

Introduction to Logic Programming and Prolog

Pitikakis Marios, PhD



Introduction

- **Logic Programming** (LP) belongs to Declarative Programming i.e. to a different category/style of programming than the “classic” approach of Imperative Programming (e.g. Procedural, Object-oriented etc.)
- **Prolog** (PROgramming in LOGic) is a representative language of this type. It is based on Mathematical Logic, emerged in the early 1970s and was used successfully in the effort of developing automatic theorem proving systems.
- Prolog is very useful for the development of intelligent systems in different problem areas of Artificial Intelligence (**AI**)
 - like Natural Language Processing, Expert systems, Knowledge systems and Decision Making systems
 - but also in other areas like Relational Database Systems and the Semantic Web.
- ... but pretty useless in others, such as graphics or numerical algorithms.



Applications of Prolog

- **Expert systems (Εμπειρα συστήματα)**
 - They are computer systems emulating the decision making ability of a human expert in a specific field. E.g. diagnosis of heart disease and treatment proposal, damage detection in cars and establishment of an appropriate procedure for their repair, investment advice for the Stock Market etc.
- **Natural Language Processing - NLP (Κατανόηση/επεξεργασία φυσικής γλώσσας)**
 - Automatic “understanding” of text or even a spoken language. Such applications are intended for the automatic translation from one natural language to another, automatic extraction of summaries from texts, voice commands etc. For the successful comprehension of a natural language, it is necessary to encode both the linguistic knowledge and the knowledge of the world (domain) to which the text or sentences refer to.



Applications of Prolog

- **Search and Optimization problems**
 - Search problems constitute a vast category of Artificial Intelligence problems, in which, given an initial state of the problem and a set of legal actions from one state to another, the question is to find a sequence of transitions between states to achieve a certain goal or arrive at a final state.
 - Such problems include route planning where the appropriate actions must be found to achieve a specific goal, scheduling problems, problems of assigning a set of resources without violating the involved constraints etc.
 - e.g. movement decisions for robot navigation, drones etc.



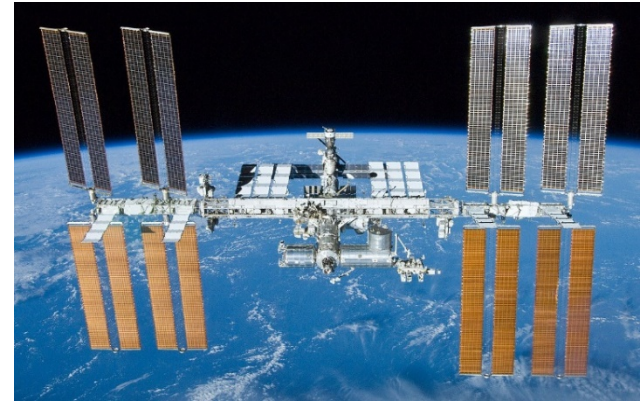
Applications of Prolog

- **Theorem Provers (Απόδειξη θεωρημάτων)**
 - Implementation of automated theorem proving systems (also known as ATP or automated deduction) which deal with proving mathematical theorems from a set of logical axioms. Commercial use of automated theorem proving is mostly concentrated in integrated circuit design and proof verification (where an existing proof for a theorem is certified valid).
- **Symbolic processing (Συμβολική επεξεργασία)**
 - Applications that need symbolic processing e.g. systems of symbolic derivation and symbolic integration, systems of symbolic (not numerical) solution of equations, systems of polynomials simplification and, in general, mathematical formulas, etc.



Real world examples of Prolog

- Prolog was used by NASA to program the natural language interface in the International Space Station (ISS) (in 2005)
 - Spoken Language Processing (Clarissa Procedure Browser)
 - The first spoken dialog system in space.
- Parts of the Watson QA supercomputer (IBM) were programmed in Prolog (in 2011).
 - Competed and won the first prize of \$1 million on the quiz show *Jeopardy!* against two champions
 - Commercial application: decision support in lung cancer treatment at Memorial Sloan Kettering Cancer Center



Real world examples of Prolog

- Web applications (<http://www.pathwayslms.com/swipltuts/html/index.html>)



Programming in Prolog

- Programming in Prolog consists of defining data and relationships between **facts** and **rules** through which new relationships between data can be induced with the help of other already defined relationships.
- A Prolog program is a collection of clauses/expressions (facts and rules) that specify **what problem you want to solve**, without explicitly specifying **how to solve it**.
- The clauses of the program constitute a **Knowledge Base (Βάση Γνώσης)** from which we can deduce new knowledge.
- When answering a **query**, Prolog tries to prove that the corresponding goal is satisfiable (can be made true). This is done using the rules and facts given in a program.



Advantages of Prolog

- The high level of abstraction, coming from its declarative nature, makes it particularly “human centric”
 - enables the developer to focus directly on how to describe the problem.
 - makes the code more readable and smaller in size
- Simple syntax
 - a useful feature for the representation of knowledge and reasoning.
- It has a built-in unification mechanism, which allows easy pattern matching
 - used in many applications of Artificial Intelligence
- It has an automatic backtracking mechanism that makes it easy to describe search algorithms.



Implementations of Prolog

- There is a large number of Prolog implementations, and development environments that support it.
- Indicatively:
 - SWI-Prolog (<http://www.swi-prolog.org>) is a free version of the language and is one of the most popular Prolog environments used by the research and education community.
 - ECLiPSe (<http://eclipseclp.org>)
 - B-Prolog (<http://www.picat-lang.org/bprolog/>)
 - SICStus Prolog (<https://sicstus.sics.se>)
- There are also a lot of online Prolog compilers (that mainly use GNU Prolog v1.4.x) e.g.
 - https://www.tutorialspoint.com/execute_prolog_online.php
 - <https://www.jdoodle.com/execute-prolog-online>
 - https://www.onlinegdb.com/online_prolog_compiler
 - <https://swish.swi-prolog.org/> (SWI-Prolog, supports Notebooks)



Imperative VS Declarative Programming

- In "conventional" programming languages (like C/C++, Java) programs implement algorithms, mixing in their code two basic elements:
 - the **logic** of the algorithm, which describes what we want the program to achieve
 - **control**, which describes the sequence of steps to be followed to find the solution
- Symbolic equation of Kowalski:

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

- On the other hand, in Declarative Programming, **flow control** commands are completely **absent** and only the definition of the logical part of the problem is stated.



Imperative VS Declarative Programming

- In a procedural style of programming, the program explicitly describes the individual steps of computation. It requires that we already know before hand the algorithm (the way to solve) the problem and we could potentially solve it without the use of a computer. Algorithmic programming does not use the computer as a "smart" machine. The **programmer finds the algorithms** and it is he/she who solves the problem and not the computer.
- In Declarative Programming to solve a problem we do **not need to know an algorithm in advance**. As long as we fully **specify** and **define** the problem, the computer finds the solution for us.



Imperative VS Declarative Programming

- Once we define what the problem is, Prolog does a full investigation and finds all possible solutions.
- The difficulty has now been shifted from finding the right algorithm, to **creating a correct and complete definition of the problem**.
- So, what have we gained? Another "tool".
We use languages like Prolog for problems that:
 - either we can not find a solution algorithm (and we can not use classical algorithmic languages)
 - either are more easily coded with Declarative Programming



Example: Calculate the factorial of N (N!)

- Abstract mathematical description of the problem:
 - If $N = 0$ then $N! = 1$
 - If $N > 0$ then $N! = 1 \times 2 \times 3 \times \dots \times (N-1) \times N$
- Code in C (using a loop and recursion)

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int i, n;
6      long factorial = 1;
7
8      printf("Enter a number to calculate its factorial\n");
9      scanf("%d", &n);
10
11     if (n < 0)
12         printf("Factorial of negative integers isn't defined.\n");
13     else
14     {
15         for (i = 1; i <= n; i++)
16             factorial = factorial * i;
17
18         printf("Factorial of %d = %d\n", n, factorial);
19     }
20
21     return 0;
22 }
```

```
1  #include<stdio.h>
2
3  long factorial(int);
4
5  int main()
6  {
7      int n;
8      long f;
9
10     printf("Enter an integer to find its factorial\n");
11     scanf("%d", &n);
12
13     if (n < 0)
14         printf("Factorial of negative integers isn't defined.\n");
15     else
16     {
17         f = factorial(n);
18         printf("%d! = %ld\n", n, f);
19     }
20
21     return 0;
22 }
23
24 long factorial(int n)
25 {
26     if (n == 0)
27         return 1;
28     else
29         return(n * factorial(n-1));
30 }
```



Example: Calculate the factorial of N (N!)

- Prolog code (using recursion)

