ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΣΗΜΕΙΩΣΕΙΣ

ΛΟΓΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

ΠΑΝΑΓΙΩΤΗΣ ΣΤΑΜΑΤΟΠΟΥΛΟΣ ΙΖΑΜΠΩ ΚΑΡΑΛΗ

ΑΘΗΝΑ 2011

ΜΕΡΟΣ Α΄

Συγγραφή: ΠΑΝΑΓΙΩΤΗΣ ΣΤΑΜΑΤΟΠΟΥΛΟΣ

Οι σημειώσεις αυτές θα είχαν μείνει στην αρχική χειρόγραφη μορφή τους,
αν δεν προσφερόταν ευγενικά ο Παναγιώτης Κόκκαλης, φοιτητής του
Τμήματος Πληροφορικής και Τηλεπικοινωνιών του Πανεπιστημίου Αθηνών,
να τις μεταφέρει σε Microsoft PowerPoint® το καλοκαίρι του 2001.
Τον ευχαριστώ πολύ και εκ μέρους των συναδέλφων του.

Παναγιώτης Σταματόπουλος

Λογική

Κάθε άνθρωπος είναι θνητός
Ο Σωκράτης είναι άνθρωπος
→ Ο Σωκράτης είναι θνητός

Κατηγορηματική λογική πρώτης τάξης

(First order predicate logic)

$((\forall x)(man(x) \Rightarrow fallible(x)) \wedge man(socrates)) \models fallible(socrates)$

Prolog

```
fallible(X) :- man(X).

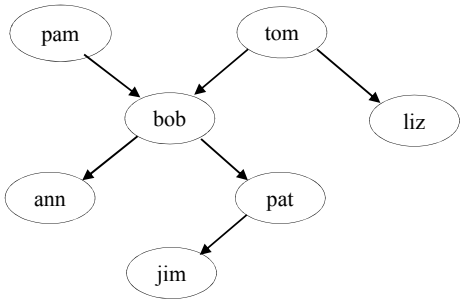
man(socrates).


?- fallible(socrates).
   yes
```

PROLOG (**PRO**gramming in **LOG**ic)

Ένα απλό πρόγραμμα

```
parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
```



Ερωτήσεις

```
?- parent(bob, pat).
   yes
?- parent(liz, pat).
   no
?- parent(tom, ben).
   no
?- parent(X, liz).
   X = tom
   yes
?- parent(bob, X).
   X = ann        -> ;
   X = pat
   yes
```

```
?- parent(X, Y).
   X = pam
   Y = bob      -> ;
   X = tom
   Y = bob      -> ;
   X = tom
   Y = liz      -> ;
   X = bob
   Y = ann      -> ;
   X = bob
   Y = pat      -> ;
   X = pat
   Y = jim
   yes
```


Πιο σύνθετες ερωτήσεις

```
?- parent(Y, jim), parent(X, Y).
   X = bob
   Y = pat
   yes
?- parent(X, Y), parent(Y, jim).
   X = bob
   Y = pat
   yes
?- parent(tom, X), parent(X, Y).
   X = bob
   Y = ann       -> ;
   X = bob
   y = pat
   yes
?- parent(X, ann), parent(X, pat).
   X = bob
   yes
```

```
?- parent(jim, X).
?- parent(X, jim).
?- parent(pam, X), parent(X, pat).
?- parent(pam, X), parent(X, Y), parent(Y, jim).
```

ΑΠΑΝΤΗΣΕΙΣ;

Ποιος είναι γονιός της pat;

Έχει η liz παιδί;

Ποιος είναι παππούς/γιαγιά της pat;

Επέκταση του προγράμματος

```
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).
```

- 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

```
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

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


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

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

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

Αναδρομικοί κανόνες

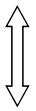
(1)
```
predecessor(X, Z) :-
        parent(X, Z).
```

(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)`

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

……..



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



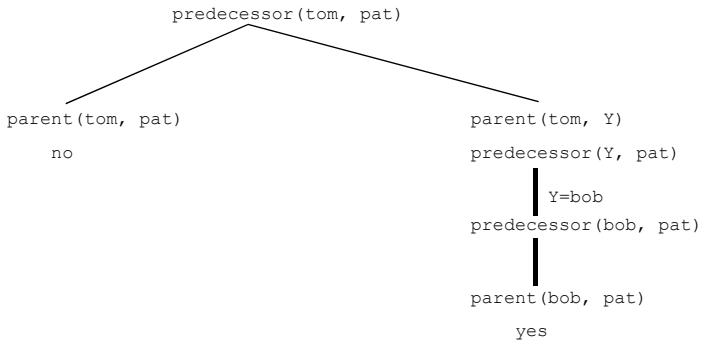
```
?- predecessor(pam, X).
   X = bob     -> ;
   X = ann     -> ;
   X = pat     -> ;
   X = jim
   yes
```

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

? ⇕ ?

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

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




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

ΕΚΤΕΛΕΣΗ;

- 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: _

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

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

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

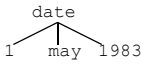
```
somebody_has_child :- parent(_, _).
```

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

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


Δομές (structures)

```
date(1, may, 1983)
```



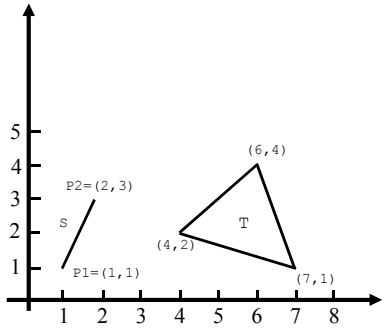
- functor (συναρτησιακό σύμβολο)
- arguments (ορίσματα)
- arity (βαθμός)
- terms (όροι)
  - άτομα
  - αριθμοί
  - μεταβλητές
  - δομές (ή σύνθετοι όροι)

```
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

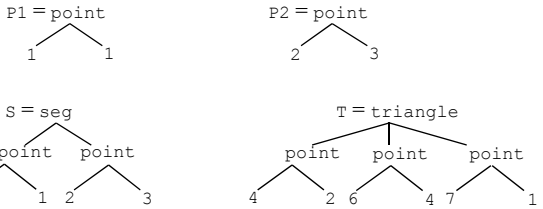


P1 = point(1, 1)

P2 = point(2, 3)

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

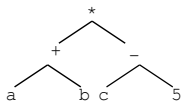
T = triangle(point(4, 2), point(6, 4), point(7, 1))

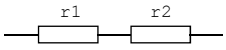


3-D: point3(X, Y, Z) ή point(X, Y, Z)

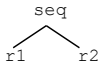
- (a+b)*(c-5)

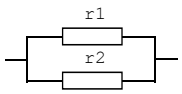
  *(+(a, b), -(c, 5))



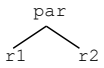
- 

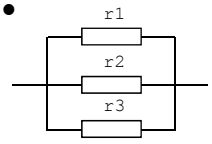
  seq(r1, r2)



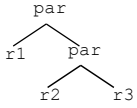
- 

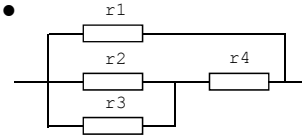
  par(r1, r2)



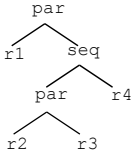
- 

  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
  - 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

```
?- 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
    ?- 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
```

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

```
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

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.

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(1))))), C).
```

2) Να οριστεί η σχέση `relatives(X, Y)` η οποία αληθεύει όταν οι `X` και `Y` είναι συγγενείς

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

- 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
- 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

(1) `big(bear).`

(2) `big(elephant).`

(3) `small(cat).`

(4) `brown(bear).`

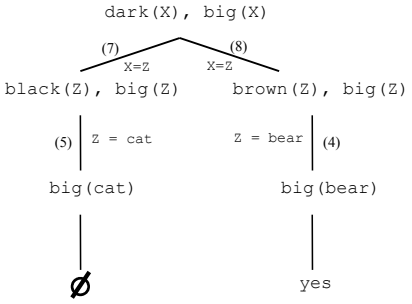
(5) `black(cat).`

(6) `gray(elephant).`

(7) `dark(Z) :- black(Z).`

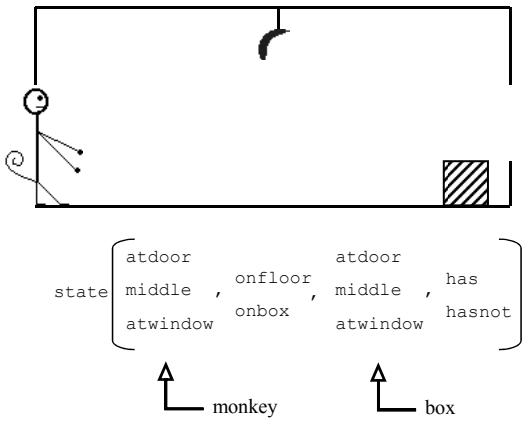
(8) `dark(Z) :- brown(Z).`

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




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

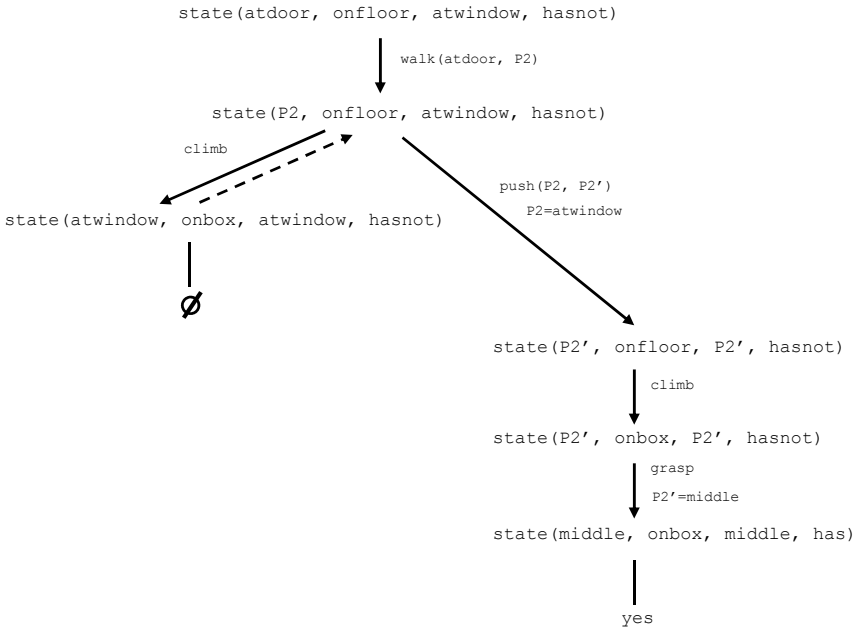

Ο πίθηκος και η μπανάνα




```
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)).
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
```



Ατέρμονες ανακυκλώσεις

- ```
  p :- p.
  ?- p.
  ```
- Πίθηκος και μπανάνα:

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

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

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

Συμπέρασμα:

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

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

```
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).
```

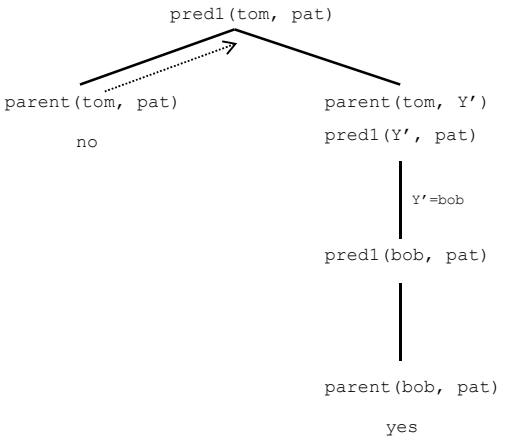
(4)

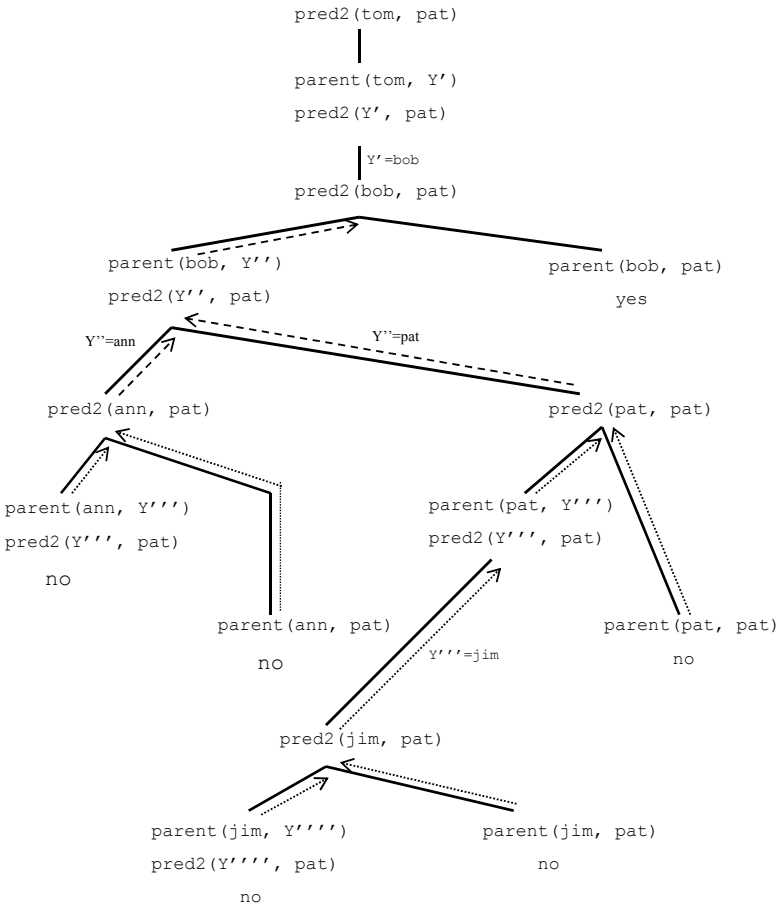
```
?- pred1(tom, pat).
   yes
```

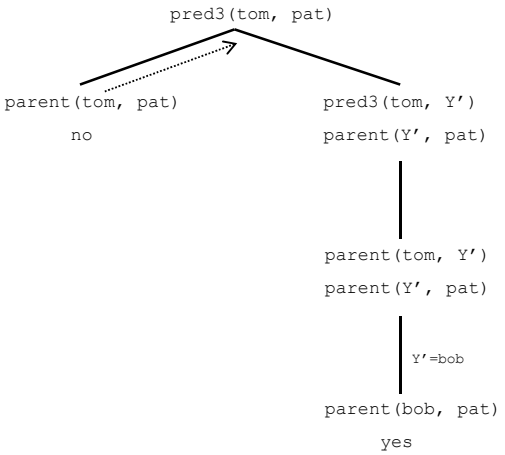
```
?- pred2(tom, pat).
   yes
```

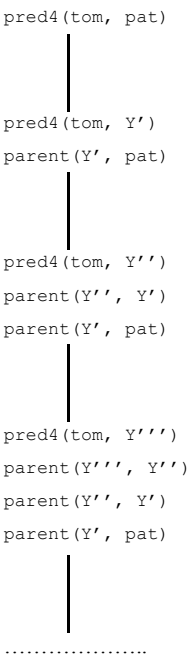
```
?- pred3(tom, pat).
   yes
```

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


pred2(tom, pat)
parent(tom, Y')
pred2(Y', pat)
Y'=bob
pred2(bob, pat)
parent(bob, Y'')
pred2(Y'', pat)
parent(bob, pat)
yes
Y''=ann
Y''=pat
pred2(ann, pat)
pred2(pat, pat)
parent(ann, Y''')
pred2(Y''', pat)
no
parent(pat, Y''')
pred2(Y''', pat)
parent(ann, pat)
no
Y'''=jim
parent(pat, pat)
no
pred2(jim, pat)
parent(jim, Y'''')
pred2(Y'''', pat)
no
parent(jim, pat)
no


pred3(tom, pat)
parent(tom, pat)
no
pred3(tom, Y')
parent(Y', pat)
parent(tom, Y')
parent(Y', pat)
Y'=bob
parent(bob, pat)
yes



