Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

Contents

- **Logic Programming**
- **Overview of Prolog**

Literature

- P.M. Nugues, **An Introduction to Prolog**. In: Language Processing with Perl and Prolog, Cognitive Technologies, Springer 2014.

- **SWI Prolog**: https://www.swi-prolog.org/

- Cordell Green, **The Application of Theorem Proving to Question Answering Systems**, Ph.D. thesis, Stanford University, 1969.
  http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.222.8760&rep=rep1&type=pdf

Logic Programming Languages

- Logic programming can be classified as declarative programming paradigm, characterized by expressing the logic of computations rather than the control flow (*what* vs. *how*).

- A logic program consists of facts and rules about some problem domain based on formal logic (e.g., first-order logic).

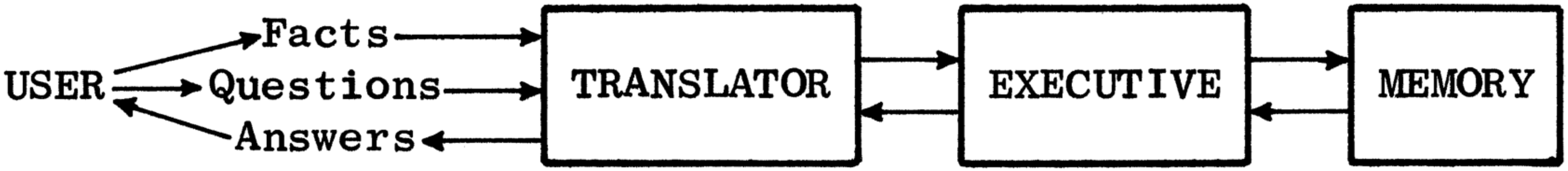
- Examples of logic programming languages include Prolog and Answer Set Programming (ASP).

```
father(dan,phil).
mother(sue,jack).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Logic Programming Languages

The foundations of logic programming go back to the 1960ies, e.g.,

the work on Question-Answering Systems by Cordell Green.



- A question-answering system may be broadly defined as a system that accepts information and uses this information to answer questions.
- The user presents statements (facts and questions).
- A translator converts them into an internal form. Facts are stored in memory.
- Answers to questions are formed in two ways:
  - the explicit answer is found in memory, or
  - the answer is computed from the information stored in memory.
- The executive program controls the process of storing information, finding information, and computing answers.

Cordell Green, The Application of Theorem Proving to Question Answering Systems, Ph.D. thesis, Stanford University, 1969.

Prolog

- Most-widely known/successful logic programming language.
- Declarative programming with facts, rules, and queries. Logic programs are queried about the provability of a goal.
- Turing complete: Prolog is as powerful as other programming languages.
- Homoiconic: treat code as data. In prolog fact and rules make up a database.
- General purpose but mainly used in expert systems, planning, AI, etc. (cf. IBM Watson)

Prolog

- As a declarative, logic-based language Prolog requires a different mindset than imperative/procedural/OO programming languages .

- The programmer/user
  - defines facts, and logical rules that constitute a so-called database
  - and queries the database by stating logical goals.

- Prolog will answer these queries by attempting to prove these goals.

Prolog

- SWI Prolog: https://www.swi-prolog.org/
  - Open Source
  - Windows/Mac/Linux
  - REPL (Read Eval Print Loop)
  - Interpreter
  - Compiler
- Online Programming with SWI Prolog: https://swish.swi-prolog.org/

Note:

- Code in yellow boxes corresponds to Prolog programs.
- Gray boxes are used to show the results of queries using the Prolog interpreter (REPL).
- Orange boxes are used to show Prolog syntax rules.

Running Prolog Programs

- To run a program, a Prolog system has to load the program text and add it to the current database in memory.
- Once Prolog is launched, it displays a prompt symbol "**`?-`**" and accepts commands from the user (REPL).

```
$ swipl plc.pl
?-
```

A Simple Prolog Program

```
course(plc, wednesday).                  % facts
course(plc, thursday).
good_day(sunday).

good_day(X) :- course(plc, X).          % rule with a variable X
```

```
%swipl plc.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.3)
?- good_day(sunday).
true.

?- good_day(friday).
false.

?- good_day(wednesday).
true.