```
1 factorial(0,1).  
2 factorial(N,F):-  
3     N>0,  
4     N1 is N-1,  
5     factorial(N1,F1),  
6     F is N*F1.
```

- Notice that there are no flow control commands at all in our program and the Prolog code does nothing more than give the recursive mathematical definition of the factorial.
- [A more detailed explanation of the example will be given later...]



Conclusions

- Declarative Programming, in contrast to Imperative Programming, focuses on the description of “**What**” is the problem and not “**How**” to solve it.
- In Declarative Programming, the programmer **specifies what is valid for the problem** (objects, properties, relations, restrictions) and it is up to the “system” (interpreter/compiler) to figure out how to perform the computation requested. The programmer should not have to tell the computer what to do.
- In Logic Programming, programs are structured as sets of definitions that specify at a high level "what" will be solved, focusing on the logical part of describing a problem.



Conclusions

- In Logic Programming, statements are **logical expressions** and the "system" is a theorem prover (αποδείκτης θεωρημάτων).
- In Prolog the verification of a logical program is easier than in a program written in a procedural programming language.
 - checking for correct results in a procedural program is based on the abstract description of the behavioral specifications of the program
 - running a program in Prolog is practically like proving of a theorem!



Syntactic rules

- Prolog's syntax follows the **predicate logic of Horn clauses** (κατηγορηματική λογική προτάσεων/φράσεων Horn)
- The method of **resolution with Horn clauses** (see CS180) finds application in Logical Programming and Prolog.
- In Prolog, implications are written
 - with the implication “conditions” on the right
 - and the “conclusions” on the left of the implication symbol (συνεπαγωγή), which has reverse direction.



Syntactic rules

The basic syntactic rules of the language are:

- the symbol of conjunction/AND (\wedge) is replaced by comma (“,”)
- the symbol of disjunction/OR (\vee) is replaced by semicolon (“;”)
- the symbol of implication (\leftarrow) is replaced by the symbols of colon and dash (“:-”)

	Logic	Prolog
Implication (Συνεπαγωγή)	$A \rightarrow B$	$B :- A$
Conjunction (Σύζευξη)	$A \wedge B$	A , B
Disjunction (Διάζευξη)	$A \vee B$	$A ; B$



Syntactic rules

- Constants, the names of functors and the names of predicates always start with a **lowercase letter**
- (logical) variables always start with a **capital letter**
- all Prolog clauses finish with a **full stop** symbol “.”



Anatomy of a program

- A Prolog programs consist of a set of clauses (προτάσεις).
- Prolog clauses are normally of three types:
 - **facts** (γεγονότα) declare things that are always true.
 - **rules** (κανόνες) declare things that are true depending on a given condition.
 - **questions** (ερωτήσεις) to find out if a particular goal is true.



Propositional Logic (Προτασιακή Λογική)

- Advantages:
 - Syntax simplicity Απλότητα στην σύνταξη
 - It can always come to a “conclusion” (decidable)
- Disadvantages:
 - Lack of generality
 - It implies that the "world" consists of facts which are true or false
 - There is no way to separate and access the entities of the "world".
- That's why we use Predicate Logic



Predicate Logic (Κατηγορηματική Λογική)

- **Predicate logic** is an **extension** of propositional logic and enables the separation and access of the entities of the "world" to which the statements refer to.
- Provides mechanisms for the representation and manipulation of objects, properties and relationships between them.
- It also introduces the concepts of: **terms** (ὅρων), **predicates** (κατηγορημάτων) and **quantifiers** (ποσοδεικτών).
- To assign properties to objects we use predicate symbols (for properties) and arguments (for objects).
 - E.g. to assign the property F to the object α , we write $F(\alpha)$.
 - To represent a relationship we use predicates with more than one arguments. E.g. the relation R between a and b is written as $R(a, b)$



Using logic to do reasoning

- Example: Given information about fatherhood and motherhood, determine the grandparent relationship.
- E.g. Given the following *facts*
 - *Helen is the mother of Maria*
 - *John is the father of Maria*
 - *Maria is the mother of George*
 - *Nick is the father of Stella*
- Who are the grandparents of George?
- Who are the grandparents of Stella?



Example of a Prolog program

- In Prolog, we write the following for the given **facts**:

mother(helen, maria).

mother(maria, george).

father(john, maria).

father(nick, stella).



mother-child
and father-child
pairs

- Words like mother and father are called **predicates** (κατηγορήματα). **Arguments** are separated by commas. The number of arguments is called **arity** (τάξη) of the predicate. We usually refer to a predicate with the following symbolism **name/arity** e.g. mother/2
- mother, helen, maria are **atoms** (άτομα) and a statement like *mother(helen, maria)* is called an **atomic formula** (ατομικός τύπος), stating a true fact.



Example of a Prolog program

- We express the grandparent relationship with the **rule** (κανόνα):
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y). } **parent(X, Y) :- father(X, Y) ; mother(X, Y).**
- The left part of the rule is called **head** (κεφαλή) e.g. *grandparent(X, Z)* and the left part is called **body** (σώμα) e.g. *parent(X, Y), parent(Y, Z)*.
- *parent(X, Y)* is the first subgoal of the rule and *parent(Y, Z)* is the second subgoal of the rule. We may have several subgoals in the body of a rule that are separated by commas.



Example of a Prolog program

- X, Y and Z (with capital letters) denote variables, meaning “for any” (“για κάθε”).
- E.g. the rule

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

means

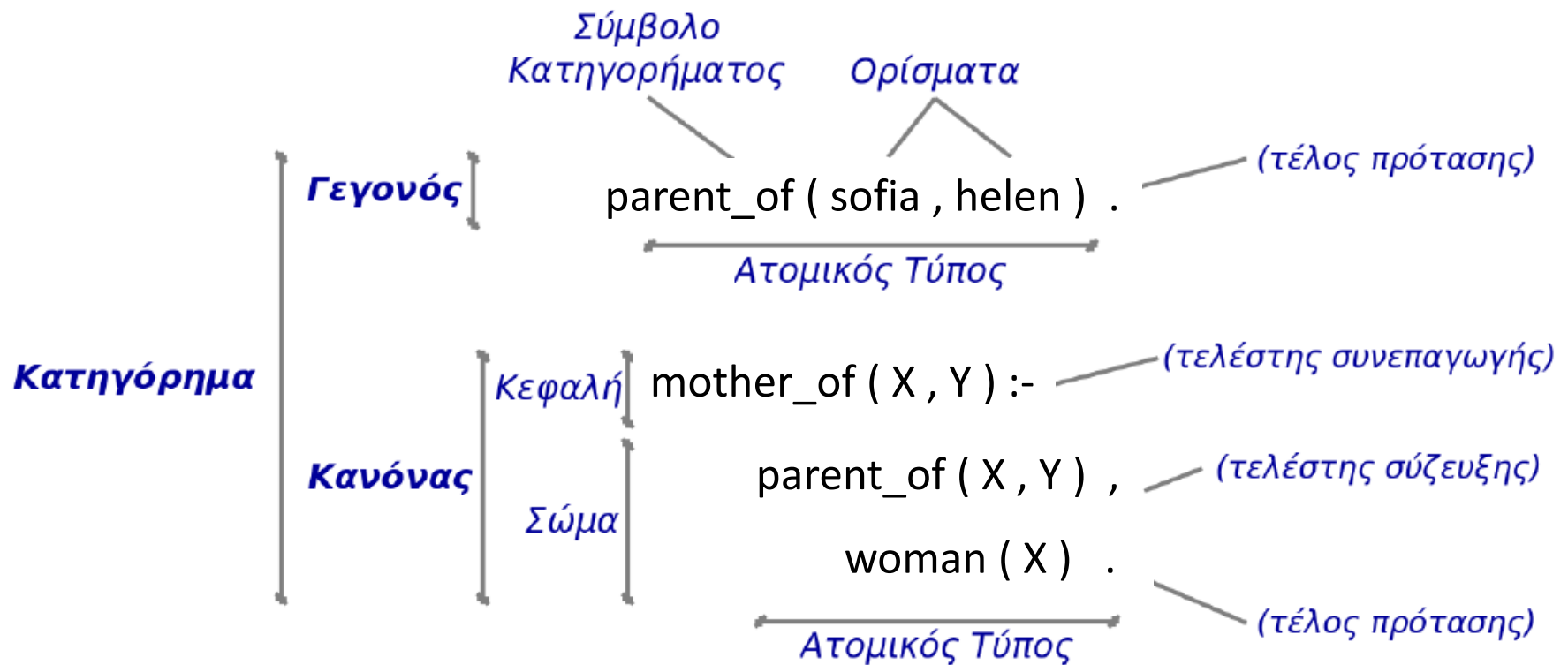
For any X, Y, Z,

if X is a parent of Y, and Y is a parent of Z,
then X is a grandparent of Z.

- Rules are **conditional statements** and :- is the implication operator (τελεστής συνεπαγωγής).
- Οι κανόνες εκφράζουν την αλήθεια μιας σχέσης **υπό συνθήκη**, δηλαδή στη συγκεκριμένη περίπτωση η σχέση grandparent είναι αληθής όταν είναι αληθής η σύζευξη (λογικό ΚΑΙ) που εμφανίζεται δεξιά του συμβόλου της συνεπαγωγής (σύμβολο :-)



Facts, Rules and Predicates



Ως σύμβολο κατηγορήματος μπορεί να είναι οποιαδήποτε συμβολοσειρά **ξεκινά με πεζό γράμμα** και περιέχει γράμματα ([a-z/A-Z]), αριθμούς [0-9] και το σύμβολο της κάτω παύλας “_”. Τα σύμβολα-ονόματα των κατηγορημάτων δεν έχουν κανένα σημασιολογικό νόημα (θα μπορούσε να είναι οτιδήποτε).



Questions in Prolog

- **Running** a program in Prolog is driven by **user queries**.
- Μπορούμε να ρωτήσουμε για την αλήθεια μιας λογικής πρότασης, να διαπιστώσουμε δηλαδή αν μια πρόταση αποτελεί λογικό συμπέρασμα του προγράμματος. **Η απόδειξη της πρότασης αποτελεί την “εκτέλεση” ενός προγράμματος.**
- Όταν η ερώτηση δεν περιέχει μεταβλητές, η απάντηση μπορεί να είναι απλά καταφατική ή αρνητική (true/false).
- Όταν η ερώτηση περιέχει μεταβλητές, τότε η απάντηση περιέχει μία ανάθεση τιμών στις μεταβλητές αυτές για τις οποίες αληθεύει η ερώτηση.



Questions in Prolog

Example:

- Is Hellen a grandparent of George ?

?- grandparent(helen, george).

Answer:

true

- Who are the grandparents of Stella ?

?- grandparent(X, stella).

Answer:

false

Facts:

```
mother(helen, maria).  
mother(maria, george).  
father(john, maria).  
father(nick, stella).
```



Questions in Prolog

Example:

- Who are the grandparents of George ?
?- grandparent(X, george).

Answer:

X = john

X = helen

- Who are the grandchildren of John ?
?- grandparent(john, X).

Answer:

X = george

Facts:

mother(helen, maria).
mother(maria, george).
father(john, maria).
father(nick, stella).



Βρίσκει **όλες** τις τιμές της μεταβλητής (ή των μεταβλητών, αν υπάρχουν πολλές) που ικανοποιούν την ερώτησή μας. Τυπώνει τις τιμές μια-μια. Γράφει μόνο την πρώτη, και στη συνέχεια περιμένει να πατήσουμε ένα πλήκτρο. Αν πατήσουμε το πλήκτρο ";" θα τυπώσει την επόμενη τιμή που ικανοποιεί την ερώτηση που κάναμε κλπ.



Questions in Prolog

Example:

– Ποιοι είναι οι γονείς της Maria ?

?- parent(X, maria).

Answer:

X = john

X = helen

or

?- father(X, maria) ; mother(Y, maria).

Answer:

X = john

Y = helen

Facts:

mother(helen, maria).

mother(maria, george).

father(john, maria).

father(nick, stella).



Prolog Terms (Όροι στη Prolog)

- All Prolog data structures are called **terms** (όροι). A term is either:
 - A **constant** (σταθερά), which can be either an **atom** or a **number**
 - A **variable** (μεταβλητή)
 - A **compound term** (σύνθετος όρος) or a **functional term** (συναρτησιακός όρος)



Constants in Prolog

- Constants are either **atoms** or **numbers**. Atoms represent the entities of the problem, and may be:
 - any sequence of characters, numbers and/or the underscore character which is preceded by a **lower case character**
e.g. monday_morning, theSun, more_Days12
 - anything enclosed in **single quotes** e.g. 'Monday Morning', 'SUN', 's o s'
 - any of the special atoms: [], {}, ; , !
 - any continuous sequence of the symbols: +, -, *, /, \, ^, <, >, =, :, ., ?, @, #, \$, &
e.g. ==>, \# =, >>>, --->, &=>
- There are build-in predicates: **atom/1** , **integer/1** , **real/1**



Variables in Prolog

- (Λογικές) μεταβλητές ορίζονται από οποιεσδήποτε συμβολοσειρές ξεκινούν **με κεφαλαίο γράμμα** και αποτελούνται από γράμματα, ψηφία και την κάτω παύλα.
π.χ. X1, Father, FileName, File_name, File_Name
- Οι μεταβλητές μπορούν να πάρουν ως τιμή οποιονδήποτε όρο, δηλαδή μια σταθερά (άτομο ή αριθμό), μια άλλη μεταβλητή ή ένα σύνθετο όρο (compound term).
- Υπάρχει το build-in κατηγορημα: **var/1**
 - μια μεταβλητή μπορεί να είναι σε δύο καταστάσεις: ελεύθερη (free/unbound), δηλαδή δεν της έχει ακόμη δοθεί τιμή, ή δεσμευμένη (bound), οπότε έχει πάρει τιμή μέσω του μηχανισμού ενοποίησης.
 - Το var/1 επιστρέφει **true** όταν το όρισμά του είναι μια ελεύθερη μεταβλητή



Variables in Prolog

- Υπάρχουν δύο σημαντικές διαφορές των λογικών μεταβλητών με τις μεταβλητές των κλασσικών διαδικαστικών γλωσσών:
 - ο **μηχανισμός ανάθεσης τιμής** σε μεταβλητή: Ο μοναδικός τρόπος για μία μεταβλητή να πάρει τιμή είναι μέσω του μηχανισμού **ενοποίησης** δύο όρων. Ο ειδικός ενθεματικός τελεστής (infix operator) που συμβολίζεται με το **ίσον** (=) καλεί ρητά τον μηχανισμό ενοποίησης στα δύο ορίσματά του. Δεν έχει σημασία τι εμφανίζεται αριστερά και δεξιά του τελεστή, τα δύο ορίσματα γίνονται συντακτικά όμοια με κατάλληλες αναθέσεις τιμών.
π.χ. $X = 5$ και $5 = X$ σημαίνουν το ίδιο πράγμα



Variables in Prolog

- Η σημαντικότερη διαφορά είναι ότι η Prolog ανήκει στην κατηγορία των γλωσσών **μοναδικής ανάθεσης (single assignment)**, δηλαδή, από την στιγμή (σημείο εκτέλεσης) στο οποίο μια μεταβλητή παίρνει τιμή, **η τιμή αυτή δεν αλλάζει**.

π.χ. η ερώτηση **?- X = 5, X = 7.** δίνει **false**

- Για να γίνει κατανοητό το αποτέλεσμα, θα πρέπει η ερώτηση να αναγνωσθεί με δηλωτικό τρόπο: ποιο είναι το X το οποίο ενοποιείται με το 5 και ταυτόχρονα ενοποιείται με το 7. Η προφανής απάντηση είναι ότι δεν υπάρχει τέτοιο X
- Η **εμβέλεια** των μεταβλητών περιορίζεται στο γεγονός ή στον κανόνα ή στην ερώτηση όπου εμφανίζονται **και όχι** στο σύνολο των προτάσεων.



Variables in Prolog

- An **underscore** character (`_`) is a special case of a variable called an **anonymous variable**. Δηλώνει ότι ο χρήστης δεν ενδιαφέρεται να “μάθει” την τιμή που θα αποδοθεί στο συγκεκριμένο όρισμα.
- Π.χ. αν θέλουμε να διατυπώσουμε το ερώτημα “Είναι γονέας ο john”, η Prolog ερώτηση είναι:

?- parent(john, _).

true

Το ερώτημα “Ποιανού είναι γονέας ο john” είναι:

?- parent(john, X).

X = maria

Το ερώτημα “Ποιοι είναι γονείς” είναι:

?- parent(X, _).



Example

- Έστω ότι έχουμε τα παρακάτω γεγονότα:
father(john, maria).
father(john, peter).
father(nick, stella).
- Αν προσπαθούσαμε να βρούμε τα αδέρφια, θα κάναμε μια ερώτηση όπως:
?- father(X, Y1) , father(X, Y2).
(δηλ. βρες δύο παιδιά που έχουν τον ίδιο πατέρα)
- Ενώ θα περιμέναμε να βρούμε την maria και τον peter, το σύνολο των λύσεων είναι πολυπληθέστερο:
X = john, Y1 = maria, Y2 = maria
X = john, Y1 = maria, Y2 = peter
X = john, Y1 = peter, Y2 = maria
X = john, Y1 = peter, Y2 = peter
X = nick, Y1 = stella, Y2 = stella



Example (cont.)

- Το προηγούμενο είναι ένα **κλασσικό λάθος**. Πρέπει να είμαστε όσο πιο σαφείς γίνεται, ώστε να αποφύγουμε τα Y1 και Y2 να πάρουν τις ίδιες τιμές (δηλ. ο καθένας να είναι αδερφός του εαυτού του). Επομένως θα γράφαμε:

?- father(X, Y1) , father(X, Y2) , Y1\=Y2.

και το σύνολο των λύσεων θα ήταν :

X = john, Y1 = maria, Y2 = peter

X = john, Y1 = peter, Y2 = maria

- Αν προσπαθούσαμε να χρησιμοποιήσουμε μια ανώνυμη μεταβλητή (αφού το όνομα του πατέρα δεν μας ενδιαφέρει) θα γράφαμε:

?- father(_ , Y1) , father(_ , Y2) , Y1\=Y2.

και το σύνολο των λύσεων θα ήταν :

Y1 = maria, Y2 = stella

Y1 = maria, Y2 = peter

Y1 = stella, Y2 = maria

Y1 = stella, Y2 = peter

Y1 = peter, Y2 = maria

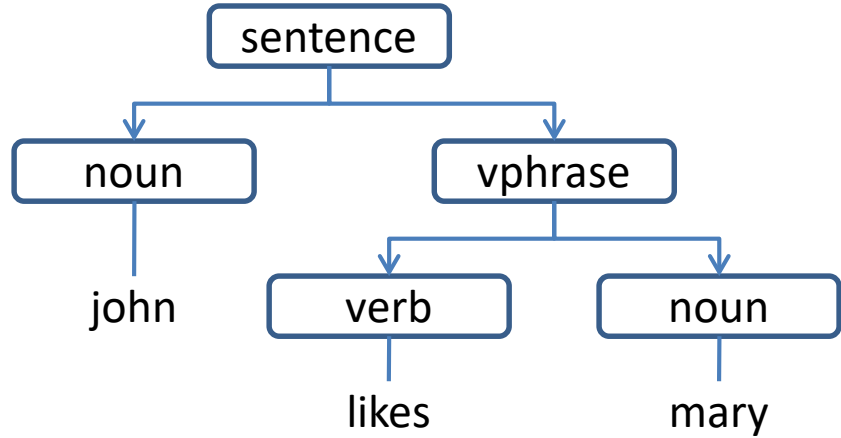
Y1 = peter, Y2 = stella

Η χρήση του τελεστή \= (διάφορο) επιβάλλει το αριστερό και δεξιό του όρισμα να μην είναι ενοποιησιμα και εξασφαλίζει ότι δεν θα προτείνει το ίδιο άτομο ως αδερφό/ή.

Αυτό όμως είναι **λάθος**, γιατί η ύπαρξη του X εξασφάλιζε ότι ο πατέρας του Y1 είναι ο ίδιος με τον πατέρα του Y2, ενώ οι ανώνυμες μεταβλητές είναι πάντα ανεξάρτητες.



Compound Terms (Σύνθετοι όροι)

- A **compound term** allows the representation of complex data with substructure (a data structure) and has the form $f(t_1, t_2, \dots, t_n)$, where f is called a **functor** (συναρτησιακό σύμβολο) and t_1, t_2, \dots, t_n are arguments.
- It sometimes helps to think of a compound term as a tree structure.

