
            t(t(nil, 6, t(nil, 7, nil)),
              8,
              t(nil, 9, nil)))).
   no
```

Στοιχείο μέλος ενός δυαδικού λεξικού (3/4)

```
?- in(5, D), in(3, D), in(8, D).
   D = t(t(_0083, 3, _0084),
         5,
         t(_0085, 8, _0086))
   yes
?- in(3, D), in(5, D), in(8, D).
   D = t(_0083,
         3,
         t(_0084,
           5,
           t(_0085,
             8,
             _0086)))
   yes
```





Στοιχείο μέλος ενός δυαδικού λεξικού (4/4)

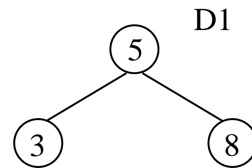
1. Να ορισθούν τα `binarytree(Object)` και `dictionary(Object)` έτσι ώστε να είναι αληθή αν το `Object` είναι δυαδικό δέντρο ή δυαδικό λεξικό αντίστοιχα.
2. Να ορισθεί το `height(BinaryTree, Height)` έτσι ώστε να υπολογίζει το ύψος ενός δυαδικού δέντρου. (Το ύψος του κενού δέντρου είναι 0).
3. Να ορισθεί το `linearize(Tree, List)` έτσι ώστενα συλλέγει όλα τα στοιχεία του δυαδικού δέντρου `Tree` στη λίστα `List`.
4. Να οριστεί το `maxelement(D, Item)` έτσι ώστε το `Item` να είναι το μεγαλύτερο στοιχείο του δυαδικού λεξικού `D`.
5. Να επεκταθεί ο ορισμός του `in/2` για την αναζήτηση στοιχείων μέσα σε δυαδικά λεξικά έτσι ώστε να επιστρέφει και την ακολουθία κόμβων από τη ρίζα του λεξικού μέχρι το στοιχείο που βρέθηκε

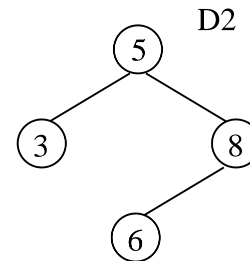
Εισαγωγή και διαγραφή κόμβων σε δυαδικό λεξικό (1/5)

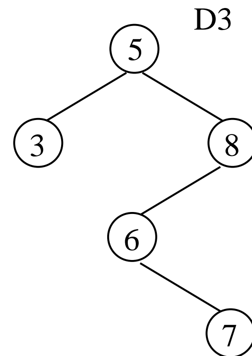
- Εισαγωγή κόμβου σε φύλλο

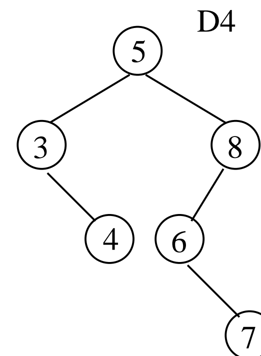
```
addleaf(nil, X, t(nil, X, nil)).
addleaf(t(Left, X, Right), X, t(Left, X, Right)).
addleaf(t(Left, Root, Right), X, t(Left1, Root,
Right)) :-
    gt(Root, X),
    addleaf(Left, X, Left1).
addleaf(t(Left, Root, Right), X, t(Left, Root,
Right1)) :-
    gt(X, Root),
    addleaf(Right, X, Right1).
```

Εισαγωγή και διαγραφή κόμβων σε δυαδικό λεξικό (2/5)










```
?- addleaf(t(t(nil, 3, nil), 5, t(nil, 8, nil)), 6,
   D2),
   addleaf(D2, 7, D3), addleaf(D3, 4, D4).
```

Εισαγωγή και διαγραφή κόμβων σε δυαδικό λεξικό (3/5)

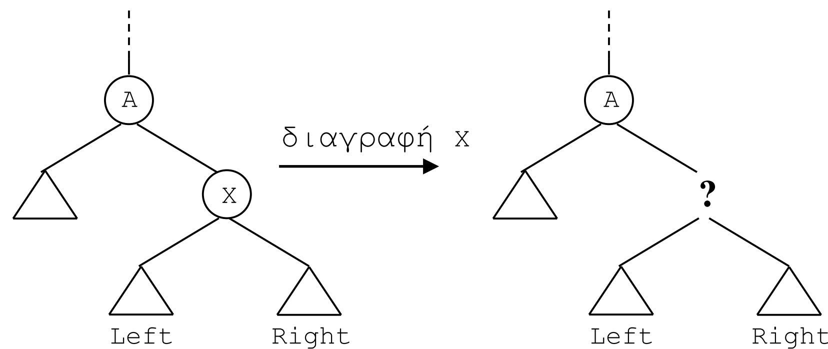
- Διαγραφή κόμβου από φύλλο

```
delleaf(D1, X, D2) :-
    addleaf(D2, X, D1).
```

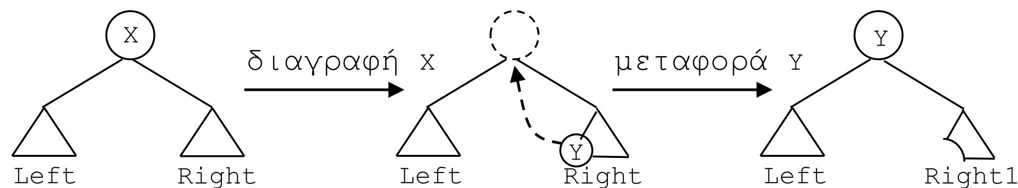
- Δεν είναι χρήσιμη γιατί δεν είναι απαραίτητο ο κόμβος `X` που θέλουμε να διαγράψουμε να είναι φύλλο του `D1`.

Εισαγωγή και διαγραφή κόμβων σε δυαδικό λεξικό (4/5)

- Διαγραφή κόμβου από οπουδήποτε



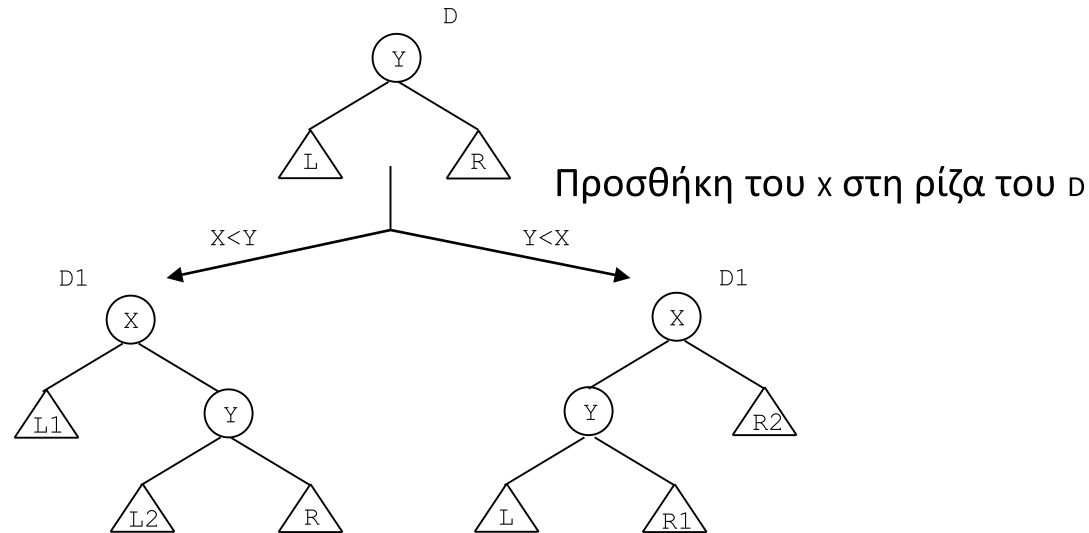
Αν κάποιο από τα `Left` ή `Right` είναι `nil`, τότε το άλλο απλώς μπορεί να συνδεθεί στο `A`. Αν όχι:



Εισαγωγή και διαγραφή κόμβων σε δυαδικό λεξικό (5/5)

