Τετάρτη 10:15 – 12:00 στο εργαστήριο 210

Πέμπτη 08:15 – 10:00 στην αίθουσα 102

Διδάσκων: Επ. Καθ. Δημήτρης Αποστόλου Τηλ: 4142314, Email: dapost στο unipi τελεία gr

Γραπτή εξέταση και προεραιτικές εργασίες (3 βαθμοί, bonus)

Σημειώσεις: Λογικός Προγραμματισμός, Θ. Παναγιωτόπουλος, http://gunet2.cs.unipi.gr

Mailing list: http://students.cs.unipi.gr/mailman/listinfo/ugrads-lpp

Prolog interpreters: http://www.swi-prolog.org/dl-stable.html, http://www.lpa.co.uk, http://sics.se/sicstus

Περί Prolog και Λογικού Προγραμματισμού

- Η λέξη Prolog βγαίνει από τα αρχικά PROgramming in LOGic, δηλαδή προγραμματίζοντας στη Λογική.

- Είναι μια γλώσσα τέταρτης γενιάς, που έχει αναπτυχθεί τα τελευταία κυρίως χρόνια, αν και ακόμη δεν έχει βρει 'το δρόμο της', δεν έχει δηλαδή καθιερωθεί μεταξύ των άλλων συμβατικών γλωσσών προγραμματισμού.

- Οι λόγοι για αυτό δεν είναι:
  - Η άγνοια όσον αφορά αυτή τη γλώσσα των υπευθύνων των εταιρειών παραγωγής λογισμικού (software houses) και των προγραμματιστών.
  - Ο προγραμματισμός με Prolog απαιτεί κάποιες στοιχειώδεις γνώσεις Λογικής.
  - Οι περισσότεροι διερμηνείς και μεταφραστές της Prolog δεν έχουν δυνατότητες επικοινωνίας με βάσεις δεδομένων, εντολές χειρισμού γραφικών, κ.λ.π.

Ιδιαιτερότητες Prolog

- Η Prolog είναι γλώσσα Λογικού Προγραμματισμού και δίνει πολύ μεγάλες δυνατότητες σε ένα προγραμματιστή να λύσει προβλήματα γρήγορα και με κομψό τρόπο.

- Όμως ο προγραμματισμός σ' αυτή τη γλώσσα έχει τελείως διαφορετικό νόημα από τον συμβατικό προγραμματισμό που ακολουθείται σε διαδικαστικές γλώσσες όπως η Pascal και η C++.
  - Αν λοιπόν θέλετε να προγραμματίσετε σε Prolog ξεχάστε τον τρόπο που προγραμματίζατε!

- Στην Prolog τα προγράμματα είναι ορισμοί!
  - Είναι ορισμοί κατηγορημάτων σε σχέση με το εννοιολογικό περιεχόμενο των κατηγορημάτων.
  - Μάλιστα είναι στις περισσότερες περιπτώσεις αναδρομικοί ορισμοί, όπως αυτοί που δίνουμε σε μερικές περιοχές των μαθηματικών (όπως η Λογική).

Εφαρμογές της Prolog

- Η γλώσσα Prolog μπορεί να χρησιμοποιηθεί σε όλες τις περιοχές που είναι εφαρμόσιμος και ο κατηγορηματικός λογισμός πρώτης τάξης.

- Επίσης, λόγω των πέρα της λογικής χαρακτηριστικών της μπορεί να βρει εφαρμογή σε όλο το εύρος της επιστήμης της Πληροφορικής (σαν μία γλώσσα προγραμματισμού).

- Δεδομένου δε ότι, όπως έχει υποστηρίξει ο Kowalski
  "Αλγόριθμος = Λογική + Έλεγχος"
  η Prolog περισσότερο από κάθε άλλη γλώσσα προγραμματισμού προσφέρει ένα εύκολο τρόπο στον χρήστη της να εκφράσει το λογικό μέρος των αλγορίθμων.
  - Τον έλεγχο τον προσφέρει η ίδια αυτοματοποιημένο, μέσα από τον μηχανισμό συμπερασματολογίας.

Βασικές έννοιες λογικών προγραμμάτων

- Ένα λογικό πρόγραμμα είναι ένα σύνολο από αξιώματα ή κανόνες οι οποίοι καθορίζουν σχέσεις ανάμεσα σε αντικείμενα.

- Υπολογισμός ενός λογικού προγράμματος είναι ένα συμπέρασμα από τα αποτελέσματα ενός προγράμματος.

- Ένα πρόγραμμα καθορίζει ένα σύνολο αποτελεσμάτων, τα οποία αποτελούν το νόημά του.

- Η ικανότητα του λογικού προγραμματισμού είναι να κατασκευάζει σαφή και κομψά προγράμματα τα οποία έχουν το επιθυμητό νόημα.

- Οι βασικές έννοιες του λογικού προγραμματισμού, όροι και δηλώσεις, είναι κληρονομημένες από την λογική.

Προτασιακή Λογική

❖ Στην προτασιακή λογική κάθε γεγονός του πραγματικού κόσμου
  1 αναπαριστάται με μια *λογική πρόταση*,
  2 χαρακτηρίζεται είτε ως *αληθής* (***T**-true*) ή ως *ψευδής* (***F**-false*)

❖ Οι λογικές προτάσεις (*άτομα - atoms*) αναπαριστώνται συνήθως από λατινικούς χαρακτήρες.

❖ Συνδυάζονται με τη χρήση λογικών συμβόλων ή συνδετικών (connectives)

Σύμβολο	**Ονομασία / Επεξήγηση**
$\wedge$	σύζευξη (λογικό "ΚΑΙ")
$\vee$	διάζευξη (λογικό "Η")
$\neg$	άρνηση
$\rightarrow$	συνεπαγωγή ("ΕΑΝ ΤΟΤΕ")
$\leftrightarrow$	Διπλή συνεπαγωγή ή ισοδυναμία ("ΑΝ ΚΑΙ ΜΟΝΟ ΑΝ")

Περιορισμός της Προτασιακής Λογικής

- Εάν με p αναπαριστούμε την πρόταση «όλοι οι σκύλοι μυρίζουν» και η p είναι αληθής, τότε θα θέλαμε να μπορούμε να αποδείξουμε οτι «ο σκύλος μου ο fido μυρίζει".

- Τότε λοιπόν μας χρειάζεται η κατηγορηματική λογική (ή λογική πρώτης τάξης).

Κατηγορηματική Λογική

- Επέκταση της προτασιακής λογικής.

- Ο κόσμος περιγράφεται σαν ένα σύνολο *αντικειμένων*, *ιδιοτήτων* και *σχέσεων*.

- Αντιμετωπίζει το πρόβλημα της μη προσπελασιμότητας των στοιχείων των γεγονότων της προτασιακής λογικής.
  - Π.χ., η πρόταση "*ο τζίμης είναι τίγρης* " αναπαριστάται με *τίγρης(τζίμης)*

- Ύπαρξη *μεταβλητών*, που αυξάνει σημαντικά την εκφραστική ικανότητά της
  - Επιτρέπει την αναπαράσταση "γενικής" γνώσης.

- Επεκτείνει την προτασιακή λογική εισάγοντας
  - όρους (terms),
  - κατηγορήματα (predicates) και
  - ποσοδείκτες (quantifiers).

Ξανά στον Λογικό Προγραμματισμό

- Υπάρχουν τρεις βασικές δηλώσεις:
  - τα γεγονότα (facts),
  - οι ερωτήσεις (queries) και
  - οι κανόνες (rules).

- Υπάρχει μια μοναδική δομή δεδομένων: ο λογικός όρος (logical term) ή απλά όρος.

Γεγονότα

- Το πιο απλό είδος της δήλωσης ονομάζεται **γεγονός (fact)**.

- Τα γεγονότα είναι ένα μέσο καθορισμού μιας σχέσης που ισχύει ανάμεσα στα αντικείμενα. Ένα παράδειγμα είναι:

  father(abraham,isaac).

- Αυτό το γεγονός λέει ότι ο *Abraham* είναι ο πατέρας του *Isaac*, ή ότι ο συσχετισμός *πατέρας (father)* ισχύει ανάμεσα στις μονάδες, οι οποίες ονομάζονται *abraham* και *isaac*.

- Άλλο ένα όνομα για μια σχέση είναι το **κατηγόρημα (predicate)**.

- Τα ονόματα των μονάδων είναι γνωστά ως **άτομα (atoms)**.

- Το κατηγόρημα father έχει 2 ορίσματα. Δηλ. ***father/2.***

Παραδείγματα γεγονότων σε Prolog

- “the capital of france is paris”

  has_capital(france,paris).

  - Προσέξτε οτι στην Prolog, εάν το όνομα ενός αντικειμένου αρχίζει με μικρό γράμμα τότε αναφέρεται σε συγκεκριμένο αντικείμενο ή μονάδα.

  - Επίσης, δεν πρέπει να υπάρχει κενό (space) ανάμεσα στο όνομα του κατηγορήματος και στη αριστερή παρένθεση “(“. Όλη η γραμμή τελειώνει με “.”.

Παραδείγματα γεγονότων σε Prolog

❖ Επίσης προσέξτε οτι μπορούμε να έχουμε σχέσεις ανάμεσα σε παραπάνω από δύο αντικείμενα. Για παράδειγμα:

meets(fred,jim,bridge).

- might be read as “fred meets jim by the bridge”.
- Here, three objects are related so it makes little sense to think of the relation meets as binary - it is ternary.

Ασκήσεις

❖ Αναπαραστήστε τις παρακάτω προτάσεις σε Prolog.

1. bill likes ice-cream
2. bill is tall
3. jane hits jimmy with the cricket bat
4. john travels to london by train
5. bill takes jane some edam cheese

Λύσεις Ασκήσεων

1. *likes(bill,ice_cream).*

We could, of course, have defined likes in the reverse way. This would lead to the representation *likes(ice_cream,bill)* and the reading, in this case, that «ice_cream» is «liked» by «bill».

Also note that we could get away with a one argument relation:

likes_ice_cream(bill).

Or even a zero argument relation

bill_likes_ice_cream.

We could try the representation that *bill(likes,ice_cream)*. Usually, predicates are associated with verbs.

Λύσεις Ασκήσεων

2.   is(bill,tall).

This might be chosen but there is are problems: First, with the word `is'. Here, it is associated with the idea that bill possesses an attribute which has the value `tall'.

Another reason for not using is(bill,tall) is that there may be many such statements in a database. Prolog would then have to sort through a large number of is/2 clauses such as is(bill,rich).

Finally, the predicate is/2 is a system predicate and cannot be redefined by the user!

height(bill,tall).

If we choose height(bill,tall) then we only search through the clauses that deal with height.

tall(bill).

is quite acceptable and probably the one most people will prefer. However, we should note that this representation will make it harder to pick up any relation between tall(bill) and, for example, short(fred) whereas this is easy for height(bill,tall) and height(fred,short).

Λύσεις Ασκήσεων

3. *hits(jane,jimmy,cricket_bat).*

…and many others!

4. *travels(john,london,train).*

5. *takes(bill,jane,cheese,edam).*

Later we will see that we can tidy this up by writing

takes(bill,jane,cheese(edam))

where cheese(edam) is a legitimate Prolog term.

Σύνολο Γεγονότων

- Παρόμοια το *plus(2,3,5)* εκφράζει την σχέση του 2 συν 3, η οποία είναι το 5. Η γνωστή σχέση, *πρόσθεση (plus)* μπορεί να πραγματοποιηθεί διαμέσου ενός συνόλου γεγονότων τα οποία καθορίζουν τον πίνακα της πρόσθεσης.

- Ένα αρχικό τμήμα αυτού του πίνακα είναι:

  plus(0,0,0). plus(0,1,1). plus(0,2,2). plus(0,3,3).
  plus(1,0,1). plus(1,1,2). plus(1,2,3). plus(1,3,4).

- Η επέκταση αυτού του πίνακα (ο οποίος δεν καταγράφεται εδώ ολόκληρος), με τις υπόλοιπες σχέσεις, π.χ. *plus(2,3,5)*., *plus(2,2,4)*., κ.τ.λ., θα θεωρηθεί ο ορισμός της σχέσης *πρόσθεση (plus)*.

Άσκηση

- Represent each of these statements as a set of Prolog clauses.

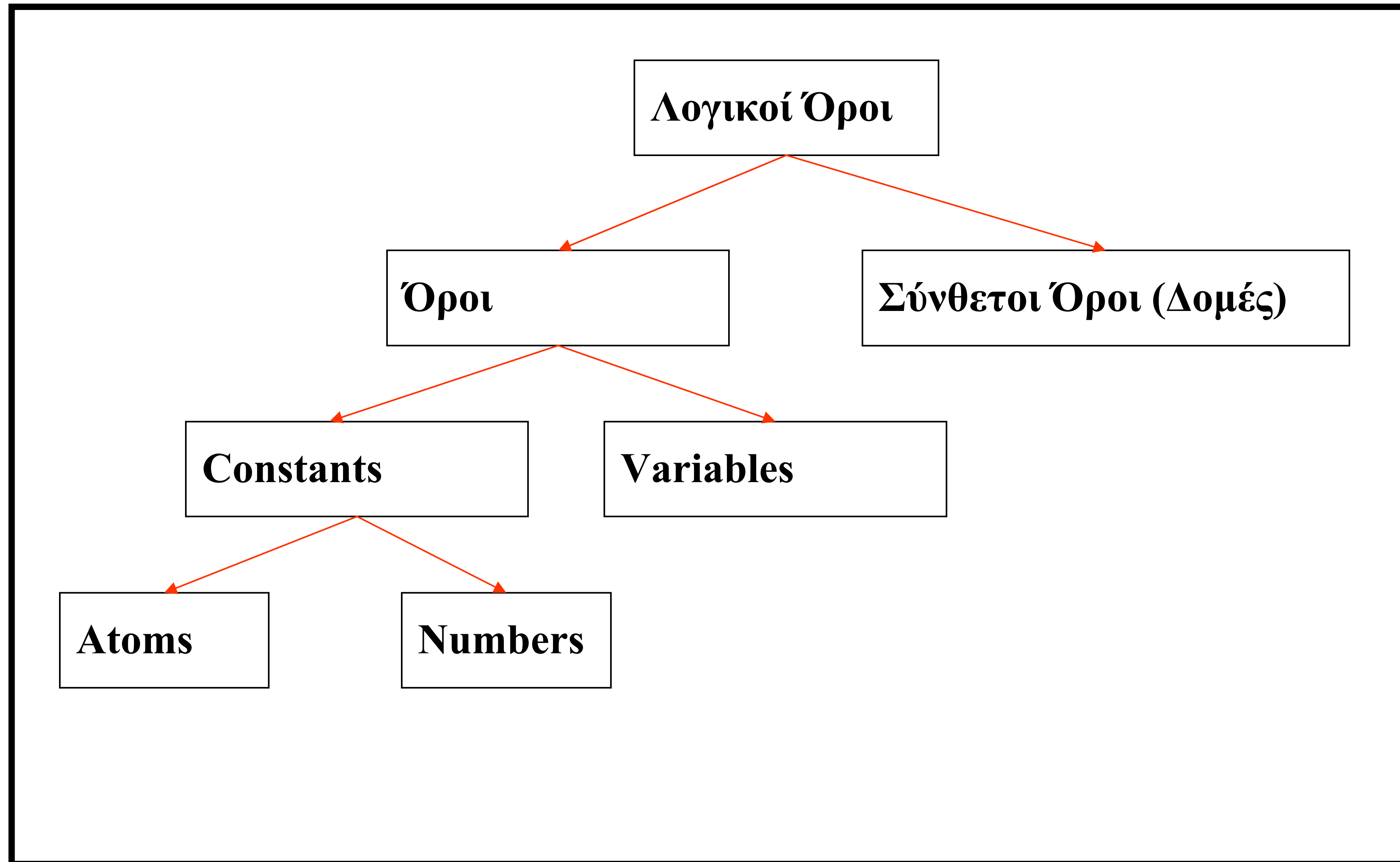
  1. bill only eats chocolate, bananas or cheese.
  2. the square root of 16 is 4 or -4.
  3. wales, ireland and scotland are all countries.

1. *eats(bill,chocolate).*
   eats(bill,bananas).
   eats(bill,cheese).

2. *square root(16,-4).*
   square root(16,4).

3. *country(wales).*
   country(ireland).
   country(scotland).

Δομές Δεδομένων στην Prolog



Σταθερές Prolog (Constants)

- If we have:

  loves(jane,jim).

  - then jane and jim refer to specific objects. Both jane and jim are constants or atoms.
  - Also, “loves" happens to be an atom too because it refers to a specific relationship.

- Generally speaking, if a string of characters starts with a **lower case letter**, the DEC-10 family of Prologs assume that the entity is an atom.

Σταθερές Prolog (Constants)

- There are constants other than atoms, including integers and real numbers.

- A constant is an atom or a number. A number is an integer or a real number

- The rules for an atom are quite complicated:

quoted item	'anything but the single quote character'
word	lower case letter followed by any letter, digit or (underscore)
symbol	any number of f+, -, *, /, n, ^, <, >, =, ', ~, :, ., ?, @, #, $, &g
special item	any of f [], fg, ;, !, %g

- So the following are all atoms:

likes_chocolate, fooX23, ++*++, ::=, 'What Ho!'

Σταθερές Prolog (Constants)

- No predicate may be a variable!!!
  - That is, we cannot have X(jane,jim) as representing the fact that jane and jim are related in some unknown way.

Αριθμοί στην Prolog

- Numbers used in Prolog include integer numbers and real numbers.

- The syntax of integers is simple, as illustrated by the following examples:

  1 1313 0 -97

- The treatment of real numbers depends on the implementation of Prolog. We will assume the simple syntax of numbers, as shown by the following examples:

  3.14 -0.0035 100.2

- Real numbers are not used very much in typical Prolog programming. The reason for this is that Prolog is primarily a language for symbolic, non-numeric computation, as opposed to number crunching oriented languages such as Fortran.

Λογικό Πρόγραμμα

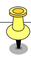
- Ένα πεπερασμένο σύνολο από γεγονότα συγκροτεί ένα *πρόγραμμα (program)*. Αυτή είναι η πιο απλή μορφή ενός λογικού προγράμματος.

- Ένα σύνολο από γεγονότα είναι επίσης η περιγραφή μιας κατάστασης. Αυτή η επίγνωση είναι η βάση για τον προγραμματισμό βάσης δεδομένων (που θα συζητηθεί στη συνέχεια.

- Ένα παράδειγμα βάσης δεδομένων των σχέσεων μιας οικογένειας από την Βίβλο είναι δοσμένο στο Πρόγραμμα 8.1 (επόμενη διαφάνεια)

- Τα κατηγορήματα:
  - πατέρας (father),
  - μητέρα (mother), αρσενικό (male) και
  - θηλυκό (female)

  εκφράζουν ολοφάνερα τις σχέσεις.



Πρόγραμμα 8.1: Μια βάση δεδομένων με οικογενειακές σχέσεις από την Βίβλο

```
father(terach,abraham).
father(terach,nachor).
father(terach,haran).
father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).

mother(sarah,isaac).

male(terach).
male(abraham).
male(nachor).
male(haran).
male(isaac).
male(lot).

female(sarah).
female(milcah).
female(yiscah).
```

Απλές Ερωτήσεις

- Η δεύτερη μορφή μιας δήλωσης στον λογικό προγραμματισμό είναι η *ερώτηση (query)*.

- Οι ερωτήσεις είναι το μέσο απόδοσης πληροφοριών από ένα λογικό πρόγραμμα.

- Με μια ερώτηση, ρωτάμε αν ισχύει κάποιος συσχετισμός ανάμεσα σε αντικείμενα.

- Για παράδειγμα η ερώτηση

  father(abraham,isaac)?

  ρωτάει, εάν η σχέση πατέρας (father) ισχύει ανάμεσα στον abraham και στον isaac.

  Με δεδομένο τα γεγονότα του προγράμματος 8.1, η απάντηση είναι “yes”.

Απλές Ερωτήσεις

- Συντακτικά, οι ερωτήσεις και τα γεγονότα μπορούν να διαφοροποιηθούν από τα συμφραζόμενα.
  - Η τελεία υποδεικνύει ένα γεγονός "*P*.", ενώ
  - το ερωτηματικό υποδεικνύει μια ερώτηση ("P? ").
- ***Στόχο (goal)*** ονομάζουμε την οντότητα χωρίς την τελεία ή το ερωτηματικό.
  - Ένα γεγονός "*A*." δηλώνει ότι ο στόχος *A* είναι αληθής.
  - Η ερώτηση "*A?*" ρωτάει εάν ο στόχος είναι αληθής.
- Μια απλή ερώτηση αποτελείται από έναν απλό στόχο.
- Απαντώντας σε μια ερώτηση ***Q?*** αναφορικά με ένα πρόγραμμα ***P*** είναι σαν να ρωτάμε αν ο στόχος ***Q*** αποδεικνύεται από το πρόγραμμα ***P*** ή αν ο στόχος ***Q*** είναι λογικό συμπέρασμα του προγράμματος ***P***.

Απλές Ερωτήσεις

- Τα λογικά συμπεράσματα έχουν παραχθεί με την εφαρμογή συμπερασματικών κανόνων.
- Ο πιο απλός κανόνας συμπεράσματος είναι η ***ταυτότητα (identity)***: από το P συμπεραίνουμε P.
- Μια ερώτηση είναι ένα λογικό συμπέρασμα ενός ίδιου γεγονότος.

Απλές Ερωτήσεις

- Για να απαντήσουμε σε απλές ερωτήσεις εργαζόμαστε ως εξής :
  - Ψάχνουμε για ένα γεγονός μέσα στο πρόγραμμα, το οποίο συμπεραίνει την ερώτηση. Αν βρεθεί ένα γεγονός ίδιο με την ερώτηση, η απάντηση είναι ναι (yes).
  - Η απάντηση όχι (no) δίνεται αν δεν βρεθεί ένα γεγονός ίδιο με την ερώτηση, γιατί τότε το γεγονός δεν είναι λογικό συμπέρασμα του προγράμματος. Αυτή η απάντηση δεν αντανακλά στην αλήθεια της ερώτησης. Απλώς λέει ότι αποτύχαμε να αποδείξουμε την ερώτηση αναφορικά με το πρόγραμμα αυτό.

- Αναφορικά με το Πρόγραμμα 8.1:
  - Και οι δύο ερωτήσεις female(abraham)? και plus(1,1,2)? θα απαντήσουν όχι (no)

Παραδείγματα απλών ερωτήσεων σε Prolog

- Δεδομένα (facts).

child (οιδίπους,ιοκάστη). *child (οιδίπους,λάιος).*
child (αντιγόνη,ιοκάστη). *child (αντιγόνη, οιδίπους).*
child (ετεοκλής,ιοκάστη). *child (ετεοκλής, οιδίπους).*
child (πολυνίκης,ιοκάστη). *child (πολυνίκης,λάιος).*
male (λάιος). *male (οιδίπους).*
male (ετεοκλής). *male (πολυνίκης).*
female (αντιγόνη). *female (ιοκάστη).*
married (λάιος, ιοκάστη). *married (ιοκάστη, λάιος).*
married (οιδίπους, ιοκάστη). *married (ιοκάστη, οιδίπους).*

- Παραδείγματα ερωτήσεων:

?- child (οιδίπους, λάιος).
YES

?- child (αντιγόνη, ετεοκλής).
NO

Λογικές Μεταβλητές

- Μια λογική μεταβλητή αντιπροσωπεύει μια ακαθόριστη μονάδα και χρησιμοποιείται ανάλογα.

- Μελετείστε την χρήση της στις ερωτήσεις. Υποθέτουμε ότι θέλουμε να γνωρίσουμε τίνος πατέρας ήταν ο *abraham*. Ένας τρόπος είναι να ρωτήσουμε μια σειρά ερωτήσεων,

  father(abraham,lot)?, *father(abraham,milcah)?,* κ.τ.λ.

  μέχρι να δοθεί μια απάντηση ναι (yes).

- Μια μεταβλητή επιτρέπει έναν καλύτερο τρόπο για να εκφράσει την ερώτηση *father(abraham,X)?* της οποίας η απάντηση είναι *X* = *isaac*.

Λογικές Μεταβλητές

- Οι μεταβλητές είναι ένα είδος περίληψης πολλών ερωτήσεων.
  - Μια ερώτηση περιέχοντας μια μεταβλητή ρωτάει αν υπάρχει μια τιμή της μεταβλητής που να κάνει την ερώτηση ένα λογικό συμπέρασμα του προγράμματος.

- Οι μεταβλητές στον λογικό προγραμματισμό συμπεριφέρονται διαφορετικά από τις μεταβλητές στις συμβατικές γλώσσες προγραμματισμού.
  - Αυτές αντιπροσωπεύουν περισσότερο μια ακαθόριστη αλλά μοναδική οντότητα, παρά μια θέση μνήμης.

Λογικές Μεταβλητές στην Prolog

- Variables usually start with a capital letter.

- The only interesting exception is the special anonymous variable written _ and pronounced underscore". In the rule:

```
process(X,Y):-
        generate(_,Z),
        test(_,Z),
        evaluate(Z,Y).
```

the underscores refer to different unnamed variables.

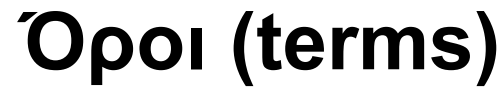
Παράδειγμα μεταβλητής _

❖ Note that, in the clause,

```
know_both_parents(X):-
        mother(_,X),
        father(_,X).
```

- the underscores do not refer to the same object. The reading is roughly that we know both the parents of X if someone(name unimportant) is the mother of X and someone else (unimportant) is the father".

❖ Note that Prolog regards the two occurrences of the anonymous variable in the above as different variables.



Όροι (terms)

- Έχοντας παρουσιάσει τις μεταβλητές, μπορούμε να ορίσουμε τους λογικούς ***όρους (terms)***, τη μοναδική δομή δεδομένων στα λογικά προγράμματα.

- Ο ορισμός είναι επαγωγικός.
  - Οι σταθερές και οι μεταβλητές είναι όροι.
  - Επίσης όροι είναι και οι σύνθετοι όροι ή οι δομές.

Σύνθετοι Όροι (compound terms)

- Ενας σύνθετος όρος περιέχει έναν ***συναρτητή (functor)*** (ονομάζεται ο κύριος συναρτητής του όρου) και μια σειρά από ένα ή περισσότερα ορίσματα, τα οποία είναι όροι.

- Ενας ***συναρτητής (functor)*** ή συναρτησιακό σύμβολο χαρακτηρίζεται από το όνομά του, το οποίο είναι ένα άτομο και από την πολλαπλότητα ή τον αριθμό των ορισμάτων του.

- Συντακτικά οι σύνθετοι όροι έχουν την μορφή $f(t_1, t_2, \ldots, t_n)$ όπου
  - ο συναρτητής έχει όνομα *f*,
  - είναι *πολλαπλότητας n*, και
  - τα t_i είναι τα ορίσματα.

- Παραδείγματα των σύνθετων όρων περιλαμβάνουν τα:
  - *hot(milk), name(john,doe), list(a,like(b,nil)), foo(X) και tree(tree(nil,3,nil),5,R)*.

Σύνθετοι Όροι (compound terms)

Κι άλλα παραδείγματα:

- The date can be viewed as a structure with three components:
  day, month, year.

- A suitable functor for our example is date.

- Then the date 1st May 1,983 can be written as:
  date(1, may, 1983)

Σύνθετοι Όροι (compound terms)

- All the components in this example are constants (two integers and one atom).
- Components can also be variables or other structures. Any day in May can be represented by the structure:

  date(Day, may, 1983)

- Note that Day is a variable and can be instantiated to any object at some later point in the execution.
- This method for data structuring is simple and powerful. It is one of the reasons why Prolog is so naturally applied to problems that involve symbolic manipulation.

Παραδείγματα Σύνθετων Όρων

happy(fred)

principal functor = happy

1st argument = a constant (atom)

sum(5,X)

principal functor = sum

1st argument = constant (integer)

2nd argument = variable

not(happy(woman))

principal functor = not

1st argument = compound term

- Nesting compound terms may be of use to the programmer.

- For example, the clause

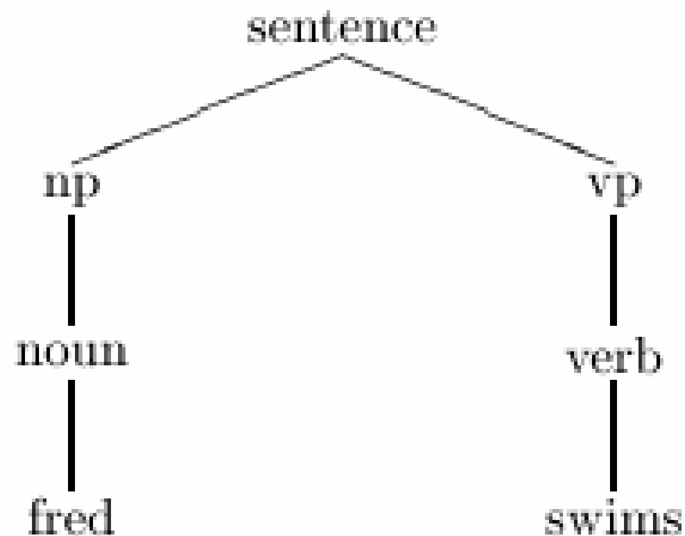
  fact(fred,10000).

  is not as informative as

  fact(name(fred),salary(10000)).

Αναπαράσταση Σύνθετων Όροι ως Δένδρα

sentence(np(noun(fred)),vp(verb(swims)))



Αντικατάσταση (substitution)

- Ένα παράδειγμα μιας αντικατάστασης που αποτελεί ένα μοναδικό ζευγάρι είναι $\{X = isaac\}$.

- Αντικαταστάσεις μπορούν να εφαρμοστούν στους όρους.
Το αποτέλεσμα από την εφαρμογή μιας αντικατάστασης θ σε έναν όρο A, που φαίνεται από το $A\theta$, είναι ο όρος ο οποίος πετυχαίνεται από την αντικατάσταση κάθε εμφάνισης του X με το t στο A, για κάθε ζευγάρι $X = t$ μέσα στο θ.

- Το αποτέλεσμα της εφαρμογής $\{X = isaac\}$ στον όρο *father(abraham,X)* είναι ο όρος *father(abraham,isaac)*.

Στιγμιότυπο (instance)

- Ο στόχος *father(abraham,isaac)* είναι ένα στιγμιότυπο του *father(abraham,X)* μέσα από την αντικατάσταση *{X = isaac}*.

- Παρόμοια ο *mother(sarah,isaac)* είναι ένα στιγμιότυπο του *mother(X,Y)* μέσα από την αντικατάσταση *{X = sarah, Y = isaac}*.

Υπαρξιακές Ερωτήσεις

- Αν εμφανίζονται μεταβλητές μέσα στις ερωτήσεις, τότε υπονοείται ότι αυτές βρίσκονται στην εμβέλεια ενός υπαρξιακού ποσοδείκτη.

- Τέτοιες ερωτήσεις θα μπορούσαμε να τις ονομάζουμε '**υπαρξιακές ερωτήσεις**' **(existential queries)**.

- Αυτό σημαίνει, ότι η ερώτηση *father(abraham,X)?* διαβάζεται: "Υπάρχει ένα *X* τέτοιο που ο *abraham* να είναι ο πατέρας του *X;*".

Υπαρξιακές Ερωτήσεις

- Ο δεύτερος κανόνας συμπεράσματος που παρουσιάζουμε είναι η **γενίκευση (generalization)**.

- Το γεγονός *father(abraham,isaac)* υπονοεί ότι υπάρχει ένα *X* τέτοιο ώστε το *father(abraham,X)* να είναι αληθές, δηλαδή *X* = *isaac*.

Υπαρξιακές Ερωτήσεις

- Λειτουργικά, για να απαντήσουμε μια υπαρκτή, μη στοιχειώδη ερώτηση χρησιμοποιώντας ένα πρόγραμμα με γεγονότα, βρίσκουμε ένα γεγονός το οποίο είναι ένα στιγμιότυπο της ερώτησης.

- Η απάντηση, ή η *λύση*, είναι αυτό το στιγμιότυπο.

- Η απάντηση είναι *όχι (no)* αν δεν υπάρχει κατάλληλο γεγονός στο πρόγραμμα.

Υπαρξιακές Ερωτήσεις

- Γενικά, μια υπαρκτή ερώτηση ίσως έχει αρκετές λύσεις.

- Το Πρόγραμμα 8.1 δείχνει ότι ο *Haran* είναι ο πατέρας τριών παιδιών. Κατά συνέπεια η ερώτηση *father(haran,X)?* έχει τις λύσεις *{X = lot}*, *{X = milcah}*, *{X = yiscah}*.

- Άλλη ερώτηση με πολλαπλές λύσεις είναι η *plus(X,Y,4)?* η οποία βρίσκει αριθμούς των οποίων το άθροισμα ισούται με το 4. Οι λύσεις είναι, για παράδειγμα, *{X = 0, Y = 4} και {X = 1, Y = 3}*.
  - Σημειώστε ότι οι διαφορετικές μεταβλητές X και Y ανταποκρίνονται σε διαφορετικά (πιθανόν) αντικείμενα.

- Μια ενδιαφέρουσα παραλλαγή της τελευταίας ερώτησης είναι η *plus(X,X,4)?* η οποία εμμένει στο ότι οι δύο αριθμοί των οποίων το άθροισμα ισούται με το 4, είναι ίδιοι. Αυτή έχει μοναδική απάντηση *{X = 2}*.

Παραδείγματα Υπαρξιακών Ερωτήσεων στην Prolog

❖ Δεδομένα (facts).

child (οιδίπους,ιοκάστη). child (οιδίπους,λάιος).
child (αντιγόνη,ιοκάστη). child (αντιγόνη, οιδίπους).
child (ετεοκλής,ιοκάστη). child (ετεοκλής, οιδίπους).
child (πολυνίκης,ιοκάστη). child (πολυνίκης,λάιος).
male (λάιος). male (οιδίπους).
male (ετεοκλής). male (πολυνίκης).
female (αντιγόνη). female (ιοκάστη).
married (λάιος, ιοκάστη). married (ιοκάστη, λάιος).
married (οιδίπους, ιοκάστη). married (ιοκάστη, οιδίπους).

❖ Παραδείγματα ερωτήσεων:

?- child (X, ιοκάστη).

X= οιδίπους

?- child (αντιγόνη,X).

X= ιοκάστη;

X= οιδίπους

?- child (X,Y).

X= οιδίπους

Y= ιοκάστη;

X= οιδίπους

Y= λάιος;

…

Καθολικά Γεγονότα

- Οι μεταβλητές είναι επίσης χρήσιμες στα γεγονότα.
  - Υπέθεσε όλα τα Βιβλικά πρόσωπα, που συμπαθούν τον Pomegranates. Αντί να συμπεριλάβουμε στο πρόγραμμα ένα κατάλληλο γεγονός για κάθε μονάδα:

    likes(abracham, pomegranates)

    likes(sarah, pomegranates).

    ...

    ένα γεγονός συμπάθειας *likes(X, pomegranates)* μπορεί να τα πει όλα. Κάνοντας χρήση αυτής της μεθόδου, *οι μεταβλητές είναι ένα μέσο περίληψης πολλών γεγονότων*.

Καθολικά Γεγονότα

- Το γεγονός *time(0, X, 0)* συνοψίζει όλα τα γεγονότα τα οποία ξεκινούν με το ότι 0 φορές κάποιος αριθμός ισούται με το 0.

- Μεταβλητές μέσα σε γεγονότα βρίσκονται στην εμβέλεια ενός καθολικού ποσοδείκτη, το οποίο σημαίνει διαισθητικά ότι το γεγονός *likes(X, pomegranates)* δηλώνει ότι για όλα τα *X*, το *X* συμπάθεια τον pomegranates. Λογικά, από ένα τέτοιο γεγονός, κάποιος μπορεί να συνάγει κάθε στιγμιότυπο από αυτό. Για παράδειγμα, από *likes(X, pomegranates)* συμπεραίνουμε *likes(abraham, pomegranates)*.

- Αυτός είναι ο τρίτος συμπερασματικός κανόνας, ο οποίος ονομάζεται ***αρχικοποίηση (instantiation).***

Καθολικά Γεγονότα

- Δύο ακαθόριστα αντικείμενα, τα οποία προσδιορίζονται από μεταβλητές, μπορούν εξαναγκασμένα να είναι όμοια, χρησιμοποιώντας το ίδιο όνομα μεταβλητής.

- Το γεγονός *plus(0,X,X)* εκφράζει ότι το *0* είναι μια αριστερή ταυτότητα της πρόσθεσης.
  - Αυτό ερμηνεύεται πως για όλες τις τιμές του *X, 0* συν *X* ισούται με *X*.
  - Μια παρόμοια χρήση γίνεται όταν μεταφράζουμε την Αγγλική δήλωση "καθένας συμπαθεί τον εαυτό του" με το *likes(X,X)*.

- Η απάντηση σε στοιχειώδη ερώτηση που βασίζεται σε ένα γεγονός το οποίο περιέχει μεταβλητές είναι προφανής.
  - Ψάξτε για ένα γεγονός για το οποίο η ερώτηση είναι ένα στιγμιότυπο.
  - Για παράδειγμα, η απάντηση στο *plus(0,2,2)?* η οποία είναι *ναι (yes),* βασίζεται πάνω στο γεγονός *plus(0,X,X)*.

Καθολικά Γεγονότα

- Η απάντηση σε μια μη στοιχειώδη ερώτηση χρησιμοποιώντας ένα μη στοιχειώδες γεγονός εμπεριέχει ένα νέο ορισμό: ένα κοινό στιγμιότυπο με δύο όρους.

- **Ορισμός**: Το *C* είναι ένα κοινό στιγμιότυπο του *A* και του *B* αν αυτό είναι ένα στιγμιότυπο του *B*. Με άλλα λόγια, αν υπάρχουν αντικαταστάσεις θ_1 και θ_2, τότε η $C = A\theta_1$, είναι συντακτικώς ταυτόσημη με τη $B\theta_2$.

- Για παράδειγμα, οι στόχοι *plus(0,1,Y)* και *plus(0,X,X)* έχουν ένα κοινό στιγμιότυπο *plus(0,1,1)*. Εφαρμόζοντας την αντικατάσταση *{Y = 1}* στο *plus(0,1,Y)* και την αντικατάσταση *{X = 1}* στο *plus(0,X,X)* και οι δύο παράγουν το *plus(0,1,1)*.

- Ως επί το πλείστον, προκειμένου να απαντήσουμε μια ερώτηση χρησιμοποιώντας ένα γεγονός, ψάχνουμε για ένα κοινό στιγμιότυπο της ερώτησης και του γεγονότος. Η απάντηση είναι το κοινό στιγμιότυπο αν υπάρχει, διαφορετικά η απάντηση είναι *όχι (no)*.