```
graph TD; sentence[sentence] --> noun1[noun]; sentence --> vphrase[vphrase]; noun1 --> john[john]; vphrase --> verb[verb]; vphrase --> noun2[noun]; verb --> likes[likes]; noun2 --> mary[mary];
```
- For example
`sentence(noun(john), vphrase(verb(likes), noun(mary)))`

Compound Terms

- Οι σύνθετοι όροι αποτελούν τη βασική δομή δεδομένων που υποστηρίζεται από την Prolog και επιτρέπουν την οργάνωση σύνθετης πληροφορίας.
- Υπάρχει το build-in κατηγορημα: **compound/1**

Άσκηση: Θέλουμε να αναπαραστήσουμε το προφίλ ενός χρήστη που περιλαμβάνει το όνομά του και επιπλέον πληροφορίες που αφορούν την εθνικότητα, την ημερομηνία γέννησης και το επάγγελμά του.



Compound Terms

Λύση

- Μια πιθανή αναπαράσταση των παραπάνω θα ήταν να χρησιμοποιηθεί ένας όρος:
info(<Nationality>, birthday(<Day>, <Month>, <Year>), job(<Job>))
- Αν ένας χρήστης έχει ελληνική υπηκοότητα, ημερομηνία γέννησης 30-6-1975 και εργάζεται σαν δάσκαλος, ο όρος που θα περιέγραφε τα παραπάνω θα ήταν:
info(greek, birthday(30,6,1975), job(teacher)).
- Το γεγονός ότι η παραπάνω πληροφορία αφορά τον χρήστη kostas, θα υλοποιούνταν στην Prolog ως:
user(kostas, info(greek, birthday(30,6,1975), job(teacher))).



The importance of correct representation

- Η επιλογή της **κατάλληλης αναπαράστασης** των γεγονότων παίζει πολύ σημαντικό ρόλο
 - καθορίζει το είδος των ερωτήσεων που μπορούν να τεθούν
 - την ευκολία με την οποία θα αποδειχθούν/απαντηθούν
- Παράδειγμα: Έστω ότι οι χρήστες ανήκουν σε μία ή περισσότερες ομάδες π.χ. admin, student, prof , και ο χρήστης George ανήκει στις ομάδες admin και student, και ο χρήστης John στις ομάδες admin και prof.



The importance of correct representation

- Μια αναπαράσταση της παραπάνω πληροφορίας (ως ιδιότητας) θα μπορούσε να είναι:

admin(george).

admin(john).

student(george).

prof(john).

- Εύκολα μπορούμε να απαντήσουμε σε ερωτήματα:
 - διαπίστωσης αν κάποιος χρήστης είναι μέλος μιας ομάδας π.χ.
?- student(george).
true
 - ποιους χρήστες περιλαμβάνει μια συγκεκριμένη ομάδα π.χ.
?- admin(User).
User = george
User = john



The importance of correct representation

- Δεν μπορούμε όμως να ρωτήσουμε σε ποιες ομάδες ανήκει ένας συγκεκριμένος χρήστης
 - Η ερώτηση **?- X(george).** ΔΕΝ είναι έγκυρη! Η χρήση των μεταβλητών στην Prolog επιτρέπεται μόνο στην θέση των ορισμάτων και όχι των ονομάτων των κατηγορημάτων.
 - Μόνο έμμεσα θα μπορούσαμε να πάρουμε απάντηση (κάνοντας 3 ερωτήσεις):
 - ?- admin(george).
true
 - ?- student(george).
true
 - ?- prof(george).
false
- Δεν μπορούμε επίσης να ρωτήσουμε αν δύο χρήστες ανήκουν στην ίδια ομάδα



The importance of correct representation

- Μια καταλληλότερη αναπαράσταση (ως ορισμός σχέσης) είναι:
belong_to(admin, george).
belong_to(admin, john).
belong_to(student, george).
belong_to(prof, john).
- Το ερώτημα “σε ποιες ομάδες ανήκει ο χρήστης george” διατυπώνεται ως:
?- belongs_to(Group, george).
Group = admin
Group = student
- Για το ερώτημα “αν δύο χρήστες ανήκουν στην ίδια ομάδα” μπορούμε να ορίσουμε τον κανόνα:
same_group(X, Y) :- belongs_to(Group, X), belongs_to(Group, Y).

όπου η χρήση της ίδιας μεταβλητής Group στους δύο όρους της σύζευξης εξασφαλίζει ότι οι χρήστες X και Y ανήκουν στην ίδια ομάδα.

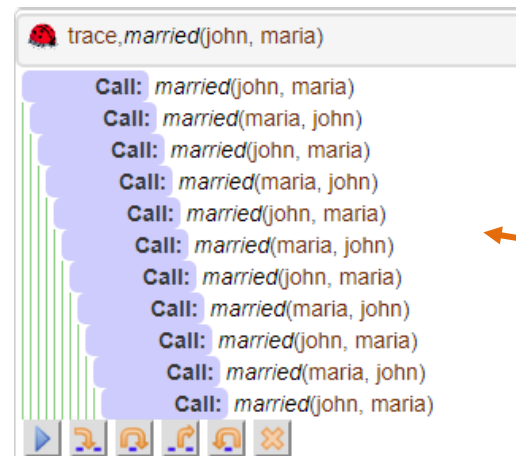


The importance of correct representation

- Ένας άλλος κίνδυνος είναι οι ατέρμονοι (άπειροι) υπολογισμοί

Παράδειγμα: Θέλουμε να εκφράσουμε ότι αν ο X είναι παντρεμένος με τον Y τότε και ο Y είναι παντρεμένος με τον X.

- Αν το εκφράσουμε με τον κανόνα: **married(X, Y) :- married(Y, X).**
- Τι θα γινόταν αν κάναμε το ερώτημα:
?- married(john, maria).



```
trace,married(john, maria)
Call: married(john, maria)
Call: married(maria, john)
Call: married(john, maria)
Call: married(maria, john)
Call: married(john, maria)
Call: married(maria, john)
Call: married(john, maria)
Call: married(maria, john)
Call: married(john, maria)
Call: married(maria, john)
Call: married(john, maria)
```

← ατέρμονος υπολογισμός

The importance of correct representation

- Μια λύση είναι να χρησιμοποιήσουμε ένα νέο κανόνα

marriedToEachOther(X, Y) :- married(X, Y).

marriedToEachOther(X, Y) :- married(Y, X).

['H marriedToEachOther(X, Y) :- married(X, Y) ; married(Y, X).]

- Οπότε τώρα κάνουμε ερωτήσεις της μορφής:

?- marriedToEachOther(john, maria).



Prolog Programming Style

- Τα σχόλια στην Prolog εισάγονται είτε με τον χαρακτήρα "%" και εκτείνονται μέχρι το τέλος της τρέχουσας γραμμής. Μεγαλύτερα σχόλια περικλείονται στους χαρακτήρες "/*" και "*/".
- Συνήθως στα σχόλια δίνεται και η περιγραφή ενός κατηγορήματος με την μορφή κατηγορήμα/τάξη, μια δηλωτική περιγραφή του κατηγορήματος και των ορισμάτων του, και μια ελεύθερη περιγραφή της χρήσης ή συμπεριφοράς του κατηγορήματος.

Για παράδειγμα:

```
% male_player/2
```

```
% male_player(TeamName, PlayerName)
```

```
% Succeeds if PlayerName is a male player of TeamName.
```

```
male_player(TeamName, PlayerName):-
```

```
    player_of(TeamName, PlayerName),
```

```
    male(PlayerName).
```



Recursion

- Η **recursion** (αναδρομή) βασίζεται στη μαθηματική επαγωγή και είναι ένα ισχυρό εργαλείο δηλωτικού προγραμματισμού
- Η Prolog δεν έχει άλλον τρόπο να εκφράσει επανάληψη εκτός της αναδρομής, η οποία αποτελεί βασική μεθοδολογία ανάπτυξης προγραμμάτων και αντικαθιστά την ανάγκη χρήσης εντολών επανάληψης.
- Τα αναδρομικά κατηγορήματα είναι σχέσεις και όχι συναρτήσεις, δηλαδή αποτιμώνται σε true ή false και δεν επιστρέφουν τιμές π.χ. παραγοντικό

```
1 factorial(0,1).  
2 factorial(N,F):-  
3     N>0,  
4     N1 is N-1,  
5     factorial(N1,F1),  
6     F is N*F1.
```



Recursion

- Πάντα οι αναδρομικοί ορισμοί αποτελούνται από μια **τερματική συνθήκη** (τερματική σχέση/κανόνας) που πρέπει να προηγείται της αναδρομικής σχέσης η οποία καλεί τον εαυτό της (δηλ. από το ειδικότερο στο γενικότερο).

Παράδειγμα

Έστω ότι έχουμε μια σειρά από γεγονότα που εκφράζουν σχέσεις γονέα-παιδιού με το `parent/2` και θέλουμε να φτιάξουμε ένα κανόνα που να ορίζει τη σχέση προγόνου-απογόνου `ancestor/2`. Το αναδρομικό κατηγορημα `ancestor(X,Y)` έχει την έννοια "ο `X` είναι πρόγονος του `Y`" δηλ. ο `X` είναι πατέρας ή παππούς ή προπάππους κτλ. του `Y`. Αυτό γίνεται με αναδρομή ως εξής:

`ancestor(X,Y) :- parent(X,Y).` ← **τερματική σχέση**
`ancestor(X,Y) :- parent(X,Z) , ancestor(Z,Y).` ← **αναδρομική σχέση**



Recursion

Παράδειγμα (συνέχεια)

ΠΡΟΣΟΧΗ: Δεν θα ήταν το ίδιο αν γράφαμε την αναδρομική σχέση
 $\text{ancestor}(X,Y) \text{ :- parent}(X,Z) , \text{ancestor}(Z,Y).$ ως εξής

$\text{ancestor}(X,Y) \text{ :- ancestor}(Z,Y) , \text{parent}(X,Z).$

ή **$\text{ancestor}(X,Y) \text{ :- ancestor}(X,Z) , \text{parent}(Z,Y).$**

γιατί ναι μεν θα έδινε σωστές απαντήσεις για ονόματα που υπάρχουν στα γεγονότα, αλλά αν ρωτούσαμε κάτι που δεν υπάρχει στα γεγονότα, η Prolog δεν θα απαντούσε false αφού θα έμπαινε σε ατέρμονο βρόχο καλώντας συνέχεια την ancestor/2.

- Βλέπουμε λοιπόν ότι η λογική της Prolog διαφέρει από τη μαθηματική Λογική. Για να δουλεύουν σωστά οι αναδρομικοί ορισμοί, το μυστικό είναι να **βάζουμε πρώτα τις ειδικότερες κλήσεις και μετά τις γενικότερες, ώστε να οδηγούμε την Prolog γρήγορα στις τερματικές σχέσεις.**



Recursion

Παράδειγμα Ακέραια διαίρεση

- Για να βρω αν ένας αριθμός X διαιρεί έναν άλλον Y υποθέτω ότι ο X διαιρεί το $Y-X$.
- Τερματική συνθήκη:
ένας αριθμός διαιρεί τον εαυτό του.

```
divides(X,X).  
divides(X,Y):-  
    X>0, Y>0,  
    D is Y-X,  
    divides(X,D).
```

Αν θέλουμε να βρούμε και πόσες φορές
ο X διαιρεί το Y (πηλίκο ακέραιας διαίρεσης):

- Για να βρω πόσες φορές ένας αριθμός X διαιρεί έναν άλλον Y υποθέτω ότι έχω βρει πόσες φορές ο X διαιρεί το $Y-X$, και προσθέτω 1.
- Τερματική συνθήκη:
ένας αριθμός διαιρεί τον εαυτό του μία φορά.