?- halt.
%
```

A Simple Prolog Program - II

```
:- initialization(main).

course(plc, wednesday).                 % facts
course(plc, thursday).

good_day(sunday).
good_day(X) :- course(plc, X).          % rule with a variable X

main :- forall(good_day(X),
            format('~s~n', X)),
        halt.
```

```
$ swipl plc2.pl
sunday
wednesday
thursday
$
```

Another Simple Program – homer.pl

```
character(priam, iliad).
character(hecuba, iliad).
character(achilles, iliad).
character(agamemnon, iliad).
character(patroclus, iliad).
character(hector, iliad).
character(andromache, iliad).
character(rhesus, iliad).
character(ulysses, iliad).
character(menelaus, iliad).
character(helen, iliad).
character(ulysses, odyssey).
character(penelope, odyssey).
character(telemachus, odyssey).
character(laertes, odyssey).
character(nestor, odyssey).
character(menelaus, odyssey).
character(helen, odyssey).
character(hermione, odyssey).
...
```

```
...
male(priam).
male(achilles).
male(agamemnon).
male(patroclus).
male(hector).
male(rhesus).
male(ulysses).
male(menelaus).
male(telemachus).
male(laertes).
male(nestor).

female(hecuba).
female(andromache).
female(helen).
female(penelope).
...
```

Another Simple Program – homer.pl

```
...
father(priam, hector).                % priam is the father of hector
father(laertes, ulysses).
father(atreus, menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).

mother(penelope, telemachus).
mother(helen, hermione).
mother(hecuba, hector).

king(ulysses, ithaca, achaean).
king(menelaus, sparta, achaean).
king(nestor, pylos, achaean).
king(agamemnon, argos, achaean).
king(priam, troy, trojan).
king(rhesus, thrace, trojan).

son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).
```

Prolog

- A Prolog program consists of one or more clauses.
- Each clause is either a fact or a rule.
- Once a program has been loaded, users can submit queries (goals), and the Prolog interpreter will give answers according to the facts and rules.

- Facts and rules may be comprised of:
  - Atoms (monday, plc,...) - begin with lower case letter!
  - Numbers (123)
  - Variables (X, Something, _123) - begin with upper case letter or _!
  - Lists ([...])

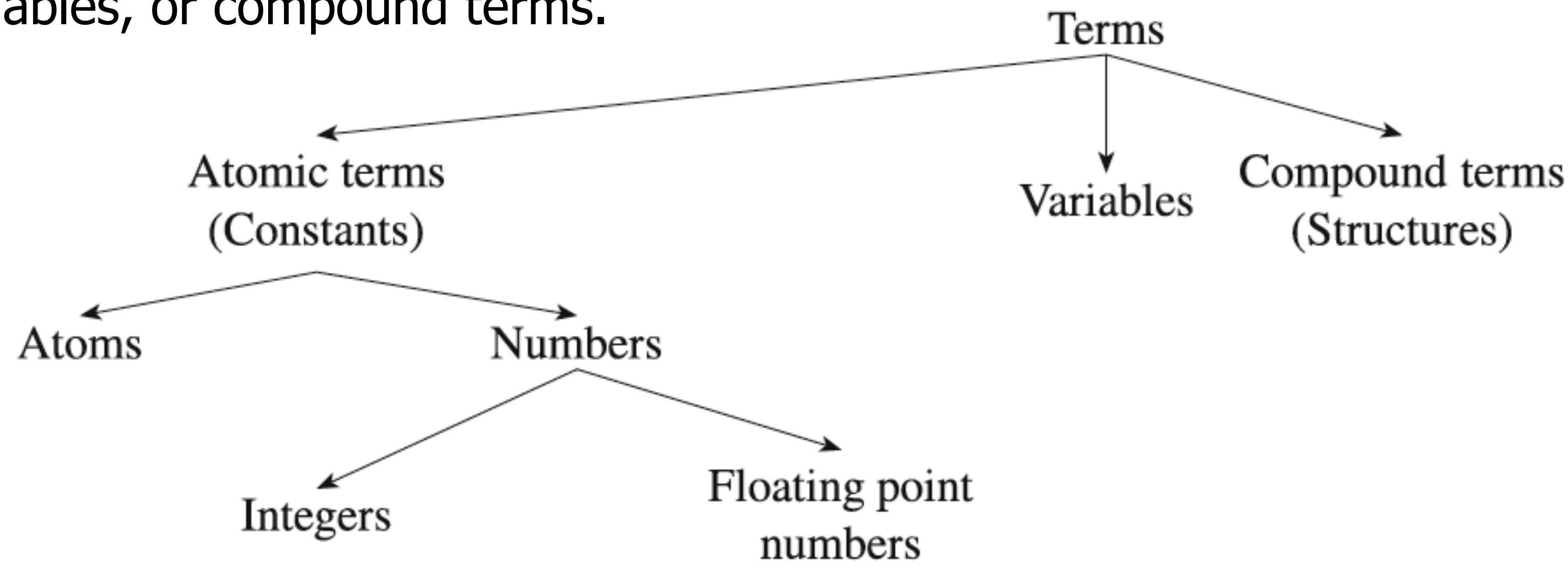
Prolog

- Prolog is a dynamically typed language.

- Prolog has a single data type, the term, which has several subtypes: atoms, numbers, variables and compound terms.

Terms

- Every data object in Prolog is a term. Terms are either atomic terms, variables, or compound terms.



- The fact **`father(laertes, ulysses)`** is a compound term or structure composed of other terms (subterms). Syntactically, a compound term consists of a functor (the name of the relation) and arguments.
- The leftmost functor of a term is the principal functor.

Facts

- Facts are statements that describe object properties or relations between objects. (Facts are always "true").
- A collection of facts and rules makes up a database.
- Such a database represents the knowledge of a certain situation in a logical format.