Συζευκτικές Ερωτήσεις

- Μια σημαντική προέκταση στο είδος των ερωτήσεων, οι οποίες συζητήθηκαν μέχρι εδώ, είναι οι συζευκτικές ερωτήσεις.

- Συζευκτική ερώτηση είναι μια σύζευξη στόχων που εκφράζονται σε μια ερώτηση, για παράδειγμα *father(terach,X), father(X,Y)?* ή γενικά, *Q1, . . ., Qn?*.

- Οι απλές ερωτήσεις είναι μια ειδική περίπτωση συζευκτικών ερωτήσεων όταν υπάρχει ένας απλός στόχος.

Συζευκτικές Ερωτήσεις

- Στις πιο απλές συζευκτικές ερωτήσεις όλοι οι στόχοι είναι στοιχειώδεις.
  - Για παράδειγμα *father(abraham,isaac), male(lot)*.
    Η απάντηση σε αυτή την ερώτηση χρησιμοποιώντας το Πρόγραμμα 8.1 είναι *ναι (yes)* καθώς και οι δύο στόχοι αποδεικνύονται από το πρόγραμμα.

- Οι συζευκτικές ερωτήσεις είναι ενδιαφέρουσες όταν υπάρχει μια ή περισσότερες διαμοιραζόμενες μεταβλητές, μεταβλητές που εμφανίζονται μέσα σε δύο διαφορετικούς στόχους ερωτήσεων.
  - Ένα παράδειγμα είναι η ερώτηση *father(haran,X), male(X)?*.
    Η έκταση μιας μεταβλητής σε μια συζευκτική ερώτηση, είναι όλη η σύζευξη.
    Κατά συνέπεια η ερώτηση *p(X), q(X)?* εκφράζει:
    "Υπάρχει ένα *X* έτσι ώστε να ισχύει και *p(X),* όσο και *q(X)?*".

- Όπως στις απλές ερωτήσεις, οι μεταβλητές στις συζευκτικές ερωτήσεις βρίσκονται στην εμβέλεια ενός ή περισσοτέρων υπαρξιακών ποσοδεικτών.

Συζευκτικές Ερωτήσεις

- Οι διαμοιραζόμενες μεταβλητές χρησιμοποιούνται σαν ένα μέσο εξαναγκασμού μιας απλής ερώτησης, έτσι ώστε να περιορίσει την έκταση μιας μεταβλητής.

- Έχουμε κιόλας δει ένα παράδειγμα με την ερώτηση *plus(X,X,4)?* όπου το 4 είναι η λύση των αριθμών που θα προστεθούν, η οποία περιοριζόταν στους αριθμούς που ήταν ίδιοι.

- Θεωρούμε την ερώτηση father(haram,X), male(X)?.
  - Εδώ οι λύσεις της ερώτησης *father(haram,X)?* είναι περιορισμένες στα παιδιά τα οποία είναι αρσενικά.
  - Το Πρόγραμμα 8.1 δείχνει ότι υπάρχει μόνο μια λύση, *{X = lot}*.
  - Αντίστοιχα, αυτή η ερώτηση μπορεί να εξεταστεί ως περιοριστική λύση στην ερώτηση *male(X)?,* στις μονάδες οι οποίες έχουν τον *Haran* για πατέρα.

Συζευκτικές Ερωτήσεις

- Μια ελαφρώς διαφορετική χρήση μιας διαμοιραζόμενης μεταβλητής μπορεί να γίνει αντιληπτή στην ερώτηση *father(terach,X), father(X,Y)?*.

  - Από την μια πλευρά περιορίζουμε τους υιούς του *terach* σε εκείνους οι οποίοι είναι οι ίδιοι πατέρες.

  - Από την άλλη πλευρά θεωρούμε μονάδες *Y*, των οποίων οι πατέρες είναι υιοί του *terach*.

- Υπάρχουν αρκετές λύσεις, για παράδειγμα $\{X = abraham,\ Y = isaac\}$, και $\{X = haran,\ Y = lot\}$.

Παραδείγματα Συζευκτικών Ερωτήσεων στην Prolog

parent(pam, bob).

parent(tom, bob). *parent(tom, liz).*

parent(bob, ann). *parent(bob, pat).*

parent(pat, jim).

?- parent(X, Y).

X = pam

Y = bob;

X = tom

Y = bob;

X = tom

Y = liz;

…

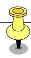
- Our example program can be asked still more complicated questions like:

  Who is a grandparent of Jim?

- As our program does not directly know the grandparent relation this query has to be broken down into two steps:

  (1) Who is a parent of Jim? Assume that this is some Y.

  (2) Who is a parent of Y? Assume that this is someX .



- Such a composed query is written in Prolog as a sequence of two simple ones:

  ?- parent(Y, jim), parent(X, Y).

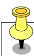
  The answer will be:

  X = bob

  Y = pat

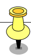
- If we change the order of the two goals the logical meaning remains the same:

  ?- parent(X, Y), parent(Y, jim).



- In a similar way we can ask: Who are Tom's grandchildren?

```
?- parent(tom, X), parent(X, Y).

X = bob
Y = ann;
X = bob
Y = pat
```



- Yet another question could be: Do Ann and Pat have a common parent? This can be expressed again in two steps:

  (1) Who is a parent, X, of Ann?

  (2) Is (this same) X a parent of Pat?

  The corresponding question to Prolog is then:

  ?- parent(X, ann), parent(X, pat).

  The answer is:

  X = bob

Ασκήσεις με Ερωτήσεις

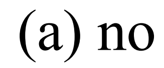
- Assuming the parent relation as defined in the previous slides, what will be Prolog's answers to the following questions?

  (a) parent(jim, X).

  (b) parent(X, jim).

  (c) parent(pam, X), parent(X, pat).

  (d) parent(pam, X), parent(X, Y), parent(Y, jim).



(a) no

(b) X = pat

(c) X = bob

(d) X = bob, Y = p a t

❖ Formulate in Prolog the following questions about the parent relation:

(a) Who is Pat's parent?

(b) Does Liz have a child?

(c) Who is Pat's grandparent

(b) ?- parent(liz, X).

(c) ?- parent(Y, pat), parent(X, Y).

Κανόνες

- Ενδιαφέρουσες συζευκτικές ερωτήσεις καθορίζουν σχέσεις ανεξάρτητα άλλων παραγόντων.

  - Η ερώτηση *father(haran,X), male(X)?* ρωτάει για έναν γιο του *Haran*.

  - Η ερώτηση *father(haran,X), father(X,Y)?* ρωτάει για εγγόνια του *Terach*.

- Αυτό μας φέρνει στην τρίτη και πιο σημαντική δήλωση στον λογικό προγραμματισμό, ο ***κανόνας (rule)***, ο οποίος μας καθιστά ικανούς στο να καθορίσουμε νέες σχέσεις στους όρους των υφιστάμενων σχέσεων.

Κανόνες

- Οι κανόνες είναι δηλώσεις της μορφής:

  $$A \leftarrow B_1, B_2, \ldots, B_n$$

  όπου $n \geq 0$.

  Το A είναι η κεφαλή του κανόνα και τα B_i είναι το σώμα του.

  Τόσο το A, όσο και τα B_i είναι στόχοι.
- Οι κανόνες, τα γεγονότα και οι ερωτήσεις καλούνται επίσης **φράσεις Horn (Horn clauses)** ή εν συντομία **φράσεις**.
  - Σημειώστε ότι ένα γεγονός είναι απλώς μια ειδική περίπτωση ενός κανόνα όταν $n = 0$.
- Τα γεγονότα καλούνται επίσης μοναδιαίες φράσεις (unit clauses).
- Υπάρχει επίσης ένα ειδικό όνομα για φράσεις με έναν στόχο μέσα στο σώμα, δηλαδή όταν $n = 1$. Μια τέτοια φράση καλείται **επαναληπτική φράση (iterative clause)**.
- Όσο για τις μεταβλητές που εμφανίζονται μέσα σε κανόνες, θεωρούνται ότι βρίσκονται στην εμβέλεια καθολικών ποσοδεικτών που η έκτασή τους είναι όλος ο κανόνας.

Κανόνες

- Ένας κανόνας ο οποίος εκφράζει την σχέση υιός (son) είναι:

$$son(X,Y) \leftarrow father(Y,X),\ male(X).$$

- Ομοίως μπορεί να καθορίσει ένας κανόνας για την σχέση της κόρης:

$$daughter(X,Y) \leftarrow father(X,Y),\ female(X).$$

- Ένας κανόνας για την σχέση παππού είναι ο εξής:

$$grandfather(X,Z) \leftarrow father(X,Y),\ father(Y,Z).$$

Κανόνες

Οι κανόνες μπορούν να εξεταστούν με <u>δύο</u> τρόπους.

- Πρώτα, είναι ένα μέσο με το οποίο εκφράζονται νέες ή σύνθετες ερωτήσεις χρησιμοποιώντας απλές ερωτήσεις.

- Μια ερώτηση *son(X,haran)?* στο πρόγραμμα το οποίο περιλαμβάνει τον προηγούμενο κανόνα είναι μεταφρασμένο στην ερώτηση *father(haran,X), male(X)?* σύμφωνα με τον κανόνα και λύνεται όπως προηγουμένως.

- Η ερμηνεία κανόνων με αυτό τον τρόπο, αποτελεί την **διαδικαστική ερμηνεία (procedural meaning)**.
  - Η διαδικαστική ερμηνεία για τον κανόνα του παππού είναι:

    "Να απαντήσουμε στην ερώτηση: *είναι ο X ο παππούς του Y?,* απαντώντας τη συζευκτική ερώτηση: *είναι ο X ο πατέρας του Z και ο Z ο πατέρας του Y*".

Κανόνες

- Η δεύτερη άποψη για τους κανόνες βγαίνει από την μετάφραση του κανόνα όπως ένα λογικό αξίωμα.

- Το ανάποδο βέλος "←" χρησιμοποιείται για να δείξει λογική συνέπεια.

- Ο κανόνας *υιός (son)* ερμηνεύεται ως:
  - Ο *X* είναι υιός του *Y*, αν ο *Y* είναι ο πατέρας του *X* και ο *X* είναι άρρεν".

- Κάτω από αυτό το πρίσμα οι κανόνες είναι ένα μέσο καθορισμού νέων ή σύνθετων σχέσεων χρησιμοποιώντας άλλες, απλές σχέσεις.

- Η ερμηνεία αυτή ενός κανόνα είναι γνωστή ως η **δηλωτική ερμηνεία (declarative meaning).**

Κανόνες

- Το να ενσωματώσουμε κανόνες συμπερασματολογίας μέσα στο πλαίσιο της λογικής απόδειξης στα πλαίσια του λογικού προγραμματισμού, χρειαζόμαστε την αρχή του Γενικευμένου Modus Ponens (M.P.).

- Ορισμός: Η αρχή του Γενικευμένου Modus Ponens (M.P.) λέει ότι από τον κανόνα

  $R = (A \leftarrow B1, B2, \ldots, Bn.)$

  και τα γεγονότα B1'. B2'. ... Bn'.

  το A' μπορεί να καταλήξει συμπέρασμα, αν

  $A' \leftarrow B1', B2', \ldots, Bn'.$

  είναι ένα στιγμιότυπο του *R*.

Κανόνες

- Ένα θέμα που σχετίζεται με τους κανόνες είναι το κατά πόσο είναι πλήρεις. Κατά πόσο δηλαδή εκφράζουν με συνέπεια αυτό που πραγματικά θέλουμε να πούμε.

- Για παράδειγμα, ο κανόνας που έχει δοθεί για το *υιό (son)* είναι σωστός, αλλά είναι ελλιπής ο ορισμός της σχέσης.
  Έτσι, δεν μπορούμε να συμπεράνουμε ότι ο *Isaac* είναι υιός της *Sarah*.

- Αυτό το οποίο λείπει, είναι ότι το παιδί μπορεί να είναι υιός μιας μητέρας όπως υιός ενός πατέρα. Ένας νέος κανόνας που εκφράζει την σχέση μπορεί να προστεθεί, συγκεκριμένα:

$$son(X,Y) \leftarrow mother(Y,X),\ male(X)$$

Κανόνες

❖ Παρόμοια, για να καθορίσουμε την συγγένεια *παππούς ή γιαγιά (grandparent)* θα μπορούσαμε να πάρουμε τέσσερις κανόνες για να συμπεριλάβουμε και τις δύο περιπτώσεις *πατέρα (father)* και *μητέρας (mother)*:

grandparent(X,Z) ← father(X,Y), father(Y,Z).
grandparent(X,Z) ← father(X,Y), mother(Y,Z).
grandparent(X,Z) ← mother(X,Y), father(Y,Z).
grandparent(X,Z) ← mother(X,Y), mother(Y,Z).

Κανόνες

Υπάρχει ένας καλύτερος, πιο συνεπτυγμένος, τρόπος για να εκφράσουμε αυτούς τους κανόνες.

- Χρειαζόμαστε να ορίσουμε την βοηθητική σχέση, *γονέας (parent)* όπως είναι ένας πατέρας ή μια μητέρα.
- Μέρος αυτής της δυνατότητας του λογικού προγραμματισμού είναι το να αποφανθεί στο τι ενδιάμεσα κατηγορήματα να καθορίσει για να επιτύχει μια πλήρη, κομψή δημιουργία ενός αξιώματος μιας σχέσης.
- Αυτοί οι κανόνες οι οποίοι ορίζουν το *γονέα (parent)* είναι ευθείς, συλλαμβάνοντας τον ορισμό ενός γονέα ως πατέρα ή μητέρα.

Κανόνες

parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).

Οι κανόνες για τον *υιό (son)* και τον *γονέα (grandparent)* είναι τώρα, περισσότερο ειδικευμένοι:

son(X,Y) ← parent(Y,X), male(X).
grandparent(X,Y) ← parent(X,Z), parent(Z,Y).

- Μια συλλογή κανόνων με το ίδιο κατηγόρημα στην κεφαλή, όπως το ζευγάρι των κανόνων του γονέα, ονομάζεται *διαδικασία (procedure)*.

- Θα δούμε αργότερα ότι κάτω από λογική μεταγλώττιση αυτών των κανόνων στην Prolog, τέτοια συλλογή κανόνων είναι όντως το ανάλογο των διαδικασιών ή υπορουτινών μέσα σε συμβατικές γλώσσες προγραμματισμού.

Κανόνες στην Prolog

Ο X είναι εγγονός του Y αν ο X είναι παιδί κάποιου Z και ο/η Z είναι παιδί του Y.

grandchild (X,Y) :- child (X,Z), child (Z,Y).

head (κεφαλή) body (σώμα)

Κανόνες και Συζεύξεις

- A man is happy if he is rich and famous might translate to:

```
happy(Person):-
        man(Person),
        rich(Person),
        famous(Person).
```

- The “,” indicates the conjunction and is roughly equivalent to the ^ of predicate calculus. Therefore, read “,” as “and”.
- In this single clause, the logical variable Person refers to the same object throughout.
- By the way, we might have chosen any name for the logical variable other than
- Person. It is common practice to name a logical variable in some way that reminds you of what kind of entity is being handled.

Κανόνες και Διαζεύξεις

- Someone is happy if they are healthy, wealthy or wise, translates to:

```
happy(Person):-
        healthy(Person).
happy(Person):-
        wealthy(Person).
happy(Person):-
        wise(Person).
```

- Note how we have had to rewrite the original informal statement into something like:

  Someone is happy if they are healthy OR

  Someone is happy if they are wealthy OR

  Someone is happy if they are wise

- Με άλλο τρόπο:

```
happy(Person):-
        healthy(Person); wealthy(Person); wise(Person).
```

Κανόνες και Συζεύξεις και Διαζεύξεις

- We combine both disjunctions and conjunctions together. Consider:

```
happy(Person):-
        healthy(Person),woman(Person).
happy(Person):-
        wealthy(Person),woman(Person).
happy(Person):-
        wise(Person),woman(Person).
```

- This can be informally interpreted as meaning that:

"A woman is happy if she is healthy, wealthy or wise"

Παραδείγματα κανόνων σε Prolog

❖ Παράδειγμα 1:

parent (X,Y) :- child (Y,X).
mother (X,Y) :- parent (X,Y), female (X).
father (X,Y) :- parent (X,Y), male (X).
sibling (X,Y) :- child (X,Z), child (Y,Z), notequal (X,Y).
sister (X,Y) :- sibling (X,Y), female (X).

Παραδείγματα κανόνων σε Prolog

❖ Παράδειγμα 2:

related (X,X).
related (X,Y) :- married (X,Y).
related (X,Y) :- child (X,Z), related (Z,Y).
related (X,Y) :- child (Z,X), related (Z,Y).

Λογικές Μεταβλητές και Κανόνες

- In the DEC-10 Prolog family, if an object is referred to by a name starting with a capital letter then the object has the status of a logical variable.

  all scots people are british

  can be turned into:

  british(Person):-
  scottish(Person).

  Note that Person is a logical variable.

- One more example:

  if you go from one country to another they you are a tourist

  turns into:

  tourist(P):-
  move(P,Country1,Country2).

  where move(P,A,B) has the informal meaning that a person P has moved from country A to country B.

- In the above rules there are two references to Person or P. All this means is that the two references are to the same object, whatever that object is.

Λογικές Μεταβλητές και Κανόνες

- The scope rule for Prolog is that two uses of an identical name for a logical variable only refer to the same object if the uses are within a single clause.

  Therefore in:

  happy(X):-
  healthy(X).
  wise(X):-
  old(X).

  the two references to X in the first clause do not refer to the same object as the references to X in the second clause.
- Do not assume that the word logical is redundant. It is used to distinguish between the nature of the variable as used in predicate calculus and the variable used in imperative languages like BASIC, FORTRAN and so on.
- In those languages, a variable name indicates a storage location which may `contain' different values at different moments in the execution of the program.

The logical variable cannot be overwritten with a new value.

Λογικές Μεταβλητές και Κανόνες

- For example, in C:

  X= 1; X= 2;

  results in the assignment of 2 to X.

- In Prolog, once a logical variable has a value, then it cannot be assigned a different one.

  The logical statement

  X=1 ^ X=2

  cannot be true as X cannot be both `2' and `1' simultaneously. An attempt to make a logical variable take a new value will fail.

❖ Represent these statements as rules:

1. all animals eat custard
2. everyone loves bergman's films
3. jim likes fred's possessions
4. if someone needs a bike then they may borrow jane's

Λύσεις Ασκήσεων

1. *eat(X,custard):- animal(X).*

This can be paraphrased as `if X is an animal then X eats custard'.
We could also, but less satisfactorily, write

custard_eater(X):- animal(X).

2. *loves(X,Y):- directed_by(bergman,Y).*

3. *likes(jim,X):- belongs_to(X,fred).*

4. *may_borrow(X,bike,jane):- need(X,bike).*

- Each of these statements should be turned into a rule with at least two subgoals (even though some statements are not immediately recognisable as such):

1. you are liable to be fined if your car is untaxed
2. two people live in the same house if they have the same address
3. two people are siblings if they have the same parents

Λύσεις Ασκήσεων (2)

1. *liable_for_fine(X):- owns_car(X,Y), untaxed(Y).*

2. *same_house(X,Y):- address(X,Z), address(Y,Z).*

3. *siblings(X,Y):- mother(X,M), mother(Y,M), father(X,P), father(Y,P), not_same(X,Y).*

ή

siblings(X,Y):-parents(X,M,P), parents(Y,M,P) , not_same(X,Y).

- Each of these statements should be turned into several rules:

  1. you are british if you are welsh, english, scottish or northern irish
  2. you are eligible for social security payments if you earn less than $ 28 per week or you are an old age pensioner
  3. those who play football, rugger or hockey are sportspeople

Λύσεις Ασκήσεων (3)

1. *british(X):- welsh(X).*
   british(X):- english(X).
   british(X):- scottish(X).
   british(X):- northern_irish(X).

2. *eligible_social_security(X):- earnings(X,Y), less_than(Y,28).*
   eligible_social_security(X):- oap(X).

3. *sportsperson(X):- plays(X,football).*
   sportsperson(X):- plays(X,rugger).
   sportsperson(X):- plays(X,hockey).

Ασκήσεις (4)

❖ Here is a small set of problems that require you to convert propositions into Prolog clauses. Make sure you explain the meaning of your representation:

1. $a \rightarrow b$
2. $a \vee b \rightarrow c$
3. $a \wedge b \rightarrow c$
4. $a \wedge (b \vee c) \rightarrow d$
5. $\neg a \vee b$

Λύσεις Ασκήσεων (4)

1. *b*:- *a*.
2. *c*:- *a*.

   c:- *b*.
3. *c*:- *a*, *b*.
4. *d*:- *a*, *b*.

   d:- *a*, *c*. (από κανόνα επιμερισμού)
5. *b*:- *a*.

Ασκήσεις (5)

❖ Represent each statement as a single Prolog clause:

1. Billy studies AI2
2. The population of France is 50 million
3. Italy is a rich country
4. Jane is tall
5. 2 is a prime number
6. The Welsh people are British
7. Someone wrote Hamlet
8. All humans are mortal
9. All rich people pay taxes
10. Bill takes his umbrella if it rains
11. If you are naughty then you will not have any supper
12. Firebrigade employees are men over six feet tall

Λύσεις Ασκήσεων (5)

1. *studies(bill,ai2).*

We have revised `AI2' to `ai2'. We could have simply put quotes around as in studies(bill,'AI2').

2. *population(france,50).*

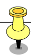
Note we have changed `France' to `france'.

3. *rich_country(italy).*

4. *height(jane,tall).*

5. *prime(2).*

6. *british(X):- welsh(X).*



7. This is a trick question. The problem lies in expressing existential statements such as “someone likes ice-cream" and so on. This is informally recast as there exists some person such that this person likes ice-cream. In first order predicate logic, we would formalise this as ∃ x likes(x,ice cream). This can be turned into likes(whatshisname,ice cream) (this is known as Skolemisation).

8. *mortal(X):- human(X).*

Note that, in the Prolog version, this `universal quantification' is implicit.

9. *pays_taxes(X):- person(X), rich(X).*

10. *takes(bill,umbrella):- raining.*

11. *no_supper(X):- naughty(X).*

Here, we might have tried to write ¬supper(X):- naughty(X). This is, however, illegal in Prolog but not for syntactic reasons.

12. *more_than(Y,6.0):- employs(firebrigade,X), man(X), height(X,Y).*

Or *over six foot(X):- firebrigade_employs(X).*

Η ερμηνεία ενός λογικού προγράμματος

- **Ορισμός**: Η **ερμηνεία (meaning)**, *M(P)*, ενός λογικού προγράμματος *P*, είναι το σύνολο των στοιχειωδών μοναδιαίων φράσεων (δηλ. στοιχειωδών κατηγορημάτων) που συμπεραίνονται από το *P*.

- Από τον ορισμό αυτόν, έπεται ότι η ερμηνεία ενός λογικού προγράμματος που αποτελείται από στοιχειώδη μόνο γεγονότα, όπως το Πρόγραμμα 8.1, είναι το ίδιο το πρόγραμμα. Με άλλα λόγια, για απλά προγράμματα, το πρόγραμμα "σημαίνει ακριβώς ότι αυτό λέει".

- Όταν το λογικό πρόγραμμα περιέχει και κανόνες ή καθολικά γεγονότα, τότε πρέπει να διατρέξουμε 'νοερά' το πρόγραμμα και να βρούμε όλες τις δυνατές στοιχειώδεις μοναδιαίες φράσεις που συμπεραίνονται από το πρόγραμμα.

- Αυτό βέβαια δεν είναι τόσο εύκολο για ένα σύνθετο λογικό πρόγραμμα.

Η ερμηνεία ενός λογικού προγράμματος

❖ Πρόγραμμα 8.2: Οι σχέσεις της Βιβλικής οικογένειας.

father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).
female(milcah).
female(yiscah).
male(lot).
male(isaac).
son(X,Y) ← *father(Y,X), male(X).*
daughter(X,Y) ← *father(Y,X), female(X).*

Η ερμηνεία ενός λογικού προγράμματος

❖ Με βάση τον παραπάνω ορισμό, η ερμηνεία του λογικού προγράμματος 8.2 είναι το παρακάτω σύνολο στοιχειωδών μοναδιαίων φράσεων :

{ father(abraham,isaac), father(haran,lot), father(haran,milcah), father(haran,yiscah), female(milcah), female(yiscah), male(lot), male(isaac), son(Isaac,abraham), son(lot,haran), daughter(milcah,haran), daughter(yiscah,haran) }

Η ερμηνεία ενός λογικού προγράμματος

- Ορίσαμε και εξηγήσαμε την έννοια της ερμηνείας ενός λογικού προγράμματος. Δηλαδή, μπορούμε τώρα να παράγουμε την ερμηνεία ενός οποιουδήποτε λογικού προγράμματος.

- Όμως αυτό δεν αρκεί. Ο λόγος είναι ότι πρέπει να συγκρίνουμε την ερμηνεία ενός προγράμματος με την προσδοκώμενη σημασία του.

- **Ορισμός**: Η **προσδοκώμενη σημασία (intended meaning)** ενός προγράμματος είναι ένα σύνολο από στοιχειώδεις μοναδιαίες φράσεις που ορίζονται από τον συγγραφέα του προγράμματος.

Η ερμηνεία ενός λογικού προγράμματος

- Ας υποθέσουμε ότι έχουμε ένα πρόγραμμα με τις σχέσεις *πατέρας*, *μητέρα*, *άρρεν*, *θήλυ* και θέλουμε να γράψουμε τον ορισμό της *κόρης (daughter)*.
  Η προσδοκώμενη σημασία του προγράμματος είναι :

  { father(peter, george), father(peter, john), father(peter,maria), mother(helen,georgia),mother(helen,roula), male(peter), male(george), male(john), female(helen), female(maria), female(georgia), female(roula), daughter(maria,peter), daughter(georgia,helen), daughter(roula,helen) }

Η ερμηνεία ενός λογικού προγράμματος

- Το πρόγραμμα που γράφουμε λοιπόν έχει ως εξής :

  father(peter, george).
  father(peter, john).
  father(peter,maria).
  mother(helen,georgia).
  mother(helen,roula).
  male(peter).
  male(george).
  male(john).
  female(helen).
  female(maria).
  female(roula).
  daughter(X,Y) ← father(Y,X), female(X).

- Όμως το πρόγραμμα αυτό έχει την παρακάτω ερμηνεία :

  { *father(peter, george), father(peter, john), father(peter,maria), mother(helen,georgia), mother(helen,roula), male(peter), male(george), male(john), female(helen), female(maria), female(roula), daughter(maria, peter)* }

Η ερμηνεία ενός λογικού προγράμματος

- Παρατηρούμε ότι η στοιχειώδης μοναδιαία φράση *daughter(roula, helen)* δεν αποδεικνύεται από το πρόγραμμα. Σκεφτόμαστε λίγο και καταλήγουμε στο συμπέρασμα ότι ξεχάσαμε να εισάγουμε τον παρακάτω κανόνα στο λογικό πρόγραμμα :

  daughter(X,Y) ← mother(Y,X), female(X).

- Πράγματι, τώρα το λογικό πρόγραμμα είναι πληρέστερο. Η ερμηνεία του προγράμματος είναι τώρα η παρακάτω :

  { father(peter, george), father(peter, john), father(peter,maria), μother(helen,georgia), mother(helen,roula), male(peter), male(george), male(john), female(helen), female(maria), female(roula), daughter(maria,peter), daughter(roula,helen) }

Η ερμηνεία ενός λογικού προγράμματος

- Παρόλα αυτά η στοιχειώδης μοναδιαία φράση *daughter(georgia, helen)* δεν αποδεικνύεται από το πρόγραμμα.
  Σκεφτόμαστε λίγο και καταλήγουμε στο συμπέρασμα ότι ξεχάσαμε να εισάγουμε τον παρακάτω γεγονός στο λογικό πρόγραμμα :

  female(georgia).

- Τώρα η ερμηνεία του προγράμματος είναι πανομοιότυπη με την προσδοκώμενη σημασία του.

Η ερμηνεία ενός λογικού προγράμματος

- Λέμε ότι ένα πρόγραμμα είναι ***ορθό (correct)*** αναφορικά προς κάποια προσδοκώμενη σημασία *M* αν η ερμηνεία του, *M(P)*, είναι ένα υποσύνολο του *M*.
  Με άλλα λόγια, ένα ορθό πρόγραμμα δεν καταλήγει σε απροσδόκητα συμπεράσματα (μπορεί όμως να μην καταλήγει σε όλα τα προσδοκώμενα συμπεράσματα).

- Ένα πρόγραμμα είναι ***πλήρες (complete)*** αναφορικά με την *M* αν η *M* είναι ένα υποσύνολο της *M(P)*.
  Με άλλα λόγια, ένα πλήρες πρόγραμμα καταλήγει σε όλα τα προσδοκώμενα συμπεράσματα (μπορεί επίσης να καταλήγει και σε άλλα συμπεράσματα).

- Έτσι, ένα πρόγραμμα *P* είναι ορθό και πλήρες αναφορικά με μια προσδοκώμενη ερμηνεία *M* αν *M = M(P)*.

Προγραμματισμός Βάσεων Γνώσης

- Ξεκινάμε ανατρέχοντας στο Πρόγραμμα 8.1, την βιβλική βάση δεδομένων, και την αύξησή της με κανόνες που εκφράζουν τις σχέσεις οικογένειας.

- Η ίδια η βάση δεδομένων είχε τέσσερα βασικά κατηγορήματα, *father/2, mother/2, male/1,* και *female/1*.

Προγραμματισμός Βάσεων Γνώσης

- Πρέπει να γραφούν νέοι κανόνες για τις σχέσεις που δεν καθορίζονται πλέον από τα γεγονότα, δηλαδή τις father και mother.

- Κατάλληλοι κανόνες είναι οι εξής:

father(Dad,Child) ← parent(Dad,Child), male(Dad).
mother(Mum,Child) ← parent(Mum,Child), female(Mum).

Προγραμματισμός Βάσεων Γνώσης

- Ένα άλλο παράδειγμα για πληροφορίες που μπορούν να ανακτηθούν από τις απλές υπάρχουσες πληροφορίες είναι οι σχέσεις αδελφών - brothers και sisters.

- Δίνουμε έναν κανόνα για τον *brother(Brother,Sibling)*.

Brother(Brother,Sib) ← *parent(Parent,Brother), parent(Parent,Sib), male(Brother)*.

- Αυτό διαβάζεται ως εξής: "Ο *Brother* είναι αδελφός του *Sib* αν ο *Parent* είναι πατέρας και των δύο *Brother* και *Sib*, και ο *Brother* είναι αρσενικός."

Προγραμματισμός Βάσεων Γνώσης

- Υπάρχει ένα πρόβλημα στον ορισμό του αδελφού που δόθηκε παραπάνω. Η ερώτηση *brother(X,X)?* ικανοποιείται για κάθε αρσενικό παιδί *X*, που δεν συμφωνεί με τον τρόπο που αντιλαμβανόμαστε την σχέση αδελφού.

- Για να αποφευχθούν τέτοιες περιπτώσεις από την έννοια του προγράμματος εισάγουμε ένα κατηγόρημα '≠', που συντάσσεται σε ενδοδιάταξη :
  - Term1 ≠ Term2. Το Term1 ≠ Term2 είναι αληθές εάν το Term1 είναι διαφορετικό από το Term2.

- Προς το παρόν, περιορίζεται σε σταθερούς όρους. Κατ' αρχήν, μπορεί να οριστεί από έναν πίνακα *X≠Y* για κάθε δύο διαφορετικά άτομα *X* και *Y* στον τομέα του ενδιαφέροντος.

Στη συνέχεια, το σχήμα 9.1 δίνει τον κατάλληλο πίνακα για το Πρόγραμμα 8.1.

Προγραμματισμός Βάσεων Γνώσης

❖ Σχήμα 9.1: Ορίζοντας την ανισότητα

abraham≠isaac	abraham≠haran	abraham≠lot
abraham≠milcah	abraham≠yiscah	isaac≠haran
isaac≠lot	isaac≠milcah	issac≠yiscah
haran≠lot	haran≠milcah	haran≠yiscah
lot≠milcah	lot≠yiscah	milcah≠yiscah

❖ Ο νέος κανόνας για τον αδελφό είναι:

brother(Brother,Sib) ←
parent(Parent,Brother),
parent(Parent,Sib),
male(Brother), Brother ≠ Sib.

Προγραμματισμός Βάσεων Γνώσης

- Όσο πιο πολλές σχέσεις υπάρχουν, τόσο πιο εύκολο είναι να οριστούν πολυπλοκότερες σχέσεις.

- Το Πρόγραμμα 9.1 ορίζει τις σχέσεις uncle(Uncle,NieceOrNephew), sibling(Sib1,Sib2), και cousin(Cousin1,Cousin2).

- Πρόγραμμα 9.1: Ορίζοντας τις σχέσεις οικογένειας

```
uncle(Uncle,Person) ←
    brother(Uncle,Parent), parent(Parent,Person).
sibling(Sib1,Sib2) ←
    parent(Parent,Sib1), parent(Parent,Sib2), Sib1≠Sib2.
cousin(Cousin1,Cousin2) ←
    parent(Parent1,Cousin1),
    parent(Parent2,Cousin2),
    sibling(Parent1,Parent2).
```

Προγραμματισμός Βάσεων Γνώσης

- Μια άλλη σχέση που δεν είναι σαφής στην βάση δεδομένων μια οικογένειας είναι αν η γυναίκα είναι μητέρα. Αυτό καθορίζεται χρησιμοποιώντας τις σχέσεις *mother/2*. Το νέο σχήμα σχέσης είναι mother(Woman) και ορίζεται από τον κανόνα:

  mother(Woman) ← *mother(Woman,Child)*.

  Αυτό διαβάζεται: Μια *Woman* είναι μητέρα εάν είναι μητέρα κάποιου *Child*.

- Προσέξτε ότι, έχουμε χρησιμοποιήσει το ίδιο όνομα κατηγόρημα, *mother*, για να περιγράψουμε δύο διαφορετικές σχέσεις *mother*.
  Το κατηγόρημα mother δέχεται διαφορετικό αριθμό ορισμάτων, λ.χ., έχει διαφορετική πολλαπλότητα στις δυο περιπτώσεις.

- Γενικά, το ίδιο όνομα κατηγορήματος δηλώνει μια διαφορετική σχέση, όταν έχει διαφορετικό αριθμό ορισμάτων.

- Η δόμηση των δεδομένων είναι σημαντική γενικά στον προγραμματισμό και ειδικά στον λογικό προγραμματισμό.

- Χρησιμοποιείται για την οργάνωση των δεδομένων κατά έναν ουσιαστικό τρόπο.

- Οι κανόνες μπορούν να γραφούν πιο αφηρημένα, αγνοώντας ασήμαντες λεπτομέρειες.

- Τα περισσότερα τμηματικά προγράμματα μπορούν να επιτευχθούν με αυτό το τρόπο, καθώς μια αλλαγή στην αναπαράσταση των δεδομένων δεν σημαίνει αλλαγή σε όλο το πρόγραμμα, όπως φαίνεται και από το παρακάτω παράδειγμα.

Προγραμματισμός Βάσεων Γνώσης

- Έστω οι ακόλουθοι δύο τρόποι αναπαράστασης ενός γεγονότος σχετικά με μια διάλεξη πάνω στην πολυπλοκότητα που δόθηκε την Δευτέρα από τις 9 ως τις 11 από τον David Harel στο κτίριο Feinberg, αίθουσα A:

  course(complexity,monday,9,11,david,harel,feinberg,a).

  και

  course(complexity,time(monday,9,11),lecturer(david,harel), location,feinberg,a)).

- Το πρώτο γεγονός αναπαριστά το *course* ως μια σχέση μεταξύ οκτώ στοιχείων - ένα όνομα μαθήματος, μια μέρα, μια ώρα εκκίνησης, μια ώρα τερματισμού, το όνομα του Λέκτορα, το επώνυμο του Λέκτορα, ένα κτίριο και μια αίθουσα.
- Το δεύτερο γεγονός θέτει το *course* ως σχέση τεσσάρων στοιχείων - ένα όνομα, έναν χρόνο, έναν λέκτορα και μια τοποθεσία με περισσότερους όρους. Ο χρόνος αποτελείται από μια μέρα, έναν χρόνο εκκίνησης και έναν χρόνο τερματισμού, οι Λέκτορες έχουν όνομα και επώνυμο, και οι τοποθεσίες προσδιορίζονται από ένα κτίριο και μια αίθουσα.
  Το δεύτερο γεγονός αντικατοπτρίζει πιο κομψά τις σχέσεις που υπάρχουν.

Προγραμματισμός Βάσεων Γνώσης

- Η έκδοση του *course* με τέσσερα ορίσματα δίνει την δυνατότητα να γραφούν πιο συνοπτικοί κανόνες αφαιρώντας τις λεπτομέρειες που δεν έχουν σχέση με την ερώτηση.
  Το Πρόγραμμα 9.4, που ακολουθεί, αποτελείται από μερικά παραδείγματα.

- Ο κανόνας *occupied* θεωρεί ένα κατηγόρημα μικρότερο ή ίσο, που αναπαρίσταται με έναν δυαδικό ένθετο τελεστή $\leq$.

- Οι κανόνες που δεν σχετίζονται με τις συγκεκριμένες τιμές ενός δομημένου ορίσματος δεν χρειάζεται να "γνωρίζουν" πως είναι δομημένο το όρισμα.
  - Για παράδειγμα, οι κανόνες για *duration* και *teaches* αναπαριστούν σαφώς τον χρόνο ως *time(Day,Start,Finish),* διότι οι χρόνοι *Day* ή *Start* ή *Finish* του μαθήματος είναι επιθυμητοί.
- Σε αντίθεση, ο κανόνας για τον *lecturer* δεν είναι. Αυτό οδηγεί σε μεγαλύτερη τμηματικότητα, καθώς η αναπαράσταση του χρόνου μπορεί να αλλάξει χωρίς να επηρεάζει τους κανόνες που δεν τον ερευνούν.

Προγραμματισμός Βάσεων Γνώσης

❖ Πρόγραμμα 9.4: Κανόνες εκμάθησης

lecturer(Lecturer,Course) $\leftarrow$
course(Course,Time,Lecturer,Location).
duration(Course,Lenght) $\leftarrow$
course(Course,time(Day,Start,Finish),Lecturer,Location)
plus(Start,Lenght,Finish).
teaches(Lecturer,Day) $\leftarrow$
course(Course,time(Day,Start,Finish),Lecturer,Location).
occupied(Room,Day,Time) $\leftarrow$
course(Course,time(Day,Start,Finish),Lecturer,Room),
$Start \leq Time,\ Time \leq Finish.$

- Δεν έχουμε ορισμένους κανόνες για να αποφασίσουμε αν θα χρησιμοποιήσουμε δεδομένα ή όχι. Εκεί όπου όλα τα δεδομένα είναι απλά, η μη-χρησιμοποίηση δομημένων δεδομένων επιτρέπει την ομοιόμορφη αναπαράσταση.