```
divides(X,X,1).  
divides(X,Y,N):-  
    X>0, Y>0,  
    D is Y-X,  
    divides(X,D,N1),  
    N is N1+1.
```



Lists

- Lists are a special class of compound term. In many cases , it's necessary to use terms with variable (and potentially infinite) number of arguments, i.e. a structure that contains a variable number of elements.
- list containing the sequence of elements a, b and c is represented by **[a, b, c]** The elements of a list are considered ordered.
- The atom **[]** represents the empty list.
- The **head** is the first element of the list. The **tail** contains all the other elements of the list, except the head, therefore it is always a list (possibly empty). The head and the tail can be separated by the symbol “|”, i.e. a list can be represented as **[H|T]** . Ο “αποκεφαλισμός” είναι η μόνη επιτρεπτή πράξη σε μια λίστα.



Lists

- Therefore **[a, b, c]** is equivalent to **[a | [b, c]]**
- The elements of a list are not necessarily of the same type.
- Compound terms can also be elements in a list
e.g. **[a, b, [4, 2,18], name(john), lang(greek)]**
- Lists are recursive structures (the stop condition of the recursive rule is the empty list).



Lists

- List management predicates
 - **member/2** επιτυγχάνει αν ο όρος του πρώτου ορίσματος αποτελεί μέλος της λίστας που εμφανίζεται στο δεύτερο όρισμα.
e.g. `?- member(5 , [2,4,5,10,1,2,2,3]).`
True
 - **length/2** is the size of a list
e.g. `?- length([a,b,c,d] , X).`
X = 4
`?- length([] , X).`
X = 0



Lists

- List management predicates
 - **reverse/2** αντιστρέφει την σειρά των στοιχείων
e.g. `?- reverse([1,2,3,4] , L).`
 $L = [4,3,2,1]$
`?- reverse([1,2,3,4] , [4,3,2,1]).`
True
 - **append/3** δημιουργία μιας νέας λίστας με συνένωση των στοιχείων από δύο άλλες λίστες.
e.g. `?- append([1,2,3], [4,5,6,7], L).`
 $L = [1,2,3,4,5,6,7]$



Lists

Example

We want to find the last element of a list

e.g. ?- lastelement([a,b,c,d] , X).

X = d

lastelement([LastOne], LastOne). % stop condition

lastelement([First | Rest], Last) :-

lastelement(Rest, Last). % ignore the first element
% and search the rest



Lists

Παράδειγμα (χρήσης τελεστών σύγκρισης)

Θέλουμε να ελέγχουμε αν μια λίστα είναι ταξινομημένη κατά αύξουσα σειρά

Π.χ. `?- sorted([13,8,11,6]).`

`false`

`?- sorted([6,8,11,13]).`

`true`

`sorted([]).` % τερματική συνθήκη (κενή λίστα)

`sorted([_]).` % τερματική συνθήκη (λίστα με ένα μόνο στοιχείο)

`sorted([X,Y | List]):-`

`X=<Y,`

`sorted([Y | List]).`



Lists

Παράδειγμα

Βρείτε το μέγιστο/ελάχιστο στοιχείο μιας λίστας αριθμών, δηλαδή δημιουργία κατηγορημάτων με την εξής συμπεριφορά:

?- list_max([8,4,5,12,1,2,2,3], Max).

Max = 12

?- list_min([8,4,5,12,1,2,2,3], Min).

Min = 1

Λύση

- Υλοποίηση με χρήση αναδρομής (παρουσιάζονται 3 τρόποι)



Lists

1^{ος} τρόπος

Δηλωτική περιγραφή:

list_max1([X], X).

Σε μια λίστα με ένα στοιχείο, μέγιστο είναι αυτό το στοιχείο.

**list_max1([H | T], H):-
list_max1(T, MT),
H > MT.**

Αν η κεφαλή της λίστας είναι όρος μεγαλύτερος από το μέγιστο στοιχείο της ουράς της, τότε η κεφαλή είναι το μέγιστο στοιχείο της λίστας.

**list_max1([H | T], MT):-
list_max1(T, MT),
H <= MT.**

Αν το μέγιστο στοιχείο της ουράς της λίστας είναι μεγαλύτερο από την κεφαλή της τότε αυτό είναι το μέγιστο στοιχείο.

- Η υλοποίηση αυτή παρουσιάζει προβλήματα απόδοσης (π.χ. στη περίπτωση που η λίστα είναι ταξινομημένη)



Lists

2^{ος} τρόπος Για να βελτιώσουμε την απόδοση μπορούμε να χρησιμοποιήσουμε την κλασική τεχνική της αποθήκευσης ενός προσωρινού μεγίστου (μεγαλύτερου στοιχείο μέχρι στιγμής)

list_max2([],MSF,MSF).

Όταν εξαντληθούν όλα τα στοιχεία της λίστας, το προσωρινό μέγιστο είναι το μέγιστο στοιχείο.

**list_max2([H | T],MSF,Max):-
H > MSF,
list_max2(T,H,Max).**

Αν η κεφαλή της λίστας είναι μεγαλύτερη από το προσωρινό μέγιστο, τότε αναζητείτε το μέγιστο της λίστας στην ουρά, με προσωρινό μέγιστο την κεφαλή.

**list_max2([H | T],MSF,Max):-
H <= MSF,
list_max2(T,MSF,Max).**

Αν η κεφαλή της λίστας είναι μικρότερη ή ίση από το προσωρινό μέγιστο, τότε αναζητείτε το μέγιστο της λίστας στην ουρά, με το ίδιο προσωρινό μέγιστο.

- Η μεταβλητή Max παραμένει αναλλοίωτη στο δεύτερο και τρίτο κανόνα και δεσμεύεται μόνο στο πρώτο γεγονός.



Lists

3^{ος} τρόπος Λύση με αποδοτικό κατηγορημα δύο ορισμάτων:

list_max3([M],M).

Σε μια λίστα με ένα στοιχείο, μέγιστο είναι αυτό το στοιχείο.

list_max3([H1,H2 | T],Max):-

H1 > H2,

list_max3([H1 | T],Max).

Αλλιώς, αν το πρώτο στοιχείο της λίστας είναι μεγαλύτερο από το δεύτερο, τότε το μέγιστο στοιχείο της είναι το μέγιστο της λίστας που αποτελείται από το πρώτο στοιχείο και τα υπόλοιπα στοιχεία της λίστας.

list_max3([H1,H2 | T],Max):-

H1 =< H2,

list_max3([H2 | T],Max).

Αλλιώς, αν το πρώτο στοιχείο της λίστας είναι μικρότερο ή ίσο από το δεύτερο, τότε το μέγιστο στοιχείο της είναι το μέγιστο της λίστας που αποτελείται από το δεύτερο στοιχείο και τα υπόλοιπα στοιχεία της λίστας.



Lists

Άσκηση: Δεδομένης μιας λίστας ακεραίων αριθμών, φτιάξτε πρόγραμμα στην Prolog που να υπολογίζει το άθροισμά τους.

Λύση

- Θα ορίσουμε ένα αναδρομικό κατηγορημα $\text{sum}/2$ της μορφής
 $\text{sum}(L,S)$

όπου L είναι η δοσμένη λίστα και το S θα χρησιμοποιηθεί για το αποτέλεσμα της πρόσθεσης.

- Για παράδειγμα, μια ερώτηση

?- $\text{sum}([2,7,3],Q)$

θα επιστρέφει την απάντηση: $Q = 12$



Lists

Λύση (συνέχεια)

- Το πρόγραμμα αποτελείται από δύο προτάσεις
 $\text{sum}([],0).$
 $\text{sum}([A|L],S) :- \text{sum}(L,S1) , S \text{ is } A+S1.$
- Το “ $S \text{ is } A+S1$ ” είναι ο τρόπος για να πούμε στην Prolog ότι “το S είναι το αποτέλεσμα της πρόσθεσης $A+S1$ ”.
- Το $[A|L]$ συμβολίζει μια λίστα με πρώτο στοιχείο το A και η υπόλοιπη λίστα είναι L
- Η σημασία των δύο προτάσεων είναι:
 - Το άθροισμα μιας κενής λίστας είναι 0
 - Το άθροισμα την λίστας $[A|L]$ είναι το αποτέλεσμα της πρόσθεσης του A με το άθροισμα της L (αναδρομικά)



Operators (Τελεστές)

- Η αναπαράσταση αριθμητικών εκφράσεων στις περισσότερες γλώσσες προγραμματισμού ακολουθεί τη συνήθη μορφή, η οποία χρησιμοποιεί ενθεματικούς (infix) τελεστές.
- Χαρακτηριστικά ενός τελεστή f
 - ο τύπος του:
 - **προθεματικός (prefix)** i.e. fx or fy (συχνότερος στην Prolog)
 - επιθεματικός (postfix) i.e. xf or yf
 - ενθεματικός (infix) i.e. xfx , xfy or yfx
 - η προτεραιότητά του (βλέπε built-in κατηγορημα **current_op/3**)
 - Π.χ. η προτεραιότητα του $+$ είναι 500 ενώ του $*$ είναι 400, δηλαδή η έκφραση $2 + 3 * 5$ γράφεται ως $+(2, *(3,5))$
 - ο τελεστής με την υψηλότερη τιμή προτεραιότητας είναι ο *principal functor* και θα εκτελεστεί τελευταίος.



Operators

Παράδειγμα (Η προτεραιότητα εκφράζεται από 0 μέχρι 1200. Μερικοί τελεστές μπορεί να έχουν πάνω από ένα τύπο.)

?- current_op(Precedence, Type, +).

Precedence=200

Type=fy ;

Precedence=500

Type=yfx

?- current_op(Precedence, Type, *).

Precedence=400

Type=yfx

?- display(2 + 3 + 4).

+(+(2,3),4)

true



Operators

- Υπάρχει η δυνατότητα ορισμού νέων τελεστών με την **op/3**
:- op(Προτεραιότητα, Τύπος, Τελεστής)
(μικρότερος αριθμός δηλώνει υψηλότερη προτεραιότητα)

Παράδειγμα: Ορισμός του παραγοντικού ως τελεστής

:-op(500, xfx, !).

N ! F :- factorial(N, F).

Ερώτηση:

?- 4 ! Result.

Result=24



Evaluating numerical expressions

ΠΡΟΣΟΧΗ: οι όροι που αναπαριστούν αριθμητικές εκφράσεις **ΔΕΝ** υπολογίζονται αυτόματα στην Prolog όπως γίνεται στις άλλες γλώσσες !

Παράδειγμα: Η παρακάτω ερώτηση αποτυγχάνει γιατί το $5 + 5$ είναι ο όρος $+(5,5)$ και το 10 μια σταθερά, άρα στην Prolog **δεν** μπορούν να ενοποιηθούν.

?- $10 = 5 + 5$.

false

- Το σύμβολο $=$ είναι δεσμευμένο για την ενοποίηση δύο όρων και **όχι** για την ισότητα ή την ανάθεση τιμών που προκύπτουν από τον υπολογισμό αριθμητικών παραστάσεων.
- Για αυτόν τον σκοπό υπάρχει το ενσωματωμένο κατηγορημα **is/2** (ενθεματικός τελεστής xfx).



Evaluating numerical expressions

Examples

?- X is 5+2.

X=7

?- X is ((2+4)*8) mod 20.

X=8

?- 5.5 is 11/2.

true

?- 5 is 11//2.

true

Operators	Meaning
+, -, *, / , ^	πρόσθεση, αφαίρεση, πολλαπλασιασμός, διαίρεση, ύψωση σε δύναμη
// , mod	ακέραια διαίρεση, υπόλοιπο ακέραιας διαίρεσης
>, <, >=, <=, :=, \=	προσοχή στο >= και στο <= αριθμητική ισότητα, ανισότητα
\=	ανισότητα μεταξύ όρων

- Υπάρχει ένας σημαντικός περιορισμός στη χρήση του is/2: το δεξιό μέλος του κατηγορήματος πρέπει να είναι πλήρως ορισμένο (χωρίς ελεύθερες μεταβλητές). Δεν μπορεί δηλαδή το is/2 να λειτουργεί ως επιλυτής εξισώσεων της μορφής 7 is X+2.



Evaluating numerical expressions

Παραδείγματα

Ποιο είναι το αποτέλεσμα των παρακάτω ερωτήσεων και γιατί;

?- $5 = \backslash 6$.

?- $5 \backslash 6$.

?- $X \backslash Y$.

?- $X = \backslash Y$.

?- $5+2 = \backslash 3+4$.

?- $5+2 \backslash 3+4$.



Evaluating numerical expressions

Παρατηρήσεις:

- Η μεταβλητή στον λογικό προγραμματισμό έχει διαφορετική σημασία από ότι στον διαδικαστικό προγραμματισμό (όπου δηλώνει μία θέση μνήμης η οποία αποθηκεύει μία τιμή που μπορεί να αλλάξει).
- Για παράδειγμα στις διαδικαστικές γλώσσες είναι σύνηθες να γράφουμε $i=i+1$, εννοώντας “πάρε την τιμή που βρίσκεται στη θέση μνήμης i , πρόσθεσε 1 και αποθήκευσε τη νέα τιμή στην ίδια θέση μνήμης.
- Στην Prolog, μια μεταβλητή αναπαριστά μία και μόνο τιμή. Έτσι εκφράσεις της μορφής **X is $X+1$** δεν είναι “λογικά” σωστές (δεν μπορεί να υπάρχει αριθμητική τιμή η οποία είναι ίση με τον εαυτό της αυξημένο κατά ένα)



Evaluating numerical expressions

Παράδειγμα

Έστω ότι αναπαριστούμε data expressions με τα παρακάτω δύο γεγονότα:

data(2).

data(3*4).

Ορίζουμε τον κανόνα ο οποίος υπολογίζει το άθροισμα κάθε ζεύγους από data expressions στην knowledge base:

calc(N):-

data(N1),

data(N2),

N is N1 + N2.

Βρίσκουμε τα αθροίσματα όλων των συνδυασμών των data expressions με την παρακάτω ερώτηση (κάνουμε backtracking για εναλλακτικές λύσεις):

?- calc(N).

N=4 ;

N=14 ;

N=14 ;

N=24

Τι απάντηση θα πάρουμε αν αλλάξουμε τον κανόνα και προσθέσουμε: $N1 \neq N2$

?- calc(N).

N=14 ;

N=14

Τι θα κάναμε για να πάρουμε μοναδική τιμή στην απάντηση?

?- distinct([N], (calc(N))).

N=14



Disjunction operator (τελεστής διάζευξης)

- In this rule

$a(X) :- b(X) ; c(X).$

the disjunction operator “;” has the following meaning:

$a(X)$ is true, if $b(X)$ is true **OR** $c(X)$ is true.

- The precedence of the disjunction operator “;” is lower than the conjunction operator (τελεστή σύζευξης) “,”. An expression like this:

$a(X) , b(X) ; c(X) , d(X)$

is equivalent with:

$(a(X) , b(X)) ; (c(X) , d(X))$



Disjunction operator (τελεστής διάζευξης)

- Παρόλο που δίνει σημαντική ευελιξία στην ανάπτυξη προγραμμάτων, η χρήση του οδηγεί σε πιο δυσανάγνωστα προγράμματα.
- Είναι προτιμότερο οι εναλλακτικές περιπτώσεις να υλοποιούνται με διαφορετικούς κανόνες. Π.χ.