```
del(t(nil, X, Right), X, Right).
del(t(Left, X, nil), X, Left).
del(t(Left, X, Right), X, t(Left, Y, Right1)) :-
   delmin(Right, Y, Right1).
del(t(Left, Root, Right), X, t(Left1, Root, Right)) :-
   gt(Root, X),
   del(Left, X, Left1).
del(t(Left, Root, Right), X, t(Left, Root, Right1)) :-
   gt(X, Root),
   del(Right, X, Right1).

delmin(t(nil, Y, Right), Y, Right).
delmin(t(Left, Root, Right), Y, t(Left1, Root, Right)) :-
   delmin(Left, Y, Left1).
```

Εισαγωγή κόμβου στη ρίζα



- L = L1   L2
- Όλα τα στοιχεία του L1 είναι μικρότερα του X που είναι μικρότερο από όλα τα στοιχεία του L2

- R = R1   R2
- Όλα τα στοιχεία του R1 είναι μικρότερα του X που είναι μικρότερο από όλα τα στοιχεία του R2

Εισαγωγή κόμβου στη ρίζα, συνέχεια

```
addroot(nil, X, t(nil, X, nil)).
   addroot(t(L, Y, R), X, t(L1, X, t(L2, Y, R))) :-
      gt(Y, X),
      addroot(L, X, t(L1, X, L2)).
   addroot(t(L, Y, R), X, t(t(L, Y, R1), X, R2)) :-
      gt(X, Y),
      addroot(R, X, t(R1, X, R2)).
```

Εισαγωγή κόμβου οπουδήποτε (μη ντετερμινιστικά)

Για να εισαχθεί ένας κόμβος X σε ένα δυαδικό λεξικό D αρκεί είτε να εισαχθεί στη ρίζα του D, είτε στο αριστερό υποδέντρο του D (αν η ρίζα του D είναι μεγαλύτερη από το X), είτε στο δεξιό υποδέντρο του D (αν η ρίζα του D είναι μικρότερη από το X).

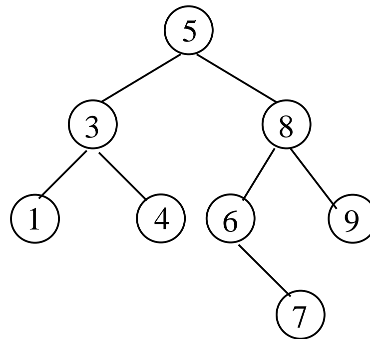
```
add(Tree, X, NewTree) :-
    addroot(Tree, X, NewTree).
add(t(L, Y, R), X, t(L1, Y, R)) :-
    gt(Y, X),
    add(L, X, L1).
add(t(L, Y, R), X, t(L, Y, R1)) :-
    gt(X, Y),
    add(R, X, R1).
```

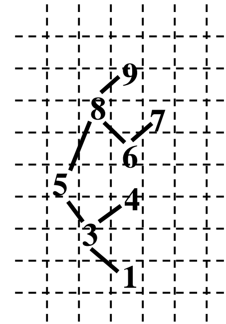
Εισαγωγή κόμβου οπουδήποτε (μη ντετερμινιστικά), συνέχεια

Η διαδικασία `add` μπορεί να χρησιμοποιηθεί και αντίστροφα, δηλαδή για τη διαγραφή στοιχείου από οπουδήποτε.

```
?- add(nil, 3, D1), add(D1, 5, D2),
   add(D2, 1, D3), add(D3, 6, D), add(DD, 5, D).
   DD =
```

Σχηματική εκτύπωση δυαδικού δέντρου





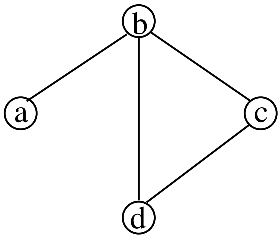

```
show(Tree) :-
     show2(Tree, 0).

  show2(nil, _).
  show2(t(Left, X, Right),
 Indent) :-
     Ind2 is Indent+2,
show2(Right, Ind2),
      spaces(Indent),
      write(X),
      nl,
      show2(Left, Ind2).

   spaces(........
```

Σχηματική εκτύπωση δυαδικού δέντρου, συνέχεια

- Να γραφεί σε Prolog διαδικασία σχηματικής εκτύπωσης δυαδικών δέντρων, αλλά με τον κλασικό προσανατολισμό (η ρίζα στην κορυφή, το αριστερό υποδέντρο αριστερά και το δεξιό υποδέντρο δεξιά).

Γράφοι (Graphs)

Αναπαραστάσεις γράφων



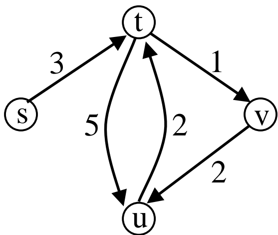

```
connected(a, b).

connected(b, c).

..........
```

Γεγονότα

```
G1 = graph([a,b,c,d],
           [e(a,b), e(b,d),
            e(b,c), e(c,d)]).

G1 = [a->[b], b->[a,c,d],

      c->[b,d], d->[b,c]]
```

Δομές Δεδομένων

Αναπαραστάσεις κατευθυνόμενων γράφων (με κόστος ακμών)



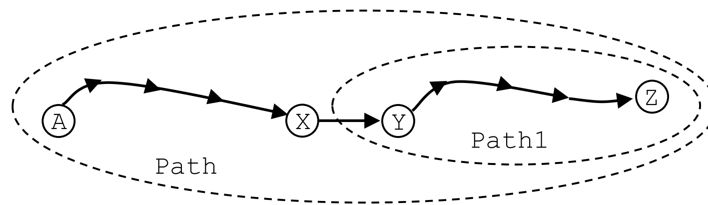

```
arc(s, t, 3).
arc(t, v, 1).
arc(u, t, 2).
.........
```

Γεγονότα

```
G2 = digraph([s,t,u,v],
        [a(s,t,3), a(t,v,1),
         a(t,u,5), a(u,t,2),
         a(v,u,2)])
G2 = [s->[t/3], t->[u/5,v/1],
      u->[t/2], v->[u/2]]
```

Δομές Δεδομένων

Εύρεση μονοπατιού σε γράφο




```
                                Graph = graph(Nodes, Edges)
path(A, Z, Graph, Path) :-
   path1(A, [Z], Graph, Path).
path1(A, [A|Path1], Graph, [A|Path1]) :-
   node(A, Graph).
path1(A, [Y|Path1], Graph, Path) :-
   adjacent(X, Y, Graph),
   not member(X, Path1),
   path1(A, [X, Y|Path1], Graph, Path).
adjacent(X, Y, graph(Nodes, Edges)) :-
   member(e(X, Y), Edges) ; member(e(Y, X), Edges).
```

Εύρεση μονοπατιού Hamilton

- (μονοπάτι χωρίς κύκλους
  που περιλαμβάνει όλους τους κόμβους του γράφου)

```
hamiltonian(Graph, Path) :-
   path(_, _, Graph, Path),
   covers(Path, Graph).
covers(Path, Graph) :-
   not (node(N, Graph), not member(N,
  Path)).
node(N, graph(Nodes, _ )) :-
   member(N, Nodes).
```

Εύρεση μονοπατιού με κόστος (1/3)

```
path(A, Z, Graph, Path, Cost) :-
   path1(A, [Z], 0, Graph, Path, Cost).
path1(A, [A|Path1], Cost1, Graph, [A|Path1], Cost1) :-
   node(A, Graph).
path1(A, [Y|Path1], Cost1, Graph, Path, Cost) :-
   adjacent(X, Y, CostXY, Graph),
   not member(X, Path1),
   Cost2 is Cost1+CostXY,
   path1(A, [X, Y|Path1], Cost2, Graph, Path, Cost).