`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`

Κατηγορηματική λογική πρώτης τάξης
(First order predicate logic)

↓

Προτασιακή μορφή
(Clause form)

↓

Προτάσεις Horn
(Horn Clauses)

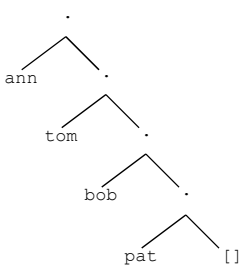
- Αρχή της ανάλυσης (Resolution principle)
- Ενοποίηση (Unification)

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 `[]`




```
.(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

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


`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).
```

```
?- 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
```

`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`)

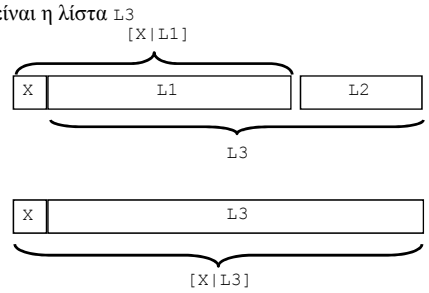
π.χ. `append([a, b], [c, d, e], [a, b, c, d, e])` : αληθές

`append([a], [[b]], [a, b])` : ψευδές

`append([1, 2], [], [1, 2])` : αληθές

i) Η συνένωση της κενής λίστας `[]` με μία τυχαία λίστα `L` 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).
```

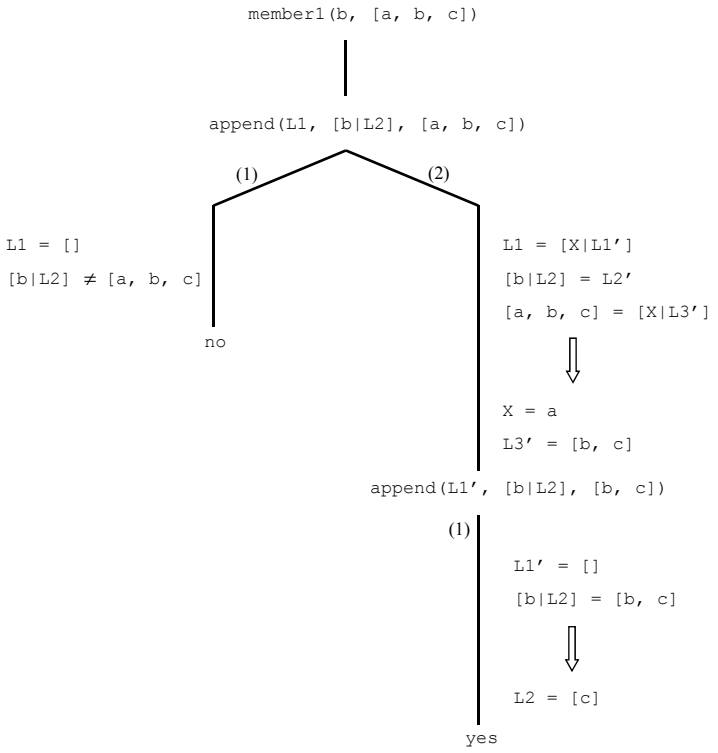


```
?- append([a, b], [c, d, e], L).
   L = [a, b, c, d, e]
   yes
?- append([a, [b, c], d], [a, [], b], L).
   L = [a, [b, c], d, a, [], b]
   yes
?- append(L1, L2, [a, b, c]).
   L1 = []
   L2 = [a, b, c]      -> ;
   L1 = [a]
   L2 = [b, c]         -> ;
   L1 = [a, b]
   L2 = [c]            -> ;
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

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

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



Εισαγωγή στοιχείου στην αρχή 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
?- 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, _ ).
```

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`

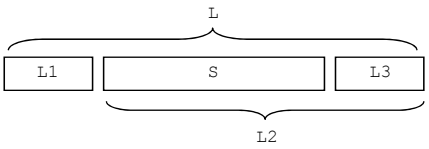
b) χωρίς την `append`

`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




```
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

```
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).
```

```
?- 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

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).

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
```

1) a) `evenlength(List)` b) `oddlength(List)`

2) `palindrome(List)`

3) `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
```

4) `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
```

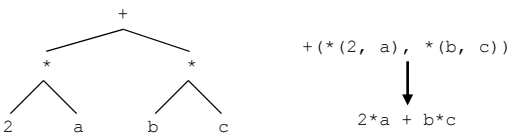
5) `dividelist(List, List1, List2)`

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

6) `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 (Precedence)

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

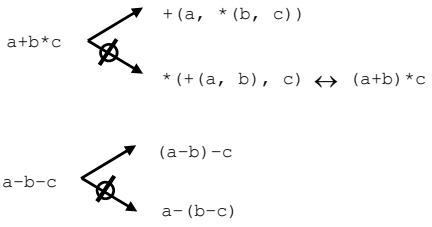
Αν ένα όρισμα κάποιου τελεστή είναι μέσα σε παρενθέσεις ή αν δεν είναι σύνθετος όρος στον οποίο συμμετέχουν τελεστές, τότε έχει προτεραιότητα μηδέν. Σε αντίθετη περίπτωση, η προτεραιότητα του είναι αυτή του τελεστή που είναι το βασικό συναρτησιακό του σύμβολο.

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:

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

`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

```
:- 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).

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

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.
```


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
?- 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

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

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

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

Ή

Ή

```
?- 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

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

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

- `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

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
```

- 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
```

- 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
```


- var/1

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

- \=/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

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



```
?- 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.
?- 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.
```

- 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) :- ............`

```
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).     -------->  ???

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).
```



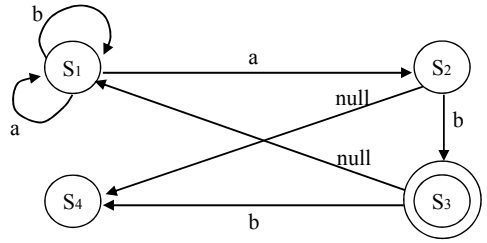
```
writefamily(family_str(H, W, C)) :-
      nl, write(parents), nl, writeperson(H), nl,
      writeperson(W), nl, write(children), nl,
      writepersonlist(C).

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).
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).

?- 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)])).
   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




```
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).
```

```
?- 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
```

Αν προστεθεί στο πρόγραμμα το:

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

Ποια είναι η απάντηση του:

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

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

Προγραμματισμός ταξιδιού:

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

timetable(edinburgh, london, [9:40/10:50/ba4733/alldays,
                              13:40/14:50/ba4773/alldays,
                              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]).
timetable(london, milan, [8:30/11:20/ba510/alldays,
                          11:00/13:50/az459/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]]).
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).

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]).

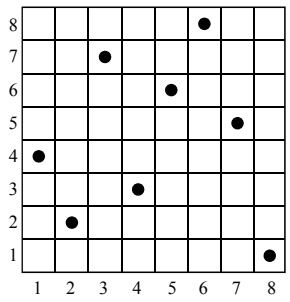
deptime([P1:P2-Fnum-Dep|_], Dep).

transfer(Hours1:Mins1, Hours2:Mins2) :-
   60*(Hours2-Hours1)+Mins2-Mins1 >= 40.

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

```
?- 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
?- 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
   FN1 = ba510          FN3 = yu323
   Dep1 = 8:30          Dep3 = 13:30
   Arr1 = 11:20         Arr3 = 14:40
   FN2 = sr621          FN4 = yu200
   Dep2 = 9:25          Dep4 = 11:10
   Arr2 = 10:15         Arr4 = 12:20
   yes
```


Το πρόβλημα των 8 βασιλισσών (1ος τρόπος)




```
[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).

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)

```
                    [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).

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).

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)

x: στήλη

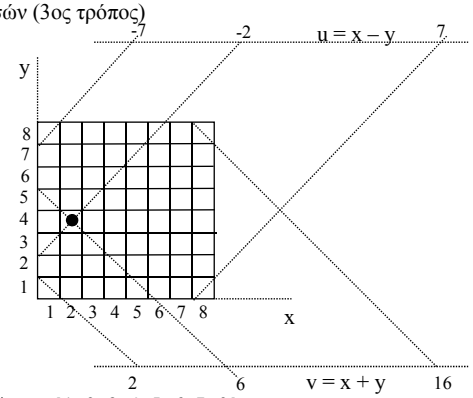
y: γραμμή

u: ανιούσα διαγώνιος

v: κατιούσα διαγώνιος

u = x – y

v = x + y




```
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).
```

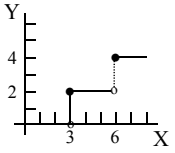
Γενίκευση για Ν βασίλισσες

```
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).

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

1) `jump(Square1, Square2) :- ......` (κίνηση αλόγου στο σκάκι)
2) `knightpath(Path) :- ......` (νόμιμη διαδρομή αλόγου σε άδεια σκακιέρα)
3) 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




```
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)

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.

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.

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)

```
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).
```

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, _),
                    !.
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>)`

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 `!`.

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)

```
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`)

```
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).
```



```
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]).
```

ΣΥΜΠΕΡΑΣΜΑ: 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
```

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
```


- 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.

```
?- 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

- `read/1`

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

```
cube(N, C) :- C is N*N*N.
?- cube(2, X).            ?- cube(12, Z).
   X = 8                     Z = 1728
   yes                       yes
```

```
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
```


- `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
```

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`

```
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).

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
```


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
```

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

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

```
?- nonvar(X).
   no
?- nonvar(bbb).
   yes
?- Z = 2, integer(Z), nonvar(Z).
   Z = 2
   yes
?- atom(22).
   no
?- atom(==>).
   yes
?- atom(X), X = costas.
   no
?- X = costas, atom(X).
   X = costas
   yes
```

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

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



```
  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).
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]).

?- 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.

```
?- 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
```

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

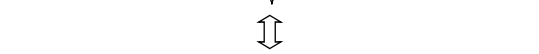
- `=../2`

```
?- 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
```

- `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
?- 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.
.............................................
```




```
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).

?- 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)),
   !.
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).

?- 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
```

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;

"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
```

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

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

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

- `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
```


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

```
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
```

- 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
```



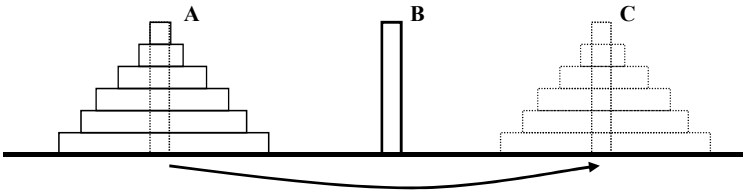
```
?- bagof(Child, Age^age(Child, Age), List).
   Child = _0084
   Age = _0098
   List = [peter, ann, pat, tom]
   yes
?- bagof(Age, Child^age(Child, Age), List).
   Age = _0084
   Child = _0098
   List = [7, 5, 8, 5]
   yes
?- setof(Child, Age^age(Child, Age), ChildList),
   setof(Age, Child^age(Child,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

```
?- 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 '),
   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

* 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.

* 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.

```
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).

* 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 είναι:

- 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].
```

```
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);

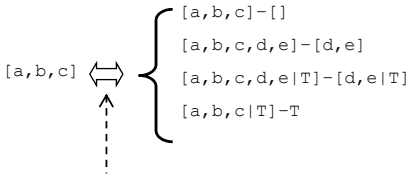
Το πρόβλημα του χρωματισμού γράφου

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

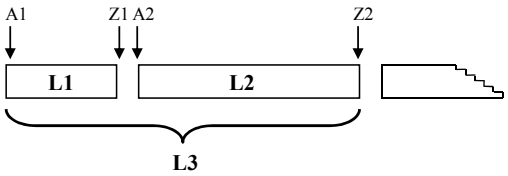


ΠΡΟΣΟΧΗ ΣΤΗ ΣΗΜΑΣΙΑ ΤΟΥ

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

$X_1 = 1$

$X_2 = 1$

$X_\nu = X_{\nu-1} + X_{\nu-2} \quad , \quad \forall \ \nu \geq 3$

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;

```
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).
```

ΣΧΟΛΙΟ;

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).
  ```

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

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

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

- 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]
```

- 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`.

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)
```

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).
```

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]).
```

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;

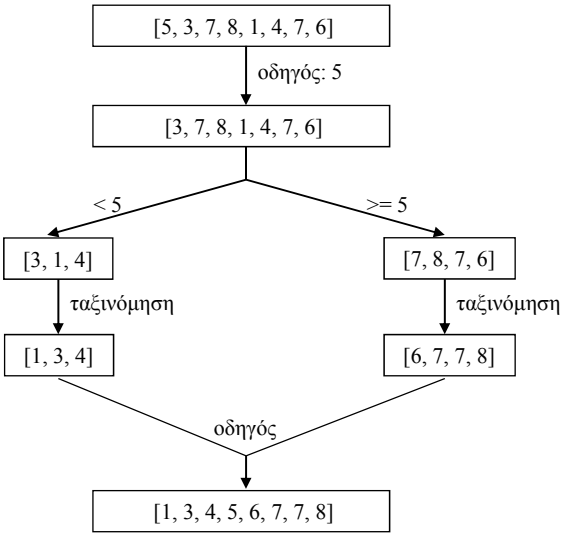
- Γρήγορη μέθοδος (quicksort)

```
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(............
```

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:

```
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`;

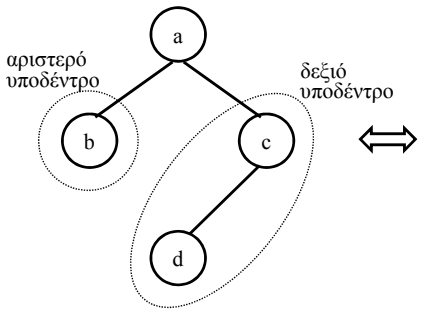
2. Μία άλλη μέθοδος ταξινόμησης είναι η `mergesort`. Σύμφωνα με αυτήν:

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

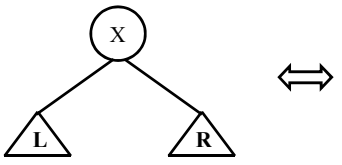
   Να υλοποιηθεί σε Prolog αυτή η μέθοδος και να συγκριθεί η απόδοσή της με αυτή της `quicksort`.

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 στην Prolog απαιτούνται ένα σύμβολο για το κενό δέντρο (έστω `nil`) και ένα σύμβολο δόμησης της ρίζας με τα δύο υποδέντρα (έστω `t`)




```
t(t(nil, b, nil),
  a,
  t(t(nil, d, nil),
    c,
    nil))
```




```
t(L, X, R)
```

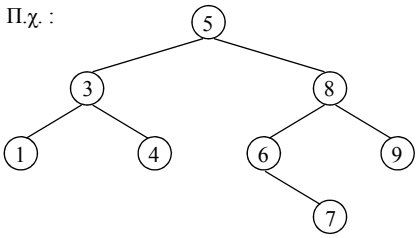
Στοιχείο μέλος ενός δυαδικού δέντρου

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


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. :



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).

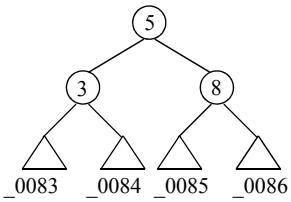
Στοιχείο μέλος ενός δυαδικού λεξικού

```
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).