```
celebrity(X):-  
  rich(X),  
  famous(X);  
  actor(X);  
  tvpersonality(X).
```



```
celebrity(X):-  
  rich(X),  
  famous(X).  
  
celebrity(X):- actor(X).  
  
celebrity(X):- tvpersonality(X).
```

- Οι δύο μορφές είναι ισοδύναμες (παράγουν τις ίδιες λύσεις)



Negation (Άρνηση)

- **Σημαντική διαφορά** της Prolog από την κλασική κατηγορηματική λογική: η σημασία που αποδίδεται στην άρνηση, δηλαδή στην απόδειξη των αρνητικών προτάσεων.
- Η Prolog υιοθετεί την “**υπόθεση του κλειστού κόσμου**” (**closed world assumption**), που διατυπώνεται ως: “**ότι δεν είναι δυνατό να αποδειχθεί αληθές, θεωρείται ψευδές**”.
- Αν μια πρόταση δεν μπορεί να αποδειχθεί βάσει των λογικών προτάσεων του προγράμματος (γεγονότα και κανόνες), τότε αυτόματα συνάγεται το συμπέρασμα ότι είναι ψευδής.
- Για την άρνηση χρησιμοποιούμε τον τελεστή “**\+**” ή το κατηγορημα **not/1**



Negation (Άρνηση)

Example : **man(peter).**
 man(jimmy).
 woman(helen).

?- \+ man(peter).
 false

?- \+ man(jenny).
 true

?- \+ woman(peter).
 true

?- \+ woman(jenny).
 true

- In other words, the operator **\+ (not)** in reality has the meaning of “unprovable”.



Negation (Άρνηση)

Example : **male(X):- \+ female(X).**

male(john).

male(kostas).

male(X) :- not(female(X)).

female(maria).



What will happen if we reverse the rule?
i.e. **female(X) :- not(male(X)).**

?- female(elen).

false

?- male(elen).

true

- not/1 : True if goal cannot be proven



Observation

- Οι λογικοί τελεστές στην Prolog δεν συμπεριφέρονται απόλυτα αντιμεταθετικά, όπως έχουμε συνηθίσει στη λογική.
Π.χ. `A AND B` στην Prolog σημαίνει να εξεταστεί πρώτα το `A` και μετά το `B`.

`man(peter).`
`man(jimmy).`
`woman(helen).`

The order sometimes affects the set of solutions! E.g.

?- `man(X); woman(X).`
X = peter ;
X = jimmy ;
X = helen ;

?- `man(X) , \+ woman(X).`
X = peter ;
X = jimmy ;

?- `woman(X); man(X).`
X = helen ;
X = peter ;
X = jimmy ;

?- `\+ woman(X) , man(X).`
false

Why this happens?

(more details later on the execution mechanism)



Substitution and Unification

- Η ύπαρξη μεταβλητών κάνει απαραίτητη την χρήση των εννοιών της **αντικατάστασης** (substitution) και της **ενοποίησης** (unification).
- Η “**αντικατάσταση**” αφορά στην αντικατάσταση των μεταβλητών που εμφανίζονται σε έναν τύπο από κάποιους όρους. Μια αντικατάσταση αναπαρίσταται με το σύνολο $\{X_i/t_i\}$ όπου X_i η μεταβλητή που θα αντικατασταθεί και t_i ο όρος με τον οποίο θα αντικατασταθεί.
- **Ενοποίηση** είναι η διαδικασία κατά την οποία δύο εκφράσεις γίνονται συντακτικά όμοιες με τη χρήση αντικαταστάσεων. Π.χ. οι προτάσεις:

επάγγελμα(γιάννης, καθηγητής, X)

επάγγελμα(γιάννης, Y, λυκείου)

ενοποιούνται με την αντικατάσταση $\{X/\lambda\kappa\epsilon\acute{\iota}\omicron\upsilon, Y/\kappa\alpha\theta\eta\gamma\eta\tau\acute{\eta}\varsigma\}$.



Unification (Ενοποίηση)

- Η ενοποίηση ανάμεσα σε δύο εκφράσεις πραγματοποιείται με τον ακόλουθο αναδρομικό αλγόριθμο:
 1. Δύο σταθερές ενοποιούνται αν και μόνο αν είναι ίδιες.
 2. Μια μεταβλητή ενοποιείται με οποιονδήποτε όρο, εισάγοντας μια νέα αντικατάσταση.
 3. Δύο συναρτησιακοί όροι ενοποιούνται αν έχουν το ίδιο συναρτησιακό σύμβολο, την ίδια τάξη και αν κάθε όρισμα του πρώτου μπορεί να ενοποιηθεί με το αντίστοιχο σε θέση όρισμα του δεύτερου όρου.
 4. Δύο ατομικοί τύποι ενοποιούνται αν έχουν το ίδιο κατηγορήμα, την ίδια τάξη και αν κάθε όρισμα του πρώτου μπορεί να ενοποιηθεί με το αντίστοιχο σε θέση όρισμα του δεύτερου ατομικού τύπου.



Unification (Ενοποίηση)

- Ο παραπάνω αλγόριθμος, αν και είναι αποδοτικός και χρησιμοποιείται σχεδόν σε όλα τα συστήματα λογικού προγραμματισμού που βασίζονται στην κατηγορηματική λογική πρώτης τάξης, όπως η Prolog, **δεν είναι ορθός !**
- Αυτό φαίνεται στις περιπτώσεις όπου η προς ενοποίηση μεταβλητή εμφανίζεται στον όρο με τον οποίο θα ενοποιηθεί. Π.χ. η ενοποίηση $X = \text{father}(X)$ θα δώσει $X = \text{father}(\text{father}(\text{father}(\dots)))$, δηλαδή δημιουργεί απειρία αντικαταστάσεων.
- Η αποφυγή τέτοιων καταστάσεων απαιτεί τη χρήση ενός αλγορίθμου με μεγάλο υπολογιστικό κόστος και για αυτό συνήθως δεν χρησιμοποιείται στις υλοποιήσεις της Prolog και η ορθότητα των προγραμμάτων επαφίεται στον προγραμματιστή, κερδίζοντας έτσι σε αποδοτικότητα.



Unification (Ενοποίηση)

- Συμπερασματικά, ο μηχανισμός απόδοσης τιμών στις μεταβλητές ώστε η ερώτηση να γίνει συντακτικά όμοια με το γεγονός, ονομάζεται **Ενοποίηση** (δηλ. ο τρόπος που η Prolog ταιριάζει δύο όρους).
- Η ιδέα είναι παρόμοια με την ενοποίηση στην λογική: έχουμε δύο όρους και θέλουμε να δούμε αν μπορούν να αναπαραστήσουν την ίδια δομή.
- Γενικός κανόνας ενοποίησης: για να είναι δύο όροι ενοποιήσιμοι θα πρέπει να έχουν το ίδιο συναρτησιακό σύμβολο, τον ίδιο αριθμό ορισμάτων και τα ορίσματά τους να είναι ενοποιήσιμα.
- Οι έννοιες της αντικατάστασης και της ενοποίησης είναι σημαντικές για την εφαρμογή των κανόνων συμπερασμού της κατηγορηματικής λογικής και την εξαγωγή νέας γνώσης.
- Ο τελεστής “=” χρησιμοποιείται για να ενοποιήσει δύο όρους.

Παραδείγματα

?- a = a. % Two identical atoms unify
true.

?- a = b. % Atoms don't unify if they aren't identical
false.



Unification (Ενοποίηση)

Παραδείγματα (συνέχεια)

?- **X = a.** % Unification instantiates a variable to an atom

X=a.

?- **X = Y.** % Unification binds two differently named variables to a single, unique variable name

?- **foo(a,b) = foo(a,b).** % Two identical complex terms unify

true.

?- **foo(a,b) = foo(X,Y).** % Two complex terms unify if they are of the same arity,

X=a, % have the same principal functor and their arguments unify

Y=b.

?- **foo(a,Y) = foo(X,b).** % Instantiation of variables may occur in either of the terms to be unified

Y=b,

X=a.

?- **foo(a,b) = foo(X,X).** % No unification because foo(X,X) must have the same 1st and 2nd arguments

false.

?- **2*3+4 = X+Y.** % The term 2*3+4 has principal functor + and therefore unifies with X+Y ,

X=2*3, % with X instantiated to 2*3 and Y instantiated to 4

Y=4.



Unification (Ενοποίηση)

Παραδείγματα (συνέχεια)

?- [a,b,c] = [X,Y,Z]. % Lists unify just like other terms

X=a,

Y=b,

Z=c.

?- [a,b,c] = [X|Y]. % Unification using the '|' symbol can be used to find the head element, X, and tail list, Y, of a list

X=a,

Y=[b,c].

?- [a,b,c] = [X,Y|Z]. % Unification on lists doesn't have to be restricted to finding the first head element

X=a,

% In this case we find the 1st and 2nd elements (X and Y) and then the tail list (Z)

Y=b,

Z=[c].

?- [a,b,c] = [X,Y,Z|T]. % The first 3 elements are unified with variables X, Y and Z, leaving T as an empty list []

X=a,

Y=b,

Z=c,

T=[].



Execution Mechanism

- Η εκτέλεση ενός προγράμματος στην Prolog ξεκινά με μια **ερώτηση** προς απόδειξη που υποβάλλει ο χρήστης. Η απάντηση στην ερώτηση είναι το αποτέλεσμα του προγράμματος.
- Για την απόδειξη μιας πρότασης ο μηχανισμός εκτέλεσης προσπαθεί να την “ταιριάξει” με ένα κατηγορημα, δηλαδή με ένα γεγονός ή την κεφαλή ενός κανόνα.
 - Στην απλούστερη περίπτωση, αν η ερώτηση ενοποιείται με ένα από τα γεγονότα του προγράμματος, τότε αποδεικνύεται η αλήθεια της χωρίς άλλη αναζήτηση.
- Αν ο υποστόχος (subgoal) ενοποιείται με περισσότερες λογικές προτάσεις τότε επιλέγεται η πρώτη από αυτές. Επίσης καταχωρούνται και οι εναλλακτικές επιλογές, δηλαδή η ενοποίηση του υποστόχου με την δεύτερη, τρίτη πρόταση κλπ., επειδή αυτές οι εναλλακτικές μπορεί να οδηγήσουν σε διαφορετικές αποδείξεις του ιδίου υποστόχου.
- Σε περίπτωση που ζητηθεί μια εναλλακτική απάντηση/λύση (π.χ. αν ο χρήστης πληκτρολογήσει τον χαρακτήρα ‘;’) τότε ενεργοποιείται ο **μηχανισμός οπισθοδρόμησης** (backtracking) για να δώσει την επόμενη (εναλλακτική) λύση.



The algorithm of the Execution Mechanism