adjacent(X, Y, CostXY, graph(Nodes, Edges)) :-
   member(e(X, Y, CostXY), Edges) ;
   member(e(Y, X, CostXY), Edges).
```

Εύρεση μονοπατιού με κόστος (2/3)

- Εύρεση μονοπατιού ελαχίστου κόστους από `node1` σε `node2`

```
?- path(node1, node2, Graph, MinPath, MinCost),
   not (path(node1, node2, Graph, _, Cost),
        Cost < MinCost).
```

Εύρεση μονοπατιού με κόστος (3/3)

- Εύρεση μονοπατιού μεγίστου κόστους μεταξύ τυχαίων κόμβων

```
?- path(_, _, Graph, MaxPath, MaxCost),
   not (path(_, _, Graph, _, Cost), Cost > MaxCost).
```

Δέντρο ανάπτυξης (spanning tree) γράφου

- Ένας γράφος λέγεται συνδεδεμένος (connected) αν για κάθε ζευγάρι κόμβων του υπάρχει μονοπάτι που τους συνδέει.
- Ένα δέντρο ανάπτυξης ενός συνδεδεμένου γράφου είναι ένας γράφος με τις ίδιες κορυφές που:
  – είναι συνδεδεμένος
  – οι ακμές του είναι και ακμές του αρχικού
  – δεν περιέχει κύκλους

Δέντρο ανάπτυξης (spanning tree) γράφου, συνέχεια

- `Graph = [a-b, b-c, c-d, b-d]`

  `SpanningTree = [a-b, b-c, c-d]`

  ή

  `[a-b, b-d, d-c]`

  ή

  `[a-b, b-d, b-c]`

- Σ’ ένα δέντρο ανάπτυξης, οποιοσδήποτε κόμβος μπορεί να είναι η ρίζα του

Εύρεση δέντρου ανάπτυξης ενός γράφου (1ος τρόπος)

```
stree(Graph, Tree) :-
    member(Edge, Graph),
    spread([Edge], Tree, Graph).

spread(Tree1, Tree, Graph) :-
    addedge(Tree1, Tree2, Graph),
    spread(Tree2, Tree, Graph).
spread(Tree, Tree, Graph) :-
    not addedge(Tree, _, Graph).
```

Εύρεση δέντρου ανάπτυξης ενός γράφου (1ος τρόπος), συνέχεια

```
addedge(Tree, [A-B|Tree], Graph) :-
    adjacent(A, B, Graph),
    node(A, Tree),
    not node(B, Tree).

adjacent(Node1, Node2, Graph) :-
    member(Node1-Node2, Graph) ;
    member(Node2-Node1, Graph).

node(Node, Graph) :-
    adjacent(Node, _, Graph).
```

Εύρεση δέντρου ανάπτυξης ενός γράφου (2ος τρόπος) (1/3)

```
stree(Graph, Tree) :-
   subset(Graph, Tree),
   tree(Tree),
   covers(Tree, Graph).
tree(Tree) :-
   connected(Tree),
   not hasacycle(Tree).
connected(Graph) :-
   not (node(A, Graph), node(B, Graph),
        not path(A, B, Graph, _)).
```

Εύρεση δέντρου ανάπτυξης ενός γράφου (2ος τρόπος) (2/3)

```
hasacycle(Graph) :-
    adjacent(Node1, Node2, Graph),
    path(Node1, Node2, Graph, [Node1, X, Y|_]).
covers(Tree, Graph) :-
    not (node(Node, Graph), not node(Node,
   Tree)).
subset(...........
path(.............
node(.............
adjacent(.........
```

Εύρεση δέντρου ανάπτυξης ενός γράφου (2ος τρόπος) (3/3)

- Αν το κόστος ενός δέντρου ανάπτυξης είναι το αθροισμάτων κοστών των ακμών του, να βρεθεί για δεδομένο γράφο με κόστη στις ακμές το δέντρο ανάπτυξης με το ελάχιστο κόστος.

Το πρόβλημα της ζέβρας (1/3)

- Πέντε άνθρωποι διαφορετικών εθνικοτήτων μένουν σε πέντε συνεχόμενα σπίτια ενός δρόμου.
- Ο καθένας τους έχει διαφορετικό επάγγελμα, του αρέσει διαφορετικό ποτό και έχει διαφορετικό κατοικίδιο ζώο.
- Τα πέντε σπίτια έχουν διαφορετικά χρώματα.
- Επιπλέον ισχύουν:

Το πρόβλημα της ζέβρας (2/3)

1. Ο Άγγλος μένει στο κόκκινο σπίτι.
2. Ο Ισπανός έχει ένα σκύλο.
3. Ο Γιαπωνέζος είναι ζωγράφος.
4. Ο Ιταλός πίνει τσάι.
5. Ο Νορβηγός μένει στο πρώτο σπίτι.
6. Αυτός που μένει στο πράσινο σπίτι πίνει καφέ.
7. Το πράσινο σπίτι είναι δεξιά από το άσπρο.

Το πρόβλημα της ζέβρας (3/3)

8. Ο γλύπτης έχει σαλιγκάρια.
9. Ο διπλωμάτης ζει στο κίτρινο σπίτι.
10. Αυτός που μένει στο μεσαίο σπίτι πίνει γάλα.
11. Ο Νορβηγός μένει δίπλα στο μπλε σπίτι.
12. Ο βιολιστής πίνει φρουτοχυμό.
13. Η αλεπού βρίσκεται δίπλα από το γιατρό.
14. Το άλογο βρίσκεται δίπλα από το διπλωμάτη.

- Ποιος έχει τη ζέβρα;
- Ποιος πίνει νερό;

1ος τρόπος (γέννησε και δοκίμασε) (1/8)

```
go(ZebraOwner, WaterDrinker) :-
    generate(People),
    test(People),
    member((ZebraOwner, _, zebra, _, _, _), People),
    member((WaterDrinker, _, _, water, _, _), People).
nationalities([english, spanish, japanese, italian,
   norwegian]).
professions([painter, sculptor, diplomat, violinist,
   doctor]).
animals([fox, dog, snails, horse, zebra]).
drinks([water, tea, coffee, milk, fruit_juice]).
house_colours([red, green, white, yellow, blue]).
house_ids([1, 2, 3, 4, 5]).
```

1^{ος} τρόπος (γέννησε και δοκίμασε) (2/8)

```
generate(People) :-
    nationalities(NList),
    professions(PList),
    animals(AList),
    drinks(DList),
    house_colours(CList),
    house_ids(HList),
    generate_people(People, NList, PList,
  AList, DList,
                          CList, HList).
```

1ος τρόπος (γέννησε και δοκίμασε) (3/8)

```
generate_people([(N, P, A, D, C, H)|More], NList,
  PList, AList, DList, CList, HList):-
   delete(N, NList, NRem),
   delete(P, PList, PRem),
   delete(A, AList, ARem),
   delete(D, DList, DRem),
   delete(C, CList, CRem),
   delete(H, HList, HRem),
   generate_people(More, NRem, PRem, ARem, DRem,
  CRem, HRem).
```

1ος τρόπος (γέννησε και δοκίμασε) (4/8)

```
generate_people([], [], [], [], [], [], []).

delete(E, [E|Rem], Rem).
delete(E, [H|T], [H|Rest]) :-
    delete(E, T, Rest).
test(People) :-
    rule1(People), rule2(People),
    rule3(People), rule4(People),
    rule5(People), rule6(People),
    rule7(People), rule8(People),
    rule9(People), rule10(People),
    rule11(People), rule12(People),
    rule13(People), rule14(People).
```

1ος τρόπος (γέννησε και δοκίμασε) (5/8)

```
rule1(People) :-
    member((english, _, _, _, red, _), People).
rule2(People) :-
    member((spanish, _, dog, _, _, _), People).
rule3(People) :-
    member((japanese, painter, _, _, _, _), People).
rule4(People) :-
    member((italian, _, _, tea, _, _), People).
rule5(People) :-
    member((norwegian, _, _, _, _, 1), People).
rule6(People) :-
    member((_, _, _, coffee, green, _), People).
```

1ος τρόπος (γέννησε και δοκίμασε) (6/8)