```
female(penelope).
male(ulysses).

mother(penelope, telemachus).
father(laertes, ulysses).

son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).
```

Facts

- The general form of a Prolog fact (with arguments) is:

```
relation(object₁, object₂, ..., objectₙ).
```

- Symbols or names representing objects, such as **`ulysses`** or **`penelope`**, are called atoms.

```
father(laertes, ulysses).
```

- Atoms are normally strings of letters, digits, or underscores "_", and begin with a lowercase letter. An atom can also be a string beginning with an uppercase letter or including white spaces, but it must be enclosed between quotes, e.g., **`'Ulysses'`** or **`'Pallas Athena'`**

Facts - predicate/arity

- In logic, the name of the symbolic relation is the predicate, the objects involved in the relation are the arguments, and the number n of the arguments is the arity.

- Traditionally, a Prolog predicate is indicated by its name and arity: **`predicate/arity`**, for example, **`raining/0, mother/2, king/3.`**

```
raining.

mother(penelope, telemachus).

king(agamemnon, argos, achaean).
```

Queries

- A query is a request to prove a goal or retrieve information from the database, for example, if a fact is true.
- The expression **`male(ulysses)`** is a goal to prove.

```
?- male(ulysses).
true.
```

- Compound queries are usually a conjunction of two or more goals:

```
?- male(menelaus), king(menelaus, sparta, achaean).
true.
```

Variables

- Logical variables begin with an uppercase letter or an underscore "_" .
- Logical variables can stand for any term: constants, compound terms, and other variables.
- A term containing variables can unify with a compatible fact.

- For example, the term **`father(X, Y)`** can unify with a compatible fact such as **`father(priam, hector)`** with the substitutions **`X = priam`** and **`Y = hector`**.

```
?- father(X,Y).
X = priam,
Y = hector
```

```
...
father(priam, hector).
father(laertes, ulysses).
father(atreus, menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).
...
```

Queries

- When there are multiple solutions to a query with variables, Prolog considers the first fact to match the query in the database.
- The user can type “`;`” (or blank) to get the next answers until there is no more solution.

```
?- father(X,Y).
X = priam,
Y = hector ;
X = laertes,
Y = ulysses
```

```
...
father(priam, hector).
father(laertes, ulysses).
father(atreus, menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).
...
```

- Note: This is a procedural aspect of Prolog, since the order in which the facts are specified may have an influence on the outcome of a program.

Shared Variables

- Goals in a conjunctive query can share variables. This is useful to constrain arguments of different goals to have the same value.
- The question *Is the king of Ithaca also a father?* , comprises the conjunction of two goals **`king(X, ithaca, Y)`** and **`father(X, Z)`**, where the variable **`X`** is shared between the goals.