Ο μηχανισμός εκτέλεσης της Prolog είναι ένας αλγόριθμος που διαχειρίζεται τις κλήσεις μιας ερώτησης ως μία στοίβα (stack) και εκτελείται έως ότου αδειάσει η στοίβα, οπότε και τερματίζει με επιτυχία. Συγκεκριμένα:

1. Μέχρις ότου δεν υπάρχουν άλλες κλήσεις στην ερώτηση ?- a_1, a_2, \dots, a_n , επιλέγονται με τη σειρά, από αριστερά προς τα δεξιά, οι κλήσεις a_i της ερώτησης.

A. Για κάθε κλήση a_i της ερώτησης, ο μηχανισμός εκτέλεσης αναλαμβάνει να βρει μια πρόταση του προγράμματος της οποίας η κεφαλή να ενοποιείται με την επιλεγμένη κλήση.

i) Αν η κλήση a_i ενοποιείται με ένα από τα γεγονότα του προγράμματος, τότε αυτή ικανοποιείται και απομακρύνεται από την ερώτηση. Αυτό υποδηλώνει ότι η αλήθεια του ερωτήματος a_i αποδεικνύεται άμεσα.

ii) Αν η κλήση a_i ενοποιείται με κάποιον κανόνα, τότε αυτή απομακρύνεται από την ερώτηση και την θέση της παίρνει το σώμα του κανόνα αυτού. Αυτό υποδηλώνει ότι για την ικανοποίηση της αρχικής κλήσης είναι απαραίτητη η ικανοποίηση των κλήσεων του σώματος του κανόνα που την αντικατέστησε.



The algorithm of the Execution Mechanism

iii) Αν υπάρχουν περισσότερες της μιας προτάσεις με τις οποίες μπορεί εν δυνάμει να ενοποιηθεί η κλήση a_i , τότε ενοποιείται με την πρώτη από αυτές. Το σημείο αυτό του προγράμματος σημειώνεται ως σημείο οπισθοδρόμησης, προστίθεται σε μία στοίβα σημείων οπισθοδρόμησης, και αντιπροσωπεύει πιθανές εναλλακτικές “απαντήσεις” στην κλήση.

B. Αν η κλήση a_i δεν μπορεί να ενοποιηθεί με καμία πρόταση (αποτυχία), τότε ενεργοποιείται ο μηχανισμός οπισθοδρόμησης (backtracking)

i) Αν η στοίβα σημείων οπισθοδρόμησης δεν είναι κενή, τότε ο μηχανισμός οπισθοδρόμησης (α) επιστρέφει την εκτέλεση στο πιο πρόσφατο σημείο οπισθοδρόμησης a_j με $j < i$ (το πρώτο στοιχείο της στοίβας σημείων οπισθοδρόμησης), (β) ακυρώνει τα υπολογιστικά βήματα (ενοποιήσεις, αντικαταστάσεις - διαγραφές κλήσεων) μεταξύ του σημείου οπισθοδρόμησης και του σημείου αποτυχίας και (γ) αναζητά στις επόμενες προτάσεις, κάποια που να μπορεί να ενοποιηθεί με την κλήση a_i .

ii) Αν η στοίβα σημείων οπισθοδρόμησης είναι κενή, τότε τερματίζει η εκτέλεση του προγράμματος με αποτυχία (fail). Εκτυπώνετε false, γίνεται έξοδος από το βρόχο και μετάβαση στο βήμα 3.



The algorithm of the Execution Mechanism

2. Όταν η στοίβα των κλήσεων μείνει κενή, τότε η ερώτηση θεωρείται ότι απαντήθηκε επιτυχώς.

A. Αν υπάρχουν μεταβλητές στην αρχική κλήση, τότε επιστρέφονται οι τιμές των μεταβλητών οι οποίες προέκυψαν από τις ενοποιήσεις των κλήσεων στα βήματα 1.A.i και 1.A.ii, παραπάνω.

i) Αν η στοίβα των σημείων οπισθοδρόμησης δεν είναι κενή, τότε δίπλα από την απάντηση η Prolog περιμένει την είσοδο κάποιου χαρακτήρα από τον χρήστη.

a. Αν δοθεί ο χαρακτήρας ';', τότε ενεργοποιείται ο μηχανισμός οπισθοδρόμησης και ζητείται μία εναλλακτική απάντηση στο αρχικό ερώτημα, δηλαδή μία εναλλακτική ανάθεση τιμών στις μεταβλητές του ερωτήματος, για τις οποίες η αρχική ερώτηση επίσης αληθεύει. Αυτό προκαλεί την επιστροφή της εκτέλεσης στο βήμα 1.B.

b. Αν δοθεί ο χαρακτήρας enter, τότε γίνεται μετάβαση στο βήμα 3.

B. Αν δεν υπάρχουν μεταβλητές στην αρχική κλήση, επιστρέφεται true και γίνεται μετάβαση στο βήμα 3.

3. Η Prolog τερματίζει και επιστρέφει στο προτρεπτικό σήμα "?-"



The Execution Mechanism (example)

- Η οπισθοδρόμηση στην Prolog γίνεται πάντα στο πιο κοντινό “χρονικά” σημείο οπισθοδρόμησης, κάτι που οφείλεται στην χρήση στοίβας.

Παράδειγμα

Έστω το απλό πρόγραμμα που αποτελείται από δύο γεγονότα:

$p(1).$

$p(2).$

και το ερώτημα:

?- $p(X), p(Y).$ % κλήση 0

Η πρώτη από αριστερά κλήση $p(X)$ ενοποιείται με το 1^ο γεγονός με την ανάθεση $\{X=1\}$ και ταυτόχρονα η *κλήση 0* τοποθετείται στην στοίβα των σημείων οπισθοδρόμησης (αφού υπάρχει και δεύτερο γεγονός με το οποίο μπορεί να ενοποιηθεί η $p(X)$). Η νέα κλήση είναι:

?- $p(Y).$ % κλήση 1



The Execution Mechanism (example)

Παράδειγμα (συνέχεια)

Η μοναδική πλέον κλήση $p(Y)$ επίσης ενοποιείται με το 1^ο γεγονός με την ανάθεση $\{Y=1\}$ και ταυτόχρονα η *κλήση 1* τοποθετείται στην κορυφή της στοίβας των σημείων οπισθοδρόμησης, πάνω από την *κλήση 0* (αφού υπάρχει και δεύτερο γεγονός με το οποίο μπορεί να ενοποιηθεί η $p(Y)$). Πλέον έχει μείνει η κενή κλήση, συνεπώς επιστρέφεται το αποτέλεσμα:

$$X=1, Y=1$$

Αν ζητηθεί εναλλακτική λύση, ο μηχανισμός οπισθοδρόμησης θα αφαιρέσει από την στοίβα σημείων οπισθοδρόμησης την κορυφή της στοίβας, δηλ. την *κλήση 1*, θα ακυρώσει την ανάθεση τιμής στην μεταβλητή Y (μόνο), και θα προσπαθήσει να ενοποιήσει την *κλήση 1* με άλλη πρόταση, δηλαδή με το 2^ο γεγονός και την ανάθεση $\{Y=2\}$. Επιστρέφεται το αποτέλεσμα:

$$X=1, Y=2$$

Καθώς δεν υπάρχει άλλο γεγονός του κατηγορήματος $p/1$, η *κλήση 1* δεν τοποθετείται ξανά στην στοίβα. Έτσι η στοίβα έχει πλέον μόνο την *κλήση 0*.



The Execution Mechanism (example)

Παράδειγμα (συνέχεια)

Αν ζητηθεί και άλλη εναλλακτική λύση, ο μηχανισμός οπισθοδρόμησης θα αφαιρέσει από την στοίβα σημείων οπισθοδρόμησης το στοιχείο στην κορυφή της στοίβας, δηλαδή την *κλήση 0*, θα ακυρώσει την ανάθεση τιμής και στις δύο μεταβλητές X και Y , και θα προσπαθήσει να ενοποιήσει την πιο αριστερή κλήση $p(X)$ της αρχικής κλήσης με άλλη πρόταση, κάτι που πραγματοποιείται με την ενοποίηση με το 2^ο γεγονός και την ανάθεση $\{X=2\}$. Στη συνέχεια επαναλαμβάνεται εκ νέου η κλήση:

?- $p(Y)$. % *κλήση 2*

Η *κλήση 0* δεν τοποθετείται εκ νέου στην στοίβα, καθώς δεν υπάρχουν εναλλακτικές λύσεις για την $p(X)$. Στη συνέχεια, η *κλήση 2* ενοποιείται με το 1^ο γεγονός και την ανάθεση $\{Y=1\}$, τοποθετείται στην στοίβα σημείων οπισθοδρόμησης, και επιστρέφεται το αποτέλεσμα:

$X=2, Y=1$



The Execution Mechanism (example)

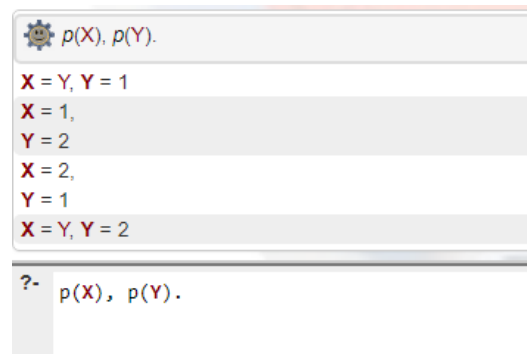
Παράδειγμα (συνέχεια)

Αν ζητηθεί εκ νέου οπισθοδρόμηση, αφαιρείται η *κλήση 2* από την στοίβα, ακυρώνεται η ανάθεση τιμής στην μεταβλητή Y , ενοποιείται η κλήση $p(Y)$ με το 2^ο γεγονός και την ανάθεση $\{Y=2\}$, και επιστρέφεται το αποτέλεσμα:

$$X=2, Y=2$$

Η *κλήση 2* δεν τοποθετείται ξανά στην στοίβα, καθώς δεν υπάρχουν εναλλακτικές λύσεις για την $p(Y)$.

Έτσι, αν ζητηθεί πάλι εναλλακτική λύση, καθώς η στοίβα σημείων οπισθοδρόμησης είναι πλέον κενή, θα επιστραφεί απάντηση **false**.



The Execution Mechanism

- Η διαδικασία της εκτέλεσης ερωτημάτων στην Prolog μπορεί να αναπαρασταθεί οπτικά με ένα **δένδρο εκτέλεσης** το οποίο πολλές φορές αναφέρεται και ως **δένδρο αναζήτησης**.
- Ο κόμβος-ρίζα είναι η αρχική κλήση-ερώτηση που θέτει ο χρήστης στην Prolog. Κάθε άλλος κόμβος αντιπροσωπεύει μία κλήση. Οι ακμές προέρχονται από την διαδικασία ενοποίησης.
- Μπορούμε να θεωρήσουμε ότι προκειμένου να βρεθεί μία απάντηση στο ερώτημα, εκτελείται αναζήτηση στον χώρο των καταστάσεων της εκτέλεσης, στον οποίο υπάρχουν δύο διαφορετικές τερματικές καταστάσεις
 - επιτυχείς και άρα αποτελούν απάντηση στο ερώτημα
 - αποτυχημένες



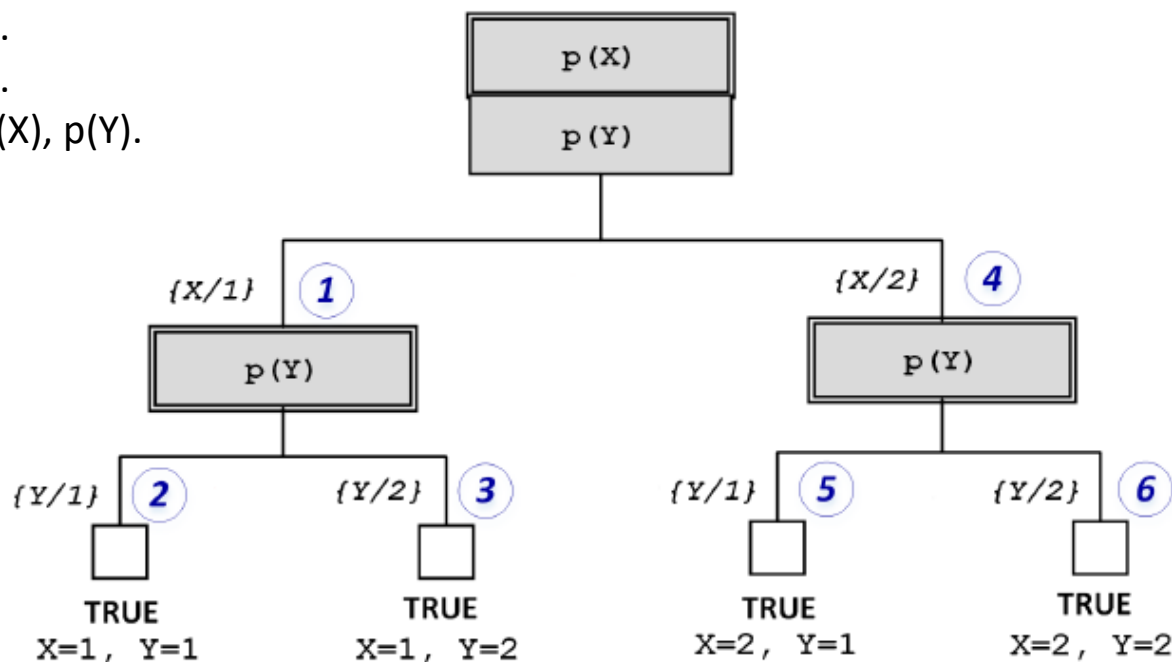
The Execution Mechanism

- Στο δέντρο εκτέλεσης του προηγούμενου παραδείγματος φαίνεται η διαδικασία της κατά βάθος αναζήτησης (depth-first search) δηλ. όταν ζητείται εναλλακτική λύση, ο μηχανισμός οπισθοδρόμησης επιστρέφει στην πιο κοντινή “διχάλα” του δένδρου, κινούμενος από κάτω-προς-τα-πάνω.

$p(1).$

$p(2).$

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



trace, p(X), p(Y)

```

Call: p(_3732)
Exit: p(1)
Call: p(_3728)
Exit: p(1)
X = Y, Y = 1
Redo: p(_3728)
Exit: p(2)
X = 1,
Y = 2
Redo: p(_3732)
Exit: p(2)
Call: p(_3728)
Exit: p(1)
X = 2,
Y = 1
Redo: p(_3728)
Exit: p(2)
X = Y, Y = 2
    
```

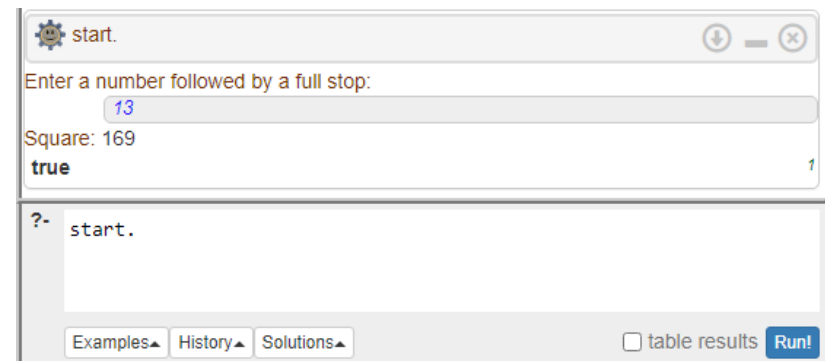
?- trace, p(X), p(Y)



Input/Output

- Υπάρχουν ενσωματωμένες διαδικασίες (built-in), οι οποίες διαβάζουν όρους ή χαρακτήρες από το προκαθορισμένο κανάλι εισόδου (πληκτρολόγιο/αρχείο) και επιστρέφουν το αποτέλεσμα στο προκαθορισμένο κανάλι εξόδου (οθόνη/αρχείο).
- Τα κατηγορήματα `read/1` και `write/1` διαβάζουν και γράφουν αντίστοιχα στο επιλεγμένο κανάλι εισόδου/εξόδου.