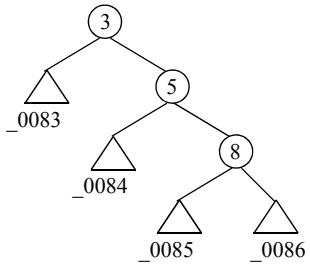
?- in(6, t(t(t(nil, 1, nil), 3, t(nil, 4, nil)),
           5,
           t(t(nil, 6, t(nil, 7, nil)),
             8,
             t(nil, 9, nil)))).
   yes
?- in(2, t(t(t(nil, 1, nil), 3, t(nil, 4, nil)),
           5,
           t(t(nil, 6, t(nil, 7, nil)),
             8,
             t(nil, 9, nil)))).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



```
?- 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
```



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 ώστε να συλλέγει όλα τα στοιχεία του δυαδικού δέντρου `Tree` στη λίστα `List`.

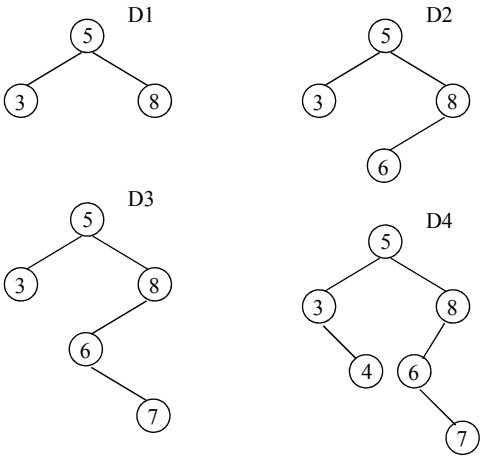
4. Να οριστεί το `maxelement(D, Item)` έτσι ώστε το `Item` να είναι το μεγαλύτερο στοιχείο του δυαδικού λεξικού `D`.

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

- 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).
```



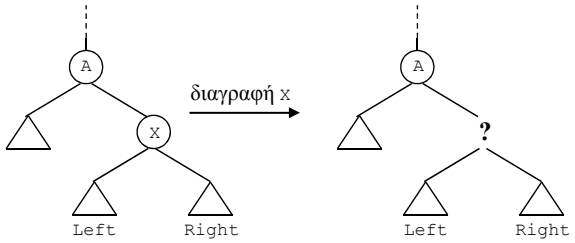

```
?- addleaf(t(t(nil, 3, nil), 5, t(nil, 8, nil)), 6, D2),
   addleaf(D2, 7, D3), addleaf(D3, 4, D4).
```

- Διαγραφή κόμβου από φύλλο

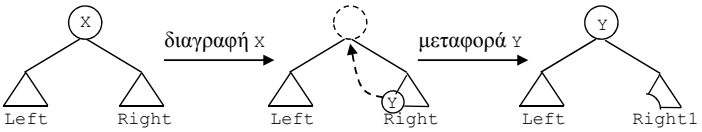
```
delleaf(D1, X, D2) :-
   addleaf(D2, X, D1).
```

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

- Διαγραφή κόμβου από οπουδήποτε



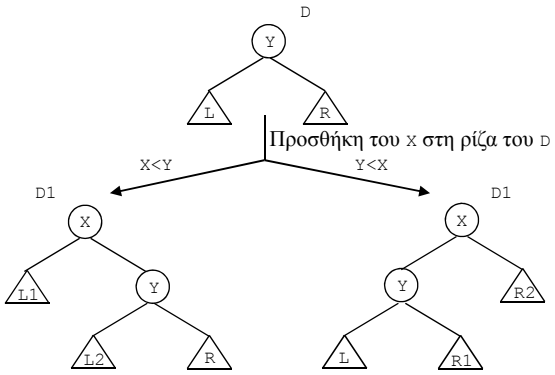
Αν κάποιο από τα `Left` ή `Right` είναι `nil`, τότε το άλλο απλώς μπορεί να συνδεθεί στο `A`. Αν όχι:




```
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

```
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)

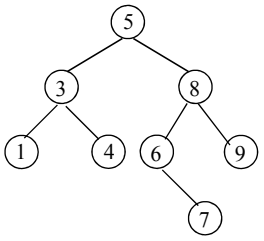
Για να εισαχθεί ένας κόμβος X σε ένα δυαδικό λεξικό D αρκεί είτε να εισαχθεί στη ρίζα του D, είτε στο αριστερό υποδέντρο του D (αν η ρίζα του D είναι μεγαλύτερη από το X), είτε στο δεξιό υποδέντρο του D (αν η ρίζα του D είναι μικρότερη από το X).

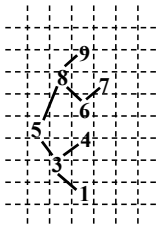
```
add(Tree, X, NewTree) :-
   addroot(Tree, X, NewTree).
add(t(L, Y, R), X, t(L1, Y, R)) :-
   gt(Y, X),
   add(L, X, L1).
add(t(L, Y, R), X, t(L, Y, R1)) :-
   gt(X, Y),
   add(R, X, R1).
```

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).

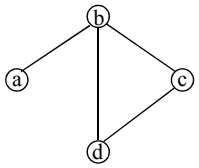
show2(nil, _).
show2(t(Left, X, Right), Indent) :-
   Ind2 is Indent+2,
   show2(Right, Ind2),
   spaces(Indent),
   write(X),
   nl,
   show2(Left, Ind2).

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

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



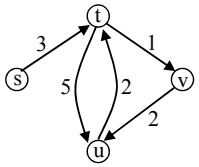

```
connected(a, b).
connected(b, c).
.........
```

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)



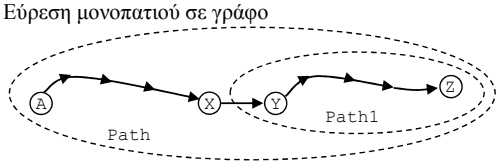

```
arc(s, t, 3).
arc(t, v, 1).
arc(u, t, 2).
.........
```

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

```
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 ελαχίστου κόστους από `node1` σε `node2`

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

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

- `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).

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)

```
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, _)).
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(.........
```

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

Το πρόβλημα της ζέβρας

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

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.
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)

```
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]).

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).
```



```
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).
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).
```

```
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).
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).
```



```
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).

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).
       .............   ΑΚΟΜΑ ΠΕΡΙΜΕΝΩ
```

2ος τρόπος (δοκίμασε και γέννησε)

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



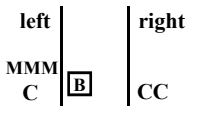
```
    generate_people([_, P, A, D, C, H)|More], PList,
                    AList, DList, CList, HList) :-
       delete(P, PList, PRem),
       delete(A, AList, ARem),
       delete(D, DList, DRem),
       delete(C, CList, CRem),
       delete(H, HList, HRem),
       generate_people(More, PRem, ARem, DRem, CRem, HRem).
    generate_people([], [], [], [], [], []).
    delete(..........
    test(............
    rule1(...........          rule2(...........
    rule3(...........          rule4(...........
    rule5(...........          rule6(...........
    rule7(...........          rule8(...........
    rule9(...........          rule10(..........
    rule11(..........          rule12(..........
    rule13(..........          rule14(..........
    nextto(..........
    rightof(.........
    member(..........

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

Το πρόβλημα των ιεραποστόλων και κανιβάλων

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.

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



`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


```
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).

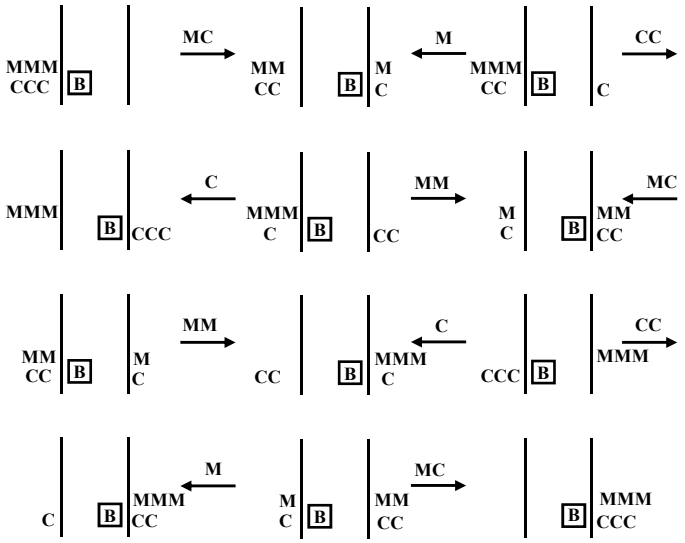
legal_move(state(MB1, CB1, B1), state(MB2, CB2, B2)) :-
   opposite(B1, B2),
   travel(MT, CT),
   MT =< MB1,
   CT =< CB1,
   MB2 is 3-MB1+MT,
   CB2 is 3-CB1+CT,
   ((MB2 =\= 0, MB2 >= CB2) ; MB2 =:= 0),
   ((MB2 =\= 3, MB2 =< CB2) ; MB2 =:= 3) .

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

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

```
?- 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



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
```

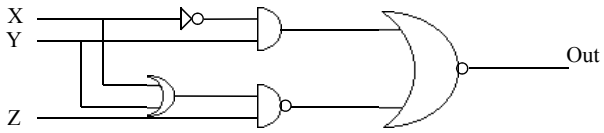
Συμβολική παραγώγιση

```
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).
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

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




```
?- 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 = ..........
```

```
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).

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).
```



```
:- 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).
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).
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)).
```

```
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)).

?- 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
?- 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)

```
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).

member(............
append(............

initial_state(.....
final_state(.......
move(..............
```

} Ορισμοί εξαρτώμενοι από το πρόβλημα

Το πρόβλημα του αγρότη, του λύκου,
της κατσίκας και του λάχανου

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);

```
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)).
```



```
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)]  -> ;
   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

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.

```
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.
```



```
?- depth_first_search(States).
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)]   -> ;
          ................................      -> ;
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)]               -> ;
          ................................      -> ;
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
```

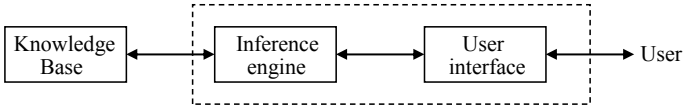
26 λύσεις

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

ΕΜΠΕΙΡΑ ΣΥΣΤΗΜΑΤΑ (EXPERT SYSTEMS)
ΚΑΙ ΛΟΓΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

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



Κέλυφος εμπείρων συστημάτων
(Expert system shell)

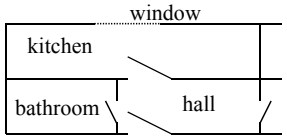
- Αναπαράσταση γνώσης στα έμπειρα συστήματα μέσω if-then κανόνων ή κανόνων παραγωγής (production rules)

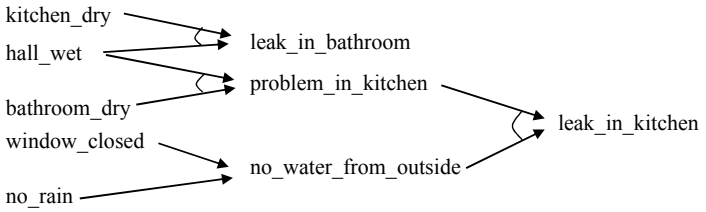
  if precondition P then conclusion C

- 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

```
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
```

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

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

```
:- 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).
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

Εισαγωγή ποσοτικού μέτρου βεβαιότητας σε γεγονότα

και if-then κανόνες: $0 \leq \text{Certainty} \leq 1$

```
given(hall_wet, 1).
given(bathroom_dry, 1).
given(kitchen_dry, 0).
given(no_rain, 0.8).
given(window_closed, 0).

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
```

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

```
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).

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
```

ΘΕΩΡΙΑ ΛΟΓΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Το θεωρητικό υπόβαθρο του λογικού προγραμματισμού είναι η λογική πρώτης τάξης (first order logic).

Στη λογική πρώτης τάξης ορίζουμε τα εξής σύνολα:

P: σύνολο κατηγορημάτων (predicates)

F: σύνολο συναρτησιακών συμβόλων (function symbols)

V: σύνολο μεταβλητών (variables)

Σε κάθε κατηγόρημα ή συναρτησιακό σύμβολο k αντιστοιχεί ένας ακέραιος αριθμός n>=0 που λέγεται βαθμός ή τάξη (arity) του k. Το k συμβολίζεται και σαν k/n.

Τα συναρτησιακά σύμβολα τάξης 0 αναφέρονται και σαν σταθερές (constants).

Ορισμός όρου (term)

Ένας όρος ορίζεται ως εξής:

1. Αν $x \in V$, το x είναι όρος
2. Αν $c/0 \in F$, το c είναι όρος
3. Αν $f/n \in F$ και $t_1, t_2, \ldots, t_n$ είναι όροι,
   το $f(t_1, t_2, \ldots, t_n)$ είναι όρος

Ορισμός ατομικού τύπου (atomic formula) ή ατόμου (atom)

Ένα άτομο ορίζεται ως εξής:

1. Αν $p/0 \in P$, το p είναι άτομο
2. Αν $p/n \in P$ και $t_1, t_2, \ldots, t_n$ είναι όροι,
   το $p(t_1, t_2, \ldots, t_n)$ είναι άτομο

Στη λογική πρώτης τάξης, τα συνδετικά (connectives) είναι εργαλεία δόμησης πιο σύνθετων τύπων.

$\neg$: άρνηση (negation)

$\wedge$: σύζευξη (conjunction)

$\vee$: διάζευξη (disjunction)

$\rightarrow$: συνεπαγωγή (implication)

$\leftrightarrow$: ισοδυναμία (equivalence)

$\forall$: καθολικός ποσοδείκτης (universal quantifier)

$\exists$: υπαρξιακός ποσοδείκτης (existential quantifier)

Ορισμός καλοσχηματισμένου τύπου (well-formed formula) ή τύπου

1. Αν το A είναι άτομο, τότε το A είναι τύπος.
2. Αν τα F και G είναι τύποι, τότε και τα
   $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$ είναι τύποι.
3. Αν το F είναι τύπος και $x \in V$, τότε τα $(\forall x\ F)$ και $(\exists x\ F)$ είναι τύποι.

- Μερικές φορές, είναι βολικό το $F \rightarrow G$ να γράφεται $G \leftarrow F$
- Παραδείγματα τύπων:
  Αν $x, y \in V$, $p/2, q/1, r/1 \in P$, $a/0, f/1, g/1 \in F$ τα
  $(\forall x\ (\exists y\ (p(x, y) \rightarrow q(x))))$
  $(\neg(\exists x\ (p(x, a) \wedge q(f(x)))))$
  $(\forall x\ (p(x, g(x)) \leftarrow (q(x) \wedge (\neg r(x)))))$ είναι τύποι
- Μία υπονοούμενη ιεραρχία $/\neg, \forall, \exists / \vee / \wedge / \rightarrow, \leftrightarrow /$ μεταξύ των συνδετικών απλοποιεί συντακτικά τους τύπους:
  $\forall x\ \exists y\ (p(x, y) \rightarrow q(x))$
  $\neg \exists x\ (p(x, a) \wedge q(f(x)))$
  $\forall x\ (p(x, g(x)) \leftarrow q(x) \wedge \neg r(x))$

- Δεδομένων συνόλων P, F και V, το σύνολο όλων των συντακτικά αποδεκτών τύπων αποτελεί μία γλώσσα πρώτης τάξης (first order language).
- Η εμβέλεια (scope) του $\forall$ στον τύπο $\forall x\ F$ ή του $\exists$ στον τύπο $\exists x\ F$ είναι ο τύπος F.
- Μια δεσμευμένη εμφάνιση (bound occurrence) μεταβλητής σε έναν τύπο είναι μια εμφάνιση αμέσως μετά από έναν ποσοδείκτη ή μέσα στην εμβέλεια ενός ποσοδείκτη που "ποσοτικοποιεί" αυτήν τη μεταβλητή. Οποιαδήποτε άλλη εμφάνιση μεταβλητής λέγεται ελεύθερη εμφάνιση (free occurrence).