```
?- king(X, ithaca, Y), father(X, Z).
X = ulysses,
Y = achaean,
Z = telemachus.
```

```
...
father(priam, hector).
father(laertes, ulysses).
father(atreus, menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).
...
king(ulysses, ithaca, achaean).
king(menelaus, sparta, achaean).
...
```

Queries – Anonymous Variables

- Anonymous variables "_" may be used if some details are not needed.

```
?- king(X, ithaca, _), father(X, _).
X = ulysses.
```

```
...
father(priam, hector).
father(laertes, ulysses).
father(atreus, menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).
...
king(ulysses, ithaca, achaean).
king(menelaus, sparta, achaean).
...
```

Rules

- Rules are a way to derive a new relation from existing ones.

```
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).
```

- Rules can call other rules.

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).

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

- Rules can be recursive.

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

Rules

- Rules consist of a term called the head or consequent, followed by the symbol ":-", read if, and a conjunction of goals in the body or antecedent of the rule.

HEAD :- G_1, G_2, G_3, ... G_n

The head is true if the body is true.

- Variables of a rule are shared between the body and the head.
- Rules can be queried just like facts:

```
?- son(telemachus, Y).
Y = ulysses ;
Y = penelope.
```

```
...
father(priam, hector).
father(laertes, ulysses).
father(atreus, menelaus).
father(menelaus, hermione).
father(ulysses, telemachus).
...
mother(penelope, telemachus).
mother(helen, hermione).
mother(hecuba, hector).
...
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).
```

Rules - Predicates

- A predicate is defined by a set of clauses (facts and rules) with the same principal functor and arity, e.g., **son/2**.

```
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).
```

- Usually, all clauses of the same predicate must be grouped together.

- Note: Facts are rules that are always true.

  true/0 is a built-in predicate that always succeeds.

```
father(laertes, ulysses).                % is equivalent

father(laertes, ulysses) :- true.        % is equivalent
```

Rules - Disjunction

- It is also possible to use a disjunction with the operator ";". Thus:

```
son(X, Y) :- father(Y, X), male(X) ; mother(Y, X), male(X).
```

is equivalent to

```
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).
```

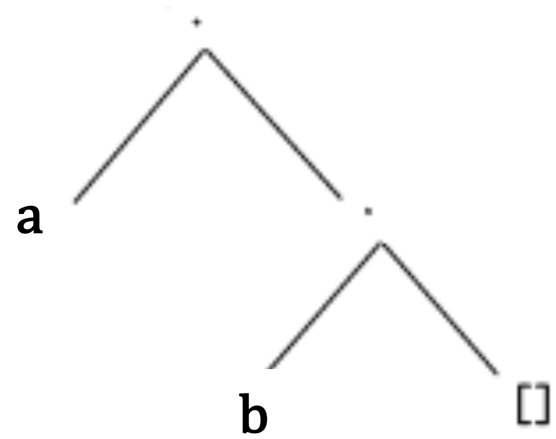
- However, the disjunctive form should not be used because it impairs the legibility of clauses and programs.

Lists

A Prolog list is a sequence of an arbitrary number of terms separated by commas and enclosed within square brackets.

```
[a]                                % list with one atom.
[a, b]                             % list with two atoms.
[a, X, father(X, telemachus)] % list w. atom, variable, comp. term
[[a, b], [[[c]]]]                  % list with 2 sublists
[]                                 % atom representing the empty list
```

The square bracketed notation is a shortcut. The list functor is a dot: "`./2`", and `[a, b]` is equivalent to the term `.(a, .(b, [])).`



Lists are recursive structures comprised of two parts: a head, the first element of a list, and a tail, the remaining list.

Lists

- Using "|" a list can be split into its head and tail.

```
?- [a, b] = [H | T].
H = a,
T = [b].

?- [a] = [H | T].
H = a, T = []

?- [a, [b]] = [H | T].
H = a, T = [[b]]

?- [a, b, c, d] = [X, Y | T].
X = a, Y = b, T = [c, d]

?- [[a, b, c], d, e] = [H | T].
H = [a, b, c], T = [d, e]
```

List-Handling Predicates

Prolog systems provide a set of built-in list predicates, including **`member`**, **`append`**, **`delete`**, **`intersection`**, **`reverse`**, **`length`**, **`maplist`** …

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

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

```
?- length([a, [a, b], c], N).
N = 3.
```

```
?- maplist(male,[laertes, ulysses]).
true.
?-
```

Constraint Logic Programming

SWIPL supports constraint logic programming (CLP) over different domains:

- CLP(FD) for integers
- CLP(B) for Boolean variables
- CLP(Q) for rational numbers
- CLP(R) for floating point numbers

A constraint logic program may contain predicates that are constraints.