```
1 start :-  
2   write('Enter a number followed by a full stop: '),  
3   read(Number),  
4   Square is Number * Number,  
5   write('Square: '),  
6   write(Square).
```



Auto-Executable Goals

- Κάθε κανόνας στην Prolog γράφεται στην μορφή **Head :- Body**.
- Ένα γεγονός είναι μια ειδική περίπτωση του κανόνα, δηλαδή **Head :- true**. και στην πράξη γράφουμε απλώς **Head**.
- Στις **άμεσα-εκτελέσιμες εντολές** (auto-executable goals) λείπει το Head και γράφουμε απλά **:- Body**. Ότι βρίσκεται στο **Body** εκτελείται αυτόματα. Π.χ. χρήση του write/1 και του nl/0
:- write('NOW LOADING PROGRAM'), nl.
:
:- write('Ready to answer your questions'), nl.
- Φυσικά, η χρήση των auto-executable goals δεν περιορίζεται μόνο στην εμφάνιση μηνυμάτων.



File Input/Output

- Για την είσοδο/έξοδο όρων και χαρακτήρων σε αρχεία χρησιμοποιούνται τα ίδια κατηγορήματα. Το μόνο που απαιτείται είναι η ανακατεύθυνση των καναλιών εισόδου/εξόδου.

see(filename): **see/1** ορίζει ως κανάλι εισόδου το αρχείο filename.

seen: **seen/0** ακυρώνει το κανάλι επικοινωνίας που ορίστηκε με την εντολή **see** και κλείνει όλα τα αρχεία εισόδου.

seeing: **seeing/0** επιστρέφει το τρέχον κανάλι εισόδου.

tell(filename): **tell/1** ορίζει ως κανάλι εξόδου το αρχείο filename.

told: **told/0** ακυρώνει το κανάλι επικοινωνίας που ορίστηκε με την εντολή **tell** και κλείνει όλα τα αρχεία εξόδου.

telling: **telling/0** επιστρέφει το τρέχον κανάλι εξόδου.

Σημείωση: Στο online περιβάλλον του SWISH, η είσοδος από αρχείο μπορεί να γίνει μόνο με την χρήση ενός URL.



File Input/Output

Παράδειγμα Το παρακάτω κατηγορημα `out` δέχεται σαν είσοδο μία λίστα με στοιχεία και τα γράφει (ένα στοιχείο ανά γραμμή) σε ένα προκαθορισμένο αρχείο, καλώντας το αναδρομικό κατηγορημα `write_list`

```
out(L):-  
    tell('myfile.txt'),  
    write_list(L),  
    told.  
write_list([]).  
write_list([H|T]):-  
    write(H),  
    nl,  
    write_list(T).
```



File Input/Output

?- see('students.txt').

Yes

?- read(Next).

Next = student(Anna, 44711, pass)

Yes

?- read(Next).

Next = student(John, 50815, pass)

Yes

?- read(Next).

Next = student(Bob, 41018, fail)

Yes

?- read(Next).

Next = end_of_file

Yes

?- seen.

Yes

Reading Terms from a File

E.g. content of file *students.txt*:

```
% Database of students
student(Anna, 44711, pass).
student(John, 50815, pass).
student(Bob, 41018, fail).
```



Cut (Αποκοπή)

- Η **αποκοπή** παρέχει στον προγραμματιστή τη δυνατότητα να «κλαδέψει» δυναμικά το δένδρο υπολογισμού που δημιουργεί η Prolog. Είναι ένας τρόπος ελέγχου του μηχανισμού οπισθοδρόμησης, ο οποίος απαγορεύει τη δημιουργία κλαδιών που ξέρουμε ότι δεν θα οδηγήσουν σε λύση.
- Η αποκοπή είναι το κατηγορήμα ελέγχου της οπισθοδρόμησης, συμβολίζεται με το χαρακτήρα «**!**» και πετυχαίνει πάντα.
- Όταν ο μηχανισμός αναζήτησης της Prolog συναντήσει μία αποκοπή
 - Αγνοούνται όλες οι προτάσεις που ανήκουν στη διαδικασία και έπονται της πρότασης που περιέχει την αποκοπή.
 - Αγνοούνται όλες οι εναλλακτικές λύσεις των ατομικών τύπων που βρίσκονται πριν από την αποκοπή μέσα στο σώμα του κανόνα που εμφανίζεται.



Cut (Αποκοπή)

- Το cut λειτουργεί σαν δίοδος μονής κατεύθυνσης. Όταν η ροή είναι φυσιολογική (από αριστερά προς τα δεξιά) το cut γίνεται TRUE και δεν εμποδίζει καθόλου (είναι σαν να μην υπάρχει). Στο backtracking, όμως, **το cut απαγορεύει τη "διέλευση" προς τα αριστερά του**.
- Σαν αποτέλεσμα έχει την καλύτερη απόδοση (και χωρικά και χρονικά).
- Περιπτώσεις χρήσης:
 - Για να σταματήσει η διαδικασία σε συγκεκριμένο κανόνα
 - Για να σταματήσει την προσπάθεια ικανοποίησης συγκεκριμένου στόχου
 - Για να αποτρέψει την οπισθοδρόμηση, δηλ. την εύρεση εναλλακτικών λύσεων



Επίδραση της αποκοπής στο δέντρο αναζήτησης

Παράδειγμα

- Έστω ότι έχουμε τα εξής κατηγορήματα:
- Στην ερώτηση $b(X)$ η Prolog θα επιστρέψει τρεις λύσεις:
- Αν στον κανόνα του κατηγορήματος $b/1$ εισάγουμε σαν τελευταίο υποστόχο το κατηγορήμα της αποκοπής, η ερώτηση $b(X)$ επιστρέφει μια και μοναδική λύση:

$a(1).$
 $a(2).$
 $b(X) :- a(X).$
 $b(3).$

$?- b(X).$
 $X = 1 ;$
 $X = 2 ;$
 $X = 3.$

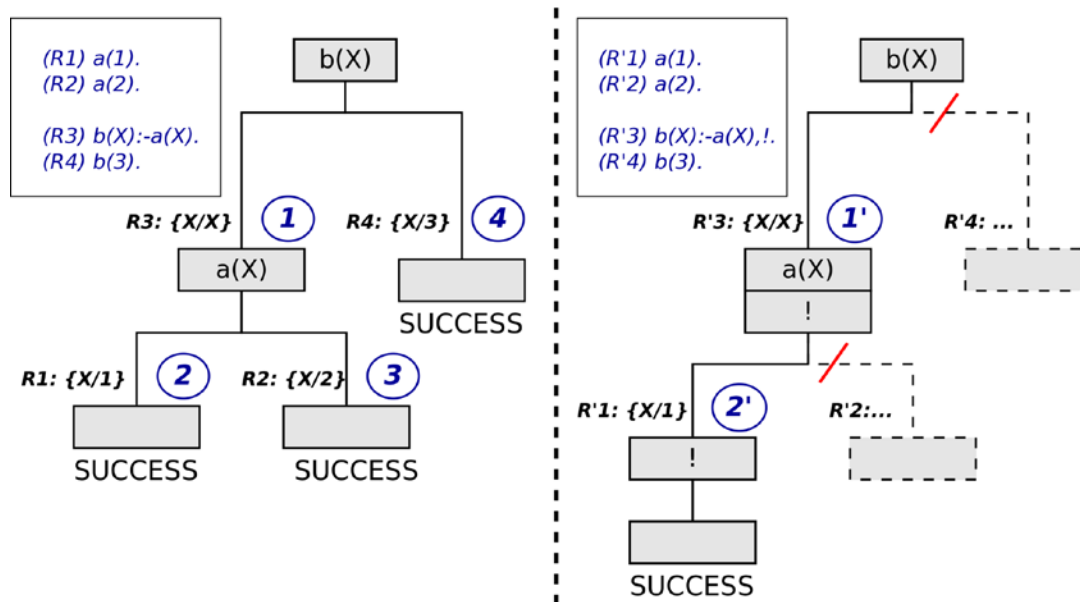
$b(X) :- a(X), !.$

$?- b(X).$
 $X = 1.$



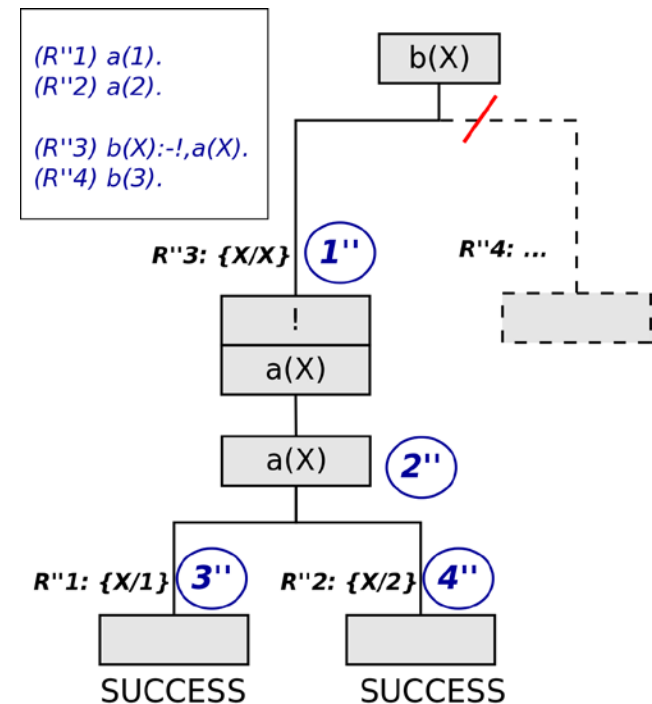
Επίδραση της αποκοπής στο δέντρο αναζήτησης

- Αυτό συμβαίνει γιατί ο μηχανισμός εκτέλεσης αποδεικνύει τον υποστόχο $a(X)$ για $X = 1$, και έπειτα εκτελεί το κατηγορήμα της αποκοπής, με αποτέλεσμα οι εναλλακτικές λύσεις για το κατηγορήμα $a/1$ αλλά και για το $b/1$ μέσα στο οποίο εμφανίζεται η αποκοπή να μην λαμβάνονται υπόψη.



Επίδραση της αποκοπής στο δέντρο αναζήτησης

- Έστω τώρα ότι εισάγουμε την αποκοπή στην πρώτη πρόταση του κατηγορήματος πριν από τον υποστόχο $a(X)$, δηλαδή έχουμε την πρόταση: $b(X) :- !, a(X)$.
- Στην ερώτηση $b(X)$, η Prolog θα επιστρέψει τις ακόλουθες λύσεις:
 $?- b(X).$
 $X = 1 ;$
 $X = 2.$
- Εφόσον η αποκοπή εκτελέστηκε πριν την απόδειξη του υποστόχου $a(X)$, δεν “κλάδεψε” τις εναλλακτικές λύσεις οι οποίες προκύπτουν από αυτόν, αλλά μόνο την λύση που προκύπτει από την δεύτερη πρόταση του κατηγορήματος $b(X)$



Cut (Αποκοπή)

Σε προηγούμενο παράδειγμα είχαμε ορίσει (σωστά) το παραγοντικό ως εξής:

```
factorial(0,1).
```

```
factorial(N,F):-
```

```
    N>0,
```

```
    N1 is N-1,
```

```
    factorial(N1,F1),
```

```
    F is N*F1.
```

Αν το ορίζαμε όπως παρακάτω, θα έβγαζε αποτέλεσμα αλλά θα έπεφτε σε ατέρμονο βρόχο.