```
rule7(People) :-
   member((_, _, _, _, green, IDG), People),
   member((_, _, _, _, white, IDW), People),
   rightof(IDG, IDW).
rule8(People) :-
   member((_, sculptor, snails, _, _, _),
  People).
rule9(People) :-
   member((_, diplomat, _, _, yellow, _),
  People).
rule10(People) :-
   member((_, _, _, milk, _, 3), People).
```

1ος τρόπος (γέννησε και δοκίμασε) (7/8)

```
rule11(People) :-
   member((norwegian, _, _, _, _, IDN), People),
   member((_, _, _, _, blue, IDB), People),
   nextto(IDN, IDB).
rule12(People) :-
   member((_, violinist, _, fruit_juice, _, _), People).
rule13(People) :-
   member((_, doctor, _, _, _, IDD), People),
   member((_, _, fox, _, _, IDF), People),
   nextto(IDD, IDF).
rule14(People) :-
   member((_, diplomat, _, _, _, IDD), People),
   member((_, _, horse, _, _, IDH), People),
   nextto(IDD, IDH).
```

1ος τρόπος (γέννησε και δοκίμασε) (8/8)

```
nextto(X, Y) :-
    rightof(X, Y) ;
    rightof(Y, X).

rightof(2, 1).
rightof(3, 2).
rightof(4, 3).
rightof(5, 4).
member(............

     ?- go(Z, W).

         ............
```

ακόμα περιμένω

2ος τρόπος (δοκίμασε και γέννησε) (1/5)

```
go(ZebraOwner, WaterDrinker) :-
    nationalities(People),
    test(People),
    generate(People),
    member((ZebraOwner, _, zebra, _, _, _), People),
    member((WaterDrinker, _, _, water, _, _),
   People).
nationalities([(english, _, _, _, _, _),
               (spanish, _, _, _ , _ , _),
               (japanese, _, _, _, _, _),
               (italian, _, _, _, _, _),
               (norwegian, _, _, _, _, _)]).
```

2ος τρόπος (δοκίμασε και γέννησε) (2/5)

```
professions(..........
animals(..............
drinks(...............
house_colours(........
house_ids(............
generate(People) :-
   professions(PList),
   animals(AList),
   drinks(DList),
   house_colours(CList),
   house_ids(HList),
   generate_people(People, PList, AList, DList,
  CList, HList).
```

$2^{ος}$ τρόπος (δοκίμασε και γέννησε) (3/5)

```
generate_people([_, P, A, D, C, H)|More],
  PList,
                ADist, DList, CList, HList) :-
   delete(P, PList, PRem),
   delete(A, AList, ARem),
   delete(D, DList, DRem),
   delete(C, CList, CRem),
   delete(H, HList, HRem),
   generate_people(More, PRem, ARem, DRem,
  CRem, HRem).
generate_people([], [], [], [], [], []).
delete(..........
```

2ος τρόπος (δοκίμασε και γέννησε) (4/5)

```
rule1(..........        rule2(..........
rule3(..........        rule4(..........
rule5(..........        rule6(..........
rule7(..........        rule8(..........
rule9(..........        rule10(.........
rule11(.........        rule12(.........
rule13(.........        rule14(.........
```

2ος τρόπος (δοκίμασε και γέννησε) (5/5)

```
nextto(..........
rightof(.........
member(..........
?- go(Z, W).
   Z = japanese
   W = norwegian
   yes
```

Το πρόβλημα των ιεραποστόλων και κανιβάλων (1/8)

- Τρεις ιεραπόστολοι και τρεις κανίβαλοι πρέπει να διασχίσουν ένα ποτάμι. Υπάρχει μία βάρκα η οποία χωράει δύο το πολύ άτομα.Δηλαδή, οποιοσδήποτε συνδυασμός ενός ή δύο ιεραποστόλων ή κανιβάλων μπορεί να ταξιδέψει με τη βάρκα από τη μία όχθη στην άλλη.
- Το πρόβλημα είναι να μεταφερθούν όλοι οι ιεραπόστολοι και οι κανίβαλοι από την αριστερή όχθη στη δεξιά έτσι ώστε σε καμία περίπτωση και σε καμία όχθη να υπάρχουν ιεραπόστολοι που να είναι λιγότεροι σε αριθμό από τους κανίβαλους (γιατί τότε οι δεύτεροι θα υπερισχύσουν και ... καλή τους όρεξη).
- Να επινοηθεί ένα σχέδιο (plan) για να επιτευχθεί η ζητούμενη μεταφορά με ασφάλεια

Το πρόβλημα των ιεραποστόλων και κανιβάλων (2/8)

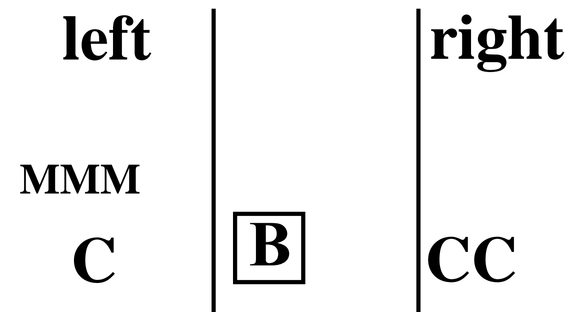
- Πρόβλημα αναζήτησης σε χώρο καταστάσεων
- Αναπαράσταση μιας κατάστασης
- Περιγραφή δυνατών μεταπτώσεων
- Αποφυγή κύκλων
- Μετάβαση από μία αρχική κατάσταση σε μία τελική

Το πρόβλημα των ιεραποστόλων και κανιβάλων (3/8)

`state(MB, CB, B)` **π.χ.**

`state(3, 1, left)`

- MB: αριθμός ιεραποστόλων στην όχθη που βρίσκεται η βάρκα
- CB: αριθμός κανιβάλων στην όχθη που βρίσκεται η βάρκα
- B: η όχθη που βρίσκεται η βάρκα



Το πρόβλημα των ιεραποστόλων και κανιβάλων (4/8)

```
plan(Plan) :-
   move(state(3, 3, left), state(3, 3, right),
        [state(3, 3, left)], Plan).

move(State, State, Plan, Plan).
move(State1, State3, Plan1, Plan3) :-
   State1 \= State3,
   legal_move(State1, State2),
   not member(State2, Plan1),          /* define it */
   append(Plan1, [State2], Plan2),     /* define it */
   move(State2, State3, Plan2, Plan3).
```

Το πρόβλημα των ιεραποστόλων και κανιβάλων (5/8)