  Παράδειγμα:

  Η τρίτη εμφάνιση της μεταβλητής x στον τύπο $\exists x\ \ p(x, y) \wedge q(x)$ είναι ελεύθερη, ενώ στον τύπο $\exists x\ (p(x, y) \wedge q(x))$ είναι δεσμευμένη.
- Ένας τύπος λέγεται κλειστός τύπος (closed formula) όταν δεν περιέχει ελεύθερες εμφανίσεις μεταβλητών.

  Παράδειγμα:

  Ο τύπος $\forall y\ \exists x\ (p(x, y) \wedge q(x))$ είναι κλειστός, αλλά ο τύπος $\exists x\ \ (p(x, y) \wedge q(x))$ δεν είναι (γιατί;)
- Ένα λεκτικό (literal) είναι ένα άτομο ή η άρνηση ενός ατόμου.
- Μία πολύ σημαντική κατηγορία τύπων είναι οι προτάσεις (clauses) που είναι τύποι της μορφής:

  $\forall x1\ \forall x2\ \ldots.\forall xs\ \ (L1 \vee L2 \vee \ldots \vee Lm)$

  όπου τα Li είναι λεκτικά και τα xj είναι όλες οι μεταβλητές που εμφανίζονται στο $L1 \vee L2 \vee \ldots \vee Lm$.

Περί προτάσεων

- Παραδείγματα:

  $\forall x \forall y \forall z\ \ (p(x, z) \vee \neg q(x, y) \vee \neg r(y, z))$

  $\forall x \forall y\ \ (\neg p(x, y) \vee r(f(x, y), a))$

  $\forall x \forall y\ \ (p(x, y) \vee q(x, y) \vee \neg r(a, x))$

  $\forall x\ \ (\neg p(x, x) \vee \neg q(x, x))$
- Εναλλακτική γραφή:

  Η πρόταση $\forall x1\ \ldots \forall xs\ \ (A1 \vee \ldots \vee Ak \vee \neg B1 \vee \ldots \vee \neg Bn)$.

  όπου τα Ai και Bj είναι άτομα, γράφεται και σαν:

  $A1\ ,\ \ldots..\ ,\ Ak\ \leftarrow B1\ ,\ \ldots\ldots\ ,\ Bn$

  Το αριστερό μέλος του $\leftarrow$ ονομάζεται κεφαλή (head) της πρότασης, ενώ το δεξιό μέλος ονομάζεται σώμα (body)
- Οι προτάσεις με $k = 1$ $(A \leftarrow B1, \ldots, Bn)$ λέγονται οριστικές (definite) προτάσεις.
- Οι οριστικές προτάσεις με $n = 0$ $(A \leftarrow)$ λέγονται μοναδιαίες (unit) προτάσεις.
- Ένα σύνολο οριστικών προτάσεων είναι ένα οριστικό πρόγραμμα.
- Οι προτάσεις με $k = 0$ $(\leftarrow B1, \ldots, Bn)$ λέγονται οριστικοί στόχοι (definite goals).
- Για $k = 0$ και $n = 0$, έχουμε την κενή (empty) πρόταση ($\leftarrow$ ή $\square$), που αναφέρεται και σαν αντίφαση (contradiction).
- Μια πρόταση Horn είναι είτε μία οριστική πρόταση, είτε ένας οριστικός στόχος (μας θυμίζει την Prolog;).

Σύνταξη και σημασιολογία (syntax and semantics)

Ό,τι έχει αναφερθεί μέχρι στιγμής στα θεωρητικά θέματα σχετίζεται μόνο με τη σύνταξη στη λογική πρώτης τάξης. Αυτό που ενδιαφέρει όμως στο λογικό προγραμματισμό είναι και η σημασιολογία, δηλαδή η μελέτη της σημασίας των λογικών προγραμμάτων. Συνήθως, τρεις μέθοδοι εφαρμόζονται:

- Μοντελοθεωρητική σημασιολογία (Model-theoretic semantics)
- Σημασιολογία σταθερού σημείου (Fixpoint semantics)
- Λειτουργική σημασιολογία (Operational semantics)

Με βάση αυτές τις μεθόδους, θα μελετηθεί η σημασία των οριστικών προγραμμάτων. Οι τρεις μέθοδοι καταλήγουν σε ισοδύναμα συμπεράσματα.

Μερικοί ορισμοί:

- Ένας όρος ή τύπος που δεν περιέχει μεταβλητές λέγεται πλήρως αποτιμημένος (ground).
- Αν L είναι μία γλώσσα πρώτης τάξης, το σύνολο των πλήρως αποτιμημένων όρων που κατασκευάζονται χρησιμοποιώντας τα συναρτησιακά σύμβολα της L (συμπεριλαμβανομένων και των σταθερών) ονομάζεται σύμπαν Herbrand (Herbrand universe) U_L. Αν δεν υπάρχει σταθερά στη γλώσσα, εισάγεται μία αυθαίρετη, για να υπάρχει μη κενό U_L.
- Για δεδομένη γλώσσα L, το σύνολο των πλήρως αποτιμημένων ατομικών τύπων (ατόμων) αποτελεί τη βάση Herbrand (Herbrand base) B_L. Δηλαδή, το B_L περιέχει όλα τα δυνατά άτομα με κατηγορήματα της L και ορίσματα από το U_L.

Μοντελοθεωρητική σημασιολογία

- Λαμβάνοντας υπόψη τις έννοιες του σύμπαντος Herbrand U_L και της βάσης Herbrand B_L για μία γλώσσα πρώτης τάξης L, ένα υποσύνολο I του B_L ονομάζεται ερμηνεία Herbrand (Herbrand interpretation), ή, στο εξής, απλά ερμηνεία.
- Ουσιαστικά, μία ερμηνεία αντιστοιχεί σε κάθε κατηγόρημα p/n της L μία σχέση επάνω στο U_L^n.

- Παράδειγμα:

  Αν $U_L = \{a, b, c\}$ και $B_L = \{p(a), p(b), p(c), q(a), q(b), q(c), r(a), r(b), r(c)\}$

  μία ερμηνεία I είναι η $I = \{p(a), p(c), q(c), r(b)\}$

- Άτυπα, μια ερμηνεία περιγράφει μία υποψήφια κατάσταση του περιβάλλοντος κόσμου, με την έννοια ότι περιέχει εκείνα τα πλήρως αποτιμημένα άτομα που αντιστοιχούν στο τι ισχύει στην κατάσταση αυτή.
- Δεδομένης μίας ερμηνείας, ένας κλειστός τύπος μπορεί να είναι αληθής ή ψευδής στην ερμηνεία αυτή. Αν ένας τύπος είναι αληθής σε μία ερμηνεία Herbrand, ή ερμηνεία αυτή είναι μοντέλο Herbrand (Herbrand model), ή, στο εξής, απλά μοντέλο του τύπου αυτού. Μία ερμηνεία είναι μοντέλο ενός συνόλου από κλειστούς τύπους, αν είναι μοντέλο για κάθε τύπο του συνόλου.

- Ένας κλειστός τύπος είναι αληθής σε μία ερμηνεία I αν:
  - είναι ένα πλήρως αποτιμημένο άτομο A και $A \in I$
  - είναι της μορφής $\neg F$ και ο τύπος F δεν είναι αληθής στην I
  - είναι της μορφής $F \wedge G$ και οι τύποι F και G είναι αληθείς στην I
  - είναι της μορφής $F \vee G$ και κάποιος από τους τύπους F ή G είναι αληθής στην I
  - είναι της μορφής $F \rightarrow G$ (ή $G \leftarrow F$) και είτε ο τύπος F δεν είναι αληθής στην I, είτε και ο F και ο G είναι αληθείς στην I.
  - είναι της μορφής $F \leftrightarrow G$ και είτε και οι δύο (F και G) είναι αληθείς στην I, είτε και οι δύο δεν είναι αληθείς στην I
  - είναι της μορφής $\forall x$ F και για όλα τα $d \in U_L$ αν αντικατασταθεί η μεταβλητή x στον τύπο F με το d, οι τύποι που προκύπτουν να είναι αληθείς στην I
  - είναι της μορφής $\exists x$ F και υπάρχει κάποιο $d \in U_L$ τέτοιο ώστε αν αντικατασταθεί η μεταβλητή x στον τύπο F με το d, ο τύπος που προκύπτει να είναι αληθής στην I
- Ένας κλειστός τύπος είναι ψευδής σε μία ερμηνεία I αν δεν είναι αληθής στην I.
- Η τιμή αλήθειας των ανοικτών τύπων μπορεί να ορισθεί μόνο σε σχέση με μία αντικατάσταση των ελευθέρων εμφανίσεων μεταβλητών τους από στοιχεία του U_L (οπότε πλέον γίνονται κλειστοί τύποι).

- Από την εφαρμογή του ορισμού απόδοσης τιμής αλήθειας σε προτάσεις προκύπτει ότι μία ερμηνεία I είναι μοντέλο για μία πρόταση, αν για κάθε δυνατή αντικατάσταση των μεταβλητών της πρότασης με στοιχεία από το U_L, η πρόταση που προκύπτει είναι αληθής στην I, δηλαδή είτε κάποιο από τα πλήρως αποτιμημένα άτομα της κεφαλής είναι αληθές στην I, είτε κάποιο από τα πλήρως αποτιμημένα άτομα του σώματος είναι ψευδές στην I.
- Αν S είναι ένα σύνολο από κλειστούς τύπους και F είναι ένας κλειστός τύπος, τότε ο F είναι λογικό επακόλουθο (logical consequence) του S (συμβολικά $S \models F$), αν κάθε μοντέλο του S είναι και μοντέλο του F. Τότε, αποδεικνύεται ότι το $S \bigcup \{\neg F\}$ δεν έχει μοντέλο.
- Αν P είναι ένα οριστικό πρόγραμμα, αποδεικνύεται ότι η τομή δύο μοντέλων του είναι επίσης μοντέλο. Η τομή Mp όλων των μοντέλων του P είναι το ελάχιστο μοντέλο Herbrand (least Herbrand model) του P.
- Αν P είναι ένα οριστικό πρόγραμμα σε μία γλώσσα πρώτης τάξης L, αποδεικνύεται ότι

  $$Mp = \{ A \in B_L \mid P \models A \}$$

  Δηλαδή, το ελάχιστο μοντέλο ενός οριστικού προγράμματος αποτελείται από όλα τα λογικά επακόλουθα του προγράμματος. Αυτό είναι το αποτέλεσμα της μοντελοθεωρητικής σημασιολογίας.

- Παράδειγμα:

  Έστω το οριστικό πρόγραμμα P:

  $p(x) \leftarrow q(x)$

  $q(x) \leftarrow r(x)$

  $p(a) \leftarrow$

  $q(b) \leftarrow$

  $r(c) \leftarrow$

  Τότε έχουμε:

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

  $B_L = \{p(a), p(b), p(c), q(a), q(b), q(c),$
  $r(a), r(b), r(c)\}$

  Το ελάχιστο μοντέλο Herbrand Mp του P είναι:

  $Mp = \{p(a), p(b), p(c), q(b), q(c), r(c)\}$

  Πόσο εύκολα μπορούμε να υπολογίσουμε το Mp;

Σημασιολογία σταθερού σημείου

- Η σημασιολογική προσέγγιση μέσω σταθερού σημείου παρέχει μία κατασκευαστική μέθοδο υπολογισμού του ελάχιστου μοντέλου Herbrand ενός οριστικού προγράμματος.
- Αν T είναι μία απεικόνιση σε ένα σύνολο L, δηλαδή $T : L \rightarrow L$, ένα στοιχείο $a \in L$ ονομάζεται σταθερό σημείο (fixpoint) της T αν ισχύει $T(a) = a$.
- Αν στο L έχει ορισθεί μία σχέση μερικής διάταξης (partial order) και με τις επιπλέον προϋποθέσεις (χωρίς να αναλύσουμε τους απαιτούμενους ορισμούς) το L να είναι πλήρες πλέγμα (complete lattice) και η T συνεχής (continuous), αποδεικνύεται ότι η T έχει ένα ελάχιστο σταθερό σημείο (least fixpoint), συμβολικά lfp(T), το οποίο ισούται με

  $lfp(T) = lub(\{\bot, T(\bot), T(T(\bot)), T(T(T(\bot))), \ldots\})$

  όπου $\bot$ είναι το ελάχιστο στοιχείο (bottom element) του L και το lub(X) είναι το ελάχιστο άνω φράγμα (least upper bound) ενός μερικώς διατεταγμένου συνόλου X.
- Δεδομένου ενός οριστικού προγράμματος P επάνω σε μία γλώσσα πρώτης τάξης L και θεωρώντας τη σχέση μερικής διάταξης $\subseteq$ στο σύνολο 2^{B_L} όλων των δυνατών ερμηνειών για το P, αποδεικνύεται ότι το 2^{B_L} είναι πλήρες πλέγμα (με $\bot = \varnothing$).

- Ορίζοντας την απεικόνιση Tp από ερμηνείες σε ερμηνείες, δηλαδή $Tp : 2^{B_L} \to 2^{B_L}$, με

  Tp (I) = { $A \in B_L$ | A ←A1 , … , An είναι πλήρως αποτιμημένο στιγμιότυπο μιας πρότασης του P και {A1 , … , An} ⊆I }

  αποδεικνύεται ότι η Tp είναι συνεχής, άρα έχει ελάχιστο σταθερό σημείο, το

  lfp(Tp) = lub ({∅, Tp (∅) , Tp (Tp (∅)) , …..})
- Τέλος, αποδεικνύεται ότι το ελάχιστο μοντέλο Herbrand Mp του P ισούται με το ελάχιστο σταθερό σημείο της Tp, δηλαδή Mp = lfp(Tp). Αυτό είναι το αποτέλεσμα της σημασιολογίας σταθερού σημείου.
- Στο παράδειγμα οριστικού προγράμματος της προηγούμενης μεθόδου σημασιολογίας έχουμε:

  - ∅
  - Tp(∅) = {p(a), q(b), r(c)}
  - Tp(Tp(∅)) = Tp({p(a), q(b), r(c)}) = = {p(a), p(b), q(b), q(c), r(c)}
  - Tp(Tp(Tp(∅))) = Tp({p(a), p(b), q(b), q(c), r(c)}) = = {p(a), p(b), p(c), q(b), q(c), r(c)}
  - Tp(Tp(Tp(Tp(∅)))) = Tp({p(a), p(b), p(c), q(b), q(c), r(c)}) = = {p(a), p(b), p(c), q(b), q(c), r(c)}
  - ……..

  Άρα: Mp = {p(a), p(b), p(c), q(b), q(c), r (c)}

Λειτουργική σημασιολογία

- Η λειτουργική σημασιολογία σχετίζεται με την παραγωγή νέας γνώσης από υπάρχουσα, μέσα από μία διαδικασία εξαγωγής συμπερασμάτων που συνίσταται σε πεπερασμένο αριθμό εφαρμογών ενός κανόνα συμπερασμού (inference rule).
- Για δεδομένη διαδικασία εξαγωγής συμπερασμάτων, αν από ένα σύνολο κλειστών τύπων S συμπεραίνεται (is inferred) ένας κλειστός τύπος F, γράφουμε S ⊢F.
- Για τα οριστικά προγράμματα, η διαδικασία εξαγωγής συμπερασμάτων που εφαρμόζεται είναι η SLD-ανάλυση (SLD-resolution).
- Για την εισαγωγή του κανόνα συμπερασμού της SLD-ανάλυσης, απαιτείται να ορισθούν οι έννοιες της αντικατάστασης, της ενοποίησης, του γενικότερου ενοποιητή, κ.λ.π.
- Αντικατάσταση (substitution)

  Μια αντικατάσταση θ είναι ένα πεπερασμένο σύνολο της μορφής {v1/t1, …., vn/tn} όπου τα vi είναι μεταβλητές και τα ti είναι όροι (κάθε ti είναι διαφορετικό από το αντίστοιχο vi και όλα τα vi είναι διαφορετικά μεταξύ τους). Ουσιαστικά, μία αντικατάσταση ορίζει ότι τα vi είναι δεσμευμένα με (έχουν πάρει σαν τιμές) τα αντίστοιχα ti. Η κενή (empty) αντικατάσταση ορίζεται από το κενό σύνολο.