- Τα πλεονεκτήματα των δομημένων δεδομένων είναι η συμπαγής αναπαράσταση, η οποία αντικατοπτρίζει με μεγαλύτερη ακρίβεια την άποψή μας για μια περίπτωση και η τμηματικότητα.

- Πιστεύουμε ότι η εμφάνιση ενός προγράμματος είναι σημαντική, ιδιαίτερα όταν αφορά δύσκολα προβλήματα. Μια καλή δόμηση των δεδομένων μπορεί να φέρει διαφοροποίηση όταν προγραμματίζουμε περίπλοκα προβλήματα.

Στρατηγική Αναζήτησης στην Prolog

«Κατά βάθος»

- Here are some facts assumed to be known:

  <u>Program Database</u>

  woman(jean).

  man(fred).

  woman(jane).

  woman(joan).

  woman(pat).

  ?- woman(jane).

- Prolog searches through the set of clauses from top to bottom. First, Prolog examines

  woman(jean).

  and finds that

  woman(jane).

  does not match.

Στρατηγική Αναζήτησης στην Prolog

- Also, it is obvious that the next clause man(fred). doesn't match either.
- Prolog then comes to look at the third clause and it finds what we want.

Μία απλή Σύζευξη

<u>Program database</u>

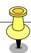
woman(jean).
man(fred).
wealthy(fred).
happy(Person):-
    woman(Person),
    wealthy(Person).
?- happy(jean).
no

- Why? Prolog tries to match:

    happy(jean)

  against

    happy(Person)



- We call this matching process unification. What happens here is that the logical variable Person gets bound to the atom jean. You could paraphrase “bound" as “is temporarily identified with".
- To solve our problem, Prolog must set up two subgoals. But we must make sure that, since Person is a logical variable, that everywhere in the rule that Person occurs we will replace Person by jean.
- We now have something equivalent to:

  happy(jean):-

  woman(jean),

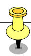
  wealthy(jean).

  So the two subgoals are:

  woman(jean)

  wealthy(jean)

  Here we come to our next problem. In which order should Prolog try to solve these subgoals?



- The answer is that the standard way to choose the subgoal to work on first is again based on the way we read (in the west)! We try to solve the subgoal woman(jean) and then the subgoal wealthy(jean). There is only one possible match for woman(jean): our subgoal is successful.
- However, we are not finished until we can find out if wealthy(jean).

  There is a possible match but we cannot unify

  wealthy(fred)

  with

  wealthy(jean)

  (However, Prolog is not finished yet. Once we reach wealthy(Person) with Person/jean and it fails we move back(backtracking) to the goal woman(Person) and break the binding for Person (because this is where we made the binding Person/jean). We now start going from left to right again (if you like, forwardtracking).)

Σύζευξη και Διάζευξη

<u>Program Database</u>

```
woman(jean).          woman(jane).
woman(joan).          woman(pat).
wise(jean).           wealthy(jane).
wealthy(jim).         healthy(jim).
healthy(jane).        healthy(jean).
happy(P):-
        healthy(P),woman(P).
happy(P):-
        wealthy(P),woman(P).
happy(P):-
        wise(P),woman(P).
```

Yes

?- *happy(joan)*

No

❖ The resolution process generates the subgoals healthy(joan) and woman(joan) from the first clause for happy/1. In all, Prolog tries three times to match healthy(joan) as there are three clauses for healthy/1. After failing healthy(joan), however, Prolog does not try to solve woman(joan) |there is no point in doing so.

❖ There is another way of trying to prove happy(joan) using the second clause of happy/1. The resolution process again generates subgoals wealthy(joan) and woman(joan)| and wealthy(joan) fails. A third attempt is made but this founders as wise(joan) fails. Now back to top level to report the complete failure to satisfy the goal.

?- happy(P)

- First, healthy(P) succeeds binding P to jim (P/jim) but when the conjunctive goal woman(jim) is attempted it fails. Prolog now backtracks. It reverses along the path until it can find a place where there was an alternative solution.
- Of course, Prolog remembers to unbind any variables exactly at the places where they were bound.

- In the example we are using we again try to resolve the goal healthy(P) succeeding with P bound to jane. Now the conjunction can be satisfied as we have woman(jane). Return to top level with P bound to jane to report success. What follows is what appears on the screen:

  ?- happy(P).

  P=jane

  Yes

Ασκήσεις

❖ Follow the execution of a number of different queries.

Program Database

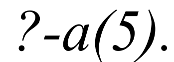
a(X):- b(X,Y), c(Y).

a(X):- c(X).

b(1,2). *b(2,2).*

b(3,3). *b(3,4).*

c(2). *c(5).*



?-a(5).

a(5) uses 1st clause new subgoals

b(5,Y) tries 1st clause fails

b(5,Y) tries 2nd clause fails

b(5,Y) tries 3rd clause fails

b(5,Y) tries 4th clause fails

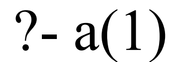
a(5) using 1st clause fails

a(5) uses 2nd clause new subgoal

c(5) tries 1st clause fails

c(5) tries 2nd clause succeeds

a(5) using 2nd clause succeeds



?- a(1)

a(1) uses 1st clause new subgoals

b(1,Y) uses 1st clause succeeds with Y=2

c(2) uses 1st clause succeeds

a(1) using 1st clause succeeds

?-a(2)

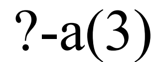
a(2) uses 1st clause new subgoals

b(2,Y) uses 1st clause fails

b(2,Y) uses 2nd clause succeeds with Y=2

c(2) uses 1st clause succeeds

a(2) using 1st clause succeeds



?-a(3)

a(3) uses 1st clause new subgoals

b(3,Y) tries 1st clause fails

b(3,Y) tries 2nd clause fails

b(3,Y) tries 3rd clause succeeds with Y=3

c(3) tries 1st clause fails

c(3) tries 2nd clause fails

b(3,Y) tries 4th clause succeeds with Y=4

c(4) tries 1st clause fails

c(4) tries 2nd clause fails

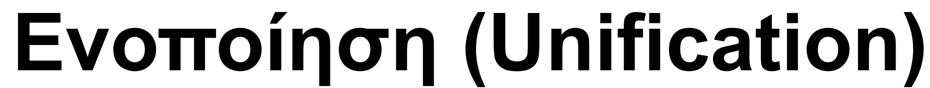
b(3,Y) no more clauses fails

a(3) uses 2nd clause new subgoal

c(3) tries 1st clause fails

c(3) tries 2nd clause fails

a(3) no more clauses fails



Ενοποίηση (Unification)

Given two terms, we say that they match if.:

(1) they are identical, or

(2) the variables in both terms can be instantiated to objects in such a way that after the substitution of variables by these objects the terms become identical.

For example, the terms date(D, M, 1983) and date(Dl, may, Y1) match. One instantiation that makes both terms identical is:

D is instantiatedto Dl

M is instantiated to may

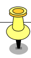
Y1 is instantiatedto 1983

This instantiation is more compactly written in the familiar form in which Prolog outputs results:

D = D1

M = may

Yl = 1983



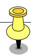
On the other hand, the terms date(D, M, l983) and date(Dl, Ml, 1444) do not match,

nor do the terms date(X, Y, Z) and point(X, Y, Z).

Matching is a process that takes as input two terms and checks whether they match.

If the terms do not match we say that this process fails.

If they do match then the process succeeds and it also instantiates the variables in both terms to such values that the terms become identical.



As an example consider the following question:

?- date(D, M, 1983) = date(Dl, may, Y1),

date(D, M, 1983) = date(15, M, Y).

To satisfy the first goal, Prolog instantiates the variables as follows:

D = Dl

M = may

Yl = 1983

After having satisfied the second goal, the instantiation becomes more specific as follows:

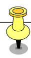
D = 15

Dl = 15

M = may

Yl = 1983

Y = 1983



- Unification is a two way matching process

- E.g. Match the term book(waverley,X) against book(Y, scott)
  Result: book(waverley,scott)

- E.g. X=fred succeeds
  jane=fred fails because you can't match two distinct atoms
  Y=fred, X=Y succeeds with X=fred, Y=fred
  X=happy(jim) succeeds
  X=Y succeeds, later, if X gets bound then so will Y and vice versa

Ενοποίηση Σύνθετων Όρων

- E.g. happy(X)=sad(jim)

  fails, because we the principal functors and their arities are the same for unification to succeed.

- data(X,salary(10000))=data(name(fred),Y).

  succeeds, because, having matched the principal functors (and checked that the arities are the same) we recursively try to match corresponding arguments.

  This generates two subgoals:

  X = name(fred)

  salary(10000) = Y

  which both succeed.

Ασκήσεις

- Decide which is the case and, if the unification succeeds, write down the substitutions made.

  1. 2+1=3

     Fails. We can tell immediately that 3 is an atom but what about 2+1? This does not look like an atom, indeed it is not.

  2. f(X,a)=f(a,X)

     succeeds with X=a.

  3. fred=fred

     succeeds.

  4. likes(jane,X)=likes(X,jim)

     Fails.

  5. f(X,Y)=f(P,P)

     succeeds with X=Y=P.

Ασκήσεις 2

greek(socrates).

human(turing).

human(socrates).

fallible(X) :- human(X).

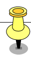
?-fallible(Y), greek(Y).

Η Prolog ξεκινά από την πρώτη από αριστερά κλήση της ερώτησης που είναι η

?-fallible(Y).

Αυτή ενοποιείται με τον κανόνα και γίνεται η αντικατάσταση:

?-human(Y), greek(Y).



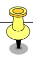
Γίνεται η ενοποίηση Y=turing και τώρα η ερώτηση είναι:

?-human(turing), greek(turing).

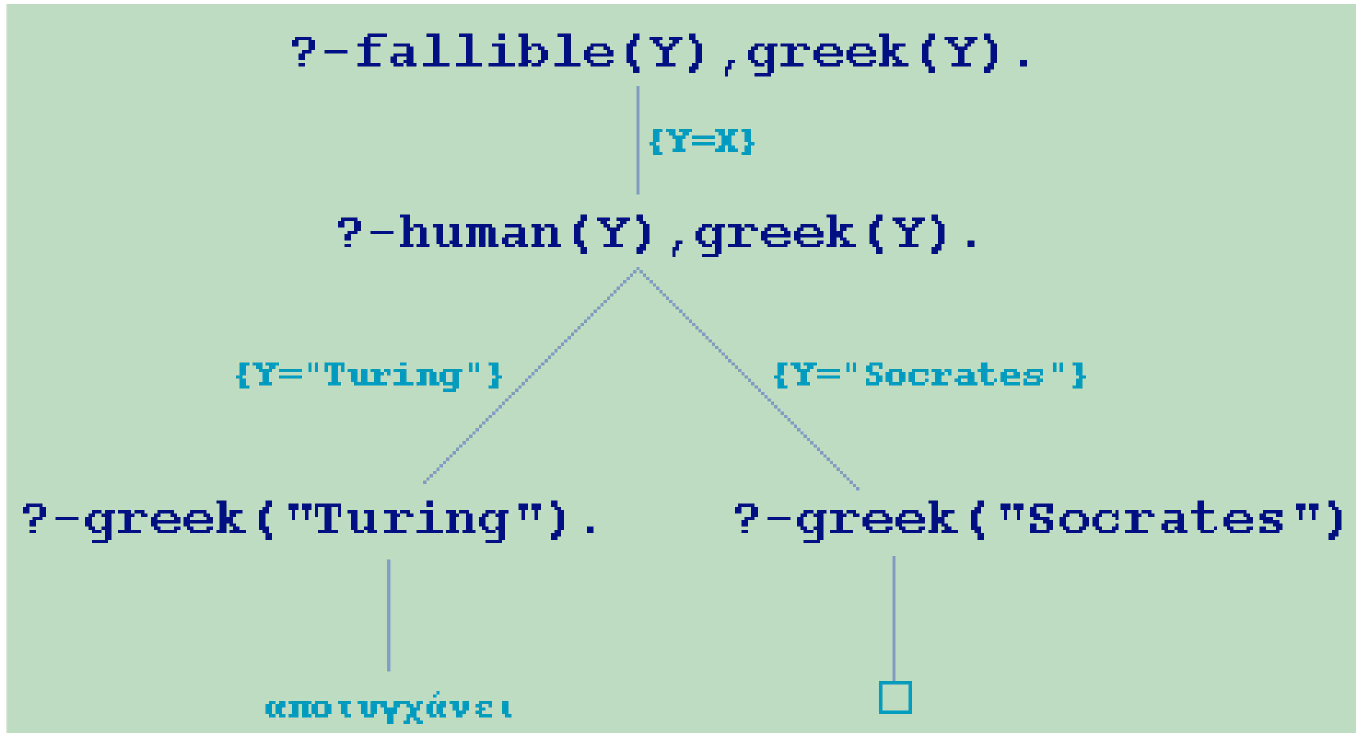
Η κλήση όμως αυτή δεν μπορεί να ταυτοποιηθεί. Στο σημείο αυτό το πρόγραμμα οπισθοδρομεί στο σημείο ενοποίησης της μεταβλητής Y, την αποδεσμεύει και την ενοποιεί με Y=socrates. Τώρα η ερώτηση γίνεται:

?-human(socrates), greek(socrates).

Η κλήση αυτή ταυτοποιείται και επειδή δεν υπάρχουν εναλλακτικές λύσεις, η απάντηση στην αρχική ερώτηση είναι Y=socrates.



...σχηματικά:



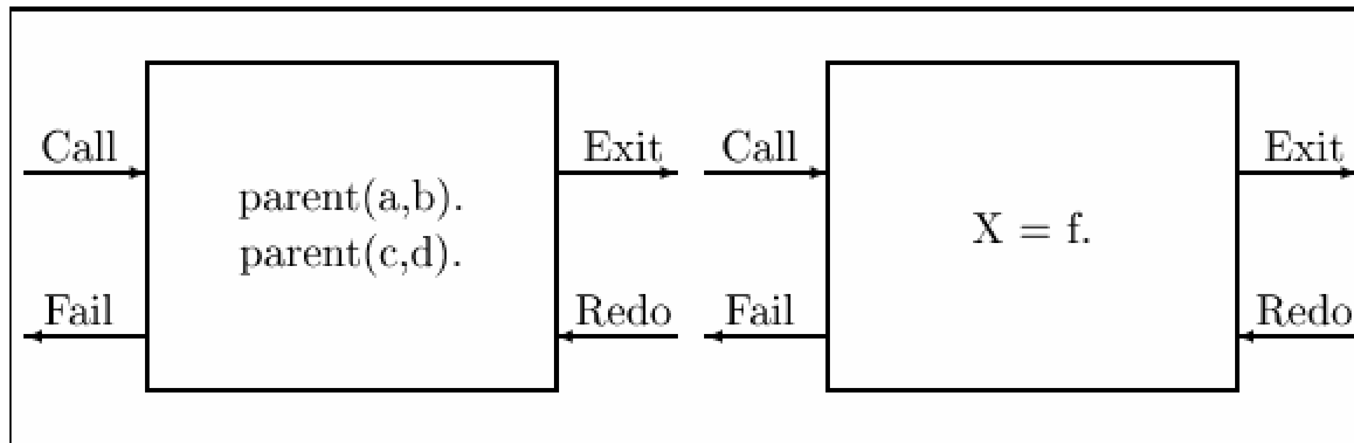
Κάθε κόμβος αντιστοιχεί στη τρέχουσα ερώτηση. Τα κλαδιά αντιπροσωπεύουν ένα υπολογιστικό βήμα και χαρακτηρίζονται από ενοποιήσεις μεταβλητών.

Σε περίπτωση που μία κλήση ταυτοποιείται με περισσότερες από μία προτάσεις του προγράμματος, τότε από τον κόμβο αυτό ξεκινούν περισσότερα του ενός κλαδιά.

Το σημείο αυτό είναι ένα σημείο οπισθοδρόμησης.

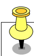
The BOX Model of Execution

We regard each box as having four ports: they are named the **Call, Exit, Fail and Redo** ports.



The **Call port** for an invocation of a procedure represents the first time the solution of the associated goal is sought. Control then flows into the box through the Call port. We then seek a clause with a head that unifies with the goal. Then, we seek solutions to all the subgoals in the body of the successful clause.

If the unification fails for all clauses (or there are no clauses at all) then control would pass out of the **Fail port**. There are also other ways to reach the Fail port.

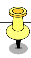


Control reaches the **Exit port** if the procedure succeeds. This can only occur if the initial goal has been unified with the head of one of the procedure's clauses and all of its subgoals have been satisfied.

The **Redo port** can only be reached if the procedure call has been successful and some subsequent goal has failed. This is when Prolog is backtracking to find some alternative way of solving some top-level goal.

Basically, backtracking is the way Prolog attempts to find another solution for each procedure that has contributed to the execution up to the point where some procedure fails. This is done back from the failing procedure to the first procedure that can contribute an alternative solution, hence, **backtracking**. When backtracking is taking place, control passes through the Redo port.

We then, with the clause which was used when the procedure was previously successful, backtrack further back through the subgoals that were previously satisfied. We can reach the Exit port again if either one of these subgoals succeeds a different way |and this leads to all the subgoals in the body of the clause succeeding| or, failing that, another clause can be used successfully.

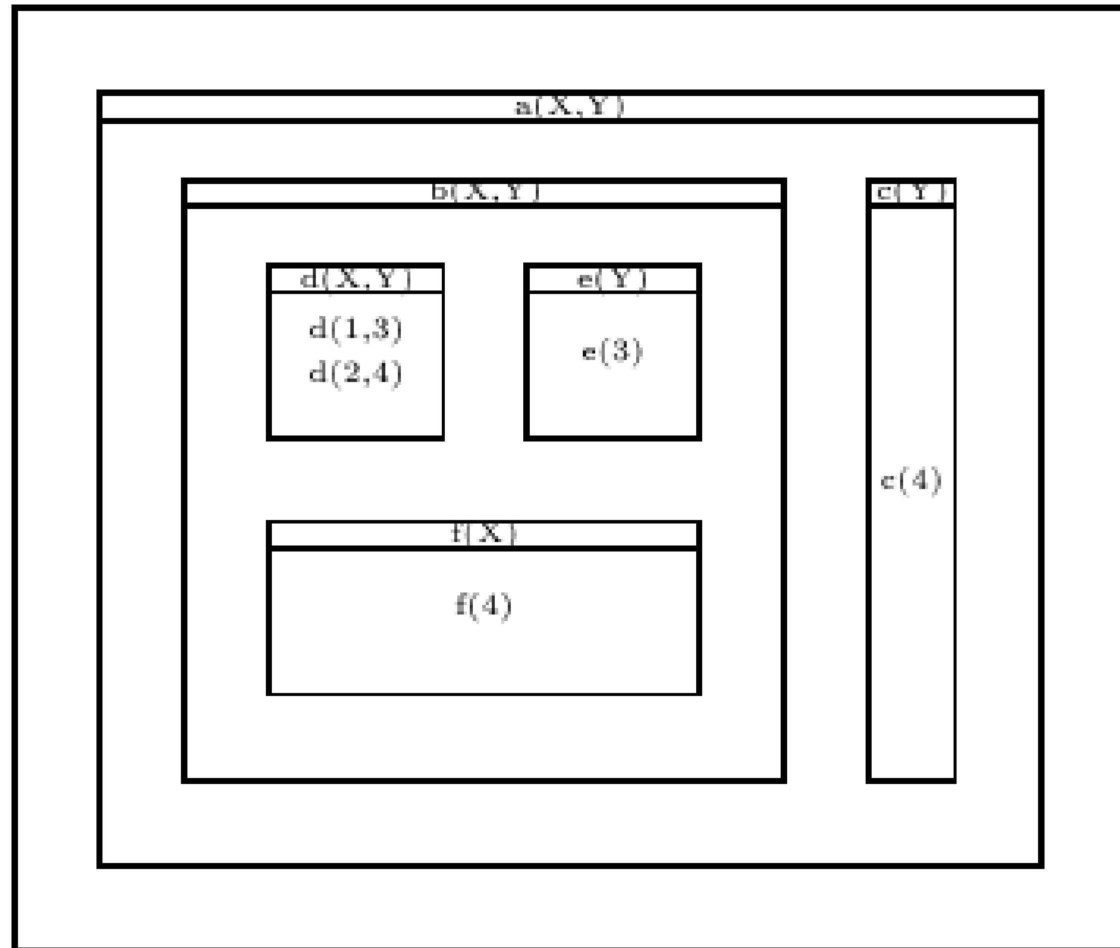


Otherwise, we reach the Fail port. Note that, for this to work out, the system has to remember the clause last used for each successful predicate. We reach the Fail port

- When we cannot find any clauses such that their heads match with the goal
- If, on the original invocation, we can find no solution for the procedure
- On backtracking, we enter the box via the Redo port but no further solution can be found

Παράδειγμα Εκτέλεσης Προγράμματος

Program Database
a(X,Y):-
b(X,Y),
c(Y).
b(X,Y):-
d(X,Y),
e(Y).
b(X,Y):-
f(X).
c(4).
d(1,3).
d(2,4).
e(3).
f(4).



?- a(X,Y).

```
Call:   a(X,Y)
                        Call:   b(X,Y)
                                                Call:   d(X,Y)
                                                Exit:   d(1,3)
                                                Call:   e(3)
                                                Exit:   e(3)
                        Exit:   b(1,3)
                        Call:   c(3)
                        Fail:   c(3)
                            Now backtracking
                        Redo:   b(X,Y)
                                                Redo:   e(3)
                                                Fail:   e(3)
                                                Redo:   d(X,Y)
                                                Exit:   d(2,4)
                                                Call:   e(4)
                                                Fail:   e(4)
                            Now backtracking
                                                Call:   f(X)
                                                Exit:   f(4)
                        Exit:   b(4,Y)
                        Call:   c(Y)
                        Exit:   c(4)
Exit:   a(4,4)
```

Αναδρομικοί κανόνες

- Οι κανόνες που περιγράφθηκαν ως τώρα ορίζουν νέες σχέσεις με βάση τις ήδη υπάρχουσες. Μια ενδιαφέρουσα επέκταση είναι οι αναδρομικοί ορισμοί των σχέσεων που ορίζουν σχέσεις με βάση τους εαυτούς τους. Κατά ένα τρόπο μπορούμε να δούμε τους αναδρομικούς κανόνες ως μια γενίκευση ενός συνόλου μη-αναδρομικών κανόνων.
- Έστω μια σειρά κανόνων που ορίζουν τους προγόνους - παππούδες και γιαγιάδες, προπαππούδες και προγιαγιάδες, κ.λπ.:

grandparent(Ancestor,Descendant) ←
parent(Ancestor,Person),
parent(Person,Descendant).
greatgrandparent(Ancestor,Descendant) ←
parent(Ancestor,Person),
grandparent(Person,Descendant).
greatgreatgrandparent(Ancestor,Descendant) ←
parent(Ancestor,Person),
greatgrandparent(Person,Descendant).

Αναδρομικοί κανόνες

❖ Μπορούμε να δούμε ένα σαφές υπόδειγμα, το οποίο μπορεί να εκφραστεί σε έναν κανόνα που ορίζει τη σχέση του *ancestor(Ancestor,Descendant)*:

Πρόγραμμα 9.5: Η σχέση προγόνου

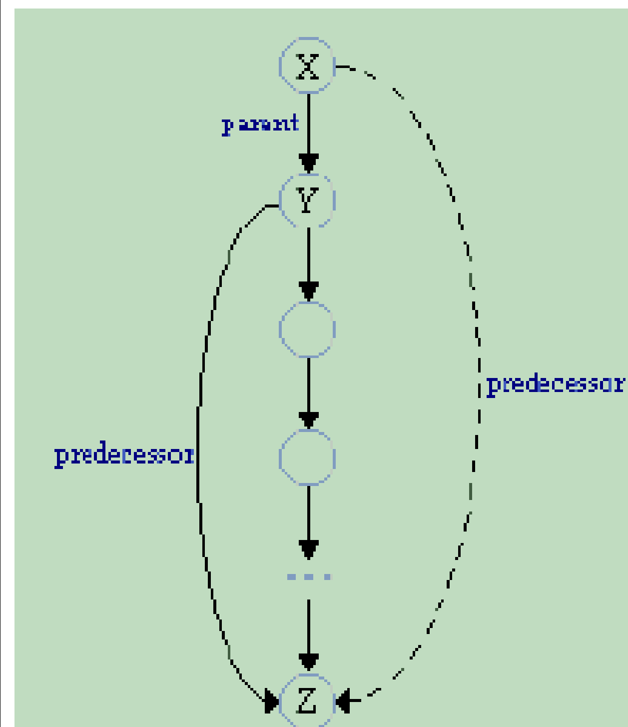
ancestor(Ancestor, Descendant) ←
parent(Ancestor, Descendant).
ancestor(Ancestor, Descendant) ←
parent(Ancestor, Person), ancestor(Person, Descendant).

Παράδειγμα σε Prolog

parent(john, george). *parent(gregory, john).* *parent(bob, gregory).*
parent(joseph, bob).
predecessor(X, Z) :- parent(X, Z).
predecessor(X, Z) :- parent(X, Y), predecessor(Y, Z).

?- predecessor(X, george)

X = john;
X = gregory;
X = bob;
X = joseph



Παράδειγμα σε Prolog 2

parent(john, george). *parent(nick, john)*

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

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

?- predecessor(nick, Y).

Αρχική ερώτηση: ?- predecessor(nick,Y).

Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
        parent(X,Z).
4:predecessor(X,Z):-
        parent(X,Y),
        predecessor(Y,Z).
```

nick=X
Y=Z

nick=X
Y=Z

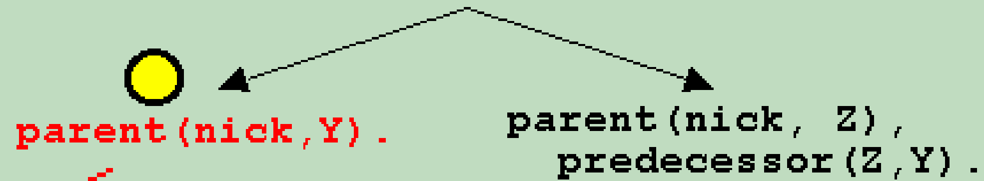
Αρχική ερώτηση: ?- predecessor(nick,Y).

```
parent(nick,Y).
parent(nick, Z),
  predecessor(Z,Y).
```

Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
        parent(X,Z).
4:predecessor(X,Z):-
        parent(X,Y),
        predecessor(Y,Z).
```

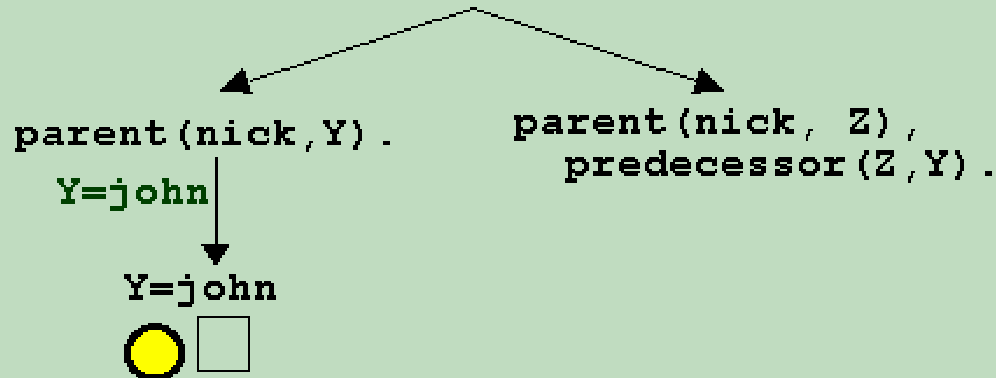
Αρχική ερώτηση: ?- predecessor(nick,Y).



Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
           parent(X,Z).
4:predecessor(X,Z):-
        parent(X,Y),
        predecessor(Y,Z).
```

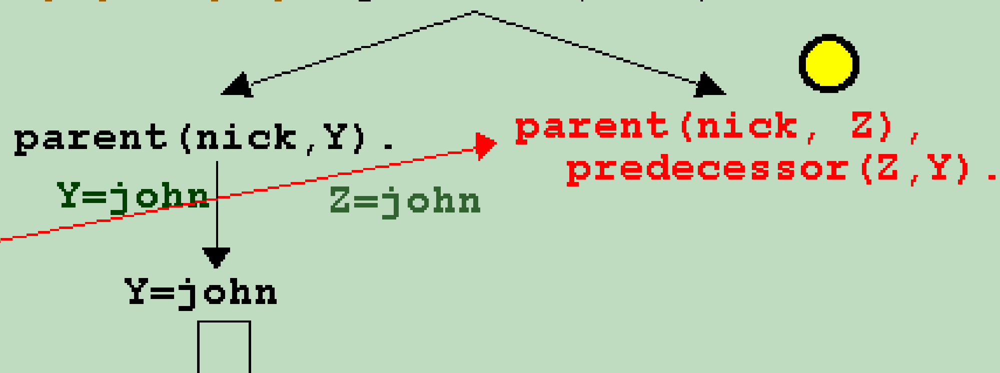
Αρχική ερώτηση: ?- predecessor(nick,Y).



Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
           parent(X,Z).
4:predecessor(X,Z):-
        parent(X,Y),
        predecessor(Y,Z).
```

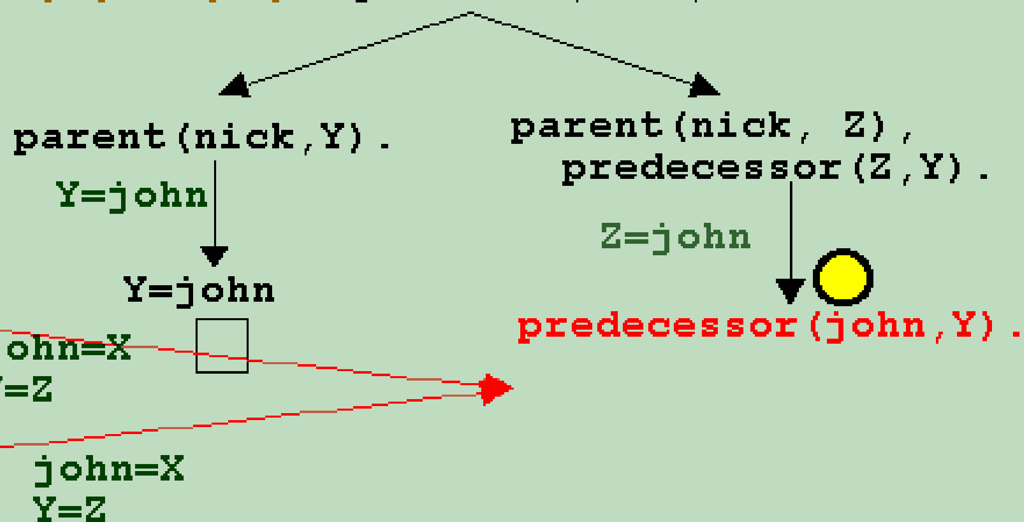
Αρχική ερώτηση: ?- predecessor(nick,Y).



Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
              parent(X,Z).
4:predecessor(X,Z):-
          parent(X,Y),
          predecessor(Y,Z).
```

Αρχική ερώτηση: ?- predecessor(nick,Y).



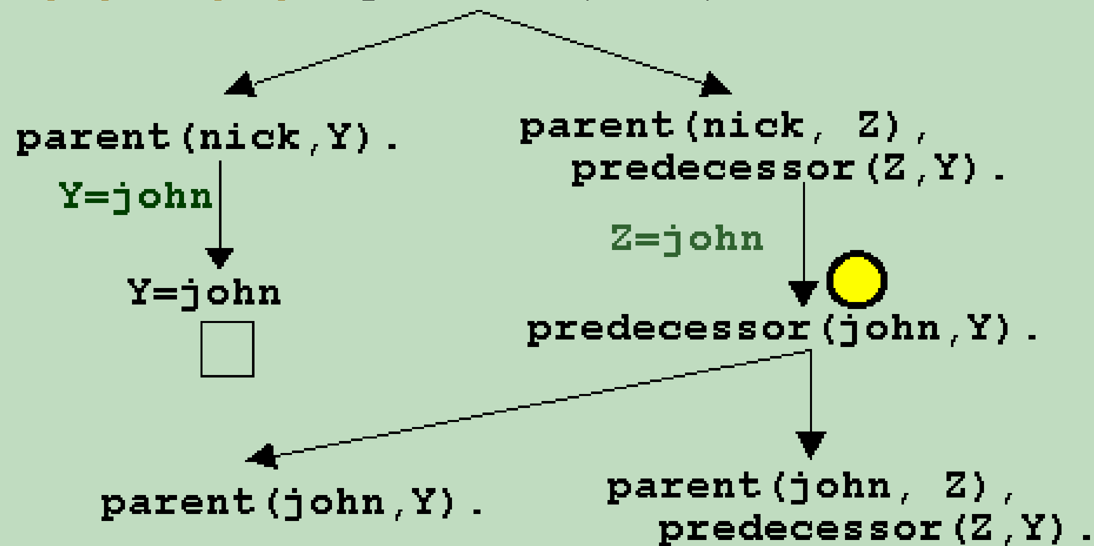
Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
              parent(X,Z).
4:predecessor(X,Z):-
          parent(X,Y),
          predecessor(Y,Z).
```

Αρχική ερώτηση: ?- predecessor(nick,Y).

Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
          parent(X,Z).
4:predecessor(X,Z):-
        parent(X,Y),
        predecessor(Y,Z).
```



Αρχική ερώτηση: ?- predecessor(nick,Y).

<u>Το πρόγραμμα:</u>

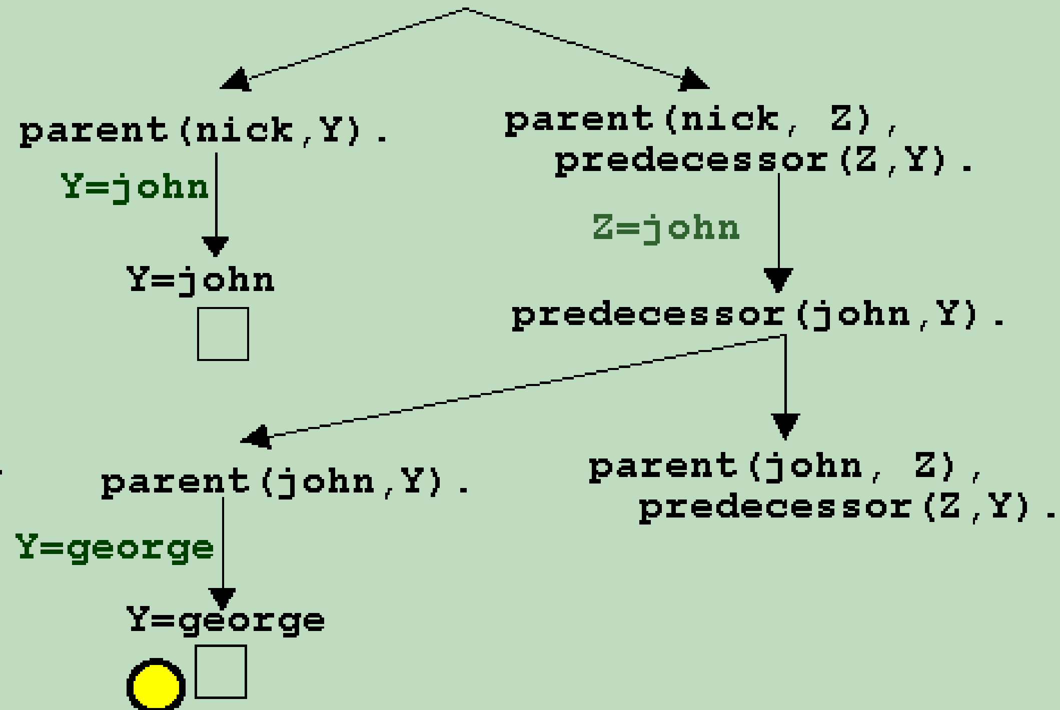
```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
            parent(X,Z).
4:predecessor(X,Z):-
        parent(X,Y),
        predecessor(Y,Z).
```

parent(nick,Y).
Y=john
Y=john

parent(nick, Z),
predecessor(Z,Y).
Z=john
predecessor(john,Y).

Y=george
parent(john,Y).

parent(john, Z),
predecessor(Z,Y).


Αρχική ερώτηση: ?- predecessor(nick,Y).
Το πρόγραμμα:
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
parent(X,Z).
4:predecessor(X,Z):-
parent(X,Y),
predecessor(Y,Z).
parent(nick,Y).
Y=john
Y=john
parent(nick, Z),
predecessor(Z,Y).
Z=john
predecessor(john,Y).
parent(john,Y).
Y=george
Y=george
parent(john, Z),
predecessor(Z,Y).

Αρχική ερώτηση: ?- predecessor(nick,Y).

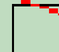
Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
        parent(X,Z).
4:predecessor(X,Z):-
        parent(X,Y),
        predecessor(Y,Z).
```

parent(nick,Y).

Y=john

Y=john



parent(nick, Z),
predecessor(Z,Y).

Z=john

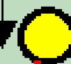
predecessor(john,Y).

Z=george

parent(john,Y).

Y=george

Y=george



parent(john, Z),
predecessor(Z,Y).

Αρχική ερώτηση: ?- predecessor(nick,Y).

Το πρόγραμμα:

```
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
        parent(X,Z).
4:predecessor(X,Z):-
        parent(X,Y),
        predecessor(Y,Z).
```

parent(nick,Y).

Y=john

Y=john

parent(nick, Z),
predecessor(Z,Y).

Z=john

predecessor(john,Y).

george=X
Y=Z

parent(john,Y).