```
legal_move(state(MB1, CB1, B1), state(MB2, CB2,
  B2)) :-
   opposite(B1, B2),
   travel(MT, CT),
   MT =< MB1,
   CT =< CB1,
   MB2 is 3-MB1+MT,
   CB2 is 3-CB1+CT,
   ((MB2 =\= 0, MB2 >= CB2) ; MB2 =:= 0),
   ((MB2 =\= 3, MB2 =< CB2) ; MB2 =:= 3) .
```

Το πρόβλημα των ιεραποστόλων και κανιβάλων (6/8)

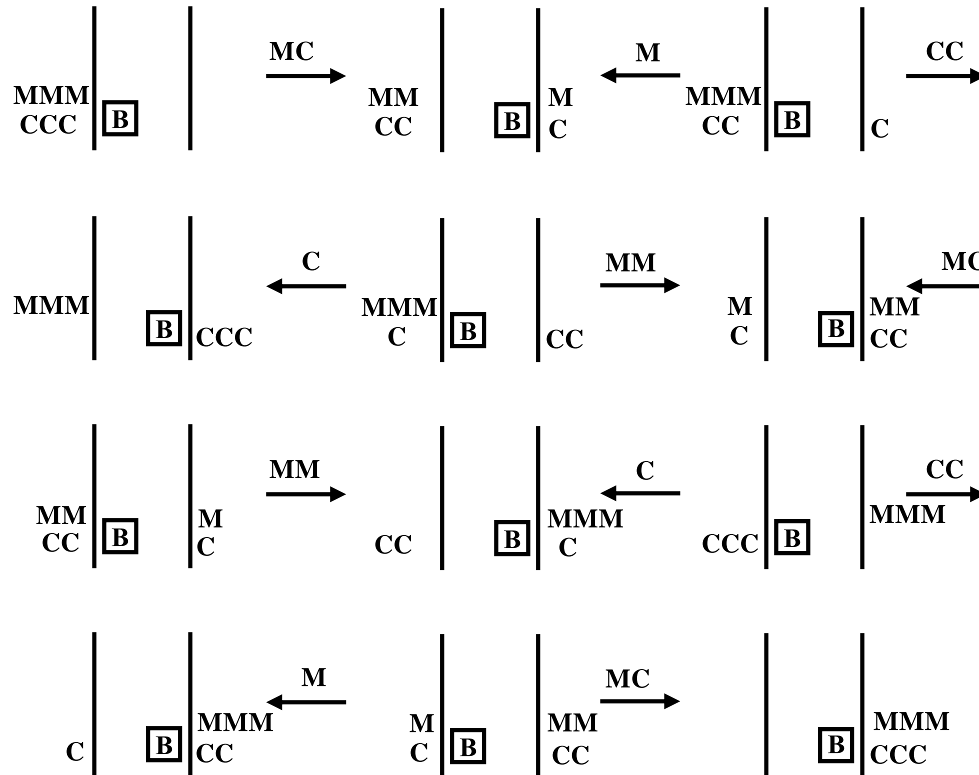
```
opposite(left, right).
opposite(right, left).

travel(1, 0).
travel(2, 0).
travel(0, 2).
travel(0, 1).
travel(1, 1).
```

Το πρόβλημα των ιεραποστόλων και κανιβάλων (7/8)

```
?- plan(Plan).
   Plan = [state(3, 3, left), state(1, 1, right), state(3, 2, left),
           state(0, 3, right), state(3, 1, left), state(2, 2, right),
           state(2, 2, left), state(3, 1, right), state(0, 3, left),
           state(3, 2, right), state(1, 1, left), state(3, 3, right)]

   -> ;
   ...........................................................
```

Υπάρχουν και άλλες 3 λύσεις

Το πρόβλημα των ιεραποστόλων και κανιβάλων (8/8)



Ένας μετα-διερμηνέας για την Prolog

```
cl(member(X, [X|_]), []).
cl(member(X, [_|L]), [member(X, L)]).

execute([]).
execute([Goal|Goals]) :-
   cl(Goal, Body),
   append(Body, Goals, NewGoals),
   execute(NewGoals).

?- execute([member(X, [1, 2, 3]),
            member(X, [2, 3, 4])]).
   X = 2     -> ;
   X = 3
   yes
```

Συμβολική παραγώγιση (1/2)

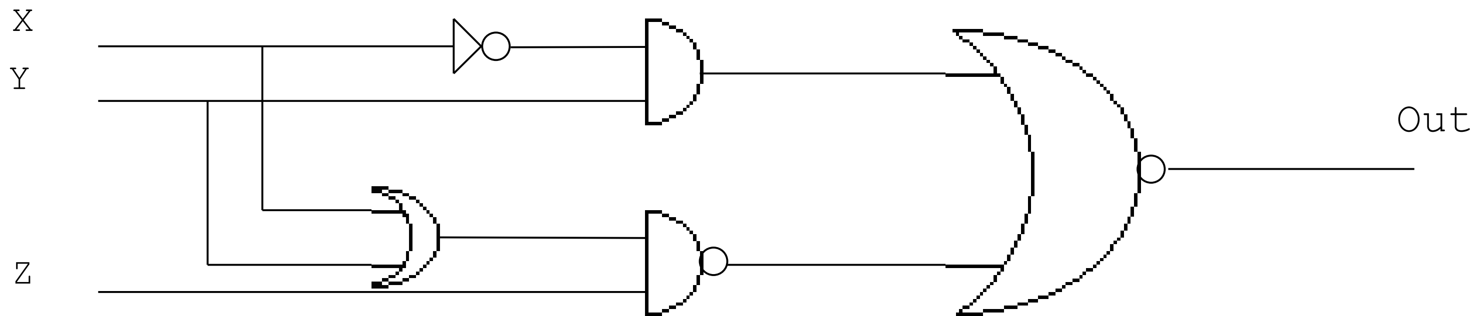
```
diff(C, X, 0) :- integer(C).
diff(X, X, 1).
diff(U+V, X, DU+DV) :- diff(U, X, DU),
                       diff(V, X, DV).
diff(-U, X, -DU) :- diff(U, X, DU).
diff(U-V, X, DU-DV) :- diff(U, X, DU),
                       diff(V, X, DV).
diff(C*U, X, C*DU) :- integer(C),
                      diff(U, X, DU).
```

Συμβολική παραγώγιση (2/2)

```
diff(U*V, X, U*DV+V*DU) :- diff(U, X, DU),
                           diff(V, X, DV).
diff(U/V, X, (V*DU-U*DV)/V^2) :- diff(U, X, DU),
                                 diff(V, X, DV).
diff(U^C, X, C*U^(C-1)*DU) :- integer(C),
                              diff(U, X, DU).
diff(sin(U), X, cos(U)*DU) :- diff(U, X, DU).

?- diff(3*x^2+x/sin(x), x, Diff).
   Diff = .........
```

Προσομοίωση λογικών κυκλωμάτων (1/4)

```
circuit(c1, nor(and(nt(X), Y),
                nand(or(X, Y), Z)),
            [X, Y, Z]).
```



Προσομοίωση λογικών κυκλωμάτων (2/4)

```
?- circuit(c1, C, [1, 0, 1]), circ(C, Out).
   C = ...........
   Out = .........

?- circuit(c1, C, Inputs), circ(C, 1).
   C = ...........
   Inputs = ......

?- circuit(c1, C, Inputs), circ(C, Out).
   C = ...........
   Inputs = ......
   Out = .........
```

Προσομοίωση λογικών κυκλωμάτων (3/4)

```
circ(1, 1) :- !.
circ(0, 0) :- !.


circ(and(X, Y), 1) :- circ(X, 1), circ(Y,
  1).
circ(and(X, Y), 0) :- circ(X, 1), circ(Y,
  0).
circ(and(X, Y), 0) :- circ(X, 0), circ(Y,
  1).
circ(and(X, Y), 0) :- circ(X, 0), circ(Y,
  0).
```

Προσομοίωση λογικών κυκλωμάτων (4/4)

```
circ(or(X, Y), 1) :- circ(X, 1), circ(Y, 1).
circ(or(X, Y), 1) :- circ(X, 1), circ(Y, 0).
circ(or(X, Y), 1) :- circ(X, 0), circ(Y, 1).
circ(or(X, Y), 0) :- circ(X, 0), circ(Y, 0).

circ(nt(X), 1) :- circ(X, 0).
circ(nt(X), 0) :- circ(X, 1).

circ(nand(X, Y), Z) :- circ(nt(and(X, Y)), Z).

circ(nor(X, Y), Z) :- circ(nt(or(X, Y)), Z).
```

Κατανόηση φυσικής γλώσσας (1/6)

```
:- op(190, yfx, :).
:- op(180, yfx, ==>).
:- op(170, yfx, &).

sentence(L, P) :-
    sentence(L, [], P).
sentence(L1, L3, P) :-
    noun_phrase(L1, L2, X, P1, P),
    verb_phrase(L2, L3, X, P1).
```

Κατανόηση φυσικής γλώσσας (2/6)