- Στιγμιότυπο (instance)

  Αν E είναι ένας όρος, λεκτικό, σύζευξη λεκτικών ή διάζευξη λεκτικών, το Eθ λέγεται στιγμιότυπο του E μέσω της αντικατάστασης $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ αν στο E αντικατασταθούν ταυτόχρονα οι εμφανίσεις των μεταβλητών v_i με τα αντίστοιχα t_i .

- Παραλλαγή (variant)

  Αν E και F είναι όροι, λεκτικά, συζεύξεις λεκτικών ή διαζεύξεις λεκτικών, το E λέγεται παραλλαγή του F (ή το F παραλλαγή του E) αν υπάρχουν αντικαταστάσεις θ και σ τέτοιες ώστε $E = F\theta$ και $F = E\sigma$.

- Σύνθεση (composition) αντικαταστάσεων

  Αν $\theta = \{u_1/s_1, \ldots, u_m/s_m\}$ και $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$ είναι δύο αντικαταστάσεις, η σύνθεση τους θσ είναι η αντικατάσταση $\{u_1/s_1\sigma, \ldots, u_m/s_m\sigma, v_1/t_1, \ldots, v_n/t_n\}$ διαγράφοντας από αυτήν κάθε $u_i/s_i\sigma$ για το οποίο $u_i = s_i\sigma$ και κάθε v_j/t_j για το οποίο $v_j \in \{u_1, \ldots, u_m\}$

- Ενοποίηση (unification)

  Αν $S = \{E_1, \ldots, E_n\}$ ένα σύνολο όρων ή ατόμων, μία αντικατάσταση θ λέγεται ενοποιητής (unifier) για το S αν το $S\theta = \{E_1\theta, \ldots, E_n\theta\}$ είναι μονοσύνολο. Ένας ενοποιητής θ για το S είναι ο γενικότερος ενοποιητής (most general unifier) mgu για το S, αν για κάθε ενοποιητή σ του S, υπάρχει αντικατάσταση γ τέτοια ώστε $\sigma = \theta\gamma$. Η διαδικασία υπολογισμού του mgu ενός S λέγεται ενοποίηση.

- Κανόνας συμπερασμού της SLD-ανάλυσης

  Αν το G είναι ένας οριστικός στόχος $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ και C μια οριστική πρόταση $A \leftarrow B_1, \ldots, B_q$, τότε το $G' = \leftarrow(A_1, \ldots, A_{m-1}, B_1 \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$ παράγεται (is derived) από τα G και C, όπου θ είναι ο mgu των ατόμων A_m και A.

- Δεδομένου ενός οριστικού προγράμματος P επάνω σε μία γλώσσα πρώτης τάξης L και ενός οριστικού στόχου G, μία SLD-απόρριψη (SLD-refutation) του $P \cup \{G\}$ είναι μία διαδοχική εφαρμογή του προηγούμενου κανόνα συμπερασμού πεπερασμένο αριθμό βημάτων, όπου σε κάθε βήμα χρησιμοποιείται ο στόχος που παρήχθη στο προηγούμενο βήμα (στο πρώτο βήμα ο G) και μία παραλλαγή κάποιας πρότασης του P (τέτοια ώστε οι μεταβλητές της να είναι διαφορετικές από αυτές που έχουν χρησιμοποιηθεί μέχρι εκείνη τη στιγμή) και επιπλέον ο τελευταίος στόχος που παράγεται είναι η κενή πρόταση □. Το σύνολο όλων των $A \in B_L$ για τα οποία το $P \cup \{\leftarrow A\}$ έχει κάποια SLD-απόρριψη ονομάζεται σύνολο επιτυχίας (success set) του P, συμβολικά SS(P).

- Αποδεικνύεται ότι το σύνολο επιτυχίας ενός οριστικού προγράμματος ισούται με το ελάχιστο μοντέλο Herbrand του προγράμματος. Δηλαδή $Mp = SS(P)$.

Στο παράδειγμα του προγράμματος P που δόθηκε, ποιες είναι οι SLD-απορρίψεις του $P \cup \{\leftarrow A\}$ για κάθε $A \in Mp$;

ΛΟΓΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΜΕ ΠΕΡΙΟΡΙΣΜΟΥΣ (CONSTRAINT LOGIC PROGRAMMING)

- Ο λογικός προγραμματισμός με περιορισμούς είναι μία επέκταση του λογικού προγραμματισμού η οποία παρέχει τη δυνατότητα δηλωτικής, και ταυτόχρονα αποδοτικής, αντιμετώπισης προβλημάτων που μπορούν να διατυπωθούν σαν προβλήματα ικανοποίησης περιορισμών (constraint satisfaction problems).

- Ένα πρόβλημα είναι πρόβλημα ικανοποίησης περιορισμών όταν μπορεί να μοντελοποιηθεί από ένα σύνολο μεταβλητών και ένα σύνολο περιορισμών (constraints) μεταξύ των μεταβλητών αυτών. Οι μεταβλητές μπορούν να παίρνουν τιμές από προκαθορισμένα σύνολα δυνατών τιμών. Το ζητούμενο είναι να βρεθούν εκείνοι οι συνδυασμοί τιμών των μεταβλητών (λύσεις του προβλήματος) που ικανοποιούν όλους τους περιορισμούς. Πολλές φορές, σε κάθε υποψήφια λύση αντιστοιχεί ένα κόστος (συνάρτηση των μεταβλητών), οπότε αυτό που ενδιαφέρει είναι να βρεθεί εκείνη η λύση που έχει το μικρότερο δυνατό κόστος. Τότε το πρόβλημα είναι και πρόβλημα βελτιστοποίησης (optimization problem).

- Ο φυσιολογικός τρόπος επίλυσης ενός προβλήματος ικανοποίησης περιορισμών είναι ίσως η μέθοδος “γέννα-και-δοκίμασε” (“generate-and-test”) . Δηλαδή, “γέννα συστηματικά τους δυνατούς συνδυασμούς τιμών των μεταβλητών και, για κάθε έναν από αυτούς, έλεγξε αν ισχύουν οι περιορισμοί”. Όμως, η μέθοδος αυτή μπορεί να δώσει αποτελέσματα μόνο όταν το πρόβλημα είναι σχετικά μικρού μεγέθους (λίγες μεταβλητές και λίγες δυνατές τιμές γι’ αυτές).
- Στο λογικό προγραμματισμό με περιορισμούς, η χρησιμοποιούμενη μέθοδος είναι η “περιόρισε-και-γέννα” (“constrain-and-generate”). Δηλαδή, “διατύπωσε τους περιορισμούς, έτσι ώστε να χρησιμοποιηθούν αυτοί ενεργά για να αποκόψουν πιθανές τιμές από τις μεταβλητές, και, στη συνέχεια, ξεκίνησε μία συστηματική διαδικασία γέννησης τιμών για τις μεταβλητές, αλλά σε κάθε βήμα να ενεργοποιείς πάλι τους περιορισμούς για να φροντίσουν για άλλες δυνατές αποκοπές τιμών”.
- Υπάρχουν διάφορες δυνατότητες υποστήριξης προγραμματισμού με περιορισμούς από γλώσσες λογικού προγραμματισμού, ανάλογα με το είδος των δυνατών τιμών για τις μεταβλητές και το είδος των περιορισμών που καλύπτονται. Η γλώσσα ECL^{i}PS^e υποστηρίζει, μεταξύ άλλων, αριθμητικούς, λογικούς, συμβολικούς κ.α. περιορισμούς σε μεταβλητές πεπερασμένων πεδίων (finite domain variables).

- Η μεθοδολογία επίλυσης ενός προβλήματος ικανοποίησης περιορισμών στην ECL^{i}PS^e συνίσταται σε:
  - Ορισμό των μεταβλητών που χρειάζονται για το πρόβλημα, καθώς και των πεδίων τους.
  - Διατύπωση των περιορισμών που μοντελοποιούν το πρόβλημα.
  - Μη-ντετερμινιστική γέννηση τιμών για τις μεταβλητές για την εύρεση μίας, όλων ή της βέλτιστης λύσης.

Το πρόβλημα των N βασιλισσών (ECL^{i}PS^e)

```
nqueens(N, Solution) :-
   length(Solution, N),
   Solution :: 1..N,
   constrain(Solution),
   generate(Solution).
constrain([]).
constrain([Column|Columns]) :-
   noattack(Column, Columns, 1),
   constrain(Columns).
noattack(_, [], _).
noattack(Column1, [Column2|Columns], Offset) :-
   Column1 ## Column2,
   Column1 ## Column2+Offset,
   Column1 ## Column2-Offset,
   NewOffset is Offset+1,
   noattack(Column1, Columns, NewOffset).
```

Ορισμός μεταβλητών πεδίων

##: ≠

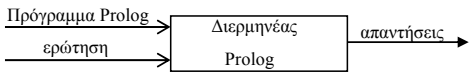

```
generate([]).
generate([Column|Columns]) :-
   indomain(Column),
   generate(Columns).
```

- Η ιδέα του προγραμματισμού με περιορισμούς, αν και "γεννήθηκε" στο περιβάλλον του λογικού προγραμματισμού, έχει ενσωματωθεί πλέον και σε άλλες προγραμματιστικές φιλοσοφίες, όπως τον αντικειμενοστραφή προγραμματισμό (ILOG Solver: C++ βιβλιοθήκη για προγραμματισμό με περιορισμούς).

- Οι περιοχές εφαρμογής του προγραμματισμού με περιορισμούς είναι πάρα πολλές, κυρίως σε προβλήματα συνδυαστικής αναζήτησης (combinational search) τα οποία έχουν αντιμετωπισθεί στο παρελθόν με μεθόδους επιχειρησιακής έρευνας (operations research), όπως:
  - Κατασκευή ωρολογίων προγραμμάτων
  - Προγραμματισμός προσωπικού
  - Σχεδίαση παραγωγής
  - Διανομή αγαθών
  - ……

ΥΛΟΠΟΙΗΣΗ ΣΥΣΤΗΜΑΤΩΝ ΛΟΓΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

- Οι προσεγγίσεις για την υλοποίηση ενός συστήματος λογικού προγραμματισμού, ουσιαστικά κάποιας γλώσσας Prolog, είναι οι συνήθεις δύο, είτε μέσω ενός διερμηνέα (interpreter) είτε μέσω ενός μεταγλωττιστή (compiler).
- Ένας διερμηνέας Prolog είναι ένα πρόγραμμα που δέχεται στην είσοδο του ένα πρόγραμμα Prolog και μία ερώτηση που απευθύνεται σ' αυτό και υπολογίζει, εφαρμόζοντας την ανάλυση και την ενοποίηση, τις ζητούμενες απαντήσεις στην ερώτηση. Σχηματικά:



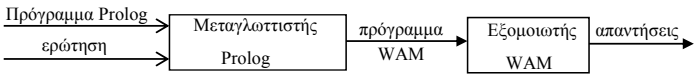
- Ο διερμηνέας κωδικοποιεί το πρόγραμμα σαν μία δομή δεδομένων στην περιοχή προγράμματος (code area) και χρησιμοποιεί μία στοίβα (stack) για να διαχειριστεί το δέντρο ανάλυσης που κατασκευάζεται κατά την προσπάθεια απάντησης της ερώτησης. Στη στοίβα φυλάσσονται ό,τι πληροφορίες απαιτούνται για την υποστήριξη, εκτός των άλλων, και της δυνατότητας οπισθοδρόμησης.
- Σε κάθε κόμβο του δέντρου ανάλυσης, που αντιστοιχεί σε μία εγγραφή της στοίβας, κωδικοποιείται η ενοποίηση ενός στόχου με την κεφαλή μίας πρότασης και δημιουργείται ένα περιβάλλον (environment), στο οποίο για κάθε μεταβλητή της επιλεγμένης πρότασης υπάρχει διαθέσιμος χώρος για να καταχωρηθεί η διεύθυνση της τιμής της.

- Επειδή μία μεταβλητή κάποιου περιβάλλοντος μπορεί να πάρει τιμή σε μεταγενέστερο κόμβο από αυτόν της δημιουργίας της, πρέπει να υπάρχει η δυνατότητα αναίρεσης της απόδοσης της τιμής αν γίνει οπισθοδρόμηση πριν το σημείο της απόδοσης. Αυτό επιτυγχάνεται μέσω των ιχνών (trails) που είτε καταχωρούνται μέσα σε περιβάλλοντα είτε συνιστούν χωριστή στοίβα.
- Τέλος, όσον αφορά την τιμή μίας μεταβλητής, αυτή μπορεί να παρασταθεί είτε από κάποιο δείκτη σε μία δομή στην περιοχή προγράμματος και ένα περιβάλλον από το οποίο θα βρεθούν οι τιμές των μεταβλητών της δομής είτε από ένα δείκτη σε μία περιοχή που λέγεται σωρός (heap) και στην οποία έχει κτισθεί η τιμή της μεταβλητής. Η πρώτη προσέγγιση χαρακτηρίζεται σαν "structure sharing" και η δεύτερη σαν "structure copying".

Ποια γλώσσα προγραμματισμού θα επιλέγατε (Prolog ή C) για να υλοποιήσετε ένα διερμηνέα της Prolog; Αν το κάνετε, τι απάντηση θα δώσει στην ερώτηση `?- a(X).` αν το πρόγραμμα που "γνωρίζει" είναι το παρακάτω;

```
a(X) :- b(X), c(X), d(X).
b(X) :- e(X), f(X).
b(X) :- e(X).
c(2).    c(3).    e(1).    e(2).    e(3).
d(1).
d(X) :- g(X).
f(1).    f(3).    g(2).
```

- Ένας μεταγλωττιστής Prolog είναι ένα πρόγραμμα που δέχεται στην είσοδο του ένα πρόγραμμα Prolog το οποίο το μεταφράζει σε μία ειδική γλώσσα χαμηλού επιπέδου (τύπου "assembly"), ονόματι WAM, της λεγόμενης αφηρημένης μηχανής του Warren (Warren Abstract Machine). Μια ερώτηση που απευθύνεται στο πρόγραμμα Prolog δεν είναι δύσκολο να ενσωματωθεί μέσα σ' αυτό, με την έννοια ότι μπορεί να θεωρηθεί ότι συνιστά το σώμα ενός κανόνα που έχει προκαθορισμένη κεφαλή (π.χ. `main`) και το ζητούμενο είναι να απαντηθεί η ερώτηση που αντιστοιχεί στην κεφαλή (`?- main`). Το πρόγραμμα WAM που προκύπτει από τη μετάφραση του προγράμματος Prolog και της αρχικής ερώτησης δίνεται στην είσοδο ενός άλλου προγράμματος, που λέγεται εξομοιωτής (emulator) WAM, το οποίο μετά την απαραίτητη "εκτέλεση" υπολογίζει τις ζητούμενες απαντήσεις. Σχηματικά:



- Η μηχανή WAM μπορεί να περιγραφεί από την αρχιτεκτονική της και το σύνολο εντολών (instruction set) της. Οι εντολές διαχειρίζονται τη μνήμη της μηχανής και τους καταχωρητές (registers) της. Η μνήμη είναι διαμερισμένη σε χώρους αντίστοιχους με αυτούς που συναντάμε στους διερμηνείς (περιοχή προγράμματος, στοίβα, σωρός, ίχνη), αν και με σημαντικά διαφορετικό τρόπο καταχώρησης των σχετικών πληροφοριών.