Y=george

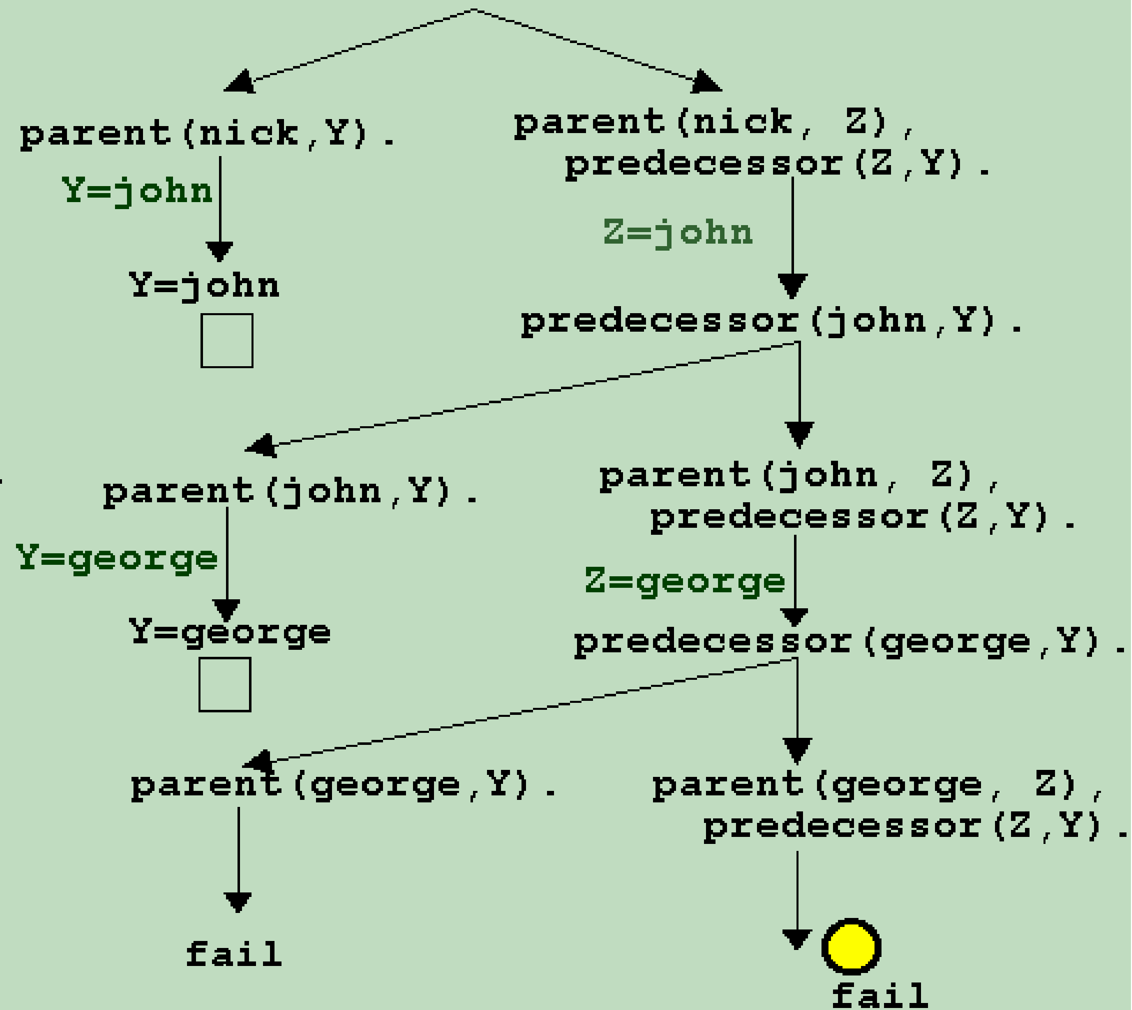
Y=george

parent(john, Z),
predecessor(Z,Y).

george=X
Y=Z

Z=george

predecessor(george,Y).


Αρχική ερώτηση: ?- predecessor(nick,Y).
Το πρόγραμμα:
1: parent(john, george).
2: parent(nick, john).
3: predecessor(X,Z):-
parent(X,Z).
4:predecessor(X,Z):-
parent(X,Y),
predecessor(Y,Z).
parent(nick,Y).
parent(nick, Z),
predecessor(Z,Y).
Y=john
Y=john
Z=john
predecessor(john,Y).
parent(john,Y).
parent(john, Z),
predecessor(Z,Y).
Y=george
Y=george
Z=george
predecessor(george,Y).
parent(george,Y).
parent(george, Z),
predecessor(Z,Y).
fail
fail

Αναδρομικές Δομές Δεδομένων (Λίστες) στη Prolog

- Οι σύνθετοι όροι χρησιμεύουν στην αναπαράσταση σύνθετων οντοτήτων ενός προβλήματος, όπως για παράδειγμα ο όρος:

  triangle(point(0,0), point(1,4), point(5,12))

  ο οποίος μπορεί να αναπαραστήσει ένα τρίγωνο.

- Μια ειδική κατηγορία αναδρομικών σύνθετων όρων είναι η λίστα (list). Πρόκειται για μια σύνθετη δομή που έχει συναρτησιακό σύμβολο (functor) την τελεία «.» ενώ τα ορίσματά της μπορεί να είναι οποιοδήποτε όροι, ακόμη και άλλες λίστες.

Λίστες στην Prolog

- Μια λίστα Prolog αναπαρίσταται με μια ακολουθία από οποιονδήποτε αριθμό στοιχείων. Τα στοιχεία της λίστας τοποθετούνται μέσα σε αγκύλες «[ ]» και χωρίζονται μεταξύ του με κόμμα «,». Για παράδειγμα, η λίστα με τρία στοιχεία, το a το b και το c αναπαρίσταται ως [a,b,c].

- Παραδείγματα:

  [ice_cream, coffee, chocolate] a list with three elements (all atoms)

  [a, b, c, c, d, e] a list with six elements (all atoms)

  [ ] a list with no elements in it (it is an atom)

  [dog(fido), cat(rufus), goldfish(jimmy)] a list with three elements (all Prolog terms)

  *[happy(fred),[ice_cream,chocolate],[1,[2],3]]*a list with three elements! (first element is happy(fred), the second is [ice cream,chocolate], a list, and the third is [1,[2],3], another list.)

Είδη Λιστών στην Prolog

- Μία λίστα μπορεί να είναι:
  - κενή (δηλαδή μία δομή χωρίς όρους), η οποία συμβολίζεται με []
  - μια δομή με δύο όρους: την κεφαλή (head) που είναι το πρώτο στοιχείο της λίστας και την ουρά (tail) που είναι το υπόλοιπο τμήμα της λίστας.

- Η λίστα που έχει κεφαλή H και ουρά T παρίσταται ως [H | T] και αντιστοιχεί στο σύνθετο όρο .(H, T).
  - Π.χ. [X | Y] = [f, r, e, d] σημαίνει: X = f, Y = [r, e, d]

- Η ουρά μιας λίστας είναι πάντα λίστα ενώ η κεφαλή μπορεί να είναι οποιοσδήποτε όρος, απλός ή σύνθετος, ακόμη και λίστα. Για παράδειγμα, η λίστα [[a, b], c] έχει κεφαλή τη λίστα [a, b] και ουρά τη λίστα [c].

Παράδειγμα Διαφορετικών Τρόπων Αναπαράστασης Λίστας

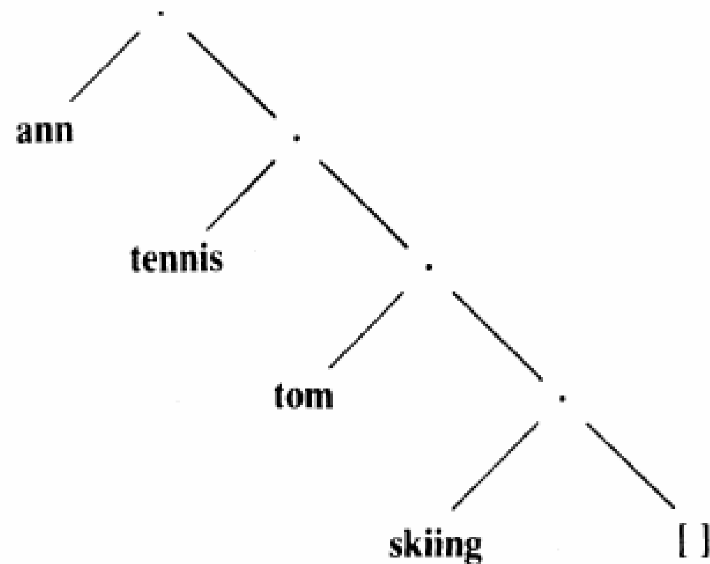
[ann, tennis, tom, skiing]

head is ann and tail is [tennis, tom, skiing]

or

.(ann, .(tennis, .(tom, .(skiing, [])))))

or



Παρατήρηση

- Note that the empty list appears in the above term.

- This is because the one but last tail is a single item list:

  [skiing]

- This list has the empty list as its tail:

  [skiing] = .(skiing, [])

Παραδείγματα

Λίστα	Κεφαλή	Ουρά
[a, b, c]	a	[b, c]
[1 | Y]	1	Y
[1,Y]	1	[Y]
[[a, b], c]	[a, b]	[c]
[1, 2 | X]	1	[2 | X]
[X, Y]	X	[Y]
[X | Y]	X	Y
[f(a,b), [a]]	f(a,b)	[[a]]

[a, [b, c], d]

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

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

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

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

Ενοποίηση Λιστών

Δύο λίστες ενοποιούνται εφόσον έχουν τον ίδιο αριθμό στοιχείων και εφόσον τα αντίστοιχα στοιχεία τους μπορούν να ενοποιηθούν.

Λίστα 1	Λίστα 2	Τιμές μεταβλητών
[a, b, c]	[a, b, c, d]	Fail
[a, b, c]	[X, Y]	Fail
[a, b]	[a, X]	{X = b}
[a, b]	[a | X]	{X = [b]}
[f(1), k, [1]]	[f(X), Y, Z]	{X = 1, Y=k, Z = [1]}
[f(1), k, [1]]	[f(X), Y | Z]	{X = 1, Y=k, Z = [[1]]}
[[a]]	[X | Y]	{X = [a], Y = []}

Παραδείγματα Ενοποίησης Λιστών

1. [b, a, d] = [d, a, b] fails as the order matters

2. [X] = [b, a, d] fails, the two lists are of different lengths

3. [X | Y] = [he, is, a, cat] succeeds with X = he, Y = [is, a, cat]

4. [X, Y | Z] = [a, b, c, d] succeeds with X = a, Y = b, Z = [c, d]

5. [X | Y] = [] fails, the empty list can't be deconstructed

6. [X | Y]=[[a, [b, c]], d] succeeds with X = [a, [b, c]], Y = [d]

7. [X | Y] = [a] succeeds with X = a, Y = []

Παραδείγματα Ενοποίησης Λιστών 2

1. [a, b | X] = [A, B, c] succeeds with A = a, B = b and X = [c] (και όχι X = c).

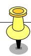
2. [a, b] = [b, a] fails.

3. [a | [b, c]] = [a, b, c] succeeds.

4. [a, [b, c]] = [a, b, c] fails (first list has two elements and the second has three).

5. [a, X] = [X, b] fails. (The first element of each list (the heads) can be unified with X = a. Looking at the second elements, we need to unify X with b, but X = a so the process fails.)

6. [a | []] = [X] succeeds with X = a. (The list [a | []] is exactly equivalent to [a]. Therefore the problem becomes [a] = [X]. This unifies with X = a.



7. [a, b, X, c] = [A, B, Y] fails. (The simple answer is that the left hand list has four elements and the right has three, therefore these two lists will not unify. To see why, we match the head elements, we get A = a. Throwing away the heads, we end up with [b, X, c] = [B, y]. Repeating, we have B = b. Again, discarding the heads, we have [X, c] = [y]. Repeating we get X = y. We now end up with [c] = []. Fails.

8. [H | T] = [[a, b], [c, d]] succeeds with H=[a, b] and T=[[c, d]]. (The right hand list has two elements: the first (head) is [a, b] and the second element is [c, d]. The head elements unify with H = [a, b]. If we now discard the head elements we are left with deciding whether T unifies with [[c, d]]. Succeeds with T=[[c, d]].)

9. [[X], Y] = [a, b] fails. (If we try to unify the head elements of these lists we have the problem [X] = a. This fails.)

Απλός Χειρισμός Λιστών στην Prolog

Suppose you want to stick the elements a, b and c onto the front of the list X to make a new list Y. This can be done with Y=[a, b, c | X].

Conversely, suppose you want to take three elements off the front of a list X in such a way that the remaining list, Y, is available for use.

This can be done with X=[A, B, C | Y]

Αναδρομικός Χειρισμός Λιστών στην Prolog

Ο χειρισμός των λιστών γίνεται συνήθως με αναδρομικούς κανόνες, οι οποίοι επιτελούν κάποια λειτουργία στην κεφαλή της λίστας και στη συνέχεια καλούν αναδρομικά τον ευατό τους ώστε να επιτελέσουν την ίδια λειτουργία και στην ουρά της. Συνήθως υπάρχει ένας τερματικός κανόνας ο οποίος ασχολείται με την περίπτωση της κενής λίστας.

1. Αν η λίστα είναι [] γύρνα την τιμή που αναμένεται για το []
2. Αλλιώς, κόψε την λίστα σε δύο τμήματα (head, tail) δούλεψε με το head, δούλεψε αναδρομικά με το tail και στο τέλος συνδύασε τα αποτελέσματα.

Παραδείγματα χειρισμού λιστών στην Prolog

Π.χ. Εύρεση τελευταίου στοιχείου λίστας.

Για να είναι ένα στοιχείο τελευταίο μίας λίστας θα πρέπει:

1. είτε να είναι το μοναδικό στοιχείο μίας λίστας
2. είτε να είναι το τελευταίο στοιχείο της ουράς της

Το κατηγόρημα last που ακολουθεί δέχεται δύο ορίσματα:

Το πρώτο όρισμα, είτε είναι μία μεταβλητή, είτε είναι ένας συν. όρος.

Το δεύτερο όρισμα είναι η λίστα.

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

?- last(3, []).

no

?- last(2, [1, a2]).

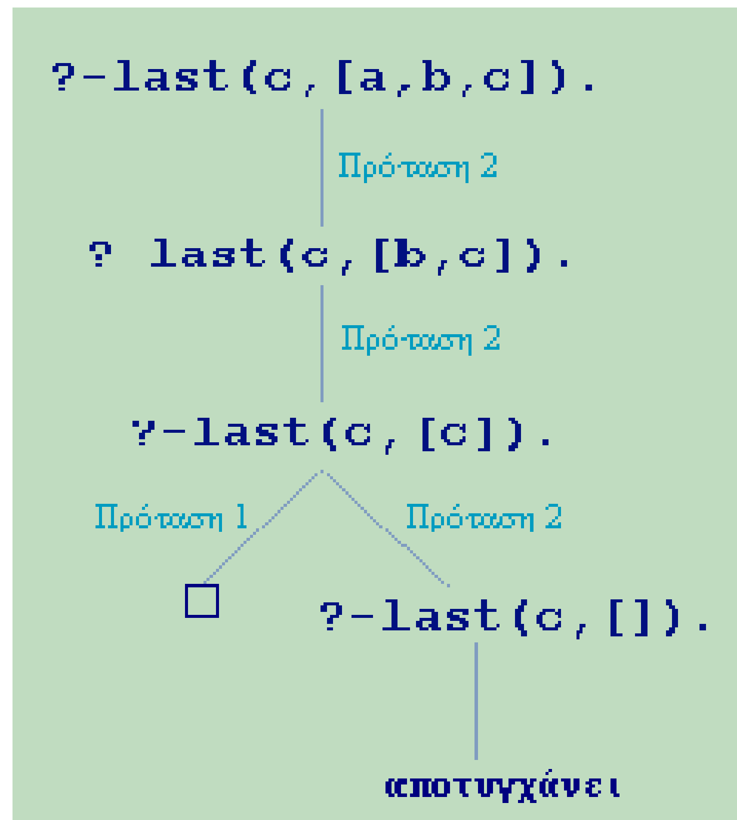
no

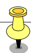
?- last(X, [2, a, 5, abc, 7]).

X=7

?- last(c, [a, b, c]).

yes





Πρόγραμμα

Η ενοποίηση αποτυγχάνει

?- last(X,[2,a,5,abc,7]). *Πρώτη Κλήση*

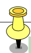
```
last(X,[X]).            ?- last(X,[2,a,5,abc,7]).

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

Πρόγραμμα

?- last(X,[2,a,5,abc,7]). *Πρώτη Κλήση*

```
last(X,[X]).                      {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[Head|Tail]):-   ?- last(X,[2,a,5,abc,7]).
      last(X,Tail).
```

Η ενοποιήση πετυχαίνει με την πρόταση 2 του προγράμματος, και παράγονται οι απαραίτητες αντικαταστάσεις μεταβλητών.



Πρόγραμμα

```
last(X,[X]).

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

```
?- last(X,[2,a,5,abc,7]).
    {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
```

Δεύτερη Κλήση

Το σώμα του κανόνα με τις αντικαταστάσεις μεταβλητών που προήλθαν από το προηγούμενο βήμα της εκτέλεσης, αποτελεί την νέα κλήση.

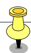
Πρόγραμμα

Η ενοποίηση αποτυγχάνει

```
last(X,[X]).          last(X,[a,5,abc,7]).

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

```
?- last(X,[2,a,5,abc,7]).
    {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
```

Δεύτερη Κλήση



Πρόγραμμα

```
last(X,[X]).

last(X,[Head|Tail]):-    last(X,[a,5,abc,7]).
      last(X,Tail).
```

Η ενοποιήση πετυχαίνει με την πρόταση 2 του προγράμματος, και παράγονται οι απαραίτητες αντικαταστάσεις μεταβλητών.

```
?- last(X,[2,a,5,abc,7]).
      {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
      {X=X, Head=a,Tail=[5,abc,7]}
```

Δεύτερη Κλήση

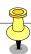
Πρόγραμμα

```
last(X,[X]).

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

Το σώμα του κανόνα με τις αντικαταστάσεις μεταβλητών που προήλθαν από το προηγούμενο βήμα της εκτέλεσης, αποτελεί την νέα κλήση.

```
?- last(X,[2,a,5,abc,7]).
      {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
      {X=X, Head=a,Tail=[5,abc,7]}

last(X,[5,abc,7]).
```

Τρίτη Κλήση



Πρόγραμμα

Η ενοποίηση αποτυγχάνει

```
last(X,[X]).          last(X,[5,abc,7]).

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

```
?- last(X,[2,a,5,abc,7]).
      {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
      {X=X, Head=a,Tail=[5,abc,7]}

last(X,[5,abc,7]).
```

Τρίτη Κλήση

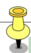
Πρόγραμμα

```
last(X,[X]).

last(X,[Head|Tail]):-   last(X,[5,abc,7]).
      last(X,Tail).
```

Η ενοποιήση πετυχαίνει με την πρόταση 2 του προγράμματος, και παράγονται οι απαραίτητες αντικαταστάσεις μεταβλητών.

```
?- last(X,[2,a,5,abc,7]).
      {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
      {X=X, Head=a,Tail=[5,abc,7]}

last(X,[5,abc,7]).
      {X=X, Head=5,Tail=[abc,7]}
```

Τρίτη Κλήση



Πρόγραμμα

```
last(X,[X]).

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

Το σώμα του κανόνα με τις αντικαταστάσεις μεταβλητών που προήλθαν από το προηγούμενο βήμα της εκτέλεσης, αποτελεί την νέα κλήση.

```
?- last(X,[2,a,5,abc,7]).
      {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
      {X=X, Head=a,Tail=[5,abc,7]}

last(X,[5,abc,7]).
      {X=X, Head=5,Tail=[abc,7]}

last(X,[abc,7]).
```

Τέταρτη Κλήση

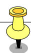
Πρόγραμμα

Η ενοποίηση αποτυγχάνει

```
last(X,[X]).            last(X,[abc,7]).

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

```
?- last(X,[2,a,5,abc,7]).
      {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
      {X=X, Head=a,Tail=[5,abc,7]}

last(X,[5,abc,7]).
      {X=X, Head=5,Tail=[abc,7]}

last(X,[abc,7]).
```

Τέταρτη Κλήση



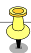
Πρόγραμμα

```
last(X,[X]).

last(X,[Head|Tail]):-   last(X,[abc,7]).
      last(X,Tail).
```

Η ενοποιήση πετυχαίνει με την πρόταση 2 του προγράμματος, και παράγονται οι απαραίτητες αντικαταστάσεις μεταβλητών.

```
?- last(X,[2,a,5,abc,7]).
      {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
      {X=X, Head=a,Tail=[5,abc,7]}

last(X,[5,abc,7]).
      {X=X, Head=5,Tail=[abc,7]}

last(X,[abc,7]).
      {X=X, Head=abc,Tail=[7]}
```

Τέταρτη Κλήση



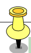
Πρόγραμμα

```
last(X,[X]).

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

Το σώμα του κανόνα με τις αντικαταστάσεις μεταβλητών που προήλθαν από το προηγούμενο βήμα της εκτέλεσης, αποτελεί την νέα κλήση.

```
?- last(X,[2,a,5,abc,7]).
    {X=X, Head=2,Tail=[a,5,abc,7]}

last(X,[a,5,abc,7]).
    {X=X, Head=a,Tail=[5,abc,7]}

last(X,[5,abc,7]).
    {X=X, Head=5,Tail=[abc,7]}

last(X,[abc,7]).
    {X=X, Head=abc,Tail=[7]}

last(X,[7]).
```

Πέμπτη Κλήση



Πρόγραμμα

```
last(X,[X]).

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

last(X,[7]).

Η κλήση ενοποιείται με το πρώτη πρόταση του προγράμματος και σταματά η αναδρομή.

```
?- last(X,[2,a,5,abc,7]).
      =2,Tail=[a,5,abc,7]}

                  ]).
      {X=X, Head=a,Tail=[5,abc,7]}

last(X,[5,abc,7]).
      {X=X, Head=5,Tail=[abc,7]}

last(X,[abc,7]).
      {X=X, Head=abc,Tail=[7]}

last(X,[7]).
      {X=7,[X]=[7]}
```

Πέμπτη Κλήση

Παραδείγματα χειρισμού λιστών στην Prolog

Π.χ. member(X, [X | Tail]).

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

Υπολογισμός αν κάποιο στοιχείο είναι μέλος λίστας: member(X,L)

Ερωτήσεις:

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

Yes

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

Yes

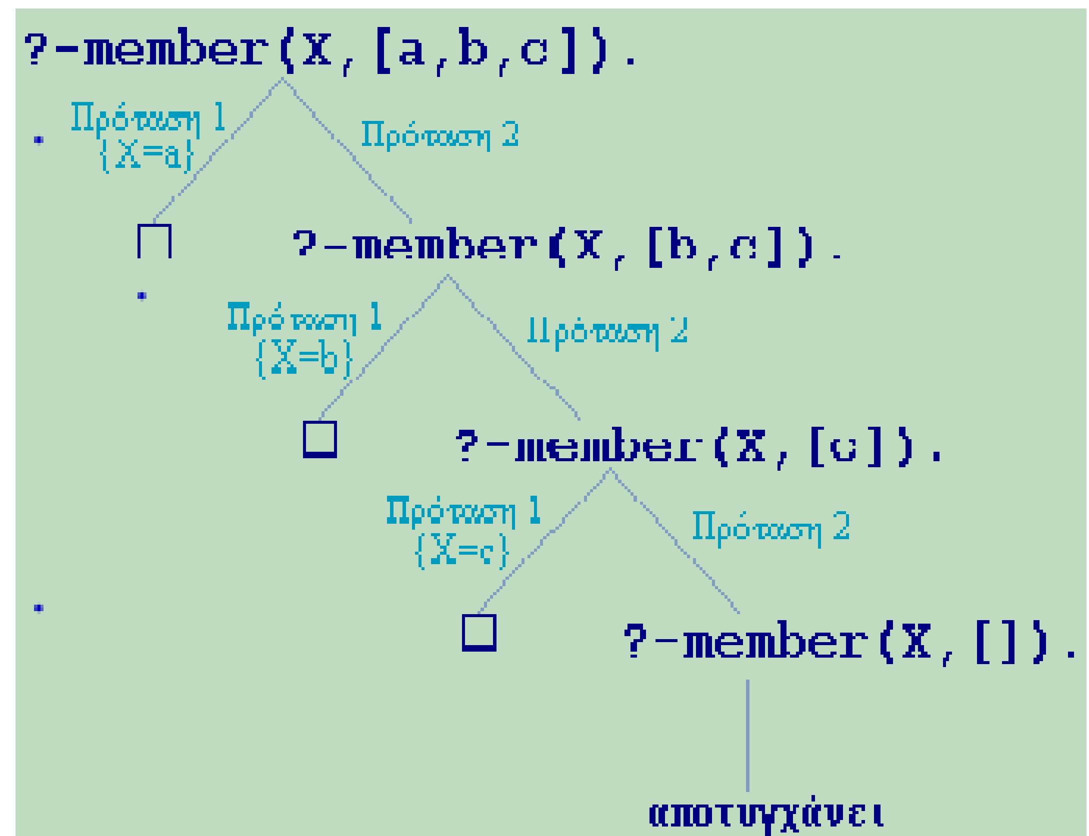
?- member(X, [a, b, c]).

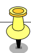
X = a;

X = b;

X = c;

no





Πρόγραμμα

```
member(X,[X|Rest]).

member(X,[Y| Rest]):
        member(X,Rest).
```

?-member(4,[1,3,4,5])
.

?-member(4,[1,3,4,5])
.

{X=4, Y= 1 Rest= [3,4,5]}. ← *Αντικαταστάσεις*

Η κλήση ταυτοποιείται με την κεφαλή της δεύτερης πρότασης του προγράμματος και παράγονται οι απαραίτητες αντικαταστάσεις (unifications).

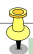
Γιατί δεν ενοποιήθηκε η κλήση με την πρώτη πρόταση;

Πρόγραμμα

```
member(X,[X|Rest]).

member(X,[Y| Rest]):
        member(X,Rest).
```

```
?-member(4,[1,3,4,5])
.
        {X=4, Y= 1 Rest= [3,4,5]}.

member(4,[3,4,5]).
```

Το σώμα του κανόνα με τις αντικαταστάσεις που προέκυψαν από το προηγούμενο βήμα δημιουργεί την επόμενη κλήση.



Πρόγραμμα

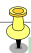
```
member(X,[X|Rest]).

member(X,[Y| Rest]):          member(4,[3,4,5]).
      member(X,Rest).
```

```
?-member(4,[1,3,4,5])
.
      {X=4, Y= 1 Rest= [3,4,5]}.
```

```
member(4,[3,4,5]).

      {X'=4, Y'=3, Rest'=[4,5]}.
```

← *Αντικαταστάσεις*

Και η δεύτερη κλήση ταυτοποιείται με την κεφαλή της δεύτερης πρότασης του προγράμματος και παράγονται οι απαραίτητες αντικαταστάσεις (unifications).

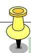


Πρόγραμμα

```
member(X,[X|Rest]).

member(X,[Y| Rest]):
        member(X,Rest).
```

```
?-member(4,[1,3,4,5])
.
        {X=4, Y= 1 Rest= [3,4,5]}.

member(4,[3,4,5]).

        {X'=4, Y'=3, Rest'=[4,5]}.

member(4,[4,5]).
```

Το σώμα του κανόνα με τις αντικαταστάσεις που προέκυψαν από το προηγούμενο βήμα δημιουργεί την επόμενη κλήση.



Πρόγραμμα

```
member(X,[X|Rest]).  member(4,[4,5]).

member(X,[Y| Rest]):
       member(X,Rest).
```

Η τρίτη κλήση ταυτοποιείται με την πρώτη πρόταση του προγράμματος, η οποία είναι γεγονός οπότε και η αναδρομή σταματά.

```
?-member(4,[1,3,4,5])
.
        {X=4, Y= 1 Rest= [3,4,5]}.

member(4,[3,4,5]).

        {X'=4, Y'=3, Rest'=[4,5]}.

member(4,[4,5]).

        {X''=4, Rest'' = [5]}.
```

← *Αντικαταστάσεις*

Επιτυχία

Παραδείγματα χειρισμού λιστών στην Prolog

Π.χ. append([], L, L).

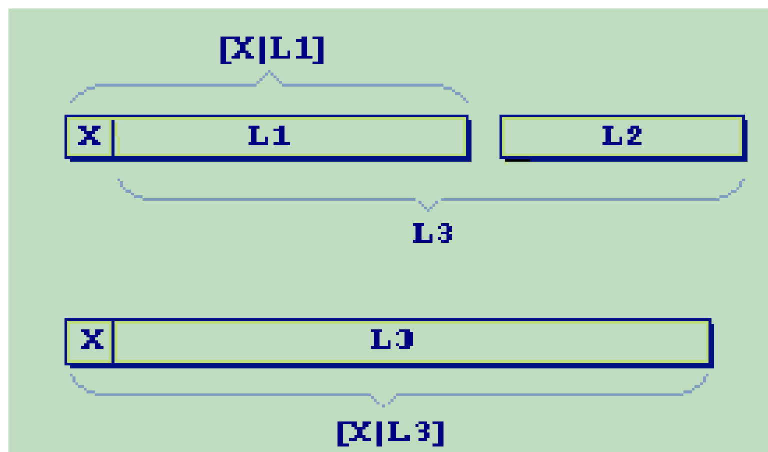
append([X | L1], L2, [X | L3]) :- append(L1, L2, L3).

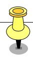
Συνένωση λιστών: append(L1, L2, L3) (βάλε το L1 στο L2 με αποτέλεσμα το L3)

Η συνένωση λιστών βασίζεται σε δύο παρατηρήσεις:

1. Η συνένωση μίας λίστα L με μία κενή λίστα είναι ο εαυτός της.

2. Αν η πρώτη λίστα δεν είναι κενή, μπορεί να χωριστεί σε κεφαλή και ουρά [X | L1]. Τότε η συνένωση της λίστας [X | L1] και της λίστα L2 είναι η λίστα [X | L3] όπου L3 είναι το αποτέλεσμα της συνένωσης των λιστών L1, L2.





Ερωτήσεις:

?- append(L, L, [a,b,d,a,b,d]).

L = [a,b,d]

?- append([a,b], [c], L3).

L3 = [a,b,c]

?- append(X, [b,c], [a,d,b,c]).

X = [a,d].

Πρόγραμμα

```
append([],List,List).

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

?- append([a,b],[c,d],L).

?- append([a,b],[c,d],L).

{X=a, L1= [b], L2 = [c,d], L = [a|L3]}. ← *Αντικαταστάσεις*

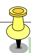
Η κλήση ταυτοποιείται με την κεφαλή της δεύτερης πρότασης του προγράμματος και παράγονται οι απαραίτητες αντικαταστάσεις (unifications).

Πρόγραμμα

```
append([],List,List).

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

```
?- append([a,b],[c,d],L).

        {X=a, L1= [b], L2 = [c,d], L = [a|L3]}.

append([b],[c,d],L3).
```

Το σώμα του κανόνα με τις αντικαταστάσεις που προέκυψαν από το προηγούμενο βήμα δημιουργεί την επόμενη κλήση.



Πρόγραμμα

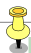
```
append([],List,List).

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

```
append([b],[c,d],L3).
```

Και η δεύτερη κλήση ταυτοποιείται με την κεφαλή της δεύτερης πρότασης του προγράμματος και παράγονται οι απαραίτητες αντικαταστάσεις (unifications).

```
?- append([a,b],[c,d],L).

{X=a, L1= [b], L2 = [c,d], L = [a|L3]}.

append([b],[c,d],L3).

{X'=b, L1'= [], L2' = [c,d], L3 = [b|L3']}.
```

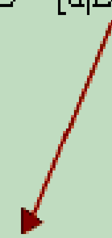
Αντικαταστάσεις



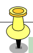
Πρόγραμμα

```
append([],List,List).

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

```
?- append([a,b],[c,d],L).

    {X=a, L1= [b], L2 = [c,d], L = [a|L3]}.

append([b],[c,d],L3).

  {X'=b, L1'= [], L2' = [c,d], L3 = [b|L3']}.

append([],[c,d],L3').
```



Το σώμα του κανόνα με τις αντικαταστάσεις που προέκυψαν από το προηγούμενο βήμα δημιουργεί την επόμενη κλήση.



Πρόγραμμα

```
append([],List,List).

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

append([],[c,d],L3').

?- append([a,b],[c,d],L).

{X=a, L1= [b], L2 = [c,d], L = [a|L3]}.

append([b],[c,d],L3).

{X'=b, L1'= [], L2' = [c,d], L3 = [b|L3']}.

Η τρίτη κλήση ταυτοποιείται με την πρώτη πρόταση του προγράμματος, η οποία είναι γεγονός οπότε και η αναδρομή σταματά.

append([],[c,d],L3').

{X''=[], List = [c,d], L3' = List = [c,d]}. ← *Αντικαταστάσεις*

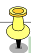
Επιτυχία

Πρόγραμμα

```
append([],List,List).

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

```
?- append([a,b],[c,d],L).

    {X=a, L1= [b], L2 = [c,d], L = [a|L3]}.

append([b],[c,d],L3).

  {X'=b, L1'= [], L2' = [c,d], L3 = [b|L3']}.
                                              {L3'=[c,d]}

append([],[c,d],L3').

    {X''=[], List = [c,d], L3' = List = [c,d]}.
```

Η λίστα που αποτελεί την συννένωση των δύο άλλων λιστών δημιουργείται κατά το "ανέβασμα" της εκτέλεσης.



Πρόγραμμα

```
append([],List,List).

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

```
?- append([a,b],[c,d],L).

        {X=a, L1= [b], L2 = [c,d], L = [a|L3]}.
                                              {L3 = [b,c,d]}

append([b],[c,d],L3).

        {X'=b, L1'= [], L2' = [c,d], L3 = [b|[c,d]]}
```

Η λίστα που αποτελεί την συννένωση των δύο άλλων λιστών δημιουργείται κατά το "ανέβασμα" της εκτέλεσης.

Πρόγραμμα

```
append([],List,List).

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

```
?- append([a,b],[c,d],L).

L = [a,b,c,d].
```

Απάντηση

Παραδείγματα χειρισμού λιστών στην Prolog

Π.χ. *print_a_list([]).*

print_a_list([H | T]) :- write(H), print_a_list(T).

Τυπώνει τα στοιχεία μίας λίστας. Το write/1είναι built-in predicate Prolog.

Π.χ. *pick(X, [X | Tail], Tail).*

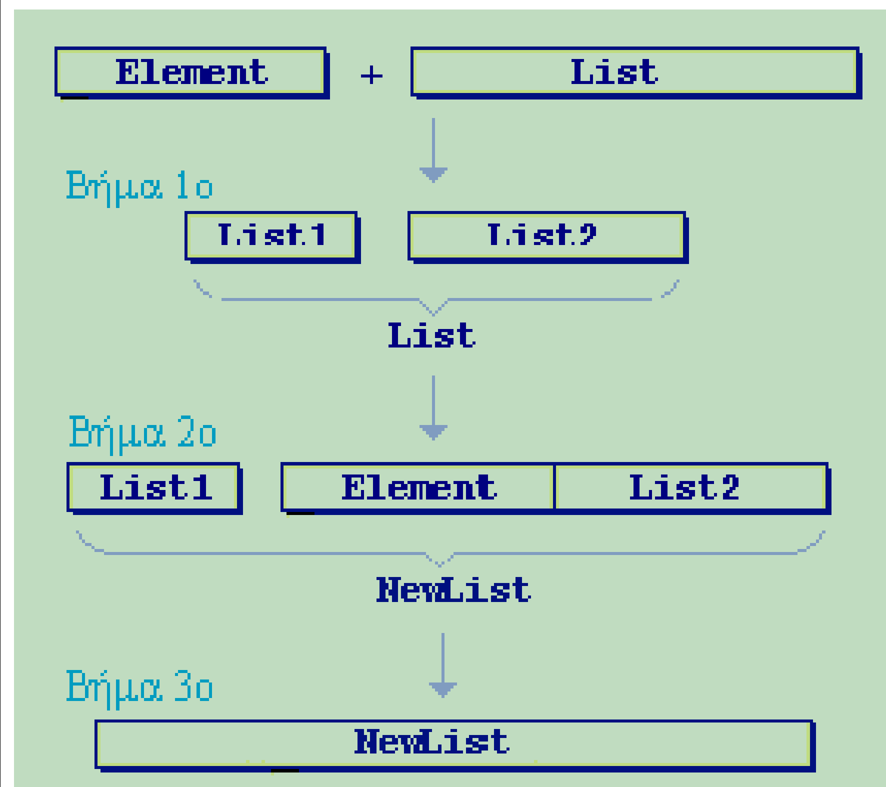
pick(X, [Head | Tail], [Head | Rem]) :- pick(X, Tail, Rem).

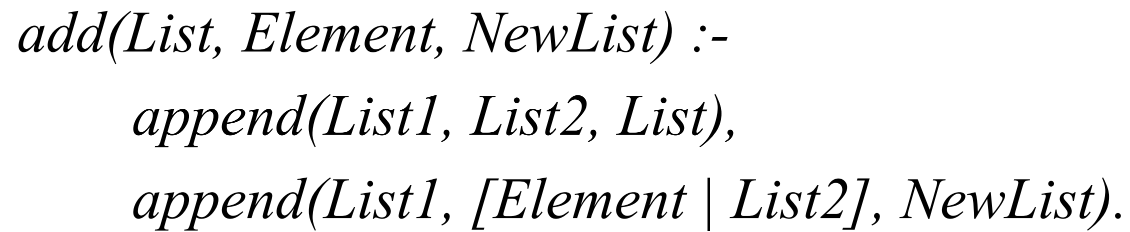
Εξαγωγή ενός στοιχείου από λίστα: pick(X, L1, L2)

Παραδείγματα χειρισμού λιστών στην Prolog

Θέλουμε να προσθέσουμε ένα στοιχείο σε μία οποιαδήποτε (όχι συγκεκριμένη) θέση μέσα σε μία λίστα.

- Βήμα 1°: Η αρχική λίστα χωρίζεται σε δύο λίστες List1 και List2 με τη βοήθεια του append/3.
- Βήμα 2°: Το στοιχείο Element που πρόκειται να προστεθεί γίνεται η κεφαλή της δεύτερης υπο-λίστας και προκύπτει η λίστα [Element | List2]
- Βήμα 3°: Συνενώνουμε την πρώτη λίστα List1 με την [Element | List2] και προκύπτει η ζητούμενη λίστα NewList.





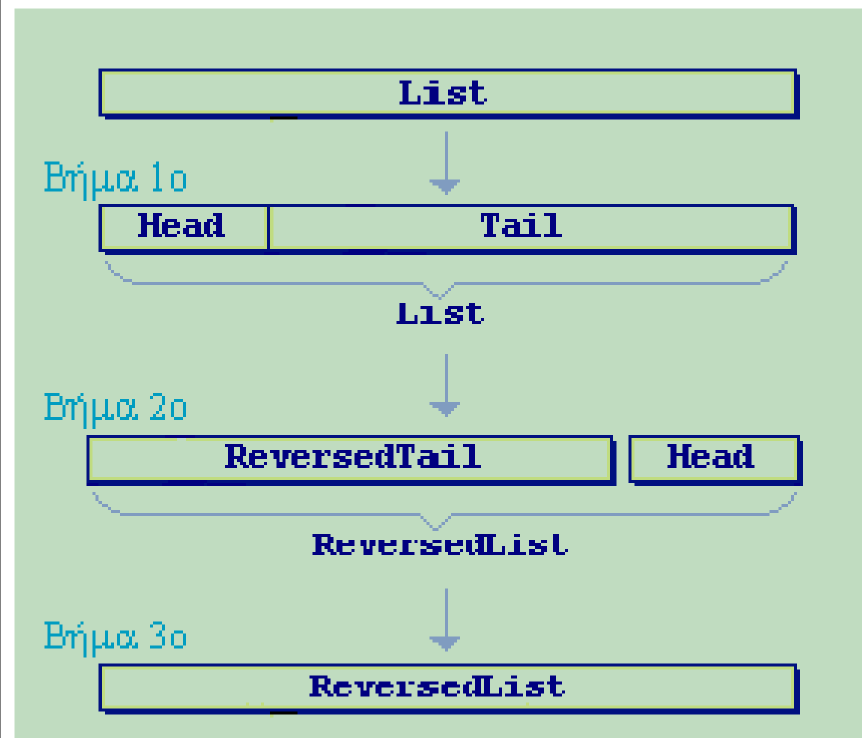
```
add(List, Element, NewList) :-
        append(List1, List2, List),
        append(List1, [Element | List2], NewList).
```

```
?- add([a, b, c], 1, L).
L = [1, a, b, c];
L = [a, 1, b, c];
L = [a, b, 1, c];
L = [a, b, c, 1];
no
```

Παραδείγματα χειρισμού λιστών στη Prolog

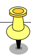
Αναστροφή λίστας

1ος Τρόπος:

❖ Βήμα 1ο: Εάν η λίστα είναι κενή, η ανεστραμμένη λίστα της είναι επίσης κενή. ❖ Βήμα 2ο: Χωρίζουμε την αρχική λίστα σε κεφαλή και ουρά. ❖ Βήμα 3ο: Αντιστρέφουμε την ουρά. ❖ Προσθέτουμε την κεφαλή της αρχικής λίστας στο τέλος της ανεστραμένης ουράς με την βοήθεια του append/3.	