```
noun_phrase(L1, L4, X, P1, P) :-
    determiner(L1, L2, X, P2, P1, P),
    noun(L2, L3, X, P3),
    rel_clause(L3, L4, X, P3, P2).
noun_phrase(L1, L2, X, P, P) :-
    name(L1, L2, X).
verb_phrase(L1, L3, X, P) :-
    trans_verb(L1, L2, X, Y, P1),
    noun_phrase(L2, L3, Y, P1, P).
```

Κατανόηση φυσικής γλώσσας (3/6)

```
verb_phrase(L1, L2, X, P) :-
    intrans_verb(L1, L2, X, P).
rel_clause([that|L1], L2, X, P1, P1&P2) :-
    verb_phrase(L1, L2, X, P2).
rel_clause(L, L, X, P, P).
determiner([every|L], L, X, P1, P2,
  all(X):(P1==>P2)).
determiner([a|L], L, X, P1, P2,
  exists(X):(P1&P2)).
```

Κατανόηση φυσικής γλώσσας (4/6)

```
noun([man|L], L, X, man(X)).
noun([woman|L], L, X, woman(X)).
name([john|L], L, john).
name([mary|L], L, mary).
trans_verb([loves|L], L, X, Y, loves(X,
  Y)).
trans_verb([hates|L], L, X, Y, hates(X,
  Y)).
intrans_verb([lives|L], L, X, lives(X)).
intrans_verb([sings|L], L, X, sings(X)).
```

Κατανόηση φυσικής γλώσσας (5/6)

```
?- sentence([john, lives], P).
   P = lives(john)
   yes
?- sentence([mary, hates, every, man], P).
   P = all(_0084):(man(_0084)==>hates(mary, _0084))
   yes
?- sentence([a, man, that, sings, loves, mary], P).
   P = exists(_0084):((man(_0084)&sings(_0084))&
                      loves(_0084, mary))
   yes
```

Κατανόηση φυσικής γλώσσας (6/6)

```
?- sentence([every, woman, that, loves, john,
  hates,
           every, woman, that, lives], P).
   P = all(_0084):(woman(_0084)&loves(_0084,
  john)==>

  all(_0086):(woman(_0086)&lives(_0086)==>
                             hates(_0084,
  _0086)))
   yes
```

Πρώτα κατά βάθος αναζήτηση (και εφαρμογές της) (1/2)

```
depth_first_search(States) :-
   initial_state(State),
   depth_first_search(State, [State], States).

depth_first_search(State, States, States) :-
   final_state(State).

depth_first_search(State1, SoFarStates, States) :-
   move(State1, State2),
   not member(State2, SoFarStates),
   append(SoFarStates, [State2], NewSoFarStates),
   depth_first_search(State2, NewSoFarStates, States).
```

Πρώτα κατά βάθος αναζήτηση (και εφαρμογές της) (2/2)

```
member(............

append(............


initial_state(.....

final_state(.......

move(..............
```

Ορισμοί εξαρτώμενοι από το πρόβλημα

Το πρόβλημα του αγρότη, του λύκου, της κατσίκας και του λάχανου (1/4)

- Ένας αγρότης θέλει να μεταφέρει από τη μία όχθη ενός ποταμού στην άλλη ένα λύκο, μία κατσίκα και ένα λάχανο.
- Για το σκοπό αυτό έχει στη διάθεσή του μια βάρκα η οποία μπορεί, φυσικά, να οδηγηθεί μόνο από τον ίδιο και η οποία μπορεί να χωρέσει, εκτός από τον αγρότη, μόνο ένα από τα “είδη” προς μεταφορά.
- Πώς πρέπει να γίνει η μεταφορά έτσι ώστε ποτέ να μη βρεθούν μαζί σε κάποια όχθη ο λύκος και η κατσίκα ή η κατσίκα και το λάχανο χωρίς να είναι ο αγρότης παρών (για τους προφανείς λόγους);

Το πρόβλημα του αγρότη, του λύκου, της κατσίκας και του λάχανου (2/4)

```
initial_state(fwgc(l, l, l, l)).
final_state(fwgc(r, r, r, r)).
move(fwgc(F1, W1, G1, C1), fwgc(F2, W2, G2,
  C2)) :-
   opposite(F1, F2),
   ((W1 = W2, G1 = G2, C1 = C2) ;
    (opposite(W1, W2), G1 = G2, C1 = C2) ;
    (W1 = W2, opposite(G1, G2), C1 = C2) ;
    (W1 = W2, G1 = G2, opposite(C1, C2))),
   not illegal(fwgc(F2, W2, G2, C2)).
```

Το πρόβλημα του αγρότη, του λύκου, της κατσίκας και του λάχανου (3/4)

```
illegal(fwgc(F, W, G, C)) :-
    opposite(F, G),
    (G = W ; G = C).

opposite(l, r).
opposite(r, l).

?- depth_first_search(States).
    States = [fwgc(l, l, l, l), fwgc(r, l, r, l),
              fwgc(l, l, r, l), fwgc(r, r, r, l),
              fwgc(l, r, l, l), fwgc(r, r, l, r),
              fwgc(l, r, l, r), fwgc(r, r, r, r)]  -> ;
```

Το πρόβλημα του αγρότη, του λύκου, της κατσίκας και του λάχανου (4/4)

```
States = [fwgc(l, l, l, l), fwgc(r, l, r, l),
          fwgc(l, l, r, l), fwgc(r, l, r, r),
          fwgc(l, l, l, r), fwgc(r, r, l, r),
          fwgc(l, r, l, r), fwgc(r, r, r, r)]
  yes
```

Το πρόβλημα των κανατών (1/4)

- Έχουμε δύο κανάτες, χωρίς ενδείξεις όγκου περιεχομένου, χωρητικότητας 8 και 5 λίτρων αντίστοιχα.
- Υπάρχει επίσης διαθέσιμη μία βρύση από την οποία οι κανάτες μπορούν να γεμίζουν ή να συμπληρώνονται με νερό.
- Κάθε κανάτα επίσης μπορεί να γεμίσει ή να συμπληρωθεί με νερό από την άλλη κανάτα.
- Τέλος, μπορούμε κάθε κανάτα να την αδειάσουμε στο έδαφος ή στην άλλη κανάτα.
- Το πρόβλημα είναι να βρεθεί μια αλληλουχία κινήσεων μετά από την οποία η πρώτη κανάτα θα περιέχει 4 λίτρα νερό και η δεύτερη θα είναι άδεια.

Το πρόβλημα των κανατών (2/4)

```
initial_state(jugs(0, 0)).
final_state(jugs(4, 0)).

move(jugs(V1, V2), jugs(W1, W2)) :-
   C1 = 8, C2 = 5, L is V1+V2,
   ((V1 < C1, W1 = C1, W2 = V2) ;
    (V2 < C2, W1 = V1, W2 = C2) ;
    (V1 > 0, W1 = 0, W2 = V2) ;
    (V2 > 0, W1 = V1, W2 = 0) ;
    (minimax(L, C2, W2), W1 is L-W2) ;
    (minimax(L, C1, W1), W2 is L-W1)).

minimax(X, Y, X) :- X =< Y.
minimax(X, Y, Y) :- X > Y.
```

Το πρόβλημα των κανατών (3/4)

```
?- depth_first_search(States).
(1)    States = [jugs(0, 0), jugs(8, 0), jugs(8, 5),
                 jugs(0, 5), jugs(5, 0), jugs(5, 5),
                 jugs(8, 2), jugs(0, 2), jugs(2, 0),
                 jugs(2, 5), jugs(7, 0), jugs(7, 5),
                 jugs(8, 4), jugs(0, 4), jugs(4, 0)]   -> ;
                 ...................................   -> ;
(2)    States = [jugs(0, 0), jugs(8, 0), jugs(3, 5),
                 jugs(3, 0), jugs(0, 3), jugs(8, 3),
                 jugs(6, 5), jugs(6, 0), jugs(1, 5),
                 jugs(1, 0), jugs(0, 1), jugs(8, 1),
                 jugs(4, 5), jugs(4, 0)]               -> ;
                 ...................................   -> ;
```

Το πρόβλημα των κανατών (4/4)