0	STR	1
1	h / 2	
2	REF	2
3	REF	3
4	STR	5
5	f / 1	
6	REF	3
7	STR	8
8	g / 3	
9	REF	2
10	STR	1
11	STR	5

Αναπαράσταση στο σωρό του όρου:

```
g(Z, h(Z, W), f(W))
```

Αναπαράσταση στην περιοχή προγράμματος των προτάσεων:

```
p(a, X).
p(X, b).
p(X, Y) :- p(X, a), p(b, Y).
```

```
p/2: try_me_else L1          %  p
     get_structure a/0, A1   %    (a,
     get_variable X3, A2     %        X)
     proceed                 %
L1 : retry_me_else L2        %  p
     get_variable X3, A1     %    (X,
     get_structure b/0, A2   %        b)
     proceed                 %
L2 : trust_me                %
     allocate                %  p
     get_variable X3, A1     %    (X,
     get_variable Y1, A2     %        Y) :-
     put_value X3, A1        %              p(X,
     put_structure a/0, A2   %                   a
     call p/2                %                    ),
     put_structure b/0, A1   %              p(b,
     put_value Y1, A2        %                   Y
     call p/2                %                    )
     deallocate              %                    .
```

ΠΑΡΑΛΛΗΛΟΣ ΛΟΓΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ
(PARALLEL LOGIC PROGRAMMING)

- Ο όρος "παράλληλος λογικός προγραμματισμός" αναφέρεται στην προσπάθεια εκμετάλλευσης των παράλληλων υπολογιστών μέσα από γλώσσες λογικού προγραμματισμού, με σκοπό την αύξηση της απόδοσης των λογικών προγραμμάτων.
- Έχουν ορισθεί, υλοποιηθεί και χρησιμοποιηθεί πολλές γλώσσες παράλληλου λογικού προγραμματισμού, άλλες λιγότερο άλλες περισσότερο επιτυχημένες, οι πιο πολλές από τις οποίες κατατάσσονται σε μία από τις εξής δύο κατηγορίες:

  - Γλώσσες δεσμευμένης επιλογής (Committed choice languages)

    Οι γλώσσες αυτές είναι τύπου Prolog αλλά σημασιολογικά πολύ διαφορετικές από την Prolog. Δεν υποστηρίζουν μη-ντετερμινισμό, με την έννοια ότι ένας στόχος μπορεί να ικανοποιηθεί το πολύ με έναν τρόπο. Στις γλώσσες αυτές, οι στόχοι του σώματος ενός κανόνα αντιπροσωπεύουν διεργασίες που μπορούν να εκτελούνται παράλληλα και οι οποίες χρησιμοποιούν τις κοινές τους μεταβλητές σαν κανάλια επικοινωνίας. Χαρακτηριστικές γλώσσες στην κατηγορία αυτή είναι η Concurrent Prolog, η Parlog, η Strand κ.λ.π.

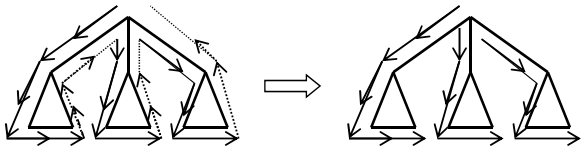
  - Γλώσσες καθαρής παραλληλίας (Pure parallelism languages)

    Οι γλώσσες αυτές αποσκοπούν σε μία πραγματικά καθαρή επέκταση της Prolog προς την κατεύθυνση της εκμετάλλευσης της παραλληλίας που, κατά κάποιο τρόπο, είναι έμφυτη στο λογικό προγραμματισμό.

- Ποια είναι η έμφυτη παραλληλία στο λογικό προγραμματισμό;

  - Ή-παραλληλία (OR-parallelism)

    Όταν ένας στόχος μπορεί να ικανοποιηθεί από περισσότερες της μίας προτάσεις, επειδή είναι ενοποιήσιμος με τις κεφαλές τους, αντί να δοκιμάζονται αυτές οι προτάσεις η μία μετά την άλλη μέσω οπισθοδρόμησης, θα ήταν δυνατόν να δοκιμασθούν παράλληλα, εφόσον υπάρχουν διαθέσιμοι υπολογιστικοί πόροι (επεξεργαστές).



  - ΚΑΙ-παραλληλία (AND-parallelism)

    Στην κλασική Prolog, οι στόχοι του σώματος ενός κανόνα που ενεργοποιήθηκε ικανοποιούνται από αριστερά προς τα δεξιά. Θα ήταν δυνατόν όμως να επιχειρηθεί οι στόχοι αυτοί να ικανοποιηθούν παράλληλα, εφόσον υπάρχουν διαθέσιμοι υπολογιστικοί πόροι, εξασφαλίζοντας όμως ότι οι κοινές τους μεταβλητές θα έχουν τελικά ίδιες τιμές.

- Διάφορες γλώσσες λογικού προγραμματισμού υποστηρίζουν μορφές Ή-παραλληλίας ή ΚΑΙ-παραλληλίας ή συνδυασμό των δύο. Η γλώσσα ECL^{i}PS^e υποστηρίζει Ή-παραλληλία και μία μορφή ΚΑΙ-παραλληλίας, την ανεξάρτητη ΚΑΙ-παραλληλία (independent AND-parallelism), μέσω της έννοιας των εργατών (workers) που αντιπροσωπεύουν υπολογιστικές μονάδες (όχι, κατ' ανάγκη, επεξεργαστές)

- ΚΑΙ-παραλληλία στην ECL^{i}PS^e

  Η χρήση του τελεστή `&` μεταξύ δύο στόχων υπονοεί την πρόθεση του προγραμματιστή να ικανοποιηθούν οι στόχοι αυτοί παράλληλα.

  Παράδειγμα:

```
compute(Z) :- (expensive_computation_1(X) &
               expensive_computation_2(Y)),
               combine_results(X, Y, Z).
expensive_computation_1(X) :- .........
expensive_computation_2(Y) :- .........
combine_results(X, Y, Z) :- ...........
```

  Παράδειγμα:

```
q(a).      q(b).      q(c).
r(c).      r(d).      r(e).
p1(X) :- q(X), r(X).
p2(X) :- q(X) & r(X).
```

  Σε τι διαφέρουν τα `p1/1` και `p2/1`;

  Παράδειγμα:

  Θυμηθείτε την `quicksort/2`. Δεν θα ήταν χρήσιμο οι δύο αναδρομικοί στόχοι στο σώμα της βασική πρότασης να ικανοποιηθούν παράλληλα;

- Ή-παραλληλία στην ECL^{i}PS^e

  Η χρήση της οδηγίας `parallel` για ένα κατηγόρημα υπονοεί την πρόθεση του προγραμματιστή να εξετασθούν οι προτάσεις / ορισμοί για το κατηγόρημα αυτό παράλληλα, αντί μέσω οπισθοδρόμησης, στην περίπτωση απόπειρας ικανοποίησης στόχου με το κατηγόρημα αυτό.

Παράδειγμα:

```
s(X) :- u(X), p(X), v(X).
u(X) :- .........      p1(X) :- .........
v(X) :- .........      p2(X) :- .........
:- parallel p/1.
p(X) :- p1(X).
p(X) :- p2(X).
```

Το κατηγόρημα `par_member/2` είναι ορισμένο όπως και το γνωστό `member/2`, αλλά είναι δηλωμένο και `parallel`.

Παράδειγμα:

```
process_value(L, Y) :-
   par_member(X, L),
   process(X, Y).
process(X, Y) :- .........
process_all(L1, L2) :-
   findall(Y, process_value(L1, Y), L2).
```

Το κατηγόρημα `par_indomain/1` έχει την ίδια σημασιολογία με το `indomain/1`, αλλά αντί να δίνει στη μεταβλητή πεδίου όλες τις δυνατές τιμές της μέσω οπισθοδρόμησης, προκαλεί την παράλληλη εξέταση όλων των διαφορετικών κλάδων στο δέντρο ανάλυσης που είναι συνυφασμένοι με τις τιμές αυτές.

Παράδειγμα:

Δοκιμάστε να αντικαταστήσετε στο πρόγραμμα που επιλύει το πρόβλημα των Ν βασιλισσών με περιορισμούς το `indomain/1` σε `par_indomain/1` και ελέγξτε την απόδοση.

ΜΕΡΟΣ Β΄

Συγγραφή: ΙΖΑΜΠΩ ΚΑΡΑΛΗ

ΑΝΑΠΑΡΑΣΤΑΣΗ ΓΝΩΣΗΣ ΚΑΙ ΣΥΛΛΟΓΙΣΤΙΚΗ

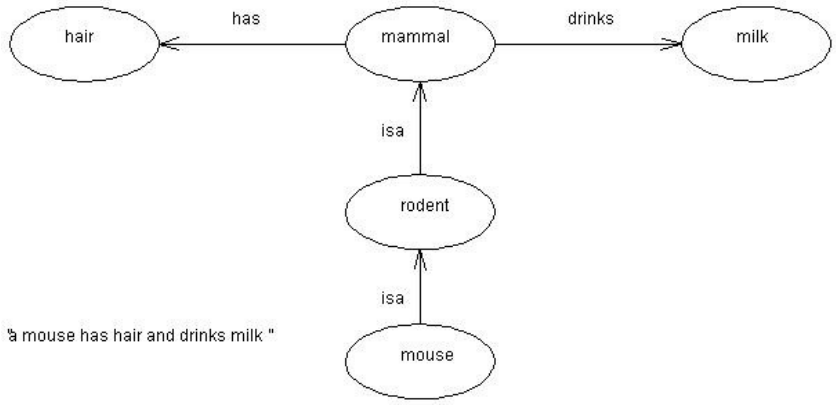
- Στην πράξη, πολλά προβλήματα εμπλέκουν ένα σύνολο από αντικείμενα τα οποία συσχετίζονται και αλληλεπιδρούν μεταξύ τους.
- Ακόμα και τα ίδια τα αντικείμενα μπορεί να εμπεριέχουν πληθώρα χαρακτηριστικών/ιδιοτήτων ή οι αλληλεπιδράσεις που έχουν μεταξύ τους να είναι πολύπλοκες.
- Για μεγάλο όγκο πληροφορίας, γίνεται αναγκαία μια πιο δομημένη οργάνωση της γνώσης.

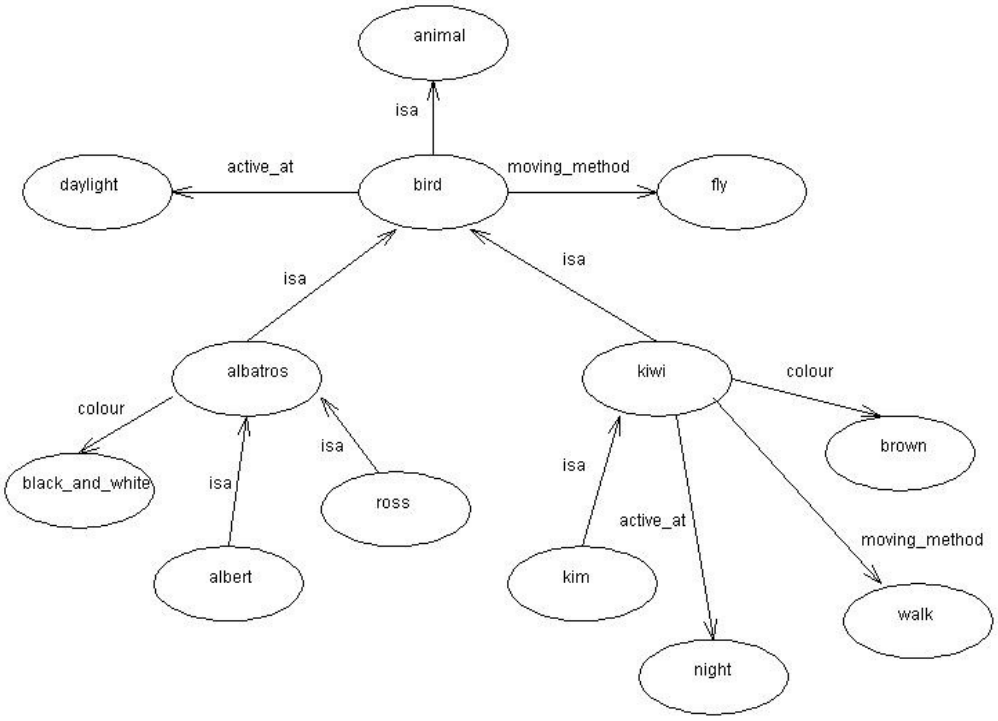
Αναπαράσταση Γνώσης μέσω Σημασιολογικών Δικτύων

- Ένα δίκτυο επιτρέπει να γίνεται εμφανής η οργάνωση της γνώσης.
- Κομμάτια γνώσης που σχετίζονται μεταξύ τους συγκεντρώνονται σε συστάδες μέσα στο δίκτυο.
- Η αναπαράσταση γνώσης γίνεται μέσα από μια εικόνα.
- Semantic Networks (Associative Networks): αναδείχθηκαν από τον Quillian (1968) για την απεικόνιση της σημασίας των προτάσεων και λέξεων της αγγλικής γλώσσας.

- Γράφος με κόμβους
  - οντότητες (entities) και
  - ιδιότητες οντοτήτων.
- Οι ακμές παριστάνουν σχέσεις (relations).
- Δεν υπάρχει καθολικά αποδεκτή σύνταξη.
- Προσπάθειες για την καθιέρωση προτύπων, π.χ. Sowa (1974): Conceptual Graphs.
- Έχουν υιοθετηθεί διάφορα σύνολα ιδιοτήτων από διάφορες κοινότητες χρηστών, π.χ. Γλωσσολογία.