```
factorial(N, F) :-
        N #> 0,      % this is a constraint
        N1 #= N - 1,
        F #= N * F1,
        factorial(N1, F1).
```

For example, Boolean Satisfiability Problems (SAT) can be expressed with CLP(B).

Magic Square Problem

https://www.aktagon.com/articles/constraint-logic-programming-in-prolog

```
magic_square(Square) :-
    Square = [A,B,C,D,E,F,G,H,I], % Square list contains variables A - I

    Square ins 1..9,          % Each cell is in the domain from 1 to 9

    all_different(Square),    % All cells must have different values

    % Row constraints: Each row should sum up to the value 'Sum'
    A+B+C #= Sum,
    D+E+F #= Sum,
    G+H+I #= Sum,

    % Column constraints: Each column should sum up to the value 'Sum'
    A+D+G #= Sum,
    B+E+H #= Sum,
    C+F+I #= Sum,

    % Both the diagonals should sum up to the value 'Sum'
    A+E+I #= Sum,  % From top-left to bottom-right diagonal
    C+E+G #= Sum,  % From top-right to bottom-left diagonal

    % Find solution that satisfies allconstraints
    label(Square).
```

Sudoku https://swish.swi-prolog.org/example/clpfd_sudoku.pl

9	8	7	6	5	4	3	2	1
2	4	6	1	7	3	9	8	5
3	5	1	9	2	8	7	4	6
1	2	8	5	3	7	6	9	4
6	3	4	8	9	2	1	5	7
7	9	5	4	6	1	8	3	2
5	1	9	2	8	6	4	7	3
4	7	2	3	1	9	5	6	8
8	6	3	7	4	5	2	1	9

All permutations of all rows `[1..9]`: $9!^9 \approx 109.1 \text{ x } 10^{48}$

All valid Sudoku grids: $6{,}670{,}903{,}752{,}021{,}072{,}936{,}960 \approx 6.7 \text{ x } 10^{21}$

Sudoku https://swish.swi-prolog.org/example/clpfd_sudoku.pl

```
:- use_module(library(clpfd)).  % constraint logic programming Integer lib

sudoku(Rows) :-
        length(Rows, 9),                          % the list rows has length 9
        maplist(same_length(Rows), Rows),         % all rows have same length
        append(Rows, Vs),                     % Vs is concatenation of all rows
        Vs ins 1..9,                          % each elem of Vs is in 1-9
        maplist(all_distinct, Rows),          % all rows must be distinct
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),       % all columns must be distinct
        Rows = [A,B,C,D,E,F,G,H,I],           % list of rows
        blocks(A, B, C),                      % constraints for 3x3(sub-)blocks
        blocks(D, E, F),
        blocks(G, H, I).

blocks([], [], []).
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :- % first sub-block
         all_distinct([A,B,C,D,E,F,G,H,I]),       % all distinct
         blocks(Bs1, Bs2, Bs3).               % recurse for remaining sub-blocks
...
```

Sudoku https://swish.swi-prolog.org/example/clpfd_sudoku.pl

```
...
problem(1, [[_,_,_, _,_,_, _,_,_],              % Sudoku puzzle to solve
            [_,_,_, _,_,3, _,8,5],
            [_,_,1, _,2,_, _,_,_],
            [_,_,_, 5,_,7, _,_,_],
            [_,_,4, _,_,_, 1,_,_],
            [_,9,_, _,_,_, _,_,_],
            [5,_,_, _,_,_, _,7,3],
            [_,_,2, _,1,_, _,_,_],
            [_,_,_, _,4,_, _,_,9]])
```