```
reverse([], []).
reverse([Head | Tail], ReversedList) :-
      reverse(Tail, ReversedTail),
      append(ReversedTail, [Head], ReversedList).
```

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



2ος Τρόπος:

Στην περίπτωση αυτή, η κεφαλή της αρχικής λίστας τοποθετείται στην κεφαλή μιας βοηθητικής λίστας. Η διαδικασία επαναλαμβάνεται για την ουρά της αρχικής λίστας.

Όταν η αρχική λίστα αδειάσει, η βοηθητική λίστα περιέχει τα στοιχεία της αρχικής σε ανάστροφη σειρά – άρα είναι το τελικό αποτέλεσμα.

```
reverse(List, ReversedList) :-
        rev(List, ReversedList, []).
rev([], List, List).
rev([Head | Tail], ReversedList, TempList) :-
        rev(Tail, ReversedList, [Head | TempList]).
```

List	TempList
[1,2,3]	[]
[2,3]	[1]
[3]	[2,1]
[]	[3,2,1]

Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
        rev(Tail,Rev,[Head|TempList]).
```

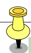
?- reverse([a,b,c],RList).

?- reverse([a,b,c],RList).

{List = [a,b,c], RevList=RList}.

Αντικαταστάσεις

Η κλήση ταυτοποιείται με το αντίστοιχο κατηγόρημα του προγράμματος και παράγονται οι απαραίτητες αντικαταστάσεις (unifications).



Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
        rev(Tail,Rev,[Head|TempList]).
```

```
?- reverse([a,b,c],RList).
        {List = [a,b,c], RevList=RList}.

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

Το σώμα του κανόνα με τις αντικαταστάσεις που προέκυψαν από το προηγούμενο βήμα δημιουργεί την επόμενη κλήση, στο κατηγόρημα rev/3.

Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
    rev(Tail,Rev,[Head|TempList]).
```

```
?- reverse([a,b,c],RList).
    {List = [a,b,c], RevList=RList}.

rev([a,b,c],RevList,[]).
    {Head=a, Tail=[b,c], Rev=RevList, TempList =[]}.
```

rev([a,b,c],RevList,[]).

Η κλήση ταυτοποιείται με την κεφαλή της δεύτερης πρότασης του κατηγορήματος rev/3 και παράγονται οι απαραίτητες αντικαταστάσεις (unifications).

Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
        rev(Tail,Rev,[Head|TempList]).
```

Το σώμα του κανόνα με τις αντικαταστάσεις που προέκυψαν από το προηγούμενο βήμα δημιουργεί την επόμενη κλήση.

```
?- reverse([a,b,c],RList).
        {List = [a,b,c], RevList=RList}.

  rev([a,b,c],RevList,[]).
      {Head=a, Tail=[b,c], Rev=RevList, TempList =[]}.

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

Πρόγραμμα

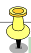
```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
    rev(Tail,Rev,[Head|TempList]).
```

rev([b,c],Rev,[a]).

Και η δεύτερη κλήση ταυτοποιείται με την κεφαλή της δεύτερης πρότασης του κατηγορήματος και παράγονται οι απαραίτητες αντικαταστάσεις (unifications).

```
?- reverse([a,b,c],RList).
    {List = [a,b,c], RevList=RList}.

rev([a,b,c],RevList,[]).
    {Head=a, Tail=[b,c], Rev=RevList, TempList =[]}.

rev([b,c],Rev,[a]).
    {Head'=b, Tail' = [c], Rev'=Rev, TempList'=[a]}.
```

Αντικαταστάσεις



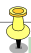
Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
        rev(Tail,Rev,[Head|TempList]).
```

Το σώμα του κανόνα με τις αντικαταστάσεις που προέκυψαν από το προηγούμενο βήμα δημιουργεί την επόμενη κλήση.

```
?- reverse([a,b,c],RList).
      {List = [a,b,c], RevList=RList}.

 rev([a,b,c],RevList,[]).
    {Head=a, Tail=[b,c], Rev=RevList, TempList =[]}.

 rev([b,c],Rev,[a]).
    {Head'=b, Tail' = [c], Rev'=Rev, TempList'=[a]}.
```

rev([c],Rev',[b,a]).



Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
    rev(Tail,Rev,[Head|TempList]).
```

Η τρίτη κλήση της rev/3 ταυτοποιείται επίσης με την δεύτερη πρόταση του κατηγορήματος rev/3, και παράγονται οι απαραίτητες

```
?- reverse([a,b,c],RList).
    {List = [a,b,c], RevList=RList}.

rev([a,b,c],RevList,[]).
    {Head=a, Tail=[b,c], Rev=RevList, TempList =[]}.

rev([c],Rev',[b,a]).

rev([b,c],Rev,[a]).
    {Head'=b, Tail' = [c], Rev'=Rev, TempList'=[a]}.

rev([c],Rev',[b,a]).
    {Head''=c, Tail'' = [ ], Rev '' = Rev', TempList''=[b,a]}.
```

Αντικαταστάσεις

Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).   rev([ ],Rev'',[c,b,a]).

rev([Head|Tail],Rev,TempList):-
        rev(Tail,Rev,[Head|TempList]).
```

Η τέταρτη κλήση της rev/3 ταυτοποιείται με την πρώτη πρόταση του κατηγορήματος, η οποία είναι γεγονός οπότε και η αναδρομή σταματά.

```
?- reverse([a,b,c],RList).
      {List = [a,b,c], RevList=RList}.

rev([a,b,c],RevList,[]).
    {Head=a, Tail=[b,c], Rev=RevList, TempList =[]}.

rev([b,c],Rev,[a]).
    {Head'=b, Tail' = [c], Rev'=Rev, TempList'=[a]}.

rev([c],Rev',[b,a]).
    {Head''=c, Tail'' = [ ], Rev '' = Rev', TempList''=[b,a]}.

rev([ ],Rev'',[c,b,a]).

{[ ]=[ ], Rev'''= Rev'', Rev''' = [c,b,a]}.
```

Αντικαταστάσεις

Επιτυχία

Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
        rev(Tail,Rev,[Head|TempList]).
```

```
?- reverse([a,b,c],RList).
        {List = [a,b,c], RevList=RList}.

  rev([a,b,c],RevList,[]).
      {Head=a, Tail=[b,c], Rev=RevList, TempList =[]}.

   rev([b,c],Rev,[a]).
      {Head'=b, Tail' = [c], Rev'=Rev, TempList'=[a]}.
                                   {Rev ' = [c,b,a]}
```

Η ανάστροφη λίστα "ανεβαίνει" στα υψηλότερα επίπεδα της εκτέλεσης μέσω της ισότητας των μεταβλητών Rev''', Rev'', Rev' και Rev'.

Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
        rev(Tail,Rev,[Head|TempList]).
```

```
?- reverse([a,b,c],RList).
        {List = [a,b,c], RevList=RList}.
                                   {RevList = [c,b,a]}
```

Η ανάστροφη λίστα "ανεβαίνει" στα υψηλότερα επίπεδα της εκτέλεσης μέσω της ισότητας των μεταβλητών Rev''', Rev'', Rev' και Rev'.

Πρόγραμμα

```
reverse(List,RevList):-
    rev(List,RevList,[]).

rev([],Rev,Rev).

rev([Head|Tail],Rev,TempList):-
        rev(Tail,Rev,[Head|TempList]).
```

```
?- reverse([a,b,c],RList).

   RList = [c,b,a].
```

Απάντηση

Αριθμητικοί Τελεστές Σύγκρισης

X > Y	Το X είναι μεγαλύτερο του Y
X < Y	Το X είναι μικρότερο του Y
X >= Y	Το X είναι μεγαλύτερο ή ίσο του Y
X =< Y	Το X είναι μικρότερο ή ίσο του Y
X =:= Y	Οι αριθμητικές τιμές του X και Y είναι ίσες
X =\= Y	Οι αριθμητικές τιμές του X και Y δεν είναι ίσες

Οι τελεστές αυτοί μπορούν να χρησιμοποιούνται στα σώματα κανόνων με σκοπό τον έλεγχο συνθηκών μεταξύ αριθμών και μεταβλητών.

Various kinds of Equality in Prolog

When do we consider two terms to be equal?

There are three kinds of equality in prolog.

The first is based on matching, written as:

X = Y

This is true if X and Y match. In contrast, when terms do not match we write

X \= Y

We also have:

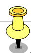
El =:= E2

This is true if the values of the arithmetic expressions E1 and E2 are equal. In contrast, when the values of two arithmetic expressions are not equal, we write

El =\= E2

Sometimes we are interested in a stricter kind of equality: the literal equality of two terms. This kind of equality is implemented as another built-in predicate written as an infix operator '=='

T1 == T2



This is true if terms T1 and T2 are identical; that is, they have exactly the same structure and all the corresponding components are the same. In particular, the names of the variables also have to be the same.

The complementary relation is 'not identical', written as:

T1 \== T2

Προσοχή! Οι τελεστές = και =:= είναι διαφορετικοί

Η κλήση X = Y προκαλεί ενοποίηση των αντικειμένων X και Y (εφόσον είναι δυνατή)

Αντίθετα στην έκφραση X =:= Y λαμβάνει χώρα αριθμητικός υπολογισμός

Το κατηγόρημα \= αληθεύει αν οι δύο όροι δεν μπορούν να ενοποιηθούν

Προσοχή! Οι τελεστές \= και =\= είναι διαφορετικοί

Απλά παραδείγματα

?- 1+2 =:= 2+1.

yes

?- 1+2 = 2+1.

no

?- 1+A = B+2.

A=2

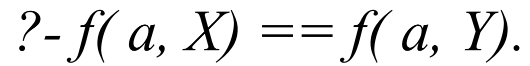
B=1

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

yes

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

no



no

?- X \== Y.

yes

?- X=Y, X == Y.

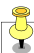
yes

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

yes

?- 3+4 == 4+3.

no



?- \+ 3+4 == 4+3. (\+ Goal succeeds if Goal does not true)

yes

?- 3+X = 3+4.

X = 4 ? yes

?- 3+X == 3+4.

no

Παράδειγμα 1

Το κατηγόρημα sorted/1 επιτυγχάνει εάν η λίστα που δέχεται σαν όρισμα εισόδου είναι ταξινομημένη σε αύξουσα σειρά, αλλιώς αποτυγχάνει. Τα βήματα στα οποία στηρίζεται η λειτουργία του κατηγορήματος είναι:

1) Εάν η λίστα έχει ένα ή κανένα στοιχείο, είναι ταξινομημένη.

2) Εάν η λίστα έχει δύο ή περισσότερα στοιχεία τότε πρέπει να ισχύουν οι ακόλουθες προϋποθέσεις:

   2α) το πρώτο στοιχείο της λίστας είναι μικρότερο ή ίσο από το δεύτερο

   2β) η ουρά της λίστας είναι ταξινομημένη (αναδρομική κλήση)

Ακολουθεί ο κώδικας του κατηγορήματος:

```
sorted([]).
sorted([_]).
sorted([X, Y | List]) :- X =< Y, sorted([Y | List]).
```

Παράδειγμα 2

Το κατηγόρημα list_max/2 βρίσκει το μέγιστο στοιχείο μιας λίστας. Βήματα:

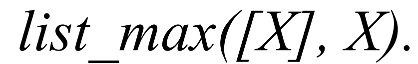
1) Εάν η λίστα έχει ένα στοιχείο, τότε αυτό είναι το μέγιστο

2) Εάν η λίστα έχει δύο ή περισσότερα στοιχεία τότε:

 2α) βρίσκει το μέγιστο στοιχείο της ουράς της (αναδρομικά)

 2β) συγκρίνει το μέγιστο στοιχείο της ουράς της με την κεφαλή της λίστας και επιστρέφει τη μέγιστη από τις δύο τιμές

Για την σύγκριση των δύο αριθμών και την επιστροφή του μεγίστου χρησιμοποιείται το κατηγόρημα max/3 που δέχεται δύο αριθμητικές τιμές σαν είσοδο και επιστρέφει το μεγαλύτερο στον τρίτο του όρισμα.



```
list_max([X], X).
list_max([X | Tail], M) :- list_max(Tail, M1), max(X, M1, M).
max(X, Y, X) :- X>Y.
max(X, Y, Y) :- X=<Y.
```

Αριθμητικοί Τελεστές

Για την εκτέλεση αριθμητικών πράξεων στην Prolog υπάρχουν οι σχετικοί τελεστές:

Τελεστής	Εξήγηση
+	Πρόσθεση
-	Αφαίρεση
*	Πολλαπλασιασμός
/	Διαίρεση
Mod	Υπόλοιπο της διαίρεσης μεταξύ ακεραίων
^	Δύναμη

The “Reversibility” of Prolog Programs

Program Database

square(1,1).

square(2,4).

square(3,9).

square(4,16).

square(5,25).

square(6,36).

There are four kinds of query:

we can ask what is the square of a specific number

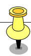
?- square(2,X).

what number has a specific square

?- square(X,5).

what entities are related by the square relation

?- square(X,Y).



whether two specific numbers are in the relation to each other of one being the square of the other.

?- square(2,3).

Unlike many other programming languages, we do not need different procedures to calculate each of these results. This is a consequence of the declarative reading of Prolog. Sometimes we say that the program for square/2 is **reversible**.

Αποτίμηση ή απόδοση τιμής στην Prolog (Evaluation)

Y = 2 + 1.

with the consequence that the term 2+1 is unevaluated and the term Y is unified with the term 2+1 with the result that Y is bound to 2+1.

Y is 2 + 1.

In this case, the term 2+1 is evaluated to 3 and Y is unified with this term resulting in Y being bound to 3.

Αποτίμηση (απόδοση τιμής) στη Prolog (Evaluation)

We can use is/2 to implement a successor relation:

```
successor(X,Y):- Y is X + 1.
```

where it is intended that successor/2 takes the first argument as input and outputs the second argument which is to be the next largest integer.

In the above, note that X + 1 is intended to be evaluated.

This means that you must use the stated calling pattern as to try to solve the goal successor(X,7) will lead to trying to evaluate X + 1 with X unbound.

This cannot be done. The result is an error message and the goal fails.

Όλες οι μεταβλητές δεξιά του is πρέπει να έχουν πάρει τιμή τη στιγμή που θα εκτελεστεί ο τελεστής is, αλλιώς η Prolog αναφέρει λάθος εκτέλεσης προγράμματος.

is/2 must always be called with its second argument as an arithmetic expression which has any variables already bound. So successor/2 is not `reversible'. For these queries,

1. successor(3,X).
2. successor(X,4).
3. successor(X,Y).
4. successor(3,5).

The 1st and 4th goals result in correct results (success and failure respectively) while the 2nd and 3rd goals produce error messages and fail.

Παράδειγμα 1

Το κατηγόρημα gcd/3 δέχεται στα δύο πρώτα ορίσματά του δύο αριθμούς και επιστρέφει στο τρίτο του όρισμα τον μέγιστο κοινό διαιρέτη (ΜΚΔ) τους.

Η λογική του είναι η ακόλουθη: εαν οι δύο αριθμοί είναι ίσοι, τότε ο ΜΚΔ είναι οι ίδιοι οι αριθμοί. Εαν οι δύο αριθμοί δεν είναι ίσοι, αφαίρεσε από τον μεγαλύτερο τον μικρότερο και υπολόγισε τον ΜΚΔ της διαφοράς και του αφαιρετέου. Αυτός είναι ο ΜΚΔ.

```
gcd(X, X, X).
gcd(X, Y, D) :- X < Y, Y1 is Y-X, gcd(X, Y1, D).
gcd(X, Y, D) :- X>Y, X1 is X-Y, gcd(X1, Y, D).
```

```
?- gcd(4, 12, D).
D = 4
```

Παράδειγμα 2

Το κατηγόρημα sum_list/2 προσθέτει όλα τα στοιχεία της λίστας που δέχεται σαν είσοδο στο πρώτο όρισμά του και επιστρέφει στο δεύτερο όρισμα του το αποτέλεσμα της άθροισης τους.

Η λογική είναι η ακόλουθη: Αν η λίστα είναι κενή, το άθροισμα των στοιχείων είναι μηδέν.

Εάν η λίστα δεν είναι κενή, τότε υπολόγισε το άθροισμα των στοιχείων της ουράς της, πρόσθεσε στο αποτέλεσμα την τιμή της κεφαλής της και επέστρεψε το τελευταίο αποτέλεσμα.

sum_list([], 0).

sum_list([H | T], Sum) :- sum_list(T, Sum1), Sum is Sum1+H.

?- sum_list([1, 2, 3, 4, 5], X).

X = 15

Παράδειγμα 3

Υπολογισμός μεγέθους λίστας: length (L, N)

length ([],0).
length ([H | Tail], N) :- length (Tail, M), N is M+1.

Παράδειγμα 4

Το κατηγόρημα n_elem/3, με παραμέτρους n_elem(N, L, X), επιτυγχάνει εάν το στοιχείο X είναι το N-οστό στοιχείο της λίστας L.

Η λογική του κατηγορήματος είναι:

Εαν N=1, και το X ταυτίζεται με την κεφαλή της λίστας, τότε τερματίζει επιτυχώς.

Εαν N>1, και N1=N-1, τότε ελέγχεται αν το X να είναι το N1-στό στοιχείο της ουράς.

```
n_elem(1, [X | T], X).
n_elem(N, [H | T], X) :- N1 is N-1, n_elem(N1, T, X).
```

```
?- n_elem(3, [a, b, c, d, e], X).
X = c
```

```
?- n_elem(N, [a, b, c, d, e], c).
error
```

Αυτό οφείλεται στο γεγονός οτι στη σχέση N1 is N-1 υπάρχουν δύο άγνωστοι. Εάν θέλαμε να είναι δυνατή η επιστροφή της θέσης ενός στοιχείου της λίστας, θα πρέπει να ορίσουμε το κατηγόρημα ως:

```
n_elem(1, [X | T], X).
n_elem(N, [H | T], X) :- n_elem(N1, T, X), N is N1+1.
```

```
?- n_elem(N, [a, b, c, d, e], c).
N = 3
```

Μερικά χρήσιμα κατηγορήματα Prolog

true/0

Always succeeds.

father(jim,fred).

is logically equivalent to

father(jim,fred):- true.

fail/0

Always fails.

lives forever(X):-fail.

is intended to mean that any attempt to solve the goal lives forever(X) will fail.

var(X) succeeds if X is currently an uninstantiated variable.

nonvar(X) succeeds if X is not a variable, or already instantiated

atom(X) is true if X currently stands for an atom

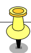
number(X) is true if X currently stands for a number

integer(X) is true if X currently stands for an integer

float(X) is true if X currently stands for a real number.

atomic(X) is true if X currently stands for a number or an atom.

compound(X)is true if X currently stands for a structure.



ground(X) succeeds if X does not contain any uninstantiated variables.

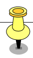
functor(T,F,N) is true if F is the principal functor of T and N is the arity of F.

arg(N,Term,A) is true if A is the Nth argument in Term.

```
?-functor(t(f(X),a,T),Func,N).
N = 3, Func = t ?
yes
```

```
?-arg(2,t(t(X),[]),A).
A = [] ?
yes
```

```
?- functor(D,date,3), arg(1,D,11), arg(2,D,oct), arg(3,D,2004).
D = date(11,oct,2004) ?
yes
```



call(X) invokes a goal X. It succeeds if X succeeds.

The goal call(X) will call the interpreter as if the system were given the goal X. Therefore X must be bound to a legal Prolog goal.

```
?- call(write(hello)).
hello
yes
```

Μερικά χρήσιμα κατηγορήματα Prolog

repeat/0

If it is asked to Redo then it will keep on succeeding.

Repeat behaves as if defined by:

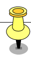
```
repeat.
repeat :- repeat.

test :- repeat, write(hello), fail.
```

The goal test produces the output:

hellohellohellohellohellohellohellohellohello...

It always succeeds the first time it is called, and it always succeeds on backtracking.

A clause body with a repeat/0 followed by fail/0 will go back and forth forever. This is one way to write an endless loop in Prolog.



A repeat/0 followed by some intermediate goals followed by a test condition will loop until the test condition is satisfied. It is equivalent to a 'do until' in other languages.

```
command_loop:-
    repeat,
    write('Enter command (end to exit): '),
    read(X),
    write(X), nl,
    X = end.
```

The last goal will fail unless end is entered. The repeat/0 always succeeds on backtracking and causes the intermediate goals to be re-executed.

Χειρισμός της άρνησης στην Prolog

Consider

man(jim).
man(fred).
?- man(bert).
no

To say that man(bert) is not true we have to assume that we known all that there is to know about man/1.

The alternative is to say the no indicates don't know and this is not a possible truth value!

Turning to Prolog, If we try to solve a goal for which there is no clause (as in the case above) then we assume that we have provided Prolog with all the necessary data to solve the problem. This is known as the Closed World Assumption.

Υπόθεση του Κλειστού Κόσμου

Η υπόθεση του κλειστού κόσμου υποστηρίζει ότι "γιά ένα κατηγόρημα τα στιγμιότυπα γιά τα οποία ισχύει το κατηγόρημα είναι μόνο εκείνα που μπορούν να αποδειχθούν με την αποδεικτική διαδικασία της Prolog".

Σύμφωνα με την υπόθεση του κλειστού κόσμου, η αποτυχία να αποδειχθεί ένα κατηγόρημα (με συγκεκριμένες παραμέτρους) συνεπάγεται ότι η αρνησή του ισχύει.

Η άρνηση στην Prolog

Με τις προτάσεις Horn δεν μπορούμε να αποδεικνύουμε αρνητικές προτάσεις.

Ωστόσο, η Prolog επιτρέπει τη χρήση άρνησης στο σώμα των κανόνων, με χρήση της δεσμευμένης λέξης *not* ή γράφοντας \+ (ανάλογα την έκδοση της Prolog που χρησιμοποιούμε):

alive(X) :- not(dead(X)).

Η λέξη not μπορεί να εμφανίζεται μόνο στο σώμα των κανόνων (όχι δηλαδή στην κεφαλή), και ερμηνεύεται ως εξής:

<u>Εάν με βάση όσα γνωρίζει το πρόγραμμα ως τώρα δεν μπορέσει να αποδείξει το dead(X), τότε μπορεί να υποθέσει ότι ισχύει το not dead(X).</u>

Η παραπάνω προσέγγιση είναι η "υπόθεση του κλειστού κόσμου", αφού υποθέτει ότι γνωρίζουμε τα πάντα σε σχέση με το συγκεκριμένο πρόβλημα, άρα ό,τι δεν μπορούμε να το αποδείξουμε δεν ισχύει.

Η άρνηση στην Prolog

We now give an example which uses a rule to define women in terms of them not being men.

Λογικά, για κάθε X που ανήκει στους ανθρώπους και για το οποίο δεν ισχύει το man(X), θα ισχύει το woman(X).

Σε Prolog:

```
man(jim).
man(fred).
woman(X):- not( man(X)).

?- woman(jim).
no
```

The strategy is: to solve the goal woman(jim) try solving man(jim). This succeeds therefore woman(jim) fails.

Σημεία Προσοχής

<u>But</u> there is a problem. Consider:

?- woman(jane).
succeeds
?- woman(anything).
succeeds
?- woman(X).

It succeeds if man(X) fails, but man(X) succeeds with X bound to jim. So woman(X) fails and, because it fails, X cannot be bound to anything.

Another problematic example:

r(a).
q(b).
p(X) :- not(r(X)).

If we now ask

?- q(X), p(X).
X = b

If we ask apparently the same question
?- p(X), q(X).
no

The reader is invited to trace the program to understand why we get different answers. The key difference between both questions is that the variable X is, in the first case, already instantiated when p(X) is executed, whereas at that point X is not yet instantiated in the second case.

Το κατηγόρημα ελέγχου cut (αποκοπή)

!/0

Το cut όταν χρησιμοποιηθεί αλλάζει την διαδικαστική συμπεριφορά των λογικών προγραμμάτων.

Η Prolog χειρίζεται το cut σύμφωνα με τον παρακάτω ορισμό.

"Σαν στόχος το cut πάντοτε επιτυγχάνει, αλλά δεσμεύει την Prolog σε όλες τις επιλογές που έχει κάνει από την στιγμή που ο αμέσως προηγούμενος στόχος είχε ενοποιηθεί με την κεφαλή του κανόνα στον οποίο βρίσκεται το cut".

Παράδειγμα αποκοπής 1

```
uncle(X,Y) :-
        brother(X,Z),
        father(Z,Y).
uncle(X,Y):-
        brother(X,Z),
        mother(Z,Y).
```

ορίζει πότε ο X είναι θείος του/της Y.
Έστω επίσης ότι έχουμε τα παρακάτω γεγονότα:

```
brother(bob, john).
brother(bob, nick).
father(john,ann).
```

Θέλουμε να ελέγξουμε εάν ο bob είναι θείος της ann. Εκτελούμε λοιπόν την ερώτηση:

```
?- uncle(bob,ann).
```

Στον πρώτο κανόνα, η παραπάνω ερώτηση πετυχαίνει, με τις ενοποιήσεις X=bob, Y=ann και Z=john.

Παράδειγμα αποκοπής 1

Η διαδικασία επιστρέφει επιτυχώς, ωστόσο θυμάται ότι πρέπει κάποια στιγμή στο μέλλον να ελέγξει και το δεύτερο κανόνα.

Με την αποκοπή μπορούμε να καθορίσουμε ότι εάν επιτύχει ο πρώτος κανόνας δεν χρειάζεται να ελεγχθεί και ο δεύτερος. Αυτό γίνεται ως εξής:

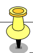
```
uncle(X,Y) :-
        brother(X,Z),
        father(Z,Y),
        !.
uncle(X,Y):-
        brother(X,Z),
        mother(Z,Y).
```

Γενικά όταν η Prolog συναντήσει μια αποκοπή, δεν ελέγχει τους επόμενους κανόνες που έχουν στην κεφαλή το ίδιο κατηγόρημα *uncle/2* (εναλλακτικές προτάσεις του ίδιου του κανόνα στον οποίο βρίσκεται), ενώ «ξεχνά» όλα τα εναλλακτικά μονοπάτια που ενδεχομένως απομένει να ελέγξει από προηγούμενες κλήσεις του ίδιου κανόνα (στην προκειμένη περίπτωση εναλλακτικές κλήσεις στα borher(X,Z) και father(Z,Y), καθώς και από προηγούμενους κανόνες, εφόσον υπήρχαν).

Παράδειγμα αποκοπής 2

a(X, Y) :- b(X, Y).
b(X, Y) :- d(X), !, e(Y).
d(c).
d(d).
d(X) :- c(X, Z), g(Z).
c(c, a).
g(a).
c(d, b).
g(b).
e(f).
e(g).
a(a, b).
b(g, h).

?- a(X, Y).
a(c, f)
a(c, g)
a(a, b)



Η Prolog θα ταιριάξει το a(X,Y) με την κεφαλή του κανόνα 1, και το b(X,Y) του κανόνα 1 με την κεφαλή του κανόνα 2.

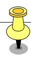
Στην συνέχεια επιλέγει το d(X) του κανόνα 2 και βρίσκει την λύση d(c) από το γεγονός 3.

Εξετάζει το cut (!) το οποίο και επιτυγχάνει, αλλά συγχρόνως δεσμεύεται τόσο στην επιλογή της λύσης d(c), όσο και στην επιλογή του κανόνα 2.

Η Prolog στην συνέχεια θα βρεί την λύση e(f) από το γεγονός 10, και τελικά θα απαντήσει a(c,f).

Αν ζητήσουμε όμως από την Prolog και άλλες λύσεις θα δώσει ακόμη δύο την a(c,g) και την a(a,b).

Γιά να βρεί την λύση a(c,g) η Prolog ενεργεί ως εξής: Αναζητεί μιά ακόμη λύση για το e(Y), και βρίσκει το e(g) (γεγονός 11).



Στην συνέχεια όμως όταν προσπαθήσει να κάνει οπισθοδρόμηση (backtracking) πάνω από το cut θα αγνοήσει άλλες λύσεις του d(X) και άλλες εναλλακτικές προτάσεις γιά το b(X, Y), και θα επιστρέψει στο a(X, Y).

Γιά το a(X,Y) θα αναζητήσει εναλλακτικές προτάσεις, και θα βρεί το γεγονός 12, δηλαδή την λύση a(a, b).

Βλέπουμε λοιπόν ότι με την χρησιμοποίηση του cut (!), η Prolog έχει "παγώσει" τις επιλογές της τόσο γιά όλα τα κατηγορήματα μεταξύ του cut και της κεφαλής του κανόνα, όσο και εναλλακτικές προτάσεις του ίδιου του κανόνα στον οποίο βρίσκεται.

Ας σημειώσουμε μόνο ότι το ίδιο πρόγραμμα χωρίς το cut θα έδινε τις ακόλουθες λύσεις:

a(g,h), a(c,f), a(c,g), a(d,f), a(d,g), a(c,f), a(c,g), a(d,f), a(d,g), a(a,b).

Η χρήση της αποκοπής:

- Αυξάνει την αποτελεσματικότητα των προγραμμάτων όταν οι κλήσεις μας γίνονται χωρίς μεταβλητές.
- Δεν μας επιστρέφει όλες τις εναλλακτικές λύσεις όταν οι κλήσεις γίνονται με μεταβλητές.
- Γενικότερα, θεωρείται ότι η χρήση της αποκοπής περιορίζει τη δηλωτικότητα (declarativeness) των προγραμμάτων, καθιστώντας τα πιο διαδικαστικά (procedural).

Παράδειγμα αποκοπής 3

Στο παρακάτω παράδειγμα η αποκοπή χρησιμοποιείται για την αποφυγή άσκοπων ελέγχων κατά την οπισθοδρόμηση, όταν γνωρίζουμε οτι μόνο μία πρόταση μπορεί να επιτύχει σε μία δεδομένη χρονική στιγμή. Το παράδειγμα υπολογίζει μία βηματική συνάρτηση.

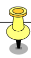
Συνάρτηση:

f(x) = 0 για x < 3

f(x) = 2 για 3 =< x < 6

f(x) = 4 για 6 =< x

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



Χρησιμοποιώντας την αποκοπή μπορούμε ακόμη και να παραλείψουμε συνθήκες σε διαδικασίες όπως η προηγούμενη. Έτσι επιτυγχάνεται η επίσπευση των προγραμμάτων καθώς εκτελούνται λιγότερες εντολές.

Για παράδειγμα το προηγούμενο πρόγραμμα μπορεί, χωρίς να μεταβληθεί η σημασία του, να γραφεί:

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

Είδη αποκοπών

Ανάλογα με το αν η αποκοπή μεταβάλει ή όχι τη δηλωτική σημασία του προγράμματος διακρίνουμε δύο είδη αποκοπών:

- Οι **κόκκινες αποκοπές (red cuts)** οι οποίες αν αφαιρεθούν από το πρόγραμμα τότε μεταβάλλεται η δηλωτική του σημασία
- Οι **πράσινες αποκοπές (green cuts)** οι οποίες αν αφαιρεθούν από το πρόγραμμα τότε δε μεταβάλλεται η δηλωτική του σημασία

Για παράδειγμα στο πρώτο παράδειγμα υλοποίησης της βηματικής συνάρτησης οι αποκοπές είναι πράσινες, ενώ στο δεύτερο πρόγραμμα είναι κόκκινες.

Προβλήματα κόκκινων αποκοπών

Αλλαγή στη σειρά που εμφανίζονται οι κανόνες συνεπάγεται συνήθως και διαφορετικά αποτελέσματα. Για παράδειγμα:

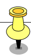
p :- a, b.

p :- c.

Το παραπάνω πρόγραμμα ισοδυναμεί με:

$p \Leftrightarrow (a \wedge b) \vee c$

Αλλάζοντας τη σειρά των κανόνων, η δηλωτική ερμηνεία του προγράμματος παραμένει η ίδια.



Αν τώρα βάλουμε ένα cut:

p :- a, !, b.
p :- c.

Το παραπάνω πρόγραμμα ισοδυναμεί με:

$p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$

Αλλάζοντας τη σειρά των κανόνων, η δηλωτική ερμηνεία του προγράμματος αλλάζει.

$p \Leftrightarrow c \vee (a \wedge b)$

Άσκηση με αποκοπή

Given the following clauses, it is required to place cut(s) in the program to achieve the given outputs: First, determine what the output will be without placing any cuts in the program.

female_author:- author(X), write(X), write(' is an author'), nl, female(X), write('and_female'), nl.

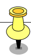
female_author:- write('no luck!'), nl, fail.

author(X):- name(X).

author(X):- write('no more found!'), nl, fail.

name(sartre). *name(calvino).* *name(joyce).*

female(murdoch). *female(bembridge).*



First, we examine the execution of the query female_author?

Using X=Sartre, the subgoals write(X), write(' is an author') and nl succeed with the side-effect of writing:

sartre is an author

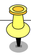
Then we solve the subgoal female(X) with X still bound to sartre.

Neither of the heads of the clauses for female/1 match the goal female(sartre) so Prolog fails and backtracks.

Now try to redo name(X) and we satisfy this with X=calvino. Again, we generate the side-effect on the screen of:

calvino is an author

and again fails to satisfy female(X). Again backtracking with X=joyce. On the screen we get:



joyce is an author

and again female(X) fails.

This time, on backtracking, there are no more solutions for name(X). We now move on to resatisfy author(X) by using its second clause. This generates, on the screen:

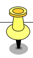
no more found!

We now backtrack and, since there are no more ways of satisfying author(X), we are through with the first clause of female author/0.

The second succeeds by writing:

no luck!

and fails.

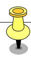


Use only one cut to get the desired output.

(a) sartre is an author

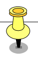
no more found!

no luck!



We should place the cut so that name(X) succeeds once only. This can be done by rewriting the first clause of name/1 as

name(sartre):-!.

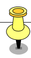


(b) sartre is an author

calvino is an author

joyce is an author

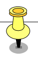
no more found!



We want to commit ourselves to the first clause of female author and not use the second at all. Hence we have the solution:

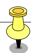
*female_author:- !,author(X),write(X),*and so on

but note that now the original query fails after producing the desired side-effect.
Also note that we have to put the cut before the call to author/1 otherwise we would only generate one of the names rather than all three.



(c) sartre is an author

no luck!



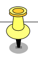
We want author(X) to succeed once and once only and go on to use the second clause of female author (this suggests that the cut cannot be one of the subgoals in the first clause of female author). We don't want to generate the phrase `no more found' so this suggests that we commit to the first clause of author/1. We will put a cut in the body of this clause but where? If we put it thus:

author(X):- !,name(X).

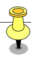
then we would generate all three names by backtracking. Hence the desired solution is:

author(X):- name(X),!.

We can read this as being committed to the first, and only the first, solution for name(X).



(d) sartre is an author

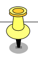


We definitely want to be committed to the first clause of female author/0.

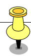
This suggests putting the cut in the body of the first clause of this predicate, but where?

If we put it after the subgoal female(X) then we would get all three solutions to name(X) and their associated side-effects. Therefore we want something like:

```
female_author:- author(X),!,write(X),and so on
```



(e) sartre is an author
calvino is an author
joyce is an author
no luck!



Now we don't get the message `no more found!'. This suggests that we want to commit to the first clause of author/1.

If we put the cut after the subgoal name(X) then we will commit to the first solution and not be able to generate the other two.

Hence we must put the cut before as in:

author(X):- !,name(X).

Αποκοπή και Άρνηση

Μία από τις πιο σημαντικές χρήσεις της αποκοπής είναι η υλοποίηση της άρνησης ως αποτυχία.

Με τη βοήθεια του συνδυασμού αποκοπής και άρνησης ορίζεται και το κατηγόρημα not/1.

```
not(X) :- X, !, fail.
not(X).
```

Το παραπάνω πρόγραμμα δέχεται σαν όρισμα μια οποιαδήποτε κλήση της Prolog. Αν η κλήση μπορεί να αποδειχθεί τότε το κατηγόρημα αποτυγχάνει (1η πρόταση), αλλιώς επιτυγχάνει (2η πρόταση).

Αποκοπή και Άρνηση

Ο συνδυασμός «αποκοπής και αποτυχίας» κάνει δυνατή την υλοποίηση κατηγορημάτων που περιέχουν άρνηση. Για παράδειγμα το παρακάτω πρόγραμμα επιτυγχάνει όταν ένα στοιχείο δεν είναι μέλος της λίστας:

```
not_member(X, []).
not_member(X, [X | _ ]) :- !, fail.
not_member(X, [ _ | RestList]) :- not_member(X, RestList).
```

Bagof, setof

- We can generate, by backtracking, all the objects, one by one, that satisfy some goal.
- Each time a new solution is generated, the previous one disappears and is not accessible any more.
- However, sometimes we would prefer to have all the generated objects available together- for example, collected into a list.
- The built-in predicates **bagof** and **setof** serve this purpose; the predicate **findall** is sometimes provided instead.

bagof

The goal

bagof(X, P, L)

will produce the list L of all the objects X such that a goal P is satisfied.