- Χρησιμοποιούνται ευρέως για να αναπαραστήσουν ιεραρχίες οντοτήτων (ή ταξονομίες).
- Μια ειδική σχέση είναι η "isa" που συνδέει ειδικότερες με γενικότερες κλάσεις αντικειμένων καθώς και συγκεκριμένα αντικείμενα/στιγμιότυπα (instances) με κλάσεις αντικειμένων.
- Κληρονομικότητα (inheritance)
- Σχέση «ανήκειν» και υποσυνόλου για σύνολα: Οι κόμβοι που αντιστοιχούν σε οντότητες που «ανήκουν» σε οντότητες άλλων κόμβων ή οι οντότητες που είναι υποσύνολα οντοτήτων άλλων κόμβων κληρονομούν ιδιότητες από προγονικούς κόμβους.
- Εκτός κι αν έχει διευκρινιστεί το αντίθετο, μια οντότητα κληρονομεί χαρακτηριστικά και ιδιότητες από προγονικές οντότητες.
- Εύκολη υλοποίηση σε prolog.


hair
has
mammal
drinks
milk
isa
rodent
isa
"a mouse has hair and drinks milk "
mouse


animal
isa
daylight
active_at
bird
moving_method
fly
isa
isa
albatros
kiwi
colour
colour
brown
black_and_white
isa
isa
ross
isa
active_at
moving_method
albert
kim
walk
night

Υλοποίηση σε Prolog

```
isa(bird,animal).
isa(albatross,bird).
isa(albert,albatross).
isa(ross,albatross).
isa(kiwi,bird).
isa(kim,kiwi).
```

```
active_at(bird,daylight).
active_at(kiwi,night).
moving_method(bird,fly).
moving_method(kiwi,walk).
colour(albatross,
          black_and_white).
colour(kiwi,brown).
```


Κληρονόμηση ιδιοτήτων σε σημασιολογικά δίκτυα: στοιχειώδης προσέγγιση

- Ειδικός ορισμός για κάθε ιδιότητα:

- ```
  moving_method(X,Method) :-
        isa(X,SuperX),
        moving_method(SuperX,Method).
  ```

- `colour(….`
- `active_at(….`
- `….`
  ```

Κληρονόμηση ιδιοτήτων σε σημασιολογικά δίκτυα: γενικευμένη προσέγγιση

- Κώδικας:

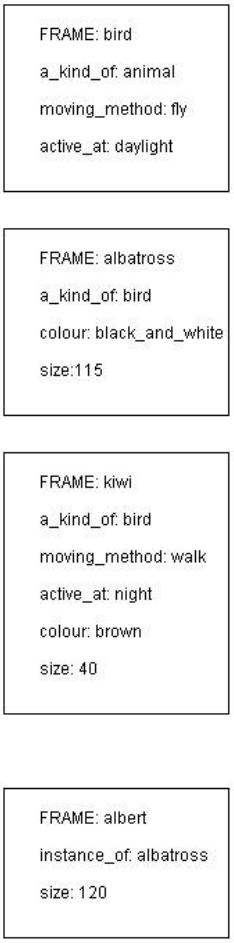
```
fact(Fact):-
     call(Fact),!.
fact(Fact):-
     Fact =.. [Rel,Arg1,Arg2],
     isa(Arg1,SuperArg),
     SuperFact =.. [Rel,SuperArg,Arg2],
     fact(SuperFact ).
```

- Παραδείγματα ερωτημάτων:

```
?- fact(moving_method(kim,Method)).
   Method = walk
?- fact(moving_method(albert,Method)).
   Method = fly
```


Πλαίσια

- Συγγενής μεθοδολογία με αυτή των σημασιολογικών δικτύων.
- Εισήχθησαν από τον Minsky (1974) για να αναπαραστήσουν ένα διανοητικό μοντέλο για μια κατάσταση, π.χ. οδήγηση αυτοκινήτου, γεύμα σε εστιατόριο.
- Η γνώση που σχετίζεται με μια οντότητα συσσωρεύεται στη μνήμη και αποτελεί μια μονάδα γνώσης.
- Οργάνωση της γνώσης σε πλαίσια που περιγράφουν αντικείμενα (συγκεκριμένα στιγμιότυπα ή γενικότερες κλάσεις).
- Ένα πλαίσιο είναι μια δομή δεδομένων που περιλαμβάνει σχισμές (slots) οι οποίες έχουν ονόματα και τιμές.
- Η τιμή μιας σχισμής μπορεί να είναι απλή, αναφορά σε άλλο πλαίσιο, διαδικασία υπολογισμού της πραγματικής τιμής ή απροσδιόριστη.
- Αντίστοιχα με τα σημασιολογικά δίκτυα, τα πλαίσια μπορούν να συνδεθούν μεταξύ τους μέσω ειδικών σχισμών (π.χ. ako) και να δομήσουν ιεραρχίες πλαισίων.
- Ειδικές σχισμές είναι οι “a_kind_of” για τη σχέση κλάσεων-υπερκλάσεων και “instance_of” για τη σχέση στιγμιοτύπων-κλάσεων.
- Η μεθοδολογία των πλαισίων παρουσιάζει πολλά κοινά στοιχεία με την αντικειμενοστραφή προσέγγιση.



Υλοποίηση σε Prolog

```
bird(a_kind_of,animal).
bird(moving_method,fly).
bird(active_at,daylight).

albatross(a_kind_of,bird).
albatross(colour,black_and_white).
albatross(size,115).

kiwi(a_kind_of,bird).
kiwi(moving_method,walk).
kiwi(active_at,night).
kiwi(size,40).
kiwi(colour,brown).
```

```
albert(instance_of,albatross).
albert(size,120).

ross(instance_of,albatross).
ross(size,40).

animal(relative_size,
   execute(relative_size(Object,Value),
           Object,
           Value)).
```


Απλός υπολογισμός των τιμών των σχισμών

- Δεν καλύπτει τον υπολογισμό τιμών με χρήση διαδικασιών
-
```
value(Frame,Slot,Value) :-
      Query =.. [Frame,Slot,Value],
      call(Query),!.
value(Frame,Slot,Value) :-
      parent(Frame,ParentFrame),
      value(ParentFrame,Slot,Value).

parent(Frame,ParentFrame) :-
      (Query =.. [Frame,a_kind_of,ParentFrame];
       Query =.. [Frame,instance_of,ParentFrame]),
      call(Query).

?-value(albert,active_at,AlbertTime).
  AlbertTime = daylight
?-value(kiwi,active_at,KiwiTime).
  KiwiTime = night
```

Έστω ότι έχουμε ακόμα τη διαδικασία

```
relative_size(Object,RelativeSize) :-
    value(Object,size,ObjSize),
    value(Object,instance_of,ObjClass),
    value(ObjClass,size,ClassSize),
    RelativeSize is ObjSize/ClassSize * 100.
```

- Θέλουμε να μπορούμε να απαντάμε και ερωτήματα όπως:

```
?-value(ross,relative_size,R).
  R = 34.78
```


Υπολογισμός των τιμών των σχισμών καλύπτοντας και την περίπτωση των διαδικασιών

```
value(Frame,Slot,Value):-
      value(Frame,Frame,Slot,Value).

value(Frame,SuperFrame,Slot,Value):-
      Query =.. [SuperFrame,Slot,Information],
      call(Query),
      process(Information,Frame,Value),!.

value(Frame,SuperFrame,Slot,Value):-
      parent(SuperFrame,ParentSuperFrame),
      value(Frame,ParentSuperFrame,Slot,Value).
```

```
process(execute(Goal,Frame,Value),Frame,Value):- !,
        call(Goal).
process(Value,_,Value).