```
?- problem(1, Rows), sudoku(Rows),
|   maplist(label, Rows), maplist(portray_clause, Rows).
[9, 8, 7, 6, 5, 4, 3, 2, 1].
[2, 4, 6, 1, 7, 3, 9, 8, 5].
[3, 5, 1, 9, 2, 8, 7, 4, 6].
[1, 2, 8, 5, 3, 7, 6, 9, 4].
[6, 3, 4, 8, 9, 2, 1, 5, 7].
[7, 9, 5, 4, 6, 1, 8, 3, 2].
[5, 1, 9, 2, 8, 6, 4, 7, 3].
[4, 7, 2, 3, 1, 9, 5, 6, 8].
[8, 6, 3, 7, 4, 5, 2, 1, 9].
```

9	8	7	6	5	4	3	2	1
2	4	6	1	7	3	9	8	5
3	5	1	9	2	8	7	4	6
1	2	8	5	3	7	6	9	4
6	3	4	8	9	2	1	5	7
7	9	5	4	6	1	8	3	2
5	1	9	2	8	6	4	7	3
4	7	2	3	1	9	5	6	8
8	6	3	7	4	5	2	1	9

See: https://www.swi-prolog.org/pldoc/man?section=clpfd-sudoku and https://youtu.be/5KUdEZTu06o

How does Prolog Work (=Answer Queries)?

- **Substitution**
- **Unification**
- **Resolution (modus ponens)**
- **Derivation Trees and Backtracking**
- **Optimizations**

Substitution

- To answer a query made of a term T containing variables, Prolog applies a substitution, replacing variables in T by values so that it proves T to be true.
- The substitution $\sigma = \{X = laertes, Y = ulysses\}$ is a solution to the query **`father(X, Y)`** because the fact **`father(laertes, ulysses)`** is in the database.
- A substitution mapping a set of variables onto another set of variables such as $\sigma = \{X = A, Y = B\}$ onto term **`father(X, Y)`** is a renaming substitution.
- The terms **`father(X, Y)`** and **`father(A, B)`** are alphabetical variants.

```
?- father(X,Y) = father(A,B).
X = A,
Y = B.
```

Unification

- To solve equations such as **`T1 = T2`**, Prolog uses unification, which substitutes variables in the terms so that they are identical.

- Two terms unify
  - if they are the same term

    or
  - if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal.

- Example: Unification of tuples

```
?- (A, 2, C) = (1, B, 3).
A = 1,
C = 3,
B = 2.
?-
```

Unification

- Prolog also uses unification in queries to match a goal or a subgoal to the head of the rule.

```
cat(lion).
cat(tiger).
zoo(X, Y, Z) :- X = lion, Y = tiger, Z = bear.
two_cats(X, Y) :- cat(X), cat(Y).
```

```
?- zoo(One, Two, Three).
One = lion,
Two = tiger,
Three = bear.
?-
```

```
?- two_cats(One, Two).
One = Two, Two = lion ;
One = lion, Two = tiger ;
One = tiger, Two = lion ;
One = Two, Two = tiger.
```

Note: order of different answers

Resolution

- The process of answering queries is referred to as resolution. The Prolog
- resolution algorithm is based on the modus ponens from traditional logic.

Major premise: All men are mortal

Minor premise: Socrates is a man

➔ Socrates is mortal

	Formal notation	Prolog notation
Facts	α	`man('Socrates').`
Rules	$\underline{\alpha \Rightarrow \beta}$	`mortal(X) :- man(X).`
Conclusion	β	`mortal('Socrates').`

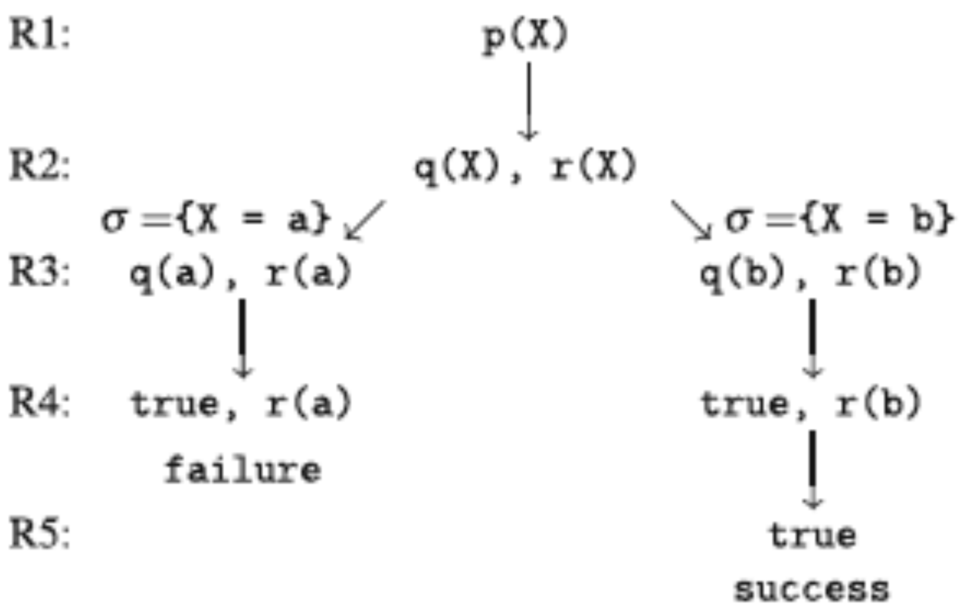
- The actual resolution mechanism in Prolog is called SLD resolution (Selective Linear Definite clause resolution).
- SLD resolution is the basic inference rule used in logic programming.

Derivation Trees and Backtracking

- The resolution process can be represented as a derivation tree. Prolog uses the depth-first strategy to search derivation trees.

- It scans clauses from top to bottom and selects the first one to match the leftmost goal in the resolvent (Ri).
- In case of failure, backtracking is used.
- Backtracking explores all possible alternatives until a solution is found or it fails.

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(b).
r(c).
```