```
(3) States = [jugs(0, 0), jugs(0, 5), jugs(5, 0),
              jugs(5, 5), jugs(8, 2), jugs(0, 2),
              jugs(2, 0), jugs(2, 5), jugs(7, 0),
              jugs(7, 5), jugs(8, 4), jugs(0, 4),
              jugs(4, 0)]                         -> ;
              ..................................
   yes
```

```
(1) ⎫
(2) ⎬   26 λύσεις
(3) ⎭
```

- Να λυθεί με βάση το γενικό πλαίσιο της πρώτα κατά βάθος αναζήτησης και το πρόβλημα των ιεραποστόλων και κανιβάλων.

Έμπειρα συστήματα (expert systems) και λογικός προγραμματισμός

Έμπειρα συστήματα (expert systems) (1/2)

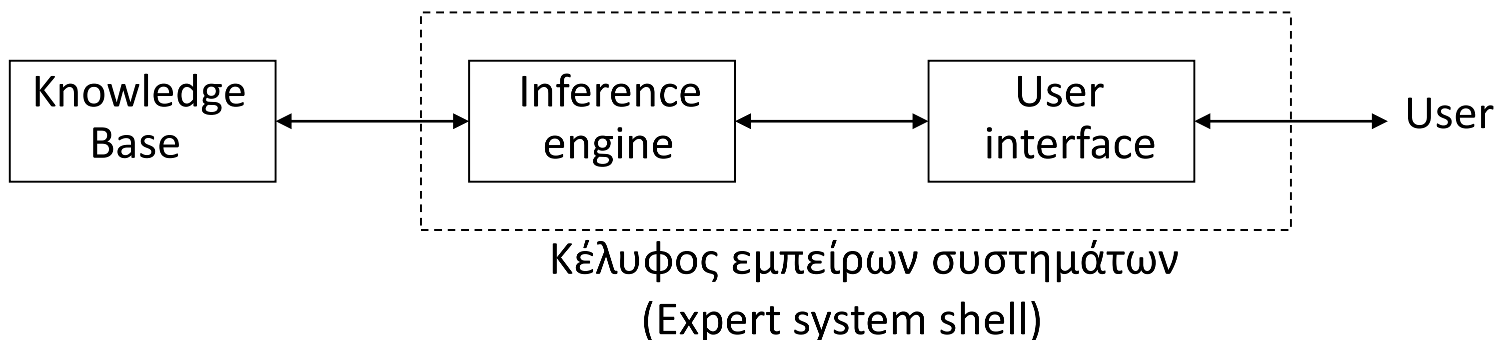
- Ένα έμπειρο σύστημα προσομοιώνει τη συμπεριφορά ενός ειδικού / εμπείρου προσώπου σε ένα περιορισμένο γνωστικό πεδίο
- Περιοχές εφαρμογής
  - Ιατρική διάγνωση
  - Διάγνωση μηχανικών βλαβών
  - Σχεδίαση
  - Σύνθεση προδιαγραφών
  - Ταξινόμηση
- Τα έμπειρα συστήματα είναι συστήματα βάσης γνώσεων (knowledge-base systems)

Έμπειρα συστήματα (expert systems) (2/2)

- Επιθυμητές δυνατότητες εμπείρων συστημάτων
  - Παροχή εξηγήσεων
  - Χειρισμός αβέβαιης ή/και ελλιπούς πληροφορίας
- Το βασικότερο πρόβλημα στην κατασκευή εμπείρων συστημάτων είναι η απόκτηση της γνώσης (knowledge acquisition) από τον άνθρωπο-ειδικό
- Ένα πολύ κλασικό έμπειρο σύστημα είναι το MYCIN (Buchanan, Shortliffe, ’75 - ’85) για τη διάγνωση μολυσματικών ασθενειών και προτάσεις θεραπευτικής αγωγής

Δομή των έμπειρων συστημάτων

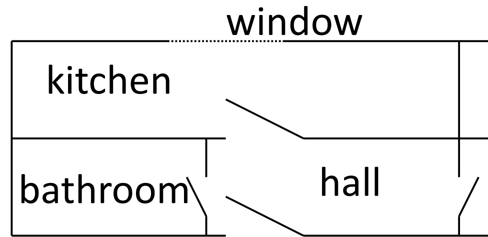
- Μηχανή συμπερασμού (Inference engine)
- Σύστημα επικοινωνίας με το χρήστη (User interface)
- Βάση γνώσης (Knowledge base)

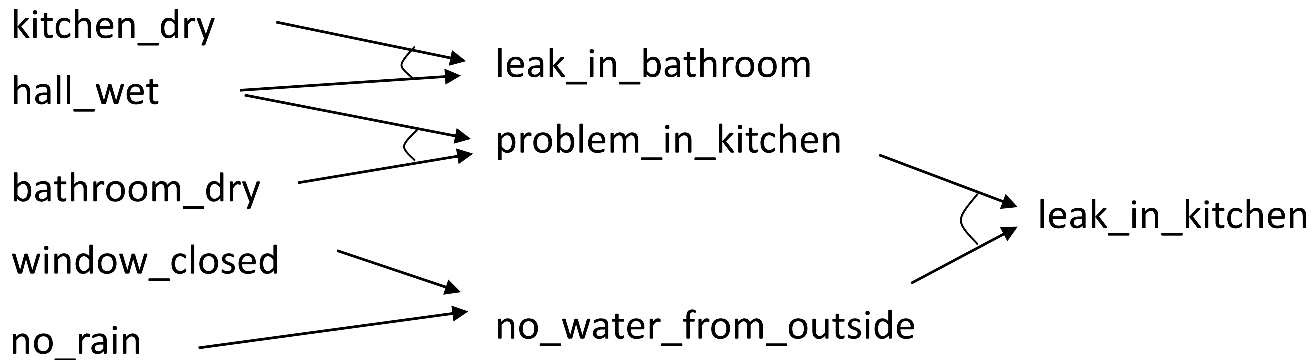


Αναπαράσταση γνώσης στα έμπειρα συστήματα

- **Μέσω if-then** κανόνων ή κανόνων παραγωγής (production rules) if precondition P then conclusion C
- Πλεονεκτήματα if-then κανόνων
  - Σταδιακή κατασκευή βάσης γνώσης
  - Διευκόλυνση απαντήσεων σε ερωτήσεις “πώς” και “γιατί”
  - Δυνατότητα επέκτασης για χειρισμό αβέβαιης πληροφορίας
  - Ικανοποιητικό μέσο κωδικοποίησης της γνώσης ενός ανθρώπου-ειδικού
- Χειρισμός if-then κανόνων
  - Με οπίσθιο τρόπο (backward chaining)
  - Με εμπρόσθιο τρόπο (forward chaining)

Ένα πρόβλημα διάγνωσης






```
if kitchen_dry and hall_wet then leak_in_bathroom

if window_closed or no_rain then
  no_water_from_outside

.................
```

Τρόποι προσέγγισης

- Οπίσθια συλλογιστική με απλή Prolog
- Οπίσθια συλλογιστική σε μέτα-επίπεδο με απλή Prolog
- Εμπρόσθια συλλογιστική σε μέτα-επίπεδο με απλή Prolog
- Χρησιμοποίηση κάποιου κελύφους εμπείρου συστήματος (υλοποιημένου σε Prolog;)

Οπίσθια συλλογιστική με απλή Prolog (1/2)

```
leak_in_bathroom :-
   kitchen_dry,

   hall_wet.

problem_in_kitchen :-
   hall_wet,

   bathroom_dry.

no_water_from_outside :-
   window_closed ;

   no_rain.
```

```
leak_in_kitchen :-
   problem_in_kitchen,

   no_water_from_outside.


hall_wet.

bathroom_dry.

window_closed.