parent(Frame,ParentFrame) :-
       (Query =.. [Frame,a_kind_of, ParentFrame];
        Query =.. [Frame,instance_of, ParentFrame]),
       call(Query).
```


Επαγωγικές (συμπερασματικές) βάσεις δεδομένων - datalog

- Δεδομένα (απλά ή σύνθετα) VS γνώση
- Συστήματα βασισμένα σε γνώση – συστήματα βάσης γνώσης VS συστήματα διαχείρισης βάσης γνώσης (τα τελευταία χρειάζονται, π.χ., αποδοτική προσπέλαση, χειρισμό δοσοληψιών)
- Δηλωτική γλώσσα
  - Για χειρισμό δεδομένων
  - Σαν φιλόξενη γλώσσα (host language)
- Οι δηλωτικές γλώσσες συχνά βασίζονται σε κάποια μορφή λογικής, π.χ. SQL βασίζεται στο σχεσιακό λογισμό (relational calculus)

- ΓΕΓΟΝΟΣ:
  - θετική η ανάμιξη της λογικής και των βάσεων δεδομένων
  - θετική η ανάμιξη της τεχνητής νοημοσύνης και των τεχνολογιών των βάσεων δεδομένων για την ανάπτυξη βάσεων γνώσης
- Η προσπάθεια στην ενοποίηση εστιάζει σε θέματα:
  - Εκφραστικότητας (expressiveness) και σημασιολογίας (semantics)
  - Βελτιστοποίηση των ερωτήσεων που εκφράζονται σαν λογικά προγράμματα
  - Το πρώτο είναι πολύ κρίσιμο για τις επαγωγικές βάσεις δεδομένων
- Απευθείας πρόσβαση στα δεδομένα από το σύστημα του λογικού προγραμματισμού: η περίπτωση της datalog

- Λογικοί κανόνες με συναρτησιακά σύμβολα είναι εξίσου ισχυροί με μια μηχανή Turing
- Ακόμα και χωρίς συναρτησιακά σύμβολα, οι λογικοί κανόνες έχουν τη δυνατότητα να εκφράσουν υπολογισμούς που δεν εκφράζονται στις συμβατικές γλώσσες χειρισμού δεδομένων, π.χ. τη μεταβατική κλειστότητα μιας σχέσης («Ποιες είναι οι ιεραρχίες managers;»)
- Η λογική πρώτης τάξης (first order logic) μπορεί να αποτελέσει τόσο ένα μέσο αναπαράστασης της γνώσης όσο και μια γλώσσα για την έκφραση πράξεων πάνω σε σχέσεις
- Υπάρχει μια ιεραρχία μοντέλων δεδομένων καθένα από τα οποία υποστηρίζει δεδομένα παρόμοια με εκείνα του σχεσιακού μοντέλου αλλά με σταδιακά πιο ισχυρές λογικές γλώσσες στις οποίες εκφράζουμε πράξεις στα δεδομένα: *το πιο απλό μοντέλο είναι η datalog*

Σύνταξη προγραμμάτων datalog

- l_0 :- l_1 & … & l_n
  - l_0: κεφαλή (head)
  - l_1 & … & l_n: σώμα (body)
  - l_i: λεκτικό (litteral) της μορφής $p_i(t_1, …, t_{ki})$ (**ατομικός τύπος**, atomic formula)
  - p_i: σύμβολο κατηγορήματος (predicate symbol)
  - t_j: όρος (term) (**σταθερά** ή **μεταβλητή**)

- Προτάσεις datalog
  - Πλήρως αποτιμημένα γεγονότα που φυλάσσονται σε σχεσιακή βάση (*άμεση βάση* – extensional database, *EDB*)
  - Λογικοί κανόνες (*έμμεση βάση* – intensional database, *IDB*)
- Κατηγορήματα IDB – κατηγορήματα EDB
- Παράδειγμα:
  - grandparent(Z,X) :- par(Y,X) & par(Z,Y).
  - father(bob,john).

Συντακτικοί Περιορισμοί

- Κανόνες Ασφάλειας:
  - Κάθε γεγονός πρέπει να είναι πλήρως αποτιμημένο (ground)
  - Κάθε μεταβλητή που εμφανίζεται στην κεφαλή ενός κανόνα πρέπει να εμφανίζεται και στο σώμα του
- Απαγορεύεται ένα κατηγόρημα EDB να εμφανίζεται σε κεφαλή κανόνα

Σημασιολογία των προγραμμάτων datalog

- Μοντελοθεωρητική Σημασιολογία (model theoretic semantics)
  - Βασίζεται στις *ερμηνείες* (interpretations) και τα *μοντέλα* (models) ενός συστήματος
  - Μια ερμηνεία κάνει άλλους ατομικούς τύπους *αληθείς* και άλλους *ψευδείς* (ως προς την ερμηνεία αυτή)
  - Συνήθως, ταυτίζουμε μια ερμηνεία με το σύνολο των πλήρως αποτιμημένων τύπων που κάνει αληθείς (*ερμηνεία Herbrand*)

- Μια ερμηνεία είναι μοντέλο για ένα σύνολο από κανόνες αν κάθε κανόνας είναι αληθής ως προς την ερμηνεία αυτή
- Θεωρούμε ότι στιγμιότυπο (instance) ενός ατομικού τύπου με κατηγόρημα EDB ισχύει εάν και μόνο εάν η αντίστοιχη σχέση περιέχει αυτό το στιγμιότυπο σαν πλειάδα

Παραδείγματα

- p(X) :- q(X).
  - I_1 = {q(a)}, δεν κάνει τον κανόνα αληθή
  - I_2 = {q(a),p(a)}, κάνει τον κανόνα αληθή
- p(X) :- q(X).
  v(X) :- r(X).
  - I_1 = {q(a),p(a)} είναι μοντέλο για τους κανόνες
  - I_2 = {q(a),p(a), r(b),v(b)} είναι μοντέλο για τους κανόνες
  - I_3 = {r(b)} δεν είναι μοντέλο για τους κανόνες

- p(X) :- q(X).
  q(X) :- r(X).
  και έστω r/1 EDB τέτοιο ώστε r(1) ισχύει από τη βάση
  - I_1={r(1),q(1),p(1)}, I_2={r(1),q(1),p(1),q(2),p(2)}, I_3={r(1),q(1),p(1),q(2),p(2),p(3)}
  - I_1,I_2,I_3 είναι μοντέλα για το παραπάνω
  - Το I_1 είναι ελάχιστο μοντέλο (minimal model)
  - Το I_1 είναι το ελάχιστο μοντέλο Herbrand (least Herbrand model)

- Ένα γεγονός F *έπεται λογικά* από ένα σύνολο προτάσεων S εάν και μόνο εάν κάθε ερμηνεία που κάνει αληθές το S κάνει αληθές και το F. Τότε λέμε ότι το F είναι *λογικό επακόλουθο* του S και γράφουμε

$$S \models F$$

- $cons(S) = \cap\{I \mid I \text{ είναι μοντέλο του } S\}$, όπου $cons(S) = \{F \in HB \mid S \models F\}$
- Πώς μπορεί να υπολογιστεί το $cons(S)$;

- Σημασιολογία Απόδειξης (proof theoretic semantics)
  - Οι κανόνες ερμηνεύονται ως αξιώματα προς χρήση σε αποδείξεις
  - Αντικαθιστούμε αποδεδειγμένα ή δεδομένα γεγονότα στο δεξί τους μέρος και συμπεραίνουμε το γεγονός που προκύπτει από την κεφαλή
  - Χρησιμοποιούνται κατά την εμπρόσθια (forward) φορά

- L_0 :- L_1 & … L_n και
- F_1, …, F_n: πλήρως αποτιμημένα γεγονότα και
- θ: αντικατάσταση (substitution): $L_i\theta = F_i$
- Τότε, με ένα βήμα, συμπεραίνουμε: $L_0\theta$
- Κανόνας συμπερασμού (inference rule)

- Έστω S ένα σύνολο από προτάσεις datalog. Ένα πλήρως αποτιμημένο γεγονός F *συμπεραίνεται* (is inferred) από το S (S ├ F) εάν και μόνο εάν είτε F ϵ S είτε μπορεί να παραχθεί από πεπερασμένο αριθμό εφαρμογών του κανόνα συμπερασμού.
- Η ακολουθία των εφαρμογών του κανόνα προκειμένου να συμπεράνουμε ένα πλήρως αποτιμημένο γεγονός F από το S καλείται *απόδειξη* (proof) του F από το S
- Ισχύει: S ╞ F⇔ S ├ F
- Το σύνολο cons(S) μπορεί να προσδιοριστεί ως το ελάχιστο σταθερό σημείο ενός μετασχηματισμού

Επεκτάσεις της datalog (I)

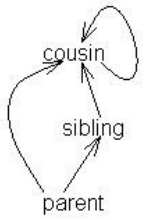
- **Ενσωματωμένα κατηγορήματα: κατηγορήματα αριθμητικών συγκρίσεων (=, ≤, ...)**
- Πρόβλημα: τα ενσωματωμένα κατηγορήματα δεν αναπαριστούν απαραίτητα πεπερασμένες σχέσεις
- Κατά τον υπολογισμό πρέπει να έχουν ικανοποιητική αποτίμηση για να μην οδηγήσουν σε μη πεπερασμένες σχέσεις (ένα πρόγραμμα datalog πρέπει να έχει πεπερασμένη έξοδο)
- Λύση: Οποτεδήποτε ένας κανόνας περιέχει στο σώμα του έναν ατομικό τύπο με ενσωματωμένο κατηγόρημα, το εύρος των τιμών των μεταβλητών του πρέπει να περιορίζεται από κάποιον άλλο ατομικό τύπο στον κανόνα
- Τα ενσωματωμένα κατηγορήματα μπορούν να θεωρηθούν σαν κατηγορήματα EDB
- Με βάση τη σχεσιακή άλγεβρα μπορούν να διερμηνευτούν ως επιλογές (selections) πάνω σε μια σχέση ή συνενώσεις (joins) σχέσεων

Γράφος εξαρτήσεων

sibling(X,Y) :- parent(X,Z) & parent(Y,Z)
& X ≠ Y.

cousin(X,Y) :- parent(X,Xp) &
parent(Y,Yp) & sibling(Xp,Yp).

cousin(X,Y) :- parent(X,Xp) &
parent(Y,Yp) & cousin(Xp,Yp).



- Κύκλοι στο γράφο δηλώνουν αναδρομή
- Ένα κατηγόρημα είναι μη αναδρομικό αν δεν εμφανίζεται μέσα σε κάποιο κύκλο
- Στα κατηγορήματα EDB δε καταλήγει κανένα βέλος

- Υπολογισμός μη αναδρομικών κανόνων μέσω μετασχηματισμού τους σε εκφράσεις της σχεσιακής άλγεβρας
- Οι σχέσεις που προκύπτουν για τα κατηγορήματα IDB ταυτίζονται τόσο με το ελάχιστο μοντέλο των κανόνων όσο και με το σύνολο των γεγονότων IDB που συμπεραίνονται από τους κανόνες και τη βάση
- Για το μετασχηματισμό εκμεταλλευόμαστε τη διάταξη που προκύπτει από το γράφο εξαρτήσεων (έναν υπάρχει ένα βέλος $p_i \rightarrow p_j$ για δυο κατηγορήματα p_i και p_j τότε $p_i < p_j$). Οπότε, όταν πάμε να υπολογίσουμε τη σχέση που αντιστοιχεί στο σώμα ενός κανόνα, έχουμε υπολογίσει όλες τις σχέσεις που αντιστοιχούν στους διάφορους υποστόχους του κανόνα.

- Παράδειγμα:

  p(X,Y) :- q(a,X) & r(X,Z,X) & s(Y,Z)

  Σχέσεις:

  $q \rightarrow Q$, $r \rightarrow R$, $s \rightarrow S$

  $T(X) = \pi_2 (\sigma_{\$1=a} (Q))$

  $U(X,Z) = \pi_{1,2}(\sigma_{\$1=\$3} (R))$

  $B(X,Y,Z) = T(X) \bowtie U(X,Z) \bowtie S(Y,Z)$

- Για τον υπολογισμό αναδρομικών κανόνων, ο γράφος εξαρτήσεων δεν μας προσφέρει καμιά διάταξη
- Οπότε, κάνουμε αλλεπάλληλους υπολογισμούς

$P_i := EVAL(p_i,R_1,\ldots,R_k,P_1,\ldots,P_m)$, όπου

$R_1,..,R_k$: δεδομένες EDB σχέσεις και

$P_1,\ldots,P_m$: IDB σχέσεις προς υπολογισμό

με τα P_j αρχικά κενά, έως ότου να μη παράγονται άλλες πλειάδες, τότε

$$P_i = EVAL(p_i,R_1,\ldots,R_k,P_1,\ldots,P_m)$$

και πλέον έχουμε τη λύση για τις σχέσεις που αντιστοιχούν στα κατηγορήματα IDB αυτών των εξισώσεων (σταθερό σημείο – fixpoint)

Ιδιότητες των προγραμμάτων datalog

- Έχουν ένα μοναδικό ελάχιστο σταθερό σημείο
- Έχουν ένα μοναδικό ελάχιστο μοντέλο
- Τα παραπάνω ταυτίζονται με το σύνολο των γεγονότων που μπορούμε να παράγουμε χρησιμοποιώντας τους κανόνες για μια δεδομένη βάση
- Η παραπάνω διαδικασία είναι μονότονη. Αρκεί σε κάθε βήμα να λαμβάνουμε υπόψη μας όχι ολόκληρες τις σχέσεις αλλά τις επαυξήσεις (διαφορές) που προέκυψαν από το προηγούμενο.

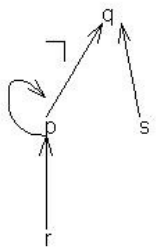
Επεκτάσεις της datalog (II)

- **Άρνηση** (negation):
- Κανόνες με άρνηση στα σώματά τους δεν είναι προτάσεις Horn (χάνουμε πολλά από τα ωραία συμπεράσματα που ισχύουν για τις προτάσεις Horn)
- trueCousin(X,Y) :- cousin(X,Y) & ¬sibling(X,Y)
- Διαισθητικά μπορούμε να πούμε ότι η άρνηση δίνει το «συμπλήρωμα» μιας σχέσης
- Προβλήματα:
  - «Συμπλήρωμα» ως προς ποιο πεδίο ορισμού;
  - Το συμπλήρωμα μπορεί να οδηγεί σε μη πεπερασμένη σχέση
  - Δεν υπάρχει απαραίτητα ένα ελάχιστο σταθερό σημείο για το λογικό πρόγραμμα
  - Υπάρχουν πολλά ελάχιστα μοντέλα για το λογικό πρόγραμμα
- C(X,Y) $\bowtie$ compl S(X,Y), όπου compl S είναι το συμπλήρωμα του S σε σχέση με κάποιο σύμπαν U το οποίο περιέχει τουλάχιστον τις πλειάδες του C

- Πρόβλημα: μεταβλητές που εμφανίζονται ΜΟΝΟ σε υποστόχους με άρνηση
- bachelor(X) :- male(X) & ¬married(X,Y)
- MRD = {<1,a>, <2,b>}
- M = {1,2,3,4}
- compl MRD = {<1,b>,<2,a>,<3,a>,…}: δεν είναι το επιθυμητό
- Λύση: απαγορεύεται η χρήση μιας μεταβλητής σ' έναν υποστόχο με άρνηση, εάν αυτή δεν εμφανίζεται σε κάποιον άλλο υποστόχο ο οποίος δεν πρέπει να περιέχει άρνηση ή άλλο ενσωματωμένο κατηγόρημα
- husband(X) :- married(X,Y).
- bachelor(X) :- male(X) & ¬husband(X).
- $H(X) = \pi_X(MRD(X,Y))$, B(X) = M(X) – H(X)

- Πρόβλημα: δεν υπάρχει ένα μοναδικό ελάχιστο σταθερό σημείο
- p(X) :- r(X) & ¬q(X).
  q(X) :- r(X) & ¬p(X).
  R{1}
  S1: P = {1}, Q = empty_set
  S2: P = empty_set, Q = {1}

- **Στρωματοποιημένη άρνηση** (stratified negation)
- p(X) :- r(X).
- p(X) :- p(X).
- q(X) :- s(X) & ¬p(X).



- Δεν υπάρχει μονοπάτι από το q στο p: στρωματοποιημένο
- R{1}, S{1,2}
- S1: P ={1}, Q={2}
- S2: P={1,2}, Q = empty_set
- Και τα δυο είναι ελάχιστα, το S1, όμως, δείχνει πιο «φυσικό»

- **Στρωματοποίηση**: Καθορισμός στρωμάτων (strata)
  - Εάν ένα κατηγόρημα p έχει έναν κανόνα που περιέχει υποστόχο με άρνηση με κατηγόρημα q, τότε το q είναι σε χαμηλότερο στρώμα από το p.
  - Εάν ένα κατηγόρημα p έχει έναν κανόνα που περιέχει υποστόχο χωρίς άρνηση που περιέχει το q τότε το στρώμα του p είναι το στρώμα του q ή ψηλότερο.
- Τα στρώματα μας δίνουν μια σειρά με την οποία μπορούν να υπολογιστούν οι σχέσεις που αντιστοιχούν στα κατηγορήματα IDB
- Επεξεργαζόμαστε κάθε φορά ένα στρώμα ξεκινώντας από το χαμηλότερο
- Έτσι, μεταχειριζόμαστε τους υποστόχους με άρνηση σαν να ήταν σχέσεις EDB
- Η σχέση για έναν υποστόχο με άρνηση ¬q(X1,…,Xn)

είναι:

$$\underbrace{DOM \times DOM \times \ldots \times DOM}_{\text{n φορές}} - Q,$$

όπου DOM είναι η ένωση όλων των σταθερών που εμφανίζονται στην IDB και στην EDB

- Κατασκευή στρωμάτων
- Κατασκευή DOM
- Υπολογισμός σχέσεων ανά στρώμα, ξεκινώντας από το χαμηλότερο
  1. Για τους υποστόχους που δεν περιέχουν άρνηση και τα κατηγορήματά τους περιέχονται σε χαμηλότερα στρώματα, θεωρούμε τις σχέσεις όπως έχουν υπολογιστεί
  2. Για τους υποστόχους που περιέχουν άρνηση, θεωρούμε τα συμπληρώματα των σχέσεων – οι σχέσεις έχουν ήδη υπολογιστεί
  3. Εφαρμόζουμε τον αλγόριθμο κατασκευής των σχέσεων μέσα στο στρώμα, θεωρώντας τις σχέσεις που αναφέρονται στα δυο πρώτα βήματα σαν να ήταν EDB
- Ο αλγόριθμος αυτός παράγει ένα ελάχιστο σταθερό σημείο (που είναι και ελάχιστο μοντέλο: το τέλειο – perfect –)

Άλλα θέματα που σχετίζονται με τη datalog

- Βελτιστοποίηση υπολογισμού ερωτήσεων datalog
- Επεκτάσεις της datalog (III): σύνθετα αντικείμενα (complex objects)
- Επεκτάσεις της datalog (IV): αντικειμενοστραφείς επεκτάσεις (object oriented extensions)

Σύγχρονες ανάγκες για αναπαράσταση γνώσης

- World wide web:
  - Πραγματικότητα: “Web of documents”
- Μεταδεδομένα
- Σημασιολογική πληροφορία
- Semantic web
  - Vision: “Web of data”

World Wide Web

- Web of documents
- URLs
- HTTP
- HTML σελίδες (HTML εμπνευσμένη από την SGML – Standard Generalized Markup Language)

- Παράδειγμα:

<H1> Λογικός Προγραμματισμός </H1>
<H2> Εξάμηνο ΣΤ’ </H2>
<H3> Περιεχόμενο </H3>
<UL>
<LI> Prolog
<LI> Θεωρία Λογικού Προγραμματισμού
<LI> …
….
</UL>

XML και Ημιδομημένα Δεδομένα

- Tags που ορίζονται από το χρήστη σύμφωνα με τις ανάγκες μιας εφαρμογής
- Θεωρούμε ότι απεικονίζουν κάποιας μορφής «νόημα» για το περιεχόμενό τους, μια που θα το χειριστεί κατάλληλα η αντίστοιχη εφαρμογή
- XML (Extensible Markup Language)
- W3 activity
- Recommendation (5η έκδοση, Feb 2008)
- Βασίζεται στην SGML (υποσύνολό της)
- Ο ορισμός του συντακτικού ενός εγγράφου XML μπορεί να γίνει από κάποιο φορμαλισμό όπως το DTD (Document Type Definition) ή το XML Schema
- Γλώσσες και μεθοδολογίες για τον υπολογισμό απαντήσεων για έγγραφα XML, π.χ. XQuery και XPath, αντίστοιχα

- Παράδειγμα:

<book>

<title> Gone with the wind </title>

<author> Margaret Mitchell </author>

<year> 1936 </year>

</book>

Ευελιξία στο σχήμα των δεδομένων – RDF

- RDF (Resource Description Framework)
- Αρχικά W3 activity, μετά semantic web activity
- Specification recommendation (2004)
- Μοντέλο δεδομένων
- Γενική μορφή: «τριπλέτες» (x,P,y), όπου:
  - P: property
  - x,y: objects
  - Αντίστοιχος λογικός τύπος: P(x,y) – P binary predicate
- URIs (Uniform Resource Identifiers) μπορούν να χρησιμοποιηθούν για να ονομάσουν τόσο τα P και x αλλά και τα y (αλλιώς το y μπορεί να είναι μια σταθερά)
- Γλώσσες και μεθοδολογίες για τον υπολογισμό απαντήσεων για αρχεία RDF, π.χ. SPARQL
- RDF και ER
- RDF και Conceptual Graphs

Ορισμός σχήματος – «ελαφρά» μεταδεδομένα

- RDF Schema (RDF Vocabulary Description Language**)**
- W3 activity
- Specification recommendation (2004)
- Classes-Instances-Properties/Resources
- Subclass/belongs/subproperty
- Κληρονομικότητα

Ανταλλαγή πληροφορίας – περιγραφή γνωστικού πεδίου

- Μοντέλο δεδομένων και μεταδεδομένων
- Για περιγραφή πολύπλοκων γνωστικών πεδίων, π.χ.
  - δυνατότητα περιγραφής κλάσεων, όχι απλά δυνατότητα ονομασίας
  - περιορισμούς στα πεδία τιμών
  - απόδοση ιδιοτήτων μεταξύ κλάσεων (π.χ. «ξένες» μεταξύ τους)

- OWL (Web Ontology Language)
- W3 activity
- Specification recommendation (2004)
- Συμβιβασμός μεταξύ εκφραστικότητας και υλοποιησιμότητας
- OWL Full (first order logic) (υπερσύνολο της RDFS)
- OWL-DL (description logic)
- OWL-Lite (απλή λειτουργικότητα στον ορισμό κλάσεων)

Λογικές Περιγραφών

- Βασικές έννοιες:
  - Concepts (unary predicates), π.χ. άνθρωπος
  - Roles (binary predicates), π.χ. hasChild
  - Individual names (σταθερές), π.χ. Mary
  - Τελεστές για να περιγράφουμε concepts και roles
  - Προσοχή: πρέπει να είναι περιορισμένοι ώστε
    - Satisfiability/subsumption is decidable and, *if possible*, of low complexity

- TBox: Το σύνολο των περιγραφών, π.χ.

  Doctor $\subseteq$ Person,
  HappyParent = Person $\cap \forall$ hasChild.(Doctor U $\exists$ hasChild.Doctor)
- ABox: Το σύνολο των δεδομένων. π.χ.

  John:HappyParent,

  John hasChild Mary
- KB = TBox U Abox

• Σημασιολογία βασισμένη στις ερμηνείες και τα μοντέλα

Συλλογιστική

- SWRL (Semantic Web Rule Language) (W3 member submission)
- RuleML:
  - Κανόνες για το WWW
  - RuleML consortium collaborates with W3C
- Κανόνες πάνω από την OWL VS Horn (datalog) κανόνες (Λογικός Προγραμματισμός)
- "open" semantics VS "closed" semantics για την άρνηση

Λογικός Προγραμματισμός

- Μπορεί να χρησιμοποιηθεί για:
  - Μοντέλο δεδομένων,
  - περιγραφή μεταδεδομένων και
  - κανόνων για χρήση σε συλλογιστική
- Απλότητα
- Θεωρία
- Υλοποίηση (prolog)
- Επεκτάσεις (constraints, modules)

Βιβλιογραφία

- Ivan Bratko, "**Prolog Programming for Artificial Intelligence**", Addison Wesley, 2000
- Jeffrey D. Ullman, "**Principles of database and knowledge-base systems, Vol. I**", Computer Science Press, Inc , 1988
- W3 Consortium (http://www.w3.org/)
- Ivan Herman, "**Introduction to the Semantic Web"** WWW2006, Edinburgh, UK, 2006-05-24 (http://www.w3.org/2006/Talks/0524-Edinburgh-IH)
- Ian Horrocks, "**Description Logics in Ontology Applications**", KI/Tableaux 2005 (http://www.cs.man.ac.uk/~horrocks/Slides/index.html)
- Carsten Lutz and Ulrike Sattler, "Description Logics Course" ESSLLI 2002 (http://lat.inf.tu-dresden.de/~clu/esslli.html)