Declarative vs. Procedural Semantics

Declarative Semantics:

- For all X and Y: if X is a parent of Z and Z is a parent of Y, then X is a grandparent of Y.
- The order of goals does not matter from a logical standpoint.
- Thus, the two rules below are logically equivalent.

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

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

Declarative vs. Procedural Semantics

Procedural Semantics:

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

- Find solution for `parent(X, Z)`, then
- find solution for `parent(Z, Y)`.
- If no solution found start over and find different solution for `parent(X, Z)` (Backtracking).

- The order of goals may matter and lead to different answers!
- Procedural semantics of Prolog require an understanding of evaluation order, search strategies, ...

Optimizing Programs

- The performance of programs can often be improved by
  - Goal reordering
  - Tabled Execution
  - Tail recursion

Goal Reordering

- Prolog tries to prove goals from left to right.

- The programmer should order the goals in such a way that goals that are easier to compute should come first.

Optimizing Recursion – Tabled Execution

- Prolog programs are often using recursion. However, recursions may incur a significant runtime overhead and should thus be used carefully.

```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        N > 1,
        N1 is N-1,
        N2 is N-2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1+F2.
```

- The complexity of executing this using SLD resolution however is 2^N and thus becomes prohibitively slow rather quickly.

```
?- time(fib(30, X)).
% 5,385,071 inferences, 0.409 CPU in 0.579 seconds (71% CPU, 13167495 Lips)
X = 1346269 .
```

Optimizing Recursion – Tabled Execution

- Tabled execution (SLG resolution) for Prolog *memoizes* predicates (remembers results from previous computations) in a table providing two properties:
- Re-evaluation of a tabled predicate is avoided by memoizing the answers. This can realize huge performance enhancements.
- Left recursion, does not lead to non-termination by suspending recursive calls and resuming it with answers from the table.

```
:- table p/1.
p(X) :- p(X), q(X).
p(a).
q(a).
```

```
?- p(a).
true.
```

Optimizing Programs containing Recursion

Using tabled execution for **`fib/2`** results in huge performance improvements.

```
:- table fib/2.

fib(0, 1) :- !.
fib(1, 1) :- !.
fib(N, F) :-
        N > 1,
        N1 is N-1,
        N2 is N-2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1+F2.
```

```
?- time(fib(1000, X)).
% 39,030 inferences, 0.008 CPU in 0.013 seconds (61% CPU, 4926786 Lips)
X =
703303677114228158218352548771835497701812698363587327426049050871545371
181969335797422494945626117334877504492417659910881863632654502236471060
120533741212738673391111981393731255987676900919022452453234035O1.
```

Tail Recursion

- Iterative algorithms can be implemented by means of recursive predicates. In a tail recursion the recursive call is the last subgoal of the last rule.
- Prolog systems typically implement tail call optimization (TCO), transforming recursion into iteration.
- With TCO, a (deterministic) tail recursive predicate can be executed with constant memory size (stack).

```
f(X) :- fact(X).
f(X) :- g(X, Y), f(Y).
```