?- leak_in_kithcen.

   yes
```

Οπίσθια συλλογιστική με απλή Prolog (2/2)

- Η οπίσθια συλλογιστική με απλή Prolog δεν είναι ο πιο ευέλικτος τρόπος λειτουργίας if-then κανόνων σε έμπειρα συστήματα (Γιατί;)

Οπίσθια συλλογιστική σε μέτα-επίπεδο με απλή Prolog

```
:- op(800, fx, if).
:- op(700, xfx, then).
:- op(300, xfy, or).
:- op(200, xfy, and).
if kitchen_dry and hall_wet
   then leak_in_bathroom.
if window_closed or no_rain
   then
  no_water_from_outside.
..................

fact(hall_wet).
fact(bathroom_dry).
fact(window_closed).
```

```
is_true(P) :- fact(P).
is_true(P) :- if Condition
   then P,

   is_true(Condition).
is_true(P1 and P2) :-
   is_true(P1),

   is_true(P2).
is_true(P1 or P2) :-
   is_true(P1) ;

   is_true(P2).


?- is_true(leak_in_kitchen).
   yes
```

Εμπρόσθια συλλογιστική σε μέτα-επίπεδο με απλή Prolog (1/2)

```
:- op(800, ............
......................
if kitchen-dry ............
......................
fact(............
......................
forward :- new_derived_fact(P), !, write('Derived: '),
            write(P), nl, assert(fact(P)), forward ;
            write('No more facts').
new_derived_fact(Concl) :- if Cond then Concl,
                           not fact(Concl),
                           composed_fact(Cond).
```

Εμπρόσθια συλλογιστική σε μέτα-επίπεδο με απλή Prolog (2/2)

```
composed_fact(Cond) :- fact(Cond).
composed_fact(Cond1 and Cond2) :- composed_fact(Cond1),
                                  composed_fact(Cond2).
composed_fact(Cond1 or Cond2) :- composed_fact(Cond1) ;
                                 composed_fact(Cond2).


?- forward.
   Derived: problem_in_kitchen
   Derived: no_water_from_outside
   Derived: leak_in_kitchen
   No more facts
   yes
```

Οπίσθια ή εμπρόσθια συλλογιστική;

- Η οπίσθια συλλογιστική είναι οδηγούμενη από το στόχο (goal driven)
- Η εμπρόσθια συλλογιστική είναι οδηγούμενη από τα δεδομένα (data driven)
- Το αν η εμπρόσθια ή η οπίσθια συλλογιστική είναι προτιμότερη, εξαρτάται από το συγκεκριμένο πρόβλημα
- Μερικές φορές είναι χρήσιμος ο συνδυασμός των δύο συλλογιστικών

Κριτήρια επιλογής

- Έχουμε όλα τα δεδομένα υπόψη μας;
- Θέλουμε να αποδείξουμε κάτι συγκεκριμένο ή δεν ξέρουμε τι θέλουμε να αποδείξουμε
- Μήπως ο AND-OR γράφος των if-then κανόνων έχει λίγους αρχικούς κόμβους και πολλούς τελικούς;
- Μήπως έχει πολλούς αρχικούς και λίγους τελικούς;
- Αν πρόκειται να χρησιμοποιήσουμε ένα συγκεκριμένο κέλυφος εμπείρων συστημάτων, τι είδους συλλογιστικές υποστηρίζονται από αυτό;

Δυνατότητα παροχής εξηγήσεων

- (Πως αποδείχθηκε κάτι;)
- Επέκταση του μέτα-διερμηνέα για οπίσθια συλλογιστική:

```
:- op(800, xfx, <=).

is_true(P, P) :- fact(P).

is_true(P, P<=CondProof) :-
    if Cond then P,
    is_true(Cond, CondProof).
..
```

```
..
is_true(P1 and P2, Proof1
   and Proof2) :-
    is_true(P1, Proof1),
    is_true(P2, Proof2).

is_true(P1 or P2, Proof) :-
    is_true(P1, Proof) ;
    is_true(P2, Proof).
```

Χειρισμός αβέβαιης πληροφορίας (1/3)

- Εισαγωγή ποσοτικού μέτρου βεβαιότητας σε γεγονότα και if-then κανόνες: 0 Certainty 1

```
given(hall_wet, 1).

given(bathroom_dry, 1).

given(kitchen_dry, 0).

given(no_rain, 0.8).

given(window_closed, 0)
```

Χειρισμός αβέβαιης πληροφορίας (2/3)

```
if hall_wet and bathroom_dry
   then problem_in_kitchen                      :  0.9
if kitchen_dry and hall_wet
   then leak_in_bathroom                        :    1
if window_closed or no_rain
   then no_water_from_outside                   :    1
if problem_in_kitchen and no_water_from_outside
   then leak_in_kitchen                         :    1
```

Χειρισμός αβέβαιης πληροφορίας (3/3)

Έστω ότι ορίζουμε:

c(P1 and P2) = min(c(P1), c(P2))

c(P1 or P2) = max(c(P1), c(P2))

c(P2) = c(P1) c αν if P1 then P2 : c

Επέκταση του μέτα-διερμηνέα για οπίσθια συλλογιστική (1/2)

```
certainty(P, Cert) :-
    given(P, Cert).
certainty(P1 and P2, Cert) :-
    certainty(P1, Cert1),
    certainty(P2, Cert2),
    minimum(Cert1, Cert2, Cert).

certainty(P1 or P2, Cert) :-
    certainty(P1, Cert1),
    certainty(P2, Cert2),
    maximum(Cert1, Cert2, Cert).
```

Επέκταση του μέτα-διερμηνέα για οπίσθια συλλογιστική (2/2)

```
certainty(P, Cert) :-
   if Cond then P : C1,
   certainty(Cond, C2),
   Cert is C1*C2.

minimum(............
maximum(............

?- certainty(leak_in_kitchen, C).
   C = 0.8
   yes
```

Τέλος Ενότητας

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στο πλαίσιο του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Αθηνών**» έχει χρηματοδοτήσει μόνο την αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Σημειώματα

Σημείωμα Ιστορικού Εκδόσεων Έργου

Το παρόν έργο αποτελεί την έκδοση 1.0

Σημείωμα Αναφοράς

Copyright Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών, Παναγιώτης Σταματόπουλος, Ιζαμπώ Καράλη. «Λογικός Προγραμματισμός, Επεκτάσεις, υλοποίηση, παραλληλία». Έκδοση: 1.0. Αθήνα 2015. Διαθέσιμο από τη δικτυακή διεύθυνση: http://opencourses.uoa.gr/courses/DI117/.

Σημείωμα Αδειοδότησης

Το παρόν υλικό διατίθεται με τους όρους της άδειας χρήσης Creative Commons Αναφορά, Μη Εμπορική Χρήση Παρόμοια Διανομή 4.0 [1] ή μεταγενέστερη, Διεθνής Έκδοση. Εξαιρούνται τα αυτοτελή έργα τρίτων π.χ. φωτογραφίες, διαγράμματα κ.λ.π., τα οποία εμπεριέχονται σε αυτό και τα οποία αναφέρονται μαζί με τους όρους χρήσης τους στο «Σημείωμα Χρήσης Έργων Τρίτων».

[1] http://creativecommons.org/licenses/by-nc-sa/4.0/

Ως **Μη Εμπορική** ορίζεται η χρήση:

- που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο
- που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο
- που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

Διατήρηση Σημειωμάτων

Οποιαδήποτε αναπαραγωγή ή διασκευή του υλικού θα πρέπει να συμπεριλαμβάνει:

- το Σημείωμα Αναφοράς
- το Σημείωμα Αδειοδότησης
- τη δήλωση Διατήρησης Σημειωμάτων
- το Σημείωμα Χρήσης Έργων Τρίτων (εφόσον υπάρχει)

μαζί με τους συνοδευόμενους υπερσυνδέσμους.

Σημείωμα Χρήσης Έργων Τρίτων

Το Έργο αυτό κάνει χρήση των ακόλουθων έργων:

Εικόνες/Σχήματα/Διαγράμματα/Φωτογραφίες