```
factorial(0,1).
```

```
factorial(N,F) :-
```

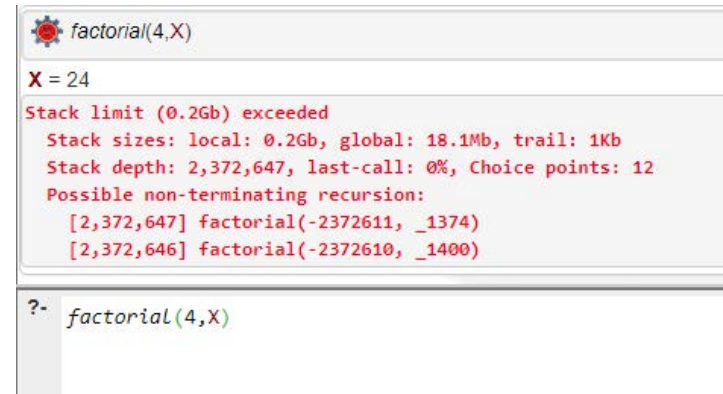
```
    N1 is N-1,
```

```
    factorial(N1,F1),
```

```
    F is N*F1.
```

Για να το αποφύγουμε αυτό, θα γράφαμε το εξής:

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

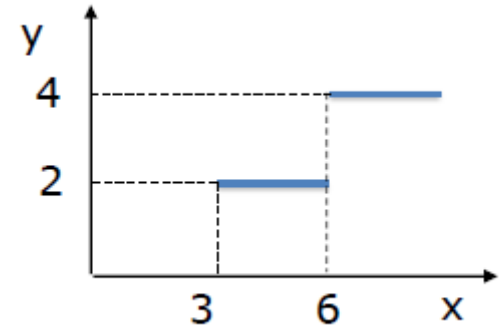


```
factorial(4,X)
X = 24
Stack limit (0.2Gb) exceeded
Stack sizes: local: 0.2Gb, global: 18.1Mb, trail: 1Kb
Stack depth: 2,372,647, last-call: 0%, Choice points: 12
Possible non-terminating recursion:
[2,372,647] factorial(-2372611, _1374)
[2,372,646] factorial(-2372610, _1400)
?- factorial(4,X)
```

Αποφυγή άσκοπων ελέγχων

- Έστω ότι θέλουμε να υλοποιήσουμε τη συνάρτηση:

$$f(x) = \begin{cases} 0, & \text{αν } x < 3 \\ 2, & \text{αν } 3 \leq x < 6 \\ 4, & \text{αν } x \geq 6 \end{cases}$$



- Στο πρόγραμμα που ακολουθεί αποφεύγονται οι περιττοί έλεγχοι κατά την οπισθοδρόμηση αφού γνωρίζουμε εκ των προτέρων ότι μόνο μία από τις τρεις προτάσεις μπορεί να είναι αληθής κάθε φορά.

$f(x, 0) :- x < 3, !.$
 $f(x, 2) :- x \geq 3, x < 6, !.$
 $f(x, 4) :- x \geq 6, !.$

Πρόγραμμα Α

$f(x, 0) :- x < 3, !.$
 $f(x, 2) :- x < 6, !.$
 $f(x, 4).$

Πρόγραμμα Β



Types of Cuts

- Μία αποκοπή είναι **πράσινη** όταν δεν επηρεάζει τη δηλωτική σημασία του προγράμματος.
 - Π.χ. στο Πρόγραμμα Α οι αποκοπές είναι πράσινες, αφού αν αφαιρεθούν το πρόγραμμα θα επιστρέψει τα ίδια αποτελέσματα σε περισσότερο χρόνο.
- Μία αποκοπή είναι **κόκκινη** όταν κατά την αφαίρεσή της από το πρόγραμμα μεταβάλλεται η δηλωτική σημασία του προγράμματος.
 - Π.χ. στο Πρόγραμμα Β οι αποκοπές είναι κόκκινες, αφού αν αφαιρεθούν το πρόγραμμα θα επιστρέψει διαφορετικά αποτελέσματα.



Cut and Negation

- Μία από τις σημαντικότερες χρήσεις της αποκοπής είναι η υλοποίηση της άρνησης σαν αποτυχία (*negation as failure*). Αυτό επιτυγχάνεται με τη χρήση του ενσωματωμένου κατηγορήματος **fail** της Prolog, το οποίο αποτυγχάνει πάντα.
- Το κατηγορήμα **fail** “προκαλεί” οπισθοδρόμηση (πίσω στο τελευταίο σημείο επιλογής).
- Ο συνδυασμός αποκοπής και αποτυχίας (εξαναγκασμός σε αποτυχία) κάνει δυνατή την υλοποίηση κατηγορημάτων που περιέχουν την άρνηση. Για παράδειγμα το παρακάτω πρόγραμμα επιτυγχάνει όταν ένα στοιχείο **δεν** είναι μέλος μίας λίστας.

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

```
not_member(X, [X|_]) :- !, fail.
```

```
not_member(X, [_|Rest]) :- not_member(X, Rest).
```

Σημείωση: Η απλή προσέγγιση είναι να χρησιμοποιηθεί άρνηση: `not_member(X,List) :- not(member(X,List)).`

Το ζητούμενο όμως ήταν να μην χρησιμοποιηθεί το `not/1`



Closed World Assumption

- Η αρχή της άρνησης σαν αποτυχία βασίζεται στην *Closed World Assumption* (Υπόθεση του Κλειστού Κόσμου). Σύμφωνα με αυτή όλη η απαραίτητη πληροφορία για την επίλυση του προβλήματος περιέχεται στο πρόγραμμα και οτιδήποτε δεν περιέχεται σε αυτό θεωρείται λάθος (οποιαδήποτε σχέση δεν μπορεί να αποδειχθεί από το σύστημα θεωρείται ως εσφαλμένη).
- Η υλοποίηση του τελεστή άρνησης του λογικού προγραμματισμού (άρνηση σαν αποτυχία) γίνεται με τη βοήθεια του συνδυασμού αποκοπής και άρνησης.

`not(X) :- X,!,fail.`

`not(X).`



Το πρόβλημα του βοσκού

- Το πρόβλημα του βοσκού αποτελεί ένα κλασσικό πρόβλημα λογικής (puzzle).
- Σε μια όχθη ενός ποταμού υπάρχουν ένας βοσκός, ένας λύκος ένα πρόβατο και ένα δεμάτι σανό. Η διαθέσιμη βάρκα χωρά μόνο δύο αντικείμενα κάθε φορά. Αν υποθέσουμε ότι σε κάθε στιγμή ο λύκος και το πρόβατο, καθώς και το πρόβατο και το δεμάτι δεν μπορούν να είναι μόνα τους σε μία όχθη (χωρίς τον βοσκό), ποια είναι η ακολουθία κινήσεων η οποία πρέπει να γίνει για να περάσουν όλοι απέναντι;
- Το παραπάνω πρόβλημα είναι ουσιαστικά ένα πρόβλημα σχεδιασμού κινήσεων (planning). Στα προβλήματα της κατηγορίας αυτής, μία λύση είναι η **αναζήτηση στο χώρο των καταστάσεων** του κόσμου του προβλήματος. Κάθε τέτοια κατάσταση περιγράφει τον "κόσμο" του προβλήματος την συγκεκριμένη χρονική στιγμή. Οι μεταβάσεις από μια συγκεκριμένη κατάσταση σε μια επόμενη γίνονται μέσω τελεστών.



Το πρόβλημα του βοσκού: Επιλογή Αναπαράστασης

- Είναι πολλές οι διαθέσιμες επιλογές με τις οποίες μπορούμε να αναπαραστήσουμε το συγκεκριμένο πρόβλημα. Η παρακάτω υλοποίηση χρησιμοποιεί για την περιγραφή της κατάστασης ένα σύνθετο όρο της μορφής:

state(shepherd(A),sheep(B),wolf(C),hey(D))

όπου τα A, B, C, D είναι οι όχθες στις οποίες βρίσκονται αντίστοιχα ο βοσκός, το πρόβατο, ο λύκος και το δεμάτι σανό.

- Με βάση την παραπάνω αναπαράσταση ορίζονται οι τελεστές, που αντιστοιχούν ουσιαστικά στις κινήσεις που μπορούν να γίνουν για να λυθεί το πρόβλημα.



Το πρόβλημα του βοσκού: Ορισμός Κατηγορημάτων

- Το κατηγορημα **opposite/2** δηλώνει τις δύο απέναντι όχθες.

```
% opposite/2
```

```
% opposite(X1,X2)
```

```
opposite(a,b).
```

```
opposite(b,a).
```

- Το κατηγορημα **move/3** περιγράφει τις επιτρεπτές κινήσεις που μπορούν να γίνουν. Η μεταβλητή **State** είναι η "τρέχουσα" κατάσταση, η μεταβλητή **Move** είναι η κίνηση που λαμβάνει χώρα για να προκύψει η νέα κατάσταση που ενοποιείται με την μεταβλητή **NewState**.

```
% move/3
```

```
% move(State,Move,NewState)
```

```
% Move the shepherd
```

```
move(state(shepherd(X1),sheep(S),wolf(W),hey(H)),
```

```
go_to_other_shore(X1,X2),
```

```
state(shepherd(X2),sheep(S),wolf(W),hey(H))):- opposite(X1,X2).
```



Το πρόβλημα του βοσκού: Ορισμός Κατηγορημάτων

% Take the sheep to the other side

```
move(state(shepherd(X1),sheep(X1),wolf(W),hey(H)),
move_sheep(X1,X2),
state(shepherd(X2),sheep(X2),wolf(W),hey(H))):-
opposite(X1,X2).
```

% Take the wolf to the other side

```
move(state(shepherd(X1),sheep(S),wolf(X1),hey(H)),
move_wolf(X1,X2),
state(shepherd(X2),sheep(S),wolf(X2),hey(H))):-
opposite(X1,X2).
```

% Take the hey to the other side

```
move(state(shepherd(X1),sheep(S),wolf(W),hey(X1)),
move_hey(X1,X2),
state(shepherd(X2),sheep(S),wolf(W),hey(X2))):-
opposite(X1,X2).
```



Το πρόβλημα του βοσκού: Ορισμός Κατηγορημάτων

- Το κατηγορημα `is_valid/1` πετυχαίνει όταν η κατάσταση `State` είναι επιτρεπτή, δηλαδή δεν παραβιάζει τους περιορισμούς του προβλήματος.

```
% is_valid/1
```

```
% is_valid(State).
```

```
is_valid(state(shepherd(X),sheep(X),wolf(_),hey(_))).
```

```
is_valid(state(shepherd(X),sheep(_),wolf(X),hey(X))).
```

- Το κατηγορημα `end_state/1` πετυχαίνει όταν η `State` είναι η τελική κατάσταση, δηλαδή όλα τα αντικείμενα είναι στην όχθη `b` (θεωρούμε ότι αρχικά όλα τα αντικείμενα είναι στην όχθη `a`).

```
% end_state/1
```

```
% end_state(State).
```

```
end_state(state(shepherd(b),sheep(b),wolf(b),hey(b))).
```



Το πρόβλημα του βοσκού: Ορισμός Κατηγορημάτων

- Το κατηγορημα **solve/3** βρίσκει την σειρά των κινήσεων που πρέπει να γίνουν για να επιλυθεί το πρόβλημα. Ουσιαστικά ο αλγόριθμος αναζήτησης που χρησιμοποιείται είναι η κατα-βάθος αναζήτηση της Prolog.
- Η μεταβλητή **State** είναι η "τρέχουσα" κατάσταση, η μεταβλητή **Moves** η λίστα των κινήσεων και η **Previous_States** η λίστα των καταστάσεων τις οποίες έχει "επισκεφτεί" μέχρι τώρα η διαδικασία αναζήτησης.
- Κάθε νέα κατάσταση που προκύπτει (**NextState**), δεν πρέπει να ανήκει στην λίστα αυτή για την αποφυγή βρόχων.

```
% solve/3
```

```
% solve(State,Moves,Previous_States)
```

```
solve(State,[],_):- end_state(State).
```

```
solve(State,[Move|Moves],Previous_States):-
```

```
    move(State,Move,NextState),
```

```
    \+ member(NextState,Previous_States),
```

```
    is_valid(NextState),
```

```
    solve(NextState,Moves,[NextState|Previous_States]).
```



Το πρόβλημα του βοσκού: Ορισμός Κατηγορημάτων

- Το επόμενο κατηγορημα τυπώνει στην οθόνη τα μέλη μιας λίστας, και χρησιμοποιείται για να τυπώνει τις κινήσεις στην οθόνη.

```
% pretty_write/1
% pretty_write(List)
pretty_write([]).
pretty_write([First|Rest]):-
    write(' -> '), write(First), nl,
    pretty_write(Rest).
```

- Το κατηγορημα `run/1`, χρησιμοποιείται για να εκτελέσουμε το πρόγραμμα.

```
run(Moves):-
    solve(state(shepherd(a),sheep(a),wolf(a),hey(a)),Moves,
    [state(shepherd(a),sheep(a),wolf(a),hey(a))]),
    pretty_write(Moves),nl.
```



Το πρόβλημα του βοσκού: εκτέλεση

```
run(M)

-> move_sheep(a, b)
-> go_to_other_shore(b, a)
-> move_wolf(a, b)
-> move_sheep(b, a)
-> move_hey(a, b)
-> go_to_other_shore(b, a)
-> move_sheep(a, b)

M = [move_sheep(a, b), go_to_other_shore(b, a), move_wolf(a, b), move_sheep(b, a), move_hey(a, b), go_to_other_shore(b, a), move_sheep(a, b)]
-> move_sheep(a, b)
-> go_to_other_shore(b, a)
-> move_wolf(a, b)
-> move_sheep(b, a)
-> move_hey(a, b)
-> go_to_other_shore(b, a)
-> move_sheep(a, b)

M = [move_sheep(a, b), go_to_other_shore(b, a), move_wolf(a, b), move_sheep(b, a), move_hey(a, b), go_to_other_shore(b, a), move_sheep(a, b)]
-> move_sheep(a, b)
-> go_to_other_shore(b, a)
-> move_hey(a, b)
-> move_sheep(b, a)
-> move_wolf(a, b)
-> go_to_other_shore(b, a)
-> move_sheep(a, b)

M = [move_sheep(a, b), go_to_other_shore(b, a), move_hey(a, b), move_sheep(b, a), move_wolf(a, b), go_to_other_shore(b, a), move_sheep(a, b)]
-> move_sheep(a, b)
-> go_to_other_shore(b, a)
-> move_hey(a, b)
-> move_sheep(b, a)
-> move_wolf(a, b)
-> go_to_other_shore(b, a)
-> move_sheep(a, b)

M = [move_sheep(a, b), go_to_other_shore(b, a), move_hey(a, b), move_sheep(b, a), move_wolf(a, b), go_to_other_shore(b, a), move_sheep(a, b)]
false

?- run(M)
```

Examples History Solutions ☐ table results Run!