Example

For example, let us assume that we have in the program a specification that classifies (some) letters into vowels and consonants:

```
class( a, vow).
class( b, con).
class( c, con).
class( d, con).
class( e, vow).
class( f, con).
```

Then we can obtain the list of all the consonants in this specification by the goal:

```
?- bagof( Letter, class( Letter, con), Letters).
Letters = [b, c, d, f]
```

Moreover,

?- bagof (Letter, class(Letter, Class), Letters).

Class : vow

Letters : [a, e];

Class : con

Letters : [b, c, d, l]

setof

The predicate setof is similar to bagof. The goal

setof(X, P, L)

will again produce a list L of objects X that satisfy P. Only this time the list L will be ordered and duplicate items, if there are any, will be eliminated.

- The ordering of the objects is according to the alphabetical order or to relation < if objects are numbers.
- If the objects are complex terms, then the principal functors are compared for the ordering.
- If these are equal then the left-most, top-most functors that are not equal in the terms compared decide.

Example

?- setof(Class/Letter, class(Letter, Class), List).

List = [con/b, con/c, con/d, con/f, vow/a, vow/e]

findall

Another predicate of this family, similar to bagof, is findall.

findall(X, P, L)

produces, again, a list of objects that satisfy P.

The difference with respect to bagof is that all the objects X are collected regardless of (possibly) different solutions for variables in P that are not shared with X.

This difference is shown in the following example:

?- findall(Letter, class(Letter, Class), Letters).

Letters = [a, b, c, d, e, f]

If there is no object X that satisfies P then findall will succeed with L = [].

Extensions to this basic communication method (questions and Prolog answers) are needed in the following areas:

- input of data in forms other than Prolog questions - for example, in the form of English sentences
- output of information in any format desired
- input from and output to any computer file and not just the user terminal

Communication with Files in Prolog

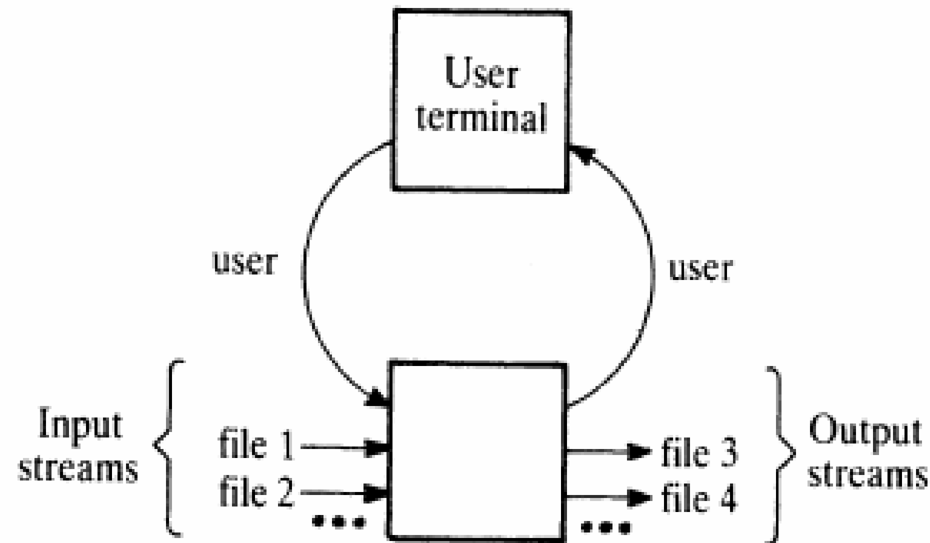


Figure shows a general situation in which a Prolog program communicates with several files.

The program can, in principle, read data from several input files, also called **input streams**, and output data to several output files, also called **output streams**.

- ❖ Data coming from the user's terminal is treated as just another input stream.
- ❖ Data output to the terminal is, analogously, treated as another output stream.

Both of these 'pseudo-files' are referred to by the name **user.**

Communication with Files in Prolog

The names of other files can be chosen by the programmer according to the rules for naming files in the computer system used.

At the beginning of the execution these two streams correspond to the user's terminal.

The current input stream can be changed to another file, Filename, by the goal:

see(Filename)

Example:

...

see(file1),

read_from_file(Information),

see(user),

…

The current output stream can be changed by a goal of the form:

tell(Filename)

A sequence of goals to output some information to fi1e, and then redirect succeeding output back to the terminal, is:

…

tell(file3),

write_on_file(Information),

tell(user),

…

Communication with Files in Prolog

The goal

seen

closes the current input file.

The goal

told

closes the current output file.

Ο στόχος

seeing *(telling)*

επιστρέφει το τρέχον κανάλι εισόδου (εξόδου).

Παράδειγμα

Το κατηγόρημα out/1 που ακολουθεί δέχεται σαν είσοδο μία λίστα με στοιχεία και τα γράφει σε ένα προκαθορισμένο αρχείο. Αρχικά κατευθύνει την έξοδο σε ένα αρχείο, στη συνέχεια καλεί το κατηγόρημα write_list, το οποίο τυπώνει τα στοιχεία της λίστας (χωρίς να προσδιορίζει που) και τέλος καλεί το κατηγόρημα told, καθιστώντας έτσι και πάλι κανάλι εξόδου την οθόνη.

```
out(L) :- tell( ‘myfile.txt’), write_list(L), told.

write_list([]).
write_list([H | T]) : - write(H), nl, write_list(T).
```

File Processing

- Files can only be processed sequentially.
- In this sense all files behave in the same way as the terminal.
- Each request to read something from an input file will cause reading at the current position in the current input stream.
- After the reading, the current position will be, of course, moved to the next unread item.
- So the next request for reading will start reading at this new current position.
- If a request for reading is made at the end of a file, then the information returned by such a request is the atom end-of-.file.
- Once some information has been read, it is not possible to re-read it again.

Viewing files in Prolog

There are two main ways in which files can be viewed in Prolog, depending on the form of information.

- One way is to consider the **character** as the basic element of the file.
  - Accordingly, one input or output request will cause a single character to be read or written.
  - The built-in predicates for this are **get, get0 and put**.
- The other way of viewing a file is to consider bigger units of information as basic building blocks of the file.
  - Such a natural bigger unit is the Prolog **term**.
  - So each input/output request of this type would transfer a whole term from the current input stream or to the current output stream respectively.
  - Predicates for transfer of terms are **read and write**.
  - Of course, in this case, the information in the file has to be in a form that is consistent with the syntax of terms.

Processing Files of Terms

- The built-in predicate **read** is used for reading terms from the current input stream.
- The goal

  read(X)
- for reading terms from the current input will cause the next term, T, to be read, and this term will be matched with X.
- If X is a variable then, as a result, **X will become instantiated to T**.
- If matching does not succeed then the goal read(X) fails.
- The predicate read is deterministic, so in the case of failure there will be **no backtracking** to input another term.
- Each term in the input file must be followed by a full stop and a space or carriage-return. Εάν ο όρος περιέχει κενά, θα πρέπει να τοποθετείται σε μονές αποστρόφους.
- If read(X) is executed when the end of the current input file has been reached then X will become instantiated to the atom **end_of_file.**

Παράδειγμα

?- read(X).

Ανάλογα με την είσοδο που θα δοθεί, θα έχουμε τα παρακάτω αποτελέσματα:

Είσοδος	Αποτέλεσμα
a.	X = a
a	no (λείπει η τελεία)
[1, k].	X = [1, k]
‘a bc	no (λείπει η δεξιά απόστροφος)

Processing Files of Terms

The built-in predicate write outputs a term. So the goal

write(X)

will output the term X on the current output file.

The goal

tab(N)

cause N spaces to be output.

The predicate nl (which has no arguments) causes the start of a new line at output.

Παράδειγμα 1

?- write(aaa), nl, write([1, 2, 3]), nl, write(f(k(1,2))).

aaa

[1, 2, 3]

f(k(1, 2))

yes

Παράδειγμα 2

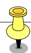
Let us assume that we have a procedure that computes the cube of a number:

```
cub( N, C) :-
        C is N*N*N.
```

Suppose we want to use this for calculating the cubes of a sequence of numbers.

We could do this by a sequence of questions:

```
?-   cube( 2, X).
X = 8
?- cube( 5, Y).
Y = 125
```



Let us now modify this program so that the cube procedure will read the data itself.

Now the program will keep reading data and outputting their cubes until the atom stop is read:

```
cube :-
        read( X),
        process(X ).
process( stop) :- !.
process( N) :-
          C is N*N*N,
          write( C),
          cube.
```

?- cube.

2.

8

5.

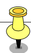
125

12.

1728

stop.

yes



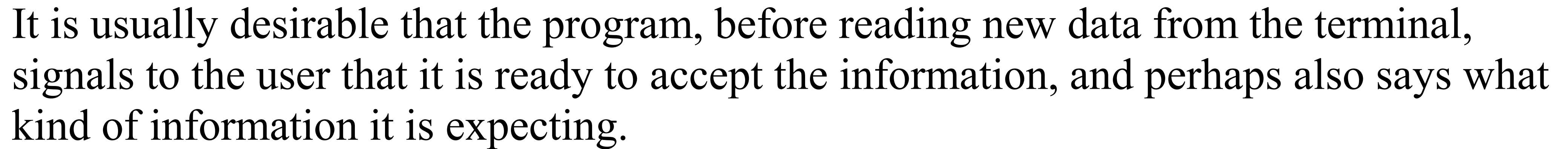
It may appear that the above cube procedure could be simplified.

However, the following attempt to simplify is not correct:

```
cube :-
    read( stop), !.
cube :-
    read( N),
    C is N*N*N,
    write( C),
    cube.
```

The reason why this is wrong can be seen easily if we trace the program with input data 5. The goal read(stop) will fail when the number is read, and this number will be lost for ever!

The next read goal will input the next term on the other hand, it could happen that the stop signal is read by the goal read(N), which would then cause a request to multiply non numeric data.



It is usually desirable that the program, before reading new data from the terminal, signals to the user that it is ready to accept the information, and perhaps also says what kind of information it is expecting.

```
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: 12.

Cube of 12 is 1728

Next item, please: stop.

yes

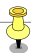
Displaying Lists

The following procedure outputs a list L so that each element of L is written on a separate line:

```
writelist( [] ).
writelist( [X | L] ) :-
        write( X), nl,
        writelist( L).
```

If we have a list of lists, one natural output form is to write the elements of each list in one line:

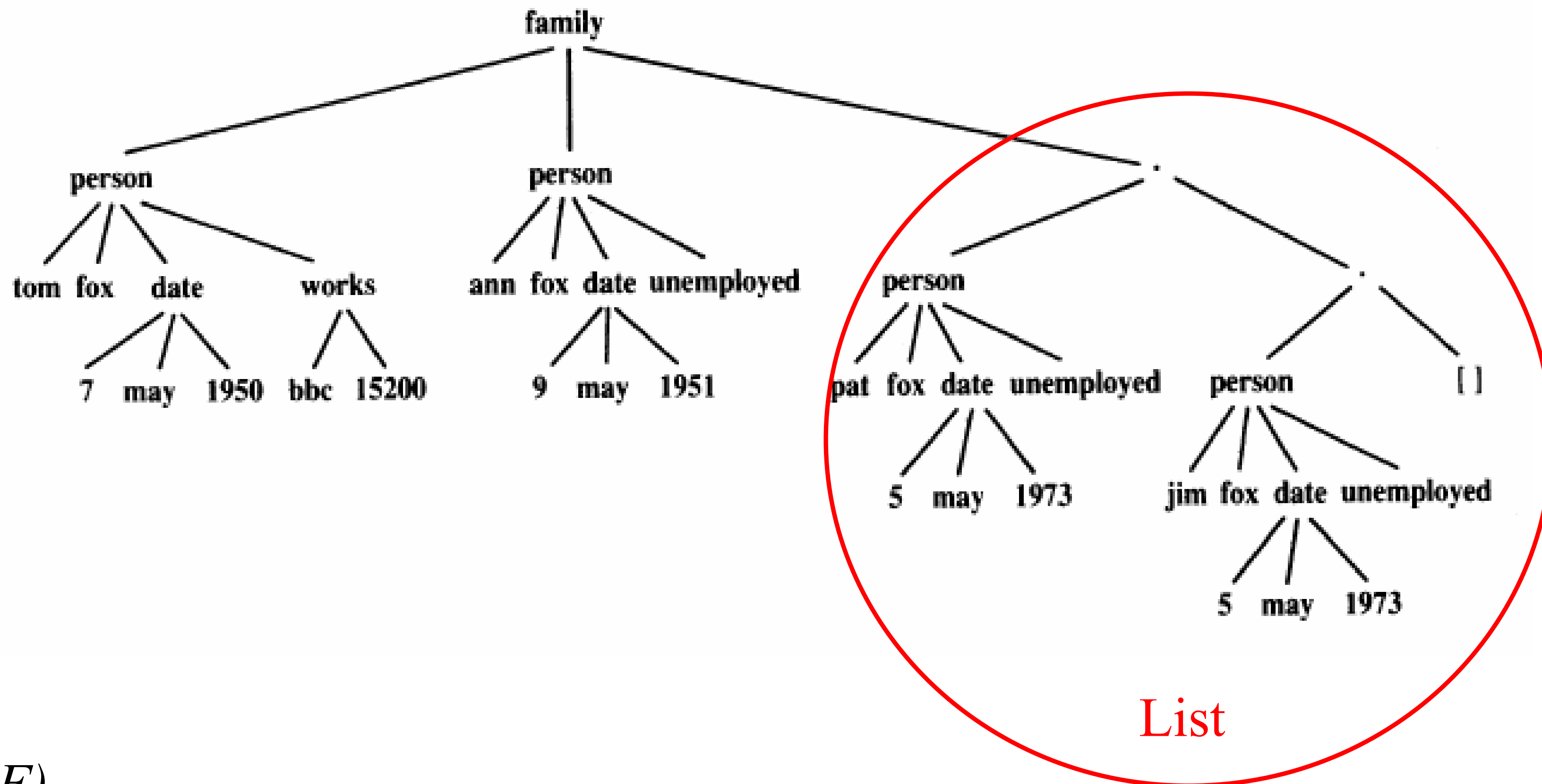
```
?- writelist2( [ [a, b, c], [d, e, f], [g, h, i] ] ).
a b c
d e f
g h i
```



A procedure that accomplishes this is:

```
writelist2( [] ).
writelist2( [L | LL] ) :-
        doline( L), nl,
        writelist2( LL).
doline( [] ).
doline( [X| L] ) :-
        write( X), tab( 1),
        doline( L).
```

Formatting Terms



write(F).

will cause this term to be output in the standard form:

family(person(tom,fox, date(7,may1, 950),works(bbc,15200)),…

Formatting Terms

parents

tom fox, born 7 may 1950, works bbc, salary 15200
ann fox, born 9 may 1951, unemployed

children

pat fox, born 5 may 1973, unemployed
jim fox, born 5 may 1973, unemployed

```
writefamily:- family(Husband,Wife,Children), nl, write('parents'), nl, nl,
              writeperson(Husband), nl,
              writeperson(Wife), nl, nl,
              write('children'), nl, nl,
              writepersonlist(Children), fail.
```

```
writeperson(person(Firstname,Secname,date(D,M,Y),Work)):-
   tab(4), write(Firstname), tab(1), write(Secname),
   write(',born '), write(D),tab(1),
   write(M),tab(1),
   write(Y),write(','),
  writework( Work).

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

writework(unemployed) :- write(unemployed).
writework(works(Comp,Sal)) :- write('works'), write(Comp), write(', salary'),
write(Sal).

family(person(michael,davis,date(2,december,1973),works(school,13000)),
person(michelle,davis,date(9,may,1974),unemployed),
[person(emelie,davis,date(5,may,2003),unemployed)]).
```

A typical sequence of goals to process a whole file, F, would look something like this:

..., see(F), processfile, see(user),. ..

Here processfile is a procedure to read and process each term in F.

```
processfile :-
        read( Term),
        process( Term).

process( end_of_file) :-!.

process( Term) :-
        treat( Term),
        processfile.
```

Processing a file of terms

Here *treat(Term)* represents whatever is to be done with each term. An example would be a procedure to display on the terminal each term together with its consecutive number:

```
showfile( N) :-
        read( Term),
        show( Term, N).
show( end_of_file, _) :- !.
show( Term, N) :-
        write( N), tab(2), write( Term),nl,
        N1 is N+1,
        showfile( N1).
```

Manipulating characters

A character is written on the current output stream with the goal put(C) where c is the ASCII code of the character to be output.

put(65), put(66), put(67).

would cause the following output:

ABC

A single character can be read from the current input stream by the goal

get0(C)

This causes the current character to be read from the input stream, and the variable C becomes instantiated to the ASCII code of this character.

get(C)

will cause the skipping over of all non-printable characters (blanks in particular) from the current input position in the input stream up to the first printable character.

First it will read the first character, output this character, and then complete the processing depending on this character.

Example 1

```
squeeze:-
    get0( C),
    put( C),
    dorest( C).

dorest( 46):- !.  %46 is ASCII for full stop, all done

dorest( Letter):-
     squeeze.
```

Example 2

```
squeeze :-
        get0( C),
        put( C),
        dorest( C).

dorest( 46):- !.            %46 is ASCII for full stop, all done

dorest( 32):- !,            % 32 is ASCII for blank
        get( C),
        put( C),
        dorest( C).
dorest( Letter) :-
        squeeze.
```

Χειρισμός Συμβολοσειρών

Στην Prolog η τακτική που ακολουθείται για τον χειρισμό των συμβολοσειρών είναι η μετατροπή τους σε λίστες και η χρήση κατηγορημάτων επεξεργασίας λιστών.

Το κατηγόρημα που μετατρέπει μια συμβολοσειρά σε λίστα είναι το:

name(Συμβολοσειρά, Λίστα).

Μετατρέπει την Συμβολοσειρά σε Λίστα ASCII κωδικών και το αντίστροφο.

Για παράδειγμα:

?- name(abc, X).

X = [97, 98, 99]

ή το αντίστροφο:

?- name(X, [97, 98, 99]).

X = abc

Παρατηρήσεις

- Η συμβολοσειρά μπορεί να είναι άτομο ή αριθμός.
- Αν το άτομο αρχίζει με κεφαλαίο ή περιέχει κενά, πρέπει να είναι τοποθετημένο σε μονά εισαγωγικά.
- Ένα άτομο ή αριθμός μέσα σε διπλά εισαγωγικά ισοδυναμεί με λίστα ASCII κωδικών.

Παράδειγμα

Το κατηγόρημα strings_concatenate/3 που ακολουθεί δέχεται δύο συμβολοσειρές σαν είσοδο στα δύο πρώτα ορίσματά του και τις συνενώνει. Το αποτέλεσμα επιστρέφεται στο τρίτο όρισμα.

```
strings_concatenate(Str1, Str2, Str) :-
        name(Str1, List1),
        name(Str2, List2),
        append(List1, List2, List),
        name(Str, List).
```

We tell Prolog to read a program from a file F with the goal:

?- consult(F).

The effect will be that all clauses in F are read and will be used by Prolog when answering further questions from the user.

If another file is consulted at some later time during the same session, clauses from this new file are simply added at the end of the current set of clauses.

We do not have to enter our program into a file and then request 'consulting' that file. Instead of reading a file, Prolog can also accept our program directly from the terminal, which correspond to the pseudo-file user.

?- consult(user).

Reading programs

A shorthand notation for consulting files is available in some Prolog systems. Files that are to be consulted are simply put into a list and stated as a goal:

?- [file1, fi1e2, file3].

This is exactly equivalent to three goals:

?- consult(file1), consult(file2), consult(file3)

Reading programs

The difference between consult and reconsult is that consult always adds new clauses while reconsult redefines previously defined relations.

reconsult(F) will, however, not affect any relation about which there is no clause in F.

Database manipulation

A Prolog program an be viewed as a database, containing partly explicit (facts) and partly implicit (rules) data.

Furthermore, built-in-predicates make it possible to update this database during the execution of the program. This is done by adding (during execution) new clauses to the program or by deleting existing ones.

The goal

assert(C)

always succeeds and, as its side effect, causes a clause C to be "asserted", that

is, added to the database.

The goal

retract(C)

does the opposite: it deletes a clause that matches C. The following conversation

with Prolog illustrates:

?- crisis.

No

?- assert(crisis).

yes

?- crisis.

yes

?- retract(crisis).

yes

Example

```
nice :-
    sunshine, not(raining).
funny :-
    sunshine, raining.
Disgusting :-
    raining,
    fog.
raining.
fog.
```

The following conversation with this program will gradually update the database:

```
?- nice.
no
?- disgusting.
yes
```

?- retract(fog).

yes

?- disgusting.

no

?- assert(sunshine).

yes

?- funny.

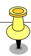
yes

?- retract(raining).

yes

?- nice.

yes

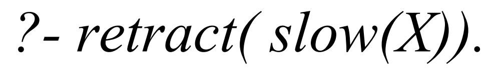


Clauses of any form can be asserted or retracted. The next example illustrates that retract is also non-deterministic: a whole set of clauses can, through backtracking, be removed by a single retract goal. Let us assume that we have the following facts in the 'consulted' program:

```
fast( ann).
slow( tom).
slow( pat).
```

We can add a rule to this program, as follows:

```
? - assert(
        ( faster(X,Y) :-
                fast(X), slow(Y) ) ).
yes
?- faster( A, B).
A = ann
B = tom
```



?- retract(slow(X)).

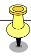
X = tom;

X = pat;

No

?- faster(ann,_).

no



Example

- |?- listing.
- yes
- |?- assert(son(tom,sue)).
- yes
- |?- assert(female(sue)).
- yes
- |?- assert((mother(X,Y):-
-     son(Y,X),female(X))).
- true ?
- yes

| ?- listing.
female(sue).
mother(A,B):-
son(B, A),
female(A).
son(tom, sue).
yes
| ?- mother(sue,tom).
yes

Asserta and Assertz

The goal

```
asserta( C)
```

adds C at the beginning of the database. The goal

```
assertz( C)
```

adds C at the end of the database.

Example

?- assert(p(a)), assertz(p(b)), asserta(p(c)).

yes

?- p(X).

X = c;

X = a ;

X = b

dynamic/1

- Asserted entries are added to the dynamic copy of the consulted Prolog programs stored in Sicstus' memory buffer.
- They are not written to the original program, i.e. the program doesn't change and once Sicstus is closed all changes to the database are lost.
- In order for a predicate already consulted in the Prolog program to be manipulated during runtime it needs to be declared as dynamic. We do this by adding a directive at the beginning of the code:

  :- dynamic PredicateIndicator
- where PredicateIndicator is the predicate followed by the arity e.g. append/3
- If the predicate is "new" then it is automatically dynamic and does not need to be declared.

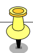
clause/2

- Once a predicate is declared as dynamic you can check for its existence using
  clause(Head, Body)
- clause(Head, Body) succeeds if there is a clause in the current Prolog database which is unifiable with:

  Head :- Body.
- E.g. :- dynamic a/2.

```
a(1,2).
a(3,4).
a(X,Y):- b(X), b(Y).
```

```
|?- clause(a(Arg1,Arg2), Body).
Arg1 = 1, Arg2 = 2, Body = true?;
Arg1 = 3, Arg2 = 4, Body = true?;
Body = b(Arg1), b(Arg2)?;
no
```



- Note that if the clause is a fact, and has no body, then the second argument of clause/2 is instantiated to true.

Comment about database manipulation

A remark on the style of programming should be made at this stage:

- The foregoing examples illustrate some obviously useful applications of assert and retract. However, their use requires special care.
- Excessive and careless use of these facilities cannot be recommended as good programming style.
- By asserting and retracting we, in fact, modify the program. Therefore relations that hold at some point will not be true at some other time. At different times the same questions receive different answers.
- A lot of asserting and retracting may thus obscure the meaning of the program and it may become hard to imagine what is true and what-is not.
- The resulting behaviour of the program may become difficult to understand, difficult to explain and to trust.

Ορισμός νέων τελεστών

- Είδαμε στα εισαγωγικά μαθήματα οτι οι σύνθετοι όροι της Prolog έχουν τη μορφή:

συναρτησιακό σύμβολο(ορίσματα)

- Η μορφή αυτή ονομάζεται προσημασμένη παρενθεσιακή μορφή, γιατί το συναρτησιακό σύμβολο προηγείται των ορισμάτων, τα οποία ακολουθούν μέσα σε παρενθέσεις.
- Ειδικά για συναρτησιακά σύμβολα 1^{ης} και 2^{ης} τάξης, η Prolog υποστηρίζει και άλλες μορφές αναπαράστασης:

Προσημασμένη μη παρενθεσιακή (prefix)

- Το συναρτησιακό σύμβολο προηγείται του ορίσματός του, το οποίο όμως δεν περικλείεται σε παρενθέσεις.
- Για παράδειγμα, αντί να γράφουμε sqrt(X), γράφουμε sqrt X.

Μετασημασμένη μη παρενθεσιακή (postfix)

- Το συναρτησιακό σύμβολο ακολουθεί του ορίσματός του, το οποίο όμως δεν περικλείεται σε παρενθέσεις.
- Για παράδειγμα, αν θεωρήσουμε οτι το συναρτησιακό σύμβολο του παραγοντικού είναι το θαυμαστικό, αντί να γράφουμε !(X), μπορούμε να γράφουμε X !.

Ενδοσημασμένη μη παρενθεσιακή (postfix)

- Για συναρτησικά σύμβολα με δύο ορίσματα μπορεί το συναρτησιακό σύμβολο να βρίσκεται ανάμεσα στα ορίσματά του.
- Για παράδειγμα, αν θεωρήσουμε οτι το συναρτησιακό σύμβολο της ακέραιας διαίρεσης είναι το div, αντί να γράφουμε div(X, Y), μπορούμε να γράφουμε X div Y.

A programmer can define his or her own operators. So, for example, we can define the atoms has and supports as infix operators and then write in the program facts like:

```
peter has information.
floor supports table.
```

These facts are exactly equivalent to:

```
has( peter, information).
supports( floor, table).
```

Ορισμός Τελεστών

- A programmer can define new operators by inserting into the program special kinds of clauses, sometimes called directives, which act as operator definitions.
- An operator definition must appear in the program before any expression containing that operator.
- For our example, the operator has can be properly defined by the directive:

op(600, xfx, has).

- This tells Prolog that we want to use ' has' as an operator, whose precedence (προτεραιότητα) is 600 and its type is 'xfx', which is a kind of infix operator.
- The form of the specifier 'xfx' suggests that the operator, denoted by 'f', is between the two arguments denoted by 'x'.

Ορισμός Τελεστών (συνέχεια)

- Notice that operator definitions do not specify any operation or action.
  - In principle, no operation on data is associated with an operator (except in very special cases).
- Operators are normally used, as functors, only to combine objects into structures and not to invoke actions on data.
- Operator names are atoms, and their precedence must be in some range which depends on the Prolog system.
  - We will assume that the range is between 1 and 1200.

Προτεραιότητα τελεστών

- Όταν έχουμε σύνθετες παραστάσεις με περισσότερους από έναν τελεστές, επειδή δεν υπάρχουν παρενθέσεις, είναι δύσκολο να αντιληφθεί η Prolog ποιο όρισμα ανήκει σε ποιόν τελεστή. Ο κανόνας λοιπόν είναι ο ακόλουθος:

Πρώτα εφαρμόζονται οι τελεστές με μικρότερο αριθμό προτεραιότητας και στη συνέχεια οι τελεστές με μεγαλύτερο αριθμό προτεραιότητας.

- Για παράδειγμα, αν ορίσουμε:

  Op(200, Μορφή, div)

  Op(100, Μορφή, ^)

- Και έστω οτι έχουμε να αναλύσουμε το σύνθετο όρο / παράσταση:

  X div Y ^ Z

- η παράσταση αυτή ισοδυναμεί με X div (Y ^ Z)
- Σημείωση: Οι σταθεροί όροι αλλά και οι μεταβλητές έχουν τη μικρότερη δυνατή προτεραιότητα.

Μορφές τελεστών (1/4)

Στους τελεστές τάξης 1, το όρισμα Μορφή μπορεί να πάρει μία από τις ακόλουθες τέσσερις τιμές:

fx, fy, xf, yf,

Στις παραπάνω τιμές το σύμβολο f αντιπροσωπεύει τον τελεστή, ενώ τα σύμβολα x και y τα ορίσματά του. Ειδικότερα:

- Το σύμβολο x δηλώνει όρισμα που θα έχει προτεραιότητα μικρότερη από την προτεραιότητα του τελεστή.
- Το σύμβολο y δηλώνει όρισμα που θα έχει προτεραιότητα μικρότερη ή ίση από την προτεραιότητα του τελεστή.

Δυνατότητα ορίσματος με προτεραιότητα μεγαλύτερη από την προτεραιότητα του τελεστή δεν τίθεται, αφού σε αυτή την περίπτωση το όρισμα (το οποίο μπορεί να είναι και τελεστής) θα αποτιμηθεί μετά τον εν λόγω τελεστή.

Μορφές τελεστών (2/4)

Παράδειγμα:

op(100, fx, sqrt)

- Με αυτή τη δήλωση μπορούμε να εκφράσουμε sqrt 16, αλλά όχι sqrt sqrt 16.
- Αυτό θα ισοδυναμούσε με sqrt(sqrt(16)), οπότε ο εξωτερικός τελεστής sqrt θα είχε σαν όρισμα έναν τελεστή ίσης προτεραιότητας, κάτι που απαγορεύεται στη μορφή fx.
- Αυτό επιτρέπεται όταν:

op(100, fy, sqrt)

Παρομοίως για xf και yf.

Μορφές τελεστών (3/4)

Παράδειγμα:

- Αν ορίσουμε τον τελεστή ακέραιας διαίρεσης σαν xfx, τότε δεν μπορούμε να γράψουμε

  8 div 4 div 2

- Η παραπάνω παράσταση επιδέχεται δύο ερμηνείες:

  8 div (4 div 2) ή div(8, div(4, 2))

και:

  (8 div 4) div 2 ή div(div(8, 4), 2)

- Και στις δύο περιπτώσεις όμως ο ένας τελεστής div έχει σαν όρισμα άλλο τελεστή div (ίσης προτεραιότητας), κάτι που όμως απαγορεύεται στη μορφή xfx.

Μορφές τελεστών (4/4)

xfy: το πρώτο όρισμα του τελεστή πρέπει να έχει μικρότερη προτεραιότητα από τον τελεστή, ενώ το δεύτερο επιτρέπεται να έχει ίση περοτεραιότητα

❖ Εάν ορίσουμε τον τελεστή της ακέραιας διαίρεσης σαν xfy, τότε η παράσταση

8 div 4 div 2

θα μεταφράζεται σαν:

8 div (4 div 2) ή 8 div(8, div(4, 2))

❖ Έτσι ώστε ο τελεστής div να έχει σαν δεύτερο όρισμα έναν ισοδύναμο, όσον αφορά την προτεραιότητα, τελεστή.

❖ Γενικά, όταν έχουμε τελεστής ίσης προτεραιότητας μορφής xfy, αυτοί εφαρμόζονται με προτεραιότητα από δεξιά προς τα αριστερά.

❖ Αντίθετα, όταν έχουμε τελεστής ίσης προτεραιότητας μορφής yfx, αυτοί εφαρμόζονται με προτεραιότητα από αριστερά προς τα δεξιά.

Γενικά

Αν και φαίνεται άσκοπο και πολλές φορές πλεονάζον από πολλούς νέους προγραμματιστές, ο τρόπος με τον οποίο γράφουμε ένα πρόγραμμα έχει μεγάλη σημασία τόσο κατά την διάρκεια της αποσφαλμάτωσης (debugging) και επέκτασης του προγράμματος. Ένα πρόγραμμα με ελλιπή τεκμηρίωση, περίεργα ονόματα μεταβλητών και κατηγορημάτων παρουσιάζει όχι μόνο μικρή αναγνωσιμότητα, αλλά και μεγάλες δυσκολίες στο να βρεθεί λάθος μέσα σε αυτό. Για παράδειγμα:

```
foo(The,[The|FOOS],FOOS).
foo(The,[FOO1|OO2],[FOO1|OO3]):-
      foo(The,OO2,OO3).
```

Αν και είναι απόλυτα σωστό από συντακτική και λογική άποψη, βοηθά πολύ λίγο τον προγραμματιστή να κατανοήσει τι ακριβώς κάνει. Αν όμως γράψουμε το παραπάνω ως:

```
delete(X,[X|RestList],RestList).
delete(X,[Y|Rest],[Y|RestList]):-
      delete(X,Rest,RestList).
```

είναι σαφώς κατανοητότερο.

Ονόματα – Κενά – Σχόλια

Μερικοί απλοί κανόνες καλού στυλ είναι οι ακόλουθοι:

- Πρέπει πάντα να χρησιμοποιούνται ονόματα για τα κατηγορήματα και τις μεταβλητές τα οποία να είναι συναφή με το ρόλο τους μέσα στο πρόγραμμα.

- Η αναγνωσιμότητα των προγραμμάτων αυξάνεται όταν υπάρχουν κενές γραμμές ανάμεσα στα κατηγορήματα και όταν το σώμα του κανόνα εμφανίζεται δεξιότερα της κεφαλής.

- Σχόλια που αφορούν τα κατηγορήματα πρέπει να εμφανίζονται πριν από αυτό, να είναι σαφή και όσο το δυνατό λιγότερα. Επίσης, είναι καλό να δηλώνονται μέσα σε σχόλια οι προδιαγραφές των κατηγορημάτων (κατηγόρημα/ τάξη).
  Για παράδειγμα, το παραπάνω κατηγόρημα μπορεί να γίνει:

```
%%% delete/3
%%% delete(X,List,RestList)
%%% Το κατηγόρημα πετυχαίνει όταν η RestList είναι η
λίστα List
%%% αν αφαιρέσουμε από αυτή το στοιχείο X.
delete(X,[X|RestList],RestList).
delete(X,[Y|Rest],[Y|RestList]):-
      delete(X,Rest,RestList).
```

Χρήσιμες συμβουλές

Οι ακόλουθοι είναι μερικοί κανόνες μεθοδολογίας οι οποίοι είναι καλό να ακολουθούνται κατά την ανάπτυξη προγραμμάτων:

- Ο αριθμός των στόχων στο σώμα ενός κανόνα πρέπει να είναι σχετικά περιορισμένος. Μεγάλοι κανόνες δυσκολεύουν σημαντικά την αποσφαλμάτωση και είναι συνήθως αποτέλεσμα κακού σχεδιασμού του κώδικα. Οργανώστε τους στόχους σε κατηγορήματα που έχουν κάποια λογική λειτουργία.
- Το συνδετικό «;» (λογικό ή) πρέπει να χρησιμοποιείται με προσοχή και μόνο όπου είναι απαραίτητο. Οι εναλλακτικές συνθήκες μπορούν να εκφραστούν με πολλαπλούς κανόνες (προτάσεις) του κατηγορήματος.
- Η αποκοπή «!» (cut) πρέπει επίσης να χρησιμοποιείται με μεγάλη προσοχή, καθώς μπορεί να αλλάξει την σημασία του προγράμματος (κόκκινη αποκοπή).
- Οι εντολές δυναμικής τροποποίησης του προγράμματος assert/1, retract/1 είναι καλό να αποφεύγονται. Οδηγούν σε προγράμματα που παρουσιάζουν μη επαναλληψιμότητα στα αποτελέσματά τους και σημαντικές αδυναμίες στην αποσφαλμάτωση, καθώς για τον εντοπισμό ενός πιθανού λάθους πρέπει να εξασφαλίσουμε ότι τα περιεχόμενα της μνήμης είναι πανομοιότυπα με εκείνα της στιγμής που εμφανίστηκε το λάθος.

Μεμονωμένες μεταβλητές (1/2)

Οι περισσότεροι διερμηνευτές της Prolog κάνουν έλεγχο για μεταβλητές οι οποίες εμφανίζονται μόνο μια φορά μέσα σε ένα κανόνα (singleton variables). Ο έλεγχος γίνεται για να αποφεύγονται τα σφάλματα τα οποία οφείλονται σε «ανορθογραφίες» στις μεταβλητές. Για παράδειγμα, στο ακόλουθο γεγονός

```
delete(X,[X|RestList],Restlist).
```

οι μεταβλητές RestList και Restlist είναι διαφορετικές (η Prolog διαχωρίζει τους πεζούς από τους κεφαλαίους χαρακτήρες). Στην περίπτωση που θα φορτώναμε τον παραπάνω κώδικα σε κάποια Prolog θα προέκυπτε προειδοποιητικό μήνυμα επισημαίνοντας ότι δύο μεταβλητές είναι singleton.

Είναι δυνατό μια μεταβλητή να εμφανίζεται μόνο μια φορά μέσα σε ένα κανόνα, για να δηλώσει ότι το συγκεκριμένο όρισμα θέλουμε να ενοποιηθεί με κάτι το οποίο δεν μας ενδιαφέρει. Στην περίπτωση αυτή στο συγκεκριμένο όρισμα τοποθετούμε μια *ανώνυμη μεταβλητή*. Οι ανώνυμες μεταβλητές δηλώνονται πολύ απλά με τον χαρακτήρα underscore (_) ή τον ίδιο χαρακτήρα ακολουθούμενο από οποιουσδήποτε χαρακτήρες (_RestList).

Μεμονωμένες μεταβλητές (2/2)

Κλασσικό παράδειγμα είναι το κατηγόρημα member/2:

```
member(X,[X|Rest]).
member(X,[Y|Rest]):-
    member(X,Rest).
```

Το παραπάνω έχει δύο singleton μεταβλητές: την Rest στην πρώτη πρόταση και την Y στην δεύτερη, και φυσικά θα προκύψουν τα αντίστοιχα προειδοποιητικά μηνύματα. Το κατηγόρημα με χρήση ανώνυμων μεταβλητών γράφεται:

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

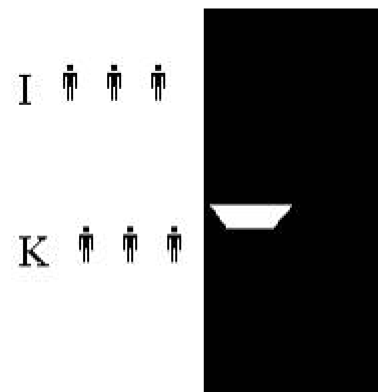
Θα πρέπει να σημειώσουμε ότι και οι δύο εκδοχές του προγράμματος είναι ακριβώς ισοδύναμες.

State-Space Search

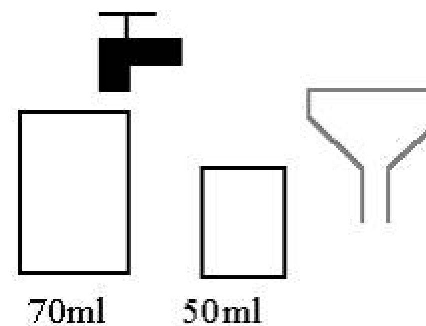
- Many problems in AI take the form of state-space search
- The states might be legal board configurations in a game, towns and cities in some sort of route map, collections of mathematical propositions, etc.
- The state-space is the configuration of the possible states and how they connect to each other e.g. the legal moves between states
- We need to search the state-space to find an optimal path from a start state to a goal state
- We can only decide what to do (or where to go), by considering the possible moves from the current state, and trying to look ahead as far as possible

Παραδείγματα

κανίβαλοι και ιεραπόστολοι (*missionaries and cannibals*)



ποτήρια (*water glass*)

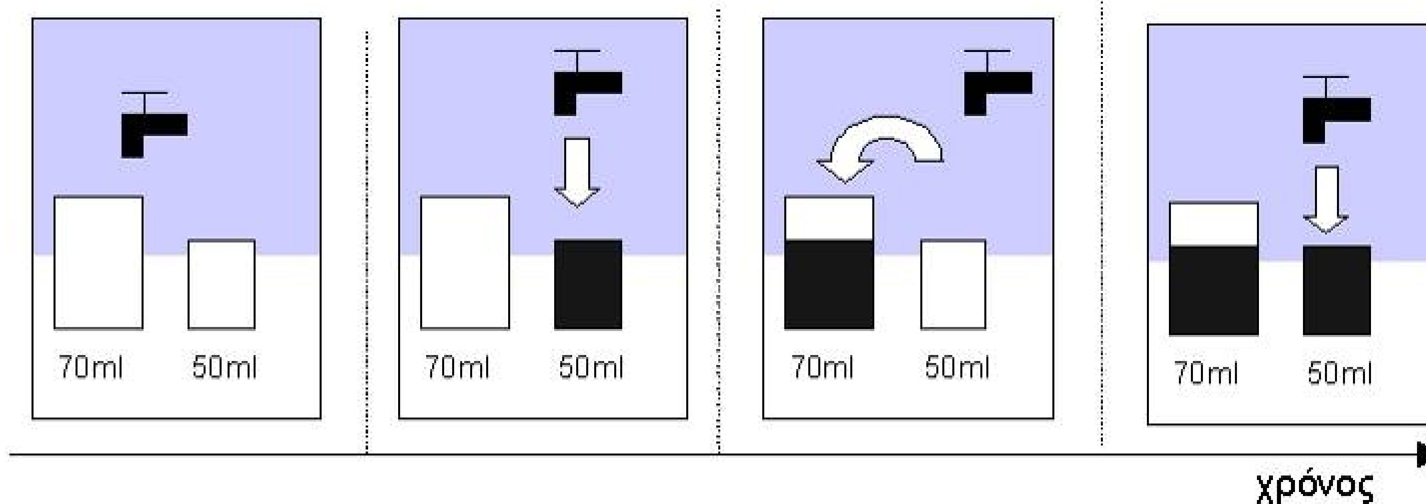


Περιγραφή Προβλημάτων με Χώρο Καταστάσεων

- Ο κόσμος ενός προβλήματος αποτελείται από τα αντικείμενα, τις ιδιότητες των αντικειμένων και τις σχέσεις που τα συνδέουν

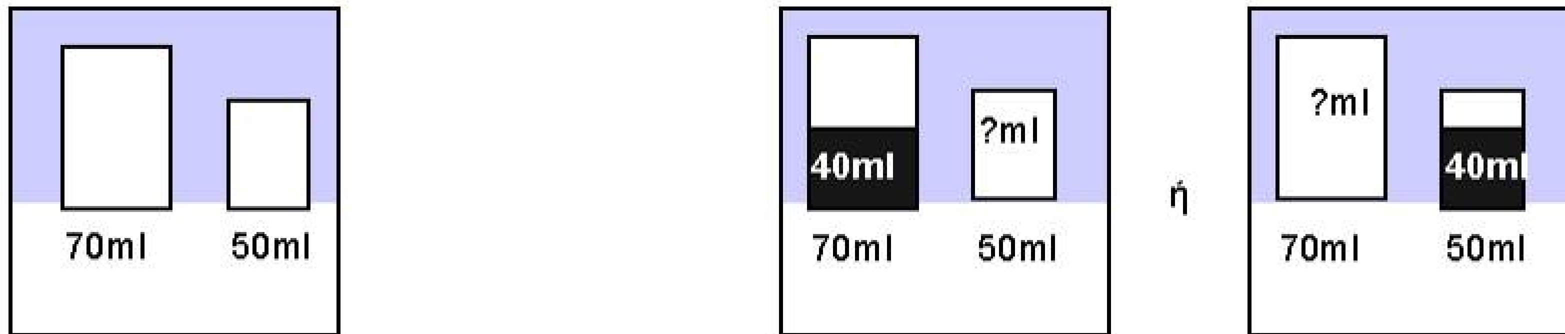
Κατάσταση προβλήματος

Κατάσταση ενός κόσμου είναι ένα *στιγμιότυπο* (*instance*) ή *φωτογραφία* (*snapshot*) μίας συγκεκριμένης χρονικής στιγμής της εξέλιξης του κόσμου.



Αρχικές και Τελικές καταστάσεις

- Η *αρχική* (*initial state*) και *τελική* (*final* ή *goal state*) ή τελικές καταστάσεις εκφράζουν το δεδομένο και το ζητούμενο αντίστοιχα.



Τελεστής (1)
Γέμισε το ποτήρι των X ml μέχρι το χείλος από τη βρύση
Προϋποθέσεις
Το ποτήρι των X ml έχει 0 ml
Αποτελέσματα
Το ποτήρι των X ml έχει X ml

Παράδειγμα
Το πρόβλημα των ποτηριών

Τελεστής (2)
Γέμισε το ποτήρι των X ml από το ποτήρι των Y ml
Προϋποθέσεις
Το ποτήρι των X ml έχει Z ml Το ποτήρι των Y ml έχει W ml ($W \neq 0$)
Αποτελέσματα
Το ποτήρι των X ml έχει X ml και Το ποτήρι των Y ml έχει W-(X-Z), αν $W \geq X-Z$ ή Το ποτήρι των X ml έχει Z+W ml και Το ποτήρι των Y ml έχει 0, αν $W < X-Z$

Τελεστής (3)
Άδειασε το ποτήρι των X ml στο νεροχύτη
Προϋποθέσεις
Το ποτήρι έχει περιεχόμενο
Αποτελέσματα
Το ποτήρι των X ml έχει 0 ml

Λύση προβλήματος

- Λύση σε ένα πρόβλημα είναι η ακολουθία τελεστών που εφαρμόζονται στην αρχική κατάσταση για να προκύψει η τελική κατάσταση.
- Παράδειγμα:

Μετάφερε 1 ιεραπόστολο και 1 κανίβαλο από την αριστερή στη δεξιά όχθη

Μετάφερε 1 ιεραπόστολο από τη δεξιά στην αριστερή όχθη

Μετάφερε 2 κανίβαλους από την αριστερή στη δεξιά όχθη

Μετάφερε 1 κανίβαλο από τη δεξιά στην αριστερή όχθη

Μετάφερε 2 ιεραπόστολους από την αριστερή στη δεξιά όχθη

Μετάφερε 1 ιεραπόστολο και 1 κανίβαλο από τη δεξιά στην αριστερή όχθη

Μετάφερε 2 ιεραπόστολους από την αριστερή στη δεξιά όχθη

Μετάφερε 1 κανίβαλο από τη δεξιά στην αριστερή όχθη

Μετάφερε 2 κανίβαλους από την αριστερή στη δεξιά όχθη

Μετάφερε 1 ιεραπόστολο από τη δεξιά στην αριστερή όχθη

Μετάφερε 1 ιεραπόστολο και 1 κανίβαλο από την αριστερή στη δεξιά όχθη

Κατηγορίες προβλημάτων (1/2)

- Κατηγοριοποίηση ανάλογα με την ερμηνεία του όρου "*λύση*".

- Προβλήματα όπου είναι πλήρως γνωστές οι τελικές καταστάσεις και επιδιώκεται η εύρεση μίας σειράς ενεργειών:

  προβλήματα σχεδιασμού ενεργειών (*planning*) και προβλήματα πλοήγησης, στρατηγικής, εφοδιαστικής, κτλ.

- Προβλήματα όπου είναι γνωστές κάποιες ιδιότητες μόνο της τελικής κατάστασης και επιδιώκεται η εύρεση ενός πλήρους στιγμιότυπου της τελικής κατάστασης,

  προβλήματα χρονοπρογραμματισμού (*scheduling*), σταυρόλεξα, κρυπτογραφικά, κτλ.

  τα προβλήματα είναι γνωστά ως *προβλήματα ικανοποίησης περιορισμών* (*constraint satisfaction problems*).

Κατηγορίες προβλημάτων (2/2)

- Προβλήματα στα οποία είναι γνωστές κάποιες ιδιότητες μόνο της τελικής κατάστασης και επιδιώκεται η εύρεση μίας πλήρως γνωστής τελικής κατάστασης και η σειρά ενεργειών που θα οδηγήσουν σε αυτή:

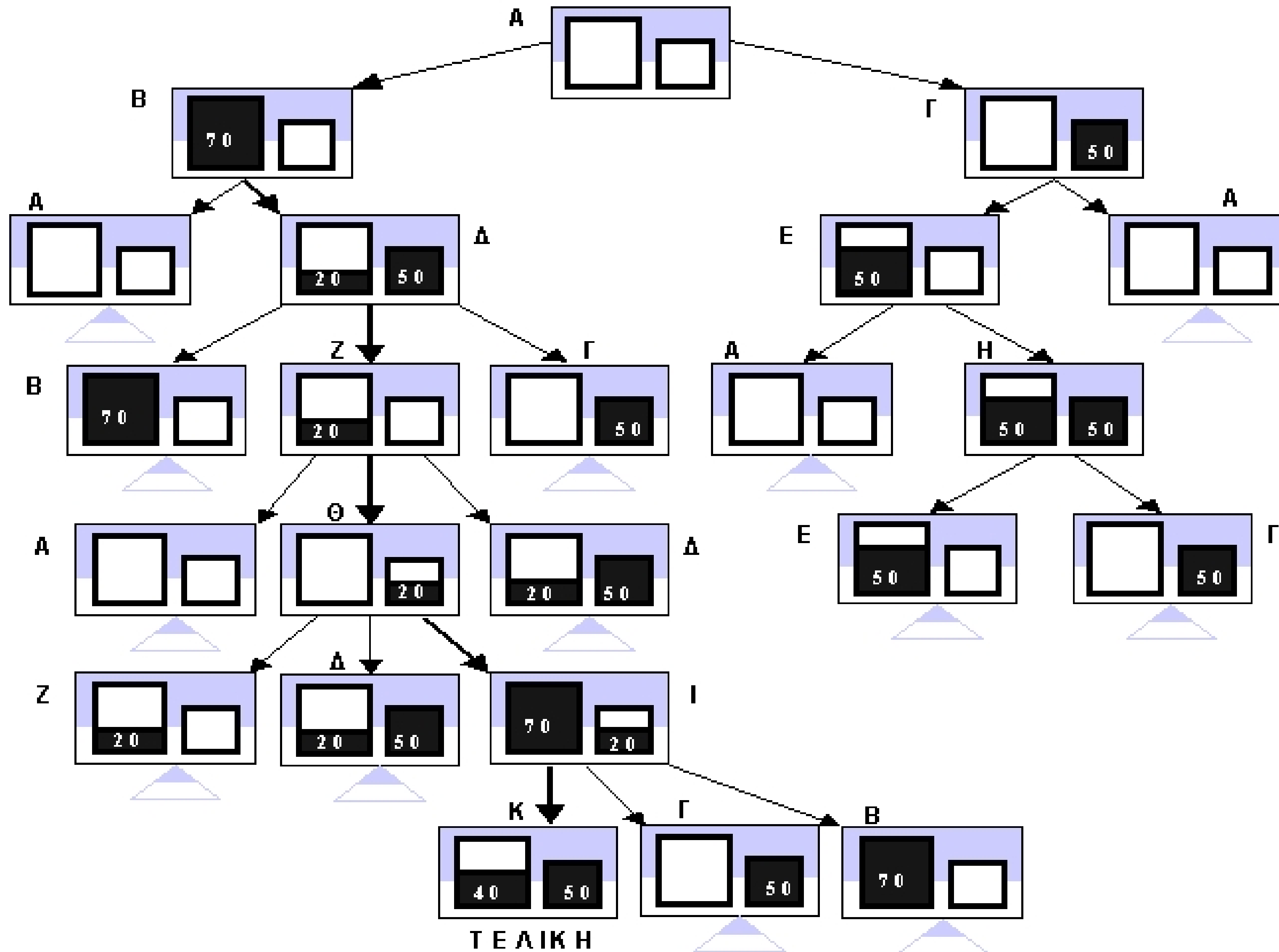
  προβλήματα διαμόρφωσης (*configuration*).

- Προβλήματα όπου είναι σχετικά εύκολο να βρεθούν λύσεις, αλλά το ζητούμενο είναι η βέλτιστη από αυτές.

  προβλήματα *βελτιστοποίησης*, στα οποία και πάλι η τελική κατάσταση δεν είναι πλήρως γνωστή αλλά είναι γνωστά κάποια χαρακτηριστικά της.

Αναζήτηση Πρώτα σε Βάθος (DFS)

Δένδρο αναζήτησης στο πρόβλημα των ποτηριών



Αναζήτηση Πρώτα σε Βάθος (DFS)
Πρόβλημα των ποτηριών

Μέτωπο της αναζήτησης	**Κλειστό Σύνολο**	**Κατάσταση**	**Παιδιά**
<Α>	{}	Α	<Β, Γ>
<Β, Γ>	{Α}	Β	<Α, Δ>
<Α, Δ, Γ>	{Α,Β}	Α	- (βρόχος)
<Δ, Γ>	{Α,Β}	Δ	<Β,Ζ,Γ>
<Β,Ζ,Γ,Γ>	{Α,Β,Δ}	Β	- (βρόχος)
<Ζ,Γ,Γ>	{Α,Β,Δ}	Ζ	<Α,Θ,Δ>
<Α,Θ,Δ,Γ,Γ>	{Α,Β,Δ,Ζ}	Α	- (βρόχος)
<Θ,Δ,Γ,Γ>	{Α,Β,Δ,Ζ}	Θ	<Ζ,Δ,Ι>
<Ζ,Δ,Ι,Δ,Γ,Γ>	{Α,Β,Δ,Ζ,Θ}	Ζ	- (βρόχος)
<Δ,Ι,Δ,Γ,Γ>	{Α,Β,Δ,Ζ,Θ}	Δ	- (βρόχος)
<Ι,Δ,Γ,Γ>	{Α,Β,Δ,Ζ,Θ}	Ι	<Κ,Γ,Β>
<Κ,Γ,Β,Δ,Γ,Γ>	{Α,Β,Δ,Ζ,Θ,Ι}	Κ	ΤΕΛΙΚΗ

2.1.1

Αναζήτηση Πρώτα σε Πλάτος (BFS)

Πρόβλημα των ποτηριών (1/2)

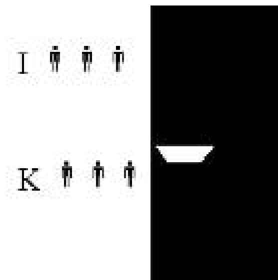
Μέτωπο αναζήτησης	Κλειστό Σύνολο	Κατάσταση	Παιδιά
<Α>	{}	Α	<Β, Γ>
<Β, Γ>	{Α}	Β	<Α, Δ>
<Γ,Α,Δ>	{Α,Β}	Γ	<Ε,Α>
<Α,Δ,Ε,Α>	{Α,Β,Γ}	Α	- (βρόχος)
<Δ,Ε,Α>	{Α,Β,Γ}	Δ	<Β,Ζ,Γ>
<Ε,Α,Β,Ζ,Γ>	{Α,Β,Γ,Δ}	Ε	<Α,Η>
<Α,Β,Ζ,Γ,Α,Η>	{Α,Β,Γ,Δ,Ε}	Α	- (βρόχος)
<Β,Ζ,Γ,Α,Η>	{Α,Β,Γ,Δ,Ε}	Β	- (βρόχος)
<Ζ,Γ,Α,Η>	{Α,Β,Γ,Δ,Ε}	Ζ	<Α,Θ,Δ>
<Γ,Α,Η,Α,Θ,Δ>	{Α,Β,Γ,Δ,Ε,Ζ}	Γ	- (βρόχος)
<Α,Η,Α,Θ,Δ>	{Α,Β,Γ,Δ,Ε,Ζ}	Α	- (βρόχος)
<Η,Α,Θ,Δ>	{Α,Β,Γ,Δ,Ε,Ζ}	Η	<Ε,Γ>
<Α,Θ,Δ,Ε,Γ>	{Α,Β,Γ,Δ,Ε,Ζ,Η}	Α	- (βρόχος)
<Θ,Δ,Ε,Γ>	{Α,Β,Γ,Δ,Ε,Ζ,Η}	Θ	<Ζ,Δ,Ι>

Αναζήτηση Πρώτα σε Πλάτος (BFS)

Πρόβλημα των ποτηριών (2/2)

Μέτωπο αναζήτησης	**Κλειστό Σύνολο**	**Κατάσταση**	**Παιδιά**
<Δ,Ε,Γ,Ζ,Δ,Ι>	{Α,Β,Γ,Δ,Ε,Ζ,Η}	Δ	- (βρόχος)
<Ε,Γ,Ζ,Δ,Ι>	{Α,Β,Γ,Δ,Ε,Ζ,Η}	Ε	- (βρόχος)
<Γ,Ζ,Δ,Ι>	{Α,Β,Γ,Δ,Ε,Ζ,Η}	Γ	- (βρόχος)
<Ζ,Δ,Ι>	{Α,Β,Γ,Δ,Ε,Ζ,Η}	Ζ	- (βρόχος)
<Δ,Ι>	{Α,Β,Γ,Δ,Ε,Ζ,Η}	Δ	- (βρόχος)
<Ι>	{Α,Β,Γ,Δ,Ε,Ζ,Η}	Ι	<Κ,Γ,Β>
<Κ,Γ,Β>	{Α,Β,Γ,Δ,Ε,Ζ,Η,Ι}	Κ	ΤΕΛΙΚΗ

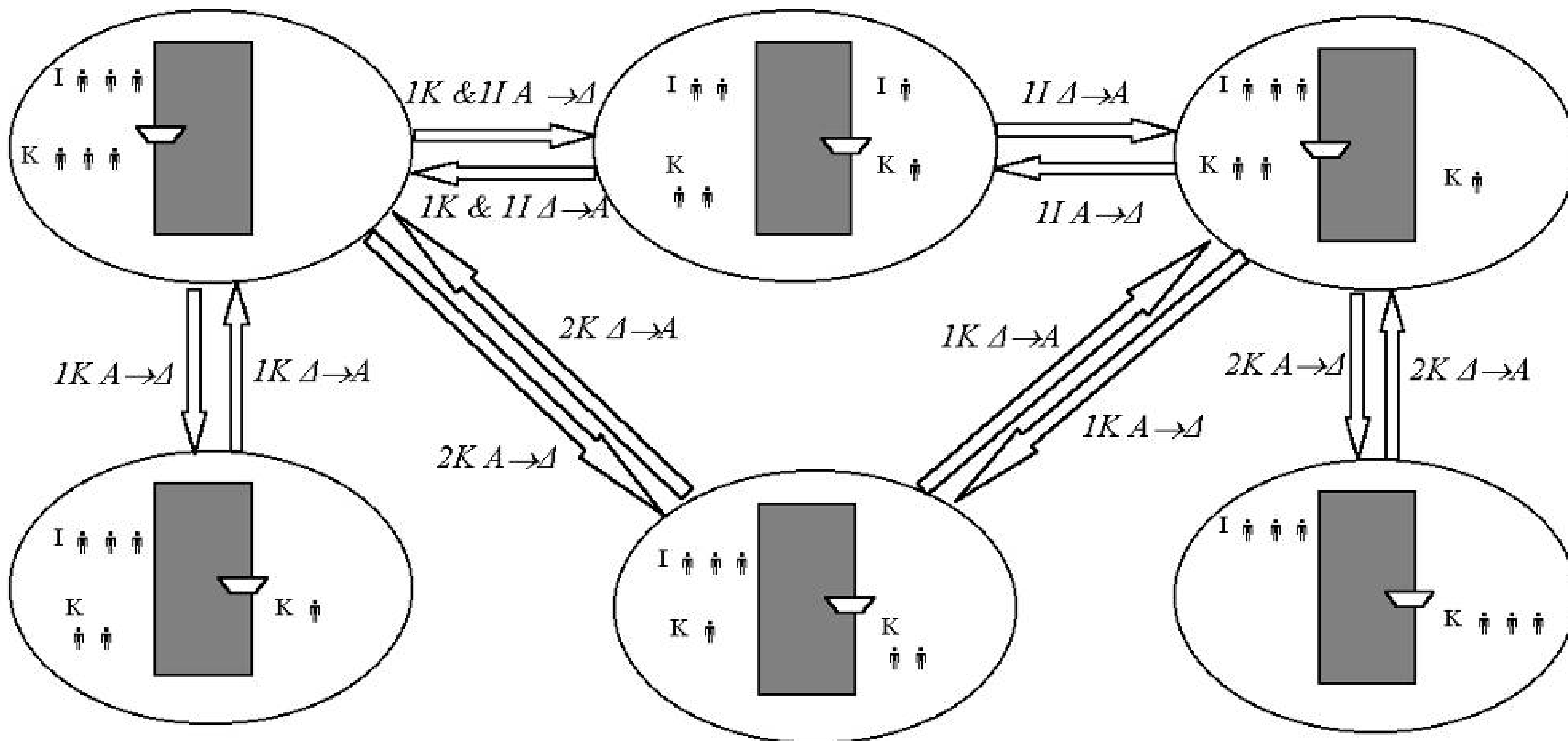
Παράδειγμα
Ιεραπόστολοι και Καννίβαλοι



Αντικείμενα	Ιδιότητες	Σχέσεις
3 Ιεραπόστολοι **3 Κανίβαλοι** Βάρκα Αριστερή Όχθη Δεξιά Όχθη	Βάρκα δύο ατόμων	Ιεραπόστολοι στην αριστερή όχθη Κανίβαλοι στην αριστερή όχθη Βάρκα στην αριστερή όχθη

Χώρος Καταστάσεων

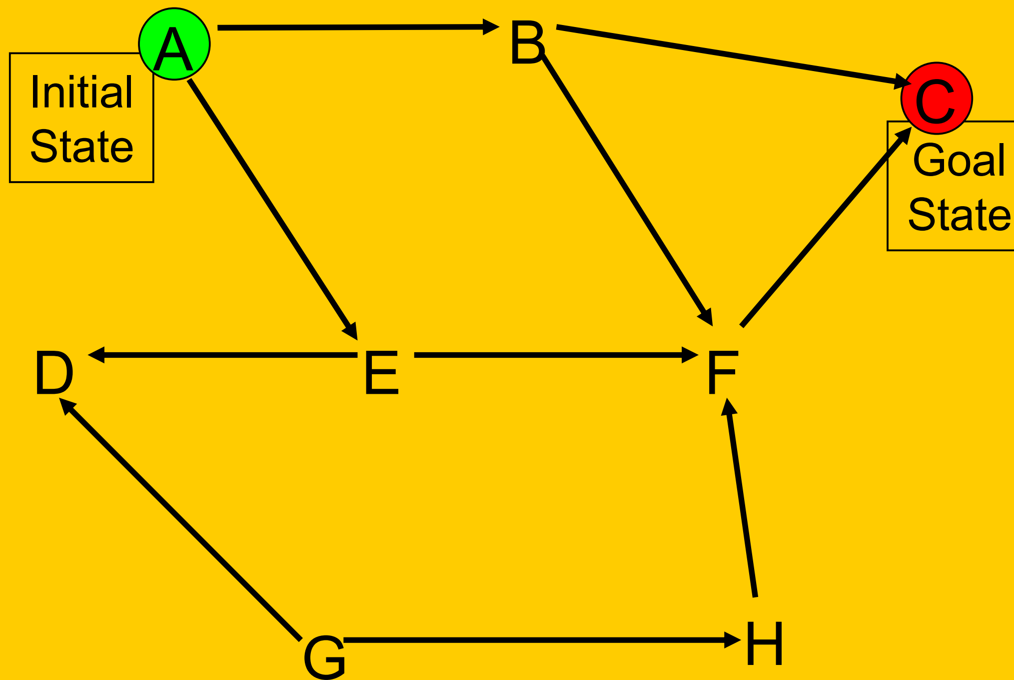
Χώρος καταστάσεων (state space ή domain space) ενός προβλήματος ονομάζεται το σύνολο όλων των έγκυρων καταστάσεων.



Παραδείγματα σε Prolog

```
link(g,h).
link(g,d).
link(e,d).
link(h,f).
link(e,f).
link(a,e).
link(a,b).
link(b,f).
link(b,c).
link(f,c).
```

State-Space




```
go(X,X,[X]).
go(X,Y,[X|T]):-
        link(X,Z),
        go(Z,Y,T).
```

Simple search algorithm

```
| ?- go(a,c,X).
X = [a,e,f,c] ? ;
X = [a,b,f,c] ? ;
X = [a,b,c] ? ;
no
```

Consultation

Implementing

❖ **To implement state-space search in Prolog, we need:**

- **A way of representing a state e.g. the board configuration**

  link(a,e).

- **A way of generating all of the next states reachable from a given state**

  go(X,Y,[X|T]):- link(X,Z), go(Z,Y,T).

- **A way of determining whether a given state is the one we're looking for. Sometimes this might be the goal state (a finished puzzle, a completed route, a checkmate position); other times it might simply be the state we estimate is the best, using some evaluation function**

  go(X,X,[X]).

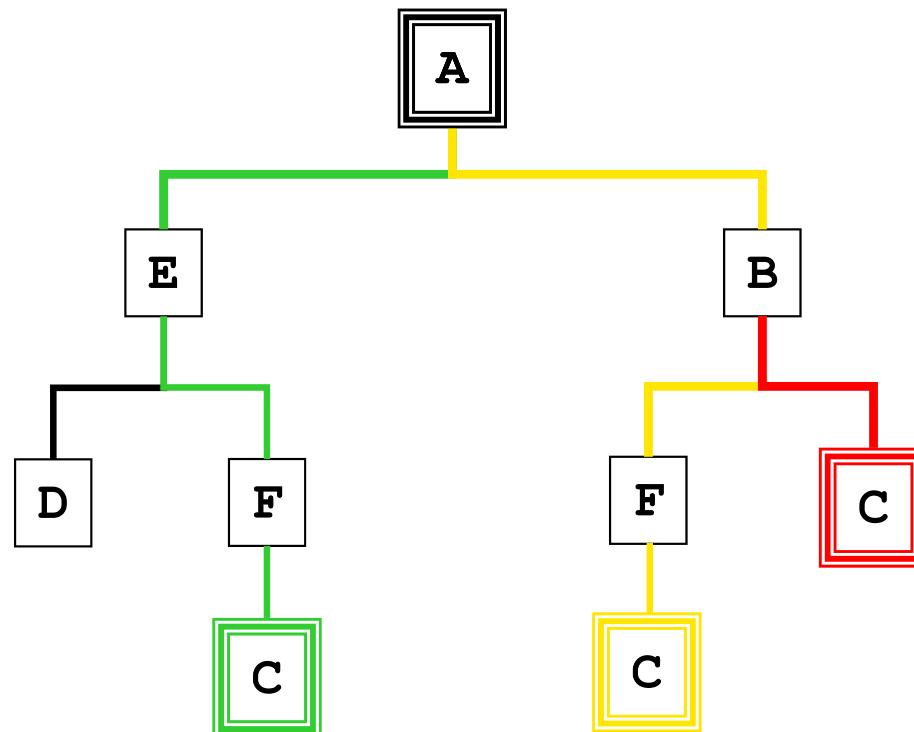
- **A mechanism for controlling how we search the space**

Depth-First Search

- This simple search algorithm uses Prolog's unification routine to find the first link from the current node then follows it
- It always follows the left-most branch of the search tree first; following it down until it either finds the goal state or hits a dead-end
- It will then backtrack to find another branch to follow

```
go(X,X,[X]).
go(X,Y,[X|T]):-
        link(X,Z),
        go(Z,Y,T).
```

```
| ?- go(a,c,X).
X = [a,e,f,c] ? ;
X = [a,b,f,c] ? ;
X = [a,b,c] ? ;
no
```



Iterative Deepening

- If the optimal solution is the shortest path from the initial state to the goal state depth-first search will usually not find this
- We need to vary the depth at which we look for a solution; increasing the depth every time we have exhausted all nodes at a particular depth
- We can take advantage of Prolog's backtracking to implement this very simply

Depth-First

```
go(X,X,[X]).

go(X,Y,[X|T]):-
      link(X,Z),
      go(Z,Y,T).
```

Iterative Deepening

```
go(X,X,[X]).

go(X,Y,[Y|T]):-
      go(X,Z,T),
      link(Z,Y).
```

Check if current node is goal.

Find an intermediate node.

Check whether intermediate links with goal.

-

Iterative Deepening search is quite useful as:

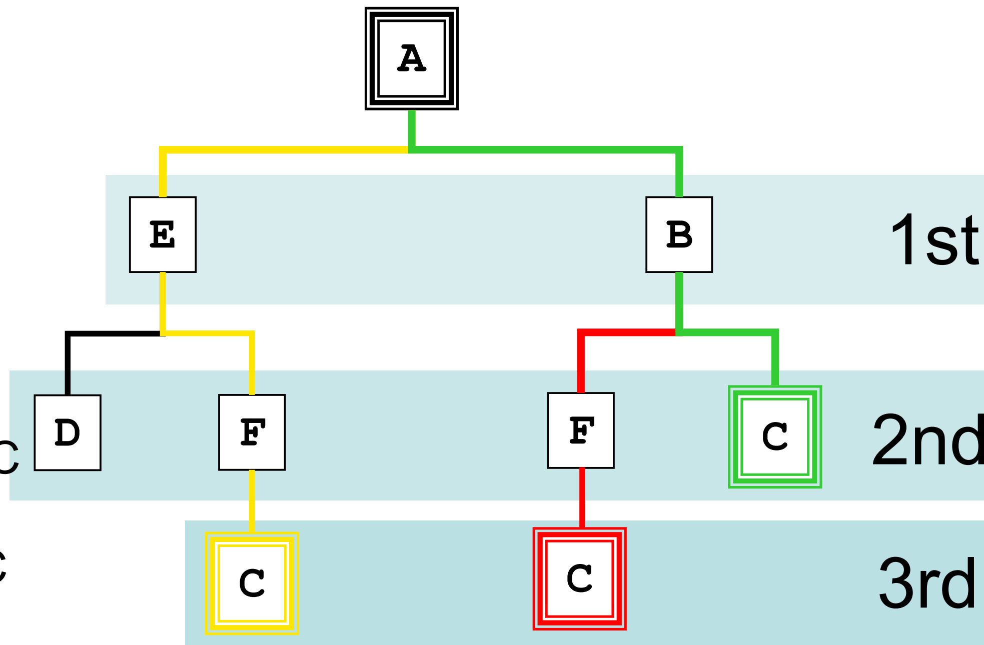
- it is simple
- reaches a solution quickly and
- with minimal memory requirements as at any point in the search it is maintaining only one path back to the initial state

However

- on each iteration it has to re-compute all previous levels and extend them to the new depth
- may not terminate (e.g. loop)
- may not be able to handle complex state-spaces
- can’t be used in conjunction with problem-specific heuristics as keeps no memory of optional paths

Breadth-First Search

```
| ?- go(a,c,X).
X = [a,b,c] ? ;
X = [a,e,f,c] ? ;
X = [a,b,f,c] ? ;
no
```



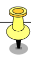
Depth-first
= A,ED,FC,BFC,C

Breadth-first
= A,EB,DFFC,CC

- **A simple, common alternative to depth-first search is breadth-first search.**
- **This checks every node at one level of the space, before moving onto the next level.**
- **It is distinct from iterative deepening as it maintains a list of alternative candidate nodes that can be expanded at each depth**

Agenda-based search

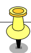
- **Both depth-first and breadth-first search can be implemented using an agenda (breadth-first can only be implemented with an agenda).**
- **The agenda holds a list of states in the state space, as we generate (‘expand') them, starting with the initial state.**
- **We process the agenda from the beginning, taking the first state each time. If that state is the one we're looking for, the search is complete.**
- **Otherwise, we expand that state, and generate the states which are reachable from it. We then add the new nodes to the agenda, to be dealt with as we come to them.**



Example Agenda = [[c,b,a],[c,f,e,a],[c,f,b,a]]

Here's a very general skeleton for agenda-based search:

```
search(Solution) :-
    initial_state(InitialState),
    agenda_search([[ InitialState ]], Solution).

agenda_search([[ Goal | Path ] | _ ], [ Goal | Path ]) :-
    is_goal(Goal).

agenda_search([[ State | Path ] | Rest], Solution) :-
    get_successors([ State | Path ], Successors),
    update_agenda(Rest, Successors, NewAgenda),
    agenda_search(NewAgenda, Solution).
```


To complete the skeleton, we need to implement:

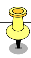
initial_state/1,
which creates the initial state for the state-space.

is_goal/1,
which succeeds if its argument is the goal state.

get_successors/2
which generates all of the states which are reachable from a given state (should take advantage of findall/3, setof/3 or bagof/3 to achieve this).

update_agenda/2,
which adds new states to the agenda (usually using append/3).

With this basic skeleton, depth-first search and breadth-first search may be implemented with a simple change:



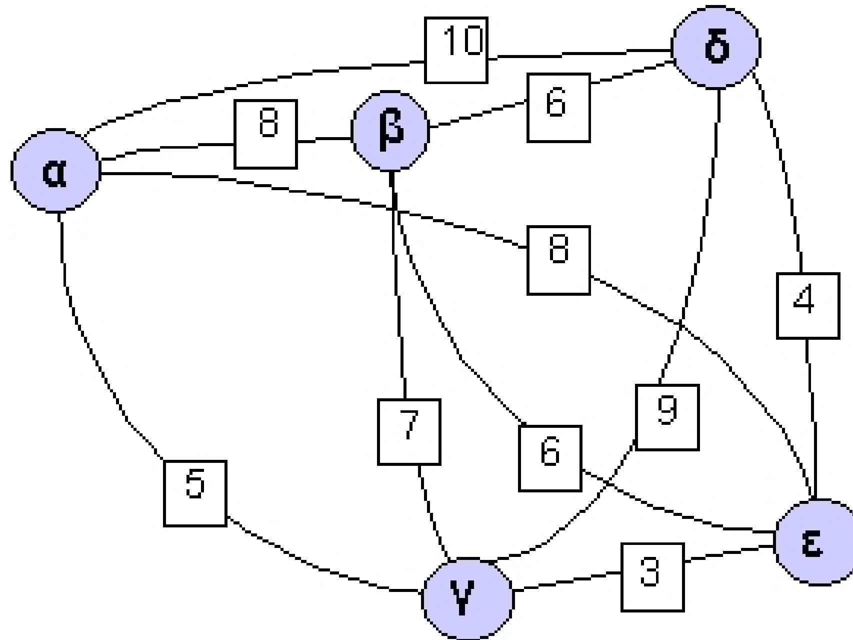
Adding newly-generated agenda items to the beginning of the agenda implements depth-first search:

```
update_agenda(OldAgenda, NewStates, NewAgenda) :-
 append(NewStates, OldAgenda, NewAgenda).
```

Adding newly-generated agenda items to the end of the agenda implements breadth-first search:

```
update_agenda(OldAgenda, NewStates, NewAgenda) :-
 append(OldAgenda, NewStates, NewAgenda).
```

Το Πρόβλημα του Πλανόδιου Πωλητή (TSP)



- Το πρόβλημα ανήκει στην κατηγορία *μη-πολυωνυμικού χρόνου* (*NP-complete*).
  - a problem is NP-complete if answers can be verified quickly, and a quick algorithm to solve this problem can be used to solve all other NP problems quickly
- Το πρόβλημα είναι πρόβλημα ελαχιστοποίησης κόστους και έχει πολλές εφαρμογές.

Ο αλγόριθμος Branch and Bound στο πρόβλημα TSP

Μέτωπο της αναζήτησης	**Κόστος Λύσης**	**Κατάσταση**	**Παιδιά**
<α>	+∞	α	αβ^{8},αγ^{5},αδ^{10},αε^{8}
<αβ^{8},αγ^{5},αδ^{10},αε^{8}>	+∞	αβ	αβγ^{15},αβδ^{14}, αβε^{14}
<αβγ^{15},αβδ^{14},αβε^{14},αγ^{5},...>	+∞	αβγ	αβγδ^{24}, αβγε^{18}
<αβγδ^{24},αβγε^{18}, αβδ^{14},αβε^{14}...>	+∞	αβγδ	αβγδε^{28}
<αβγδε^{28},αβγε^{18}, αβδ^{14},...>	+∞	αβγδε	αβγδεα^{36}
< αβγδεα^{36}, αβγε^{18}, αβδ^{14},..>	36	αβγδεα	Τελική Κατάσταση
<αβγε^{18}, αβδ^{14},... >	36	αβγε	αβγεδ^{22}
<αβγεδ^{22},αβδ^{14},...>	36	αβγεδ	αβγεδα^{32}
< αβγεδα^{32} ,αβδ^{14},αβε^{14}...>	32	αβγεδα^{32}	Τελική Κατάσταση
...	...	...	...
<αβδεγα^{26}, ...>	26	αβδεγα	Τελική Κατάσταση
...	...	...	...
<αβεγδ^{26}, ...>	26	αβεγδ	Κλάδεμα
....	...	...	...
<αεβγδ^{30}, ...>	26	αεβγδ	Κλάδεμα

Μέτωπο της αναζήτησης	**Κόστος Λύσης**	**Κατάσταση**	**Παιδιά**
. . .	. . .	. . .	. . .
<>	ΕλάχιστηΤιμή	ΤΕΛΟΣ	

Αλγόριθμοι Ευριστικής Αναζήτησης
Εισαγωγικά

- Ο χώρος αναζήτησης συνήθως αυξάνεται εκθετικά. Απαιτείται πληροφορία για αξιολόγηση των καταστάσεων (ευριστικός μηχανισμός).
- Οι αλγόριθμοι που εκμεταλλεύονται τέτοια πληροφορία ονομάζονται Αλγόριθμοι Ευριστικής Αναζήτησης.
- Παράδειγμα ευριστικής αναζήτησης είναι η συναρμολόγηση ενός puzzle.
- Αν δεν υπήρχαν ευριστικοί μηχανισμοί, τότε τα προβλήματα αυτά θα λύνονταν πολύ δύσκολα, γιατί οι συνδυασμοί που πρέπει να γίνουν είναι πάρα πολλοί.
- Ο ευριστικός μηχανισμός εξαρτάται από τη γνώση που έχουμε για το πρόβλημα.

Ευριστικός μηχανισμός

Ευριστικός μηχανισμός (heuristic) είναι μία στρατηγική, βασισμένη στη γνώση για το συγκεκριμένο πρόβλημα, η οποία χρησιμοποιείται σα βοήθημα στη γρήγορη επίλυσή του.

- Ο ευριστικός μηχανισμός υλοποιείται με ευριστική συνάρτηση (heuristic function).
- Ευριστική τιμή (heuristic value) είναι η τιμή της ευριστικής συνάρτησης και εκφράζει το πόσο κοντά βρίσκεται μία κατάσταση σε μία τελική.
- Η ευριστική τιμή δεν είναι η πραγματική τιμή της απόστασης από μία τερματική κατάσταση, αλλά μία εκτίμηση (estimate*)* που πολλές φορές μπορεί να είναι και λανθασμένη.

2.1.2

Ευριστικές Συναρτήσεις σε Μικρά Προβλήματα (1/3)

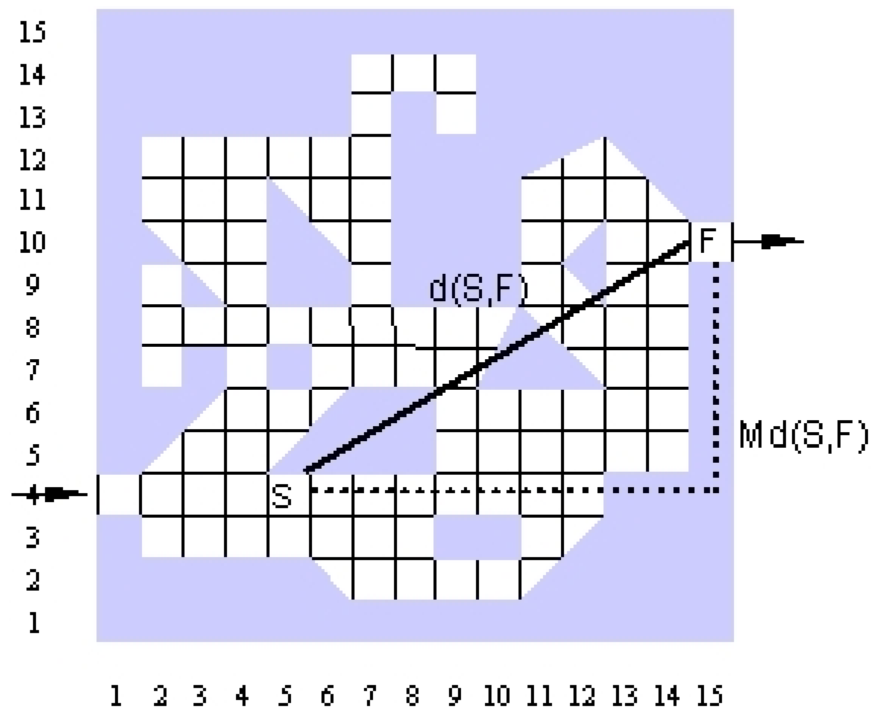
Ευριστικός μηχανισμός και συναρτήσεις σε λαβύρινθο

- Ευκλείδεια απόσταση (Euclidian distance):

$$d(S,F) = \sqrt{(X_S - X_F)^2 + (Y_S - Y_F)^2}$$

- Απόσταση Manhattan (Manhattan distance):

$$Md(S,F) = |X_S - X_F| + |Y_S - Y_F|$$



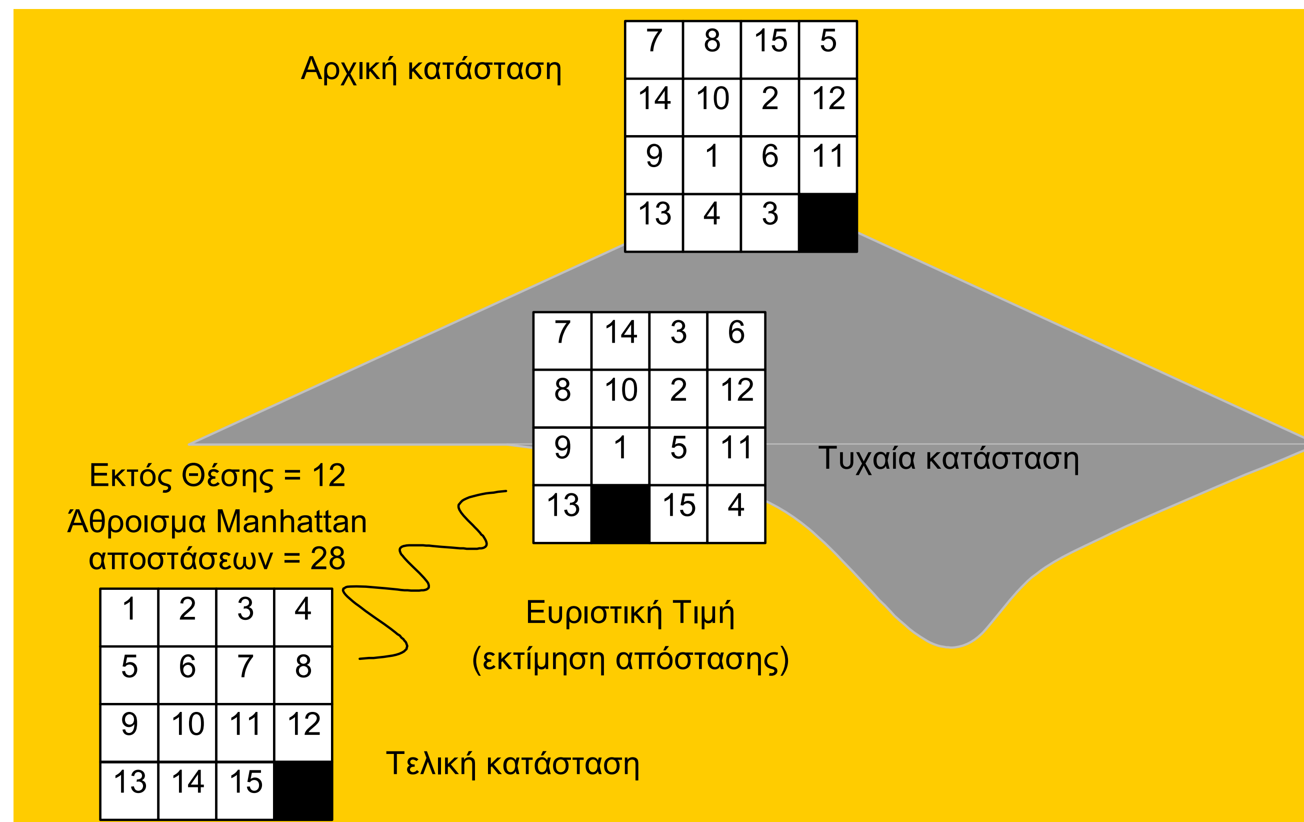
$$d(S,F) = \sqrt{(5-15)^2 + (4-10)^2} = \sqrt{(100+36)} = 11,6$$

Md(S,F)=|5-15| + |4-10|=10+6=16.

Ευριστικές Συναρτήσεις σε Μικρά Προβλήματα (2/3)

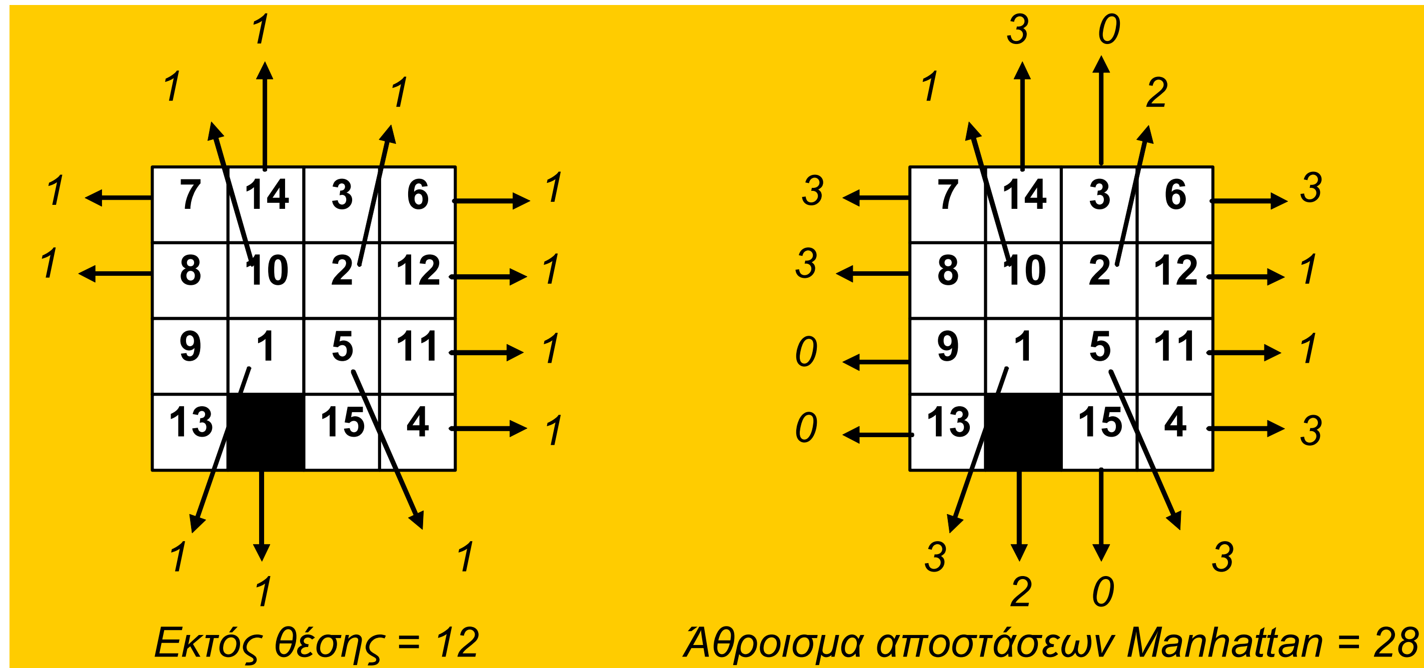
Ευριστικός μηχανισμός και συναρτήσεις στο N-Puzzle

- Πόσα πλακίδια βρίσκονται εκτός θέσης.
- Το άθροισμα των αποστάσεων Manhattan κάθε πλακιδίου από την τελική του θέση.



Ευριστικές Συναρτήσεις σε Μικρά Προβλήματα (3/3)

Αναλυτικός υπολογισμός ευριστικής τιμής για μία τυχαία κατάσταση του 15-puzzle.



Ευριστικός μηχανισμός και συναρτήσεις στο TSP

- Η κοντινότερη πόλη έχει περισσότερες πιθανότητες να οδηγήσει σε μία συνολικά καλή λύση.

2.1.3

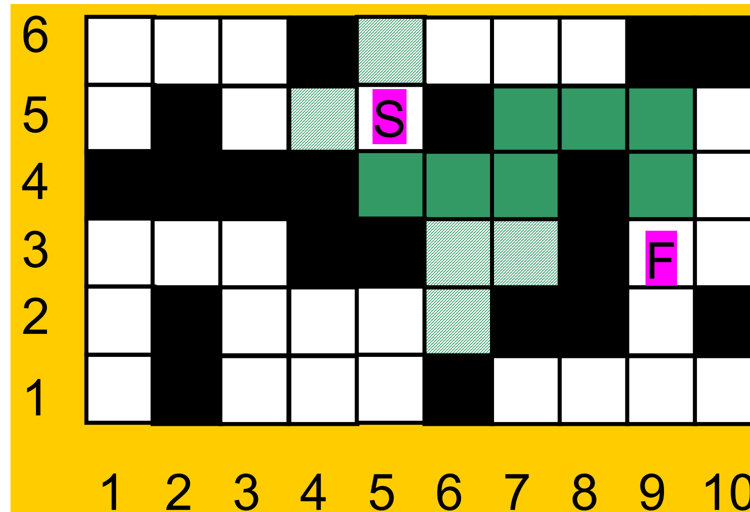
Αναζήτηση Πρώτα στο Καλύτερο BestFS

Ο αλγόριθμος *αναζήτηση* πρώτα στο καλύτερο (Best-First - *BestFS*) κρατά όλες τις καταστάσεις στο μέτωπο αναζήτησης.

Ο αλγόριθμος BestFS

1. **Βάλε την αρχική κατάσταση στο μέτωπο αναζήτησης.**
2. **Αν το μέτωπο αναζήτησης είναι κενό τότε σταμάτησε.**
3. **Πάρε την πρώτη σε σειρά κατάσταση από το μέτωπο αναζήτησης.**
4. **Αν η κατάσταση είναι μέλος του κλειστού συνόλου τότε πήγαινε στο 2.**
5. **Αν η κατάσταση είναι μία τελική τότε ανάφερε τη λύση και σταμάτα.**
6. **Εφάρμοσε τους τελεστές μεταφοράς για να παράγεις τις καταστάσεις-παιδιά.**
7. **Εφάρμοσε την ευριστική συνάρτηση σε κάθε παιδί.**
8. **Βάλε τις καταστάσεις-παιδιά στο μέτωπο αναζήτησης.**
9. **Αναδιάταξε το μέτωπο αναζήτησης, έτσι ώστε η κατάσταση με την καλύτερη ευριστική τιμή να είναι πρώτη.**
10. **Βάλε τη κατάσταση-γονέα στο κλειστό σύνολο.**
11. **Πήγαινε στο βήμα 2.**

Ο αλγόριθμος BestFS: το πρόβλημα του λαβύρινθου



Μέτωπο Αναζήτησης	Κλειστό Σύνολο	Κατάσταση	Παιδιά
<5-5>	<>	5-5	5-4^{5},5-6^{7},4-5^{7}
<5-4^{5},5-6^{7},4-5^{7}>	<5-5>	5-4	5-5^{6},6-4^{4}
<6-4^{4},5-5^{6},5-6^{7},4-5^{7}>	<5-5,5-4>	6-4	5-4^{7},6-3^{3},7-4^{3}
<6-3^{3},7-4^{3},5-5^{6},5-6^{7},...>	<5-5,5-4,6-4>	6-3	6-4^{4},6-2^{3},7-3^{2}
<7-3^{2},6-2^{3},7-4^{3},6-4^{4},5-5^{6},...>	<5-5,5-4,...>	7-3	6-3^{3},6-4^{4}
<6-3^{3},6-2^{3},7-4^{3},6-4^{4},5-5^{6},...>	<...,6-3,...>	6-3	Βρόχος
<6-2^{3},7-4^{3},6-4^{4},5-5^{6},5-6^{7},...>	<...>	6-2	5-2^{5},6-3^{3}
<7-4^{3},6-4^{4},5-2^{5},...>	<...>	7-4	7-5^{4},6-4^{4},7-3^{2}

Μέτωπο Αναζήτησης	**Κλειστό Σύνολο**	**Κατάσταση**	**Παιδιά**
<7-3^2,7-5^4,6-4^4,5-2^5,...>	<..,7-3,..>	7-3	Βρόχος
<7-5^4,6-4^4,5-2^5,...>	<...>	7-5	7-4^3,8-5^3,7-6^5
<8-5^3,7-4^3,6-4^4,...>	<...>	8-5	8-6^4,7-5^4,9-5^2
<9-5^2,7-4^3,6-4^4,8-6^4,...>	<...>	9-5	8-5^3,9-4^1
<9-4^1,8-5^3,7-4^3,...>	<...>	9-4	9-3^0,9-5^2,10-4^2
<9-3^0,9-5^2,10-4^2,...>	<...>	9-3	ΤΕΛΙΚΗ ΚΑΤΑΣΤΑΣΗ
		ΤΕΛΟΣ	

- Η λύση στο παραπάνω πρόβλημα είναι η διαδρομή που ορίζεται από τη σειρά των θέσεων: $5^5 \rightarrow 5^4 \rightarrow 6^4 \rightarrow 7^4 \rightarrow 7^5 \rightarrow 8^5 \rightarrow 9^5 \rightarrow 9^4 \rightarrow 9^3$.

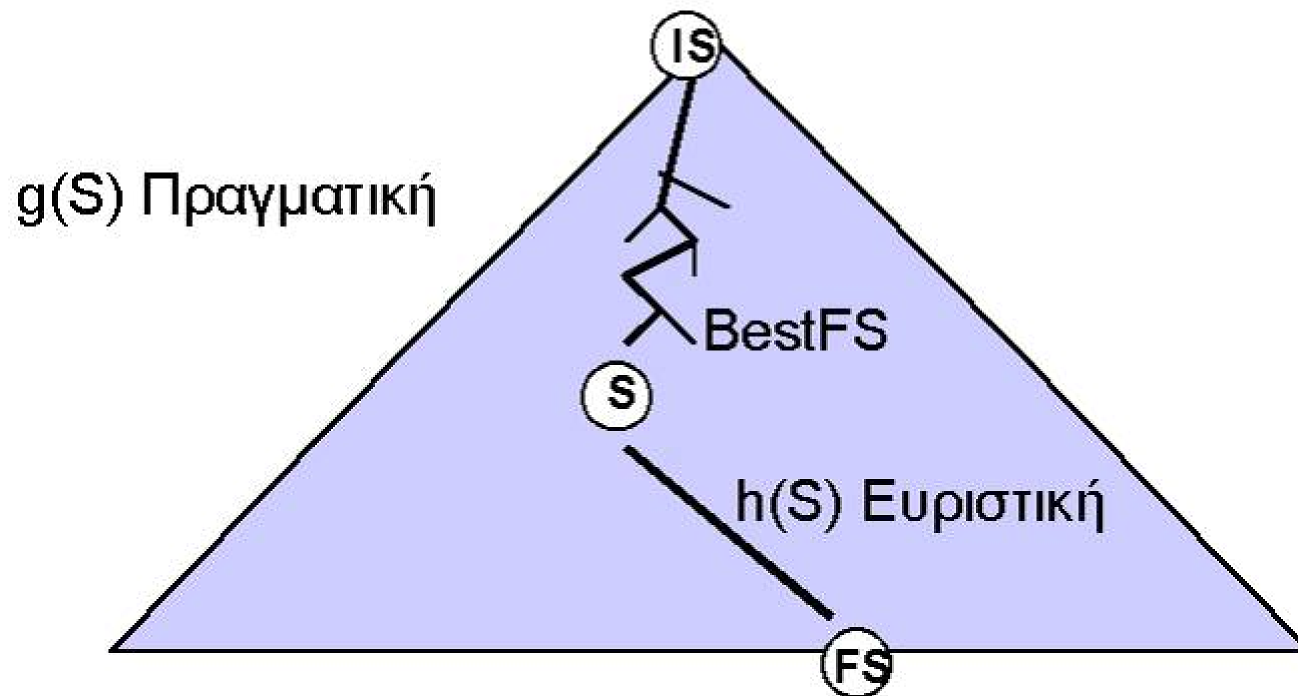
Ο Αλγόριθμος Άλφα-Άστρο (Α*)

Ο *αλγόριθμος Α* (*Άλφα Άστρο*) είναι κατά βάσει BestFS, αλλά με ευριστική συνάρτηση:

F(S) = g(S) + h(S)

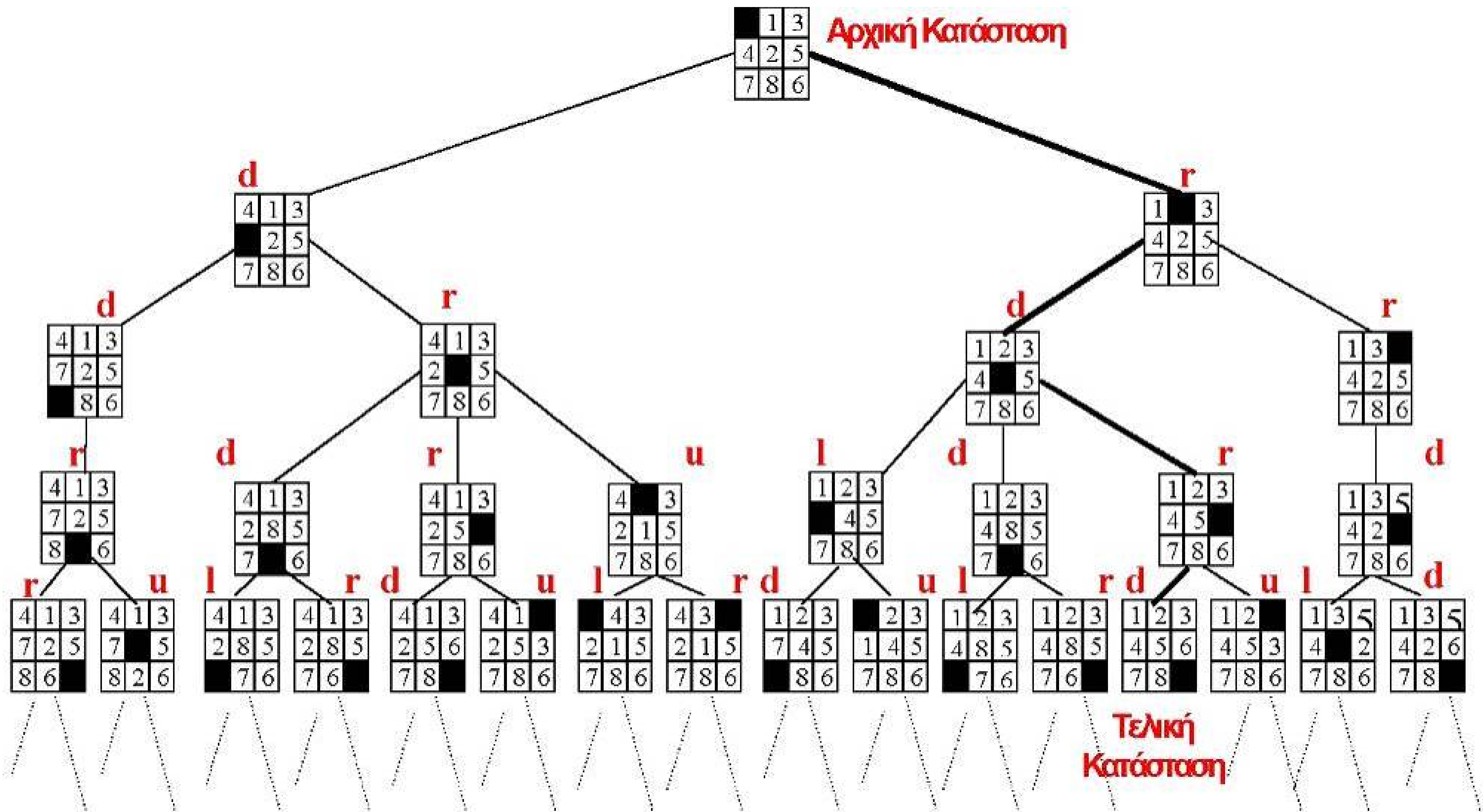
η g(S) δίνει την απόσταση της S από την αρχική κατάσταση, η οποία είναι πραγματική και γνωστή, και

η h(S) δίνει την εκτίμηση της απόστασης της S από την τελική κατάσταση μέσω μιας ευριστικής συνάρτησης, όπως ακριβώς στον BestFS.

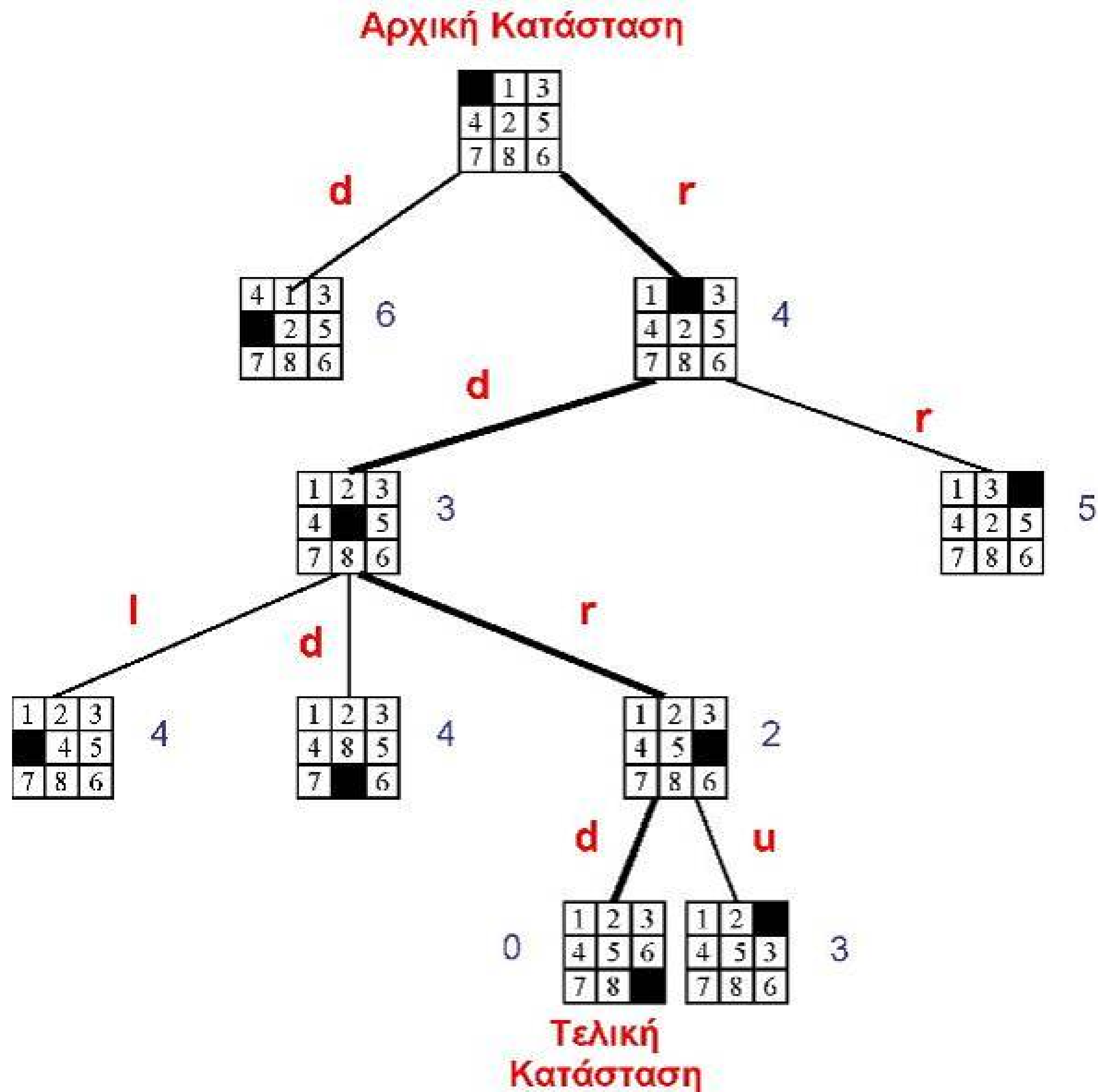


Εφαρμογή των Αλγορίθμων Ευριστικής Αναζήτησης

Χώρος Αναζήτησης στο 8-puzzle



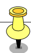
Εφαρμογή αλγορίθμου BestFS στο 8-puzzle



Παραδείγματα σε Prolog

```
sort_agenda([], NewAgenda, NewAgenda).
sort_agenda([State|NewStates], OldAgenda, SortedAgenda):-
        insert(State, OldAgenda, NewAgenda),
        sort_agenda(NewStates, NewAgenda, SortedAgenda).

insert(New, [], [New]).
insert(New, [Old|Agenda], [New,Old|Agenda]):-
      h(New,H), h(Old,H2), H =< H2.
insert(New, [Old|Agenda], [Old|Rest]):-
      insert(New, Agenda, Rest).
```

- This is a very general skeleton. By implementing **sort_agenda/3**, according to whatever domain we're looking at, we can make the search strategy *informed* by our knowledge of the domain.



- Best-first search isn't so much a search strategy, as a mechanism for implementing many *different* types of informed search.

- One simple way to sort the agenda is by the cost-so-far. This might be the number of moves we've made so far in a game, or the distance we've travelled so far looking for a route between towns (traveling salesman problem).

- If we sort the agenda so that the states with the lowest costs come first, then we'll always expand these first, and that means that we're sure we'll always find an optimal solution first.

- This is branch & bound with agenda sorting. It looks a lot like breadth-first search, except that it will find an optimal solution even if the steps between states have different costs (e.g. the distance between towns is irregular).

- However, branch & bound doesn't really direct us towards the goal we're looking for, so it isn't very informed.

Αποσφαλμάτωση (1/3)

Το γεγονός ότι η Prolog είναι μια διερμηνευόμενη γλώσσα (interpreted) διευκολύνει σημαντικά τη διαδικασία αποσφαλμάτωσης, καθώς επιτρέπει την εύκολη δοκιμή των διαφόρων επιμέρους κατηγορημάτων του προγράμματος. Έτσι μια καλή προσέγγιση στην εύρεση οποιουδήποτε σφάλματος σε ένα Prolog πρόγραμμα, είναι ο έλεγχος πρώτα των απλούστερων κατηγορημάτων του προγράμματος, και έπειτα των περισσότερο πολύπλοκων μέχρι να φτάσει η διαδικασία στο βασικό στόχο.

Ο έλεγχος πρέπει να περιλαμβάνει δοκιμές του κατηγορήματος με όσο το δυνατό περισσότερα σετ τιμών των ορισμάτων. Ιδιαίτερη προσοχή πρέπει να δοθεί στις οριακές τιμές των ορισμάτων, δηλαδή να γίνει έλεγχος για τις ακραίες τιμές για τις οποίες το κατηγόρημα θα πρέπει να μπορεί να διαχειριστεί , καθώς και οι τιμές για τις οποίες θα πρέπει να αποτυχαίνει.

Σε όλες τις εκδόσεις της γλώσσας, υπάρχει η δυνατότητα παρακολούθησης βήμα προς βήμα της εκτέλεσης του προγράμματος (tracing). Στις περισσότερες εκδόσεις της γλώσσας η δυνατότητα αυτή ενεργοποιείται με την εντολή

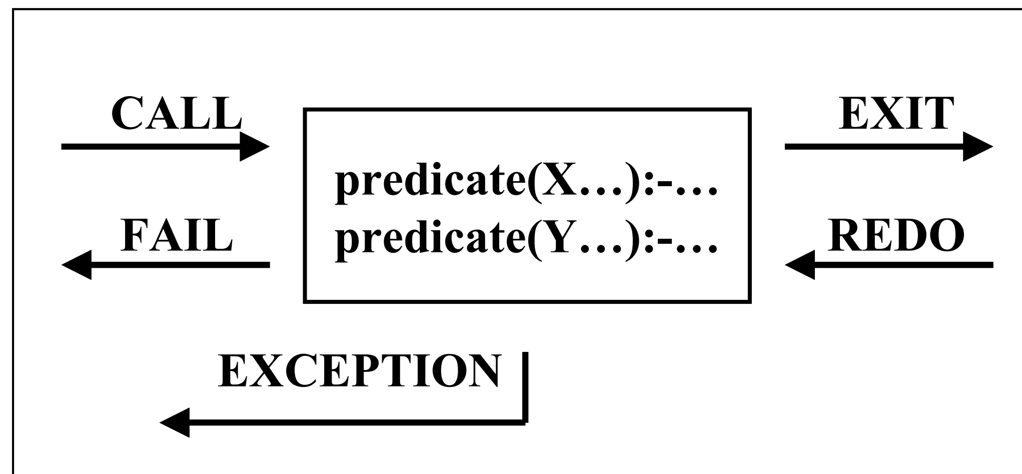
```
?- trace.
```

και απενεργοποιείται με την εντολή

```
?- notrace.
```

Αποσφαλμάτωση (2/3)

Η απεικόνιση της εκτέλεσης ακολουθεί το μοντέλο του κουτιού ελέγχου ροής εκτέλεσης διαδικασίας (procedure box flow control model). Το μοντέλο αυτό παρουσιάζεται στο ακόλουθο σχήμα:



- CALL σηματοδοτεί την αρχική κλήση του κατηγορήματος
- EXIT σηματοδοτεί την επιτυχία στην κλήση του κατηγορήματος

Αποσφαλμάτωση (3/3)

- REDO σηματοδοτεί ότι η προηγούμενη κλάση του κατηγορήματος έχει αποτύχει και δοκιμάζεται η επόμενη εναλλακτική πρόταση για την ικανοποίηση της κλήσης. Το ίδιο μπορεί να συμβαίνει και όταν ζητάμε εναλλακτικές λύσεις.
- FAIL σηματοδοτεί ότι η κλήση απέτυχε εντελώς, δηλ. δεν υπάρχουν προτάσεις που να ικανοποιούν την κλήση.
- EXCEPTION σηματοδοτεί ότι έγινε **σφάλμα** κατά την εκτέλεση της κλήσης, όπως πχ. χρησιμοποιήθηκε λάθος τύπου όρισμα σε κάποιο κατηγόρημα. Προσοχή, δεν πρέπει να συγχέεται με τη αποτυχία. Συνήθως, και αν δεν έχει ληφθεί μέριμνα από τον προγραμματιστή, η εκτέλεση του προγράμματος σταματά.

Παράδειγμα (1/3)

Παρακάτω δίνεται ένα παράδειγμα εκτέλεσης ενός απλού προγράμματος:

```
a(X):- b(X).
a(X):- c(X).
b(1).
b(2).
c(3).
| ?- trace.
trace.
{the debugger will first creep -- showing everything
(trace)}
yes
{trace}
```

Παράδειγμα (2/3)

```
| ?- a(X).
1 1 Call: a(_183) ?
2 2 Call: b(_183) ?
2 2 Exit: b(1) ?
1 1 Exit: a(1) ?
X = 1 ? ;
```

(Ζητούμε την εύρεση επόμενης λύσης)

```
1 1 Redo: a(1) ?
2 2 Redo: b(1) ?
2 2 Exit: b(2) ?
1 1 Exit: a(2) ?
X = 2 ? ;
```

(Ζητούμε την εύρεση επόμενης λύσης)

Παράδειγμα (3/3)

```
1 1 Redo: a(2) ?
2 2 Redo: b(2) ?
2 2 Fail: b(_183) ?
2 2 Call: c(_183) ?
2 2 Exit: c(3) ?
1 1 Exit: a(3) ?
X = 3 ? ;
```

(Ζητούμε την εύρεση επόμενης λύσης)

```
1 1 Redo: a(3) ?
2 2 Redo: c(3) ?
2 2 Fail: c(_183) ?
1 1 Fail: a(_183) ?
no
{trace}
```

Σειρά προτάσεων και κλήσεων (1/5)

- Στην ιδανική περίπτωση η σειρά των προτάσεων σε ένα πρόγραμμα Prolog δεν θα έπρεπε να έχει σημασία, ωστόσο αυτό δεν συμβαίνει. Η σειρά των προτάσεων και των κλήσεων μέσα σε ένα πρόγραμμα επηρεάζει όχι μόνο την απόδοση του προγράμματος αλλά και την ορθότητά του.
- Παράδειγμα: το ακόλουθο πρόγραμμα μετρά τα στοιχεία μιας λίστας

```
len_list(N,[_|Rest}):-
len_list(NN,Rest),
N is NN + 1.
len_list(0,[]).
```

Στην ερώτηση που ακολουθεί το πρόγραμμα φαίνεται να δουλεύει ικανοποιητικά

```
?- len_list(N,[a,b,c]).
N = 3
```

Σειρά προτάσεων και κλήσεων (2/5)

Αν όμως δώσουμε οποιαδήποτε από τις παρακάτω ερωτήσεις

```
| ?- len_list(3,L).
Error 1, Backtrack Stack Full, Trying len_lest/2
Aborted
| ?- len_list(N,K).
Error 1, Backtrack Stack Full, Trying len_lest/2
Aborted
```

η εκτέλεση πέφτει σε ατέρμονα βρόχο καθώς ο αριστερότερος κλάδος του δένδρου αναζήτησης , που δημιουργείται από την πρώτη πρόταση του κατηγορήματος, έχει άπειρο μήκος. Το πρόβλημα διορθώνεται αν αλλάξουμε τη σειρά των προτάσεων

```
len_list(0,[]).
len_list(N,[_|Rest]):-
len_list(NN,Rest),
N is NN + 1.
```

Σειρά προτάσεων και κλήσεων (3/5)

Στην περίπτωση αυτή οι δύο προηγούμενες ερωτήσεις θα δώσουν

```
?- len_list(N,K).
N = 0,
K = [];
N = 1,
K = [_274732];
N = 2,
K = [_274732,_274914];
N = 3,
K = [_274732,_274914,_275100]
| ?- len_list(3,L).
L = [_301566,_301588,_301610]
```

Σειρά προτάσεων και κλήσεων (4/5)

- Θα πρέπει λοιπόν να δίνεται ιδιαίτερη σημασία και προσοχή στη σειρά των προτάσεων στο πρόγραμμα, ιδιαίτερα όταν κάποιοι κλάδοι είναι άπειρου μήκους, το οποίο είναι συχνό στις αναδρομικές διαδικασίες.
- Ένα άλλο σημείο το οποίο χρίζει προσοχής είναι η σειρά των κλήσεων στο σώμα ενός κανόνα, ιδίως σε κατηγορήματα τα οποία απαιτούν δεν δέχονται ελεύθερες μεταβλητές στα ορίσματά τους. Ένα τέτοιο παράδειγμα δίνεται πάλι με το κατηγόρημα len_list/2

```
len_list(0,[]).
len_list(N,[_|Rest]):-
N is NN + 1,
   len_list(NN,Rest).
```

Στην περίπτωση αυτή το κατηγόρημα δεν θα λειτουργήσει καμιά ερώτηση καθώς η μεταβλητή NN που εμφανίζεται στο δεύτερο μέρος του is/2 δεν έχει πάρει τιμή κατά τη χρονική στιγμή που καλείται.

Σειρά προτάσεων και κλήσεων (5/5)

- Πέρα από την ορθότητα του προγράμματος η σειρά των κλήσεων επηρεάζει το δένδρο αναζήτησης και κατά συνέπεια το χρόνο εκτέλεσης του κατηγορήματος. Αν και δεν υπάρχει συγκεκριμένη μεθοδολογία εκτέλεσης, μια καλή τακτική είναι να μπαίνουν πρώτα οι κλήσεις οι οποίες είναι πιθανό να αποτύχουν συντομότερα, δηλαδή έχουν μικρότερο αριθμό εναλλακτικών λύσεων.