Programare Logică

Introducere în Prolog

25 Septembrie 2018

Conținutul cursului

Part 1: Introducere în programarea logică și programea în Prolog. Bazată pe [Clocksin and Mellish, 2003].

Conținutul cursului

Part 1: Introducere în programarea logică și programea în Prolog. Bazată pe [Clocksin and Mellish, 2003].

Part 2: O revizuire a bazei teoretice a programarii logice. Bazată pe [Ben-Ari, 2001] și [Nilsson and Maluszynski, 2000].

Conținutul cursului

Part 1: Introducere în programarea logică și programea în Prolog. Bazată pe [Clocksin and Mellish, 2003].

Part 2: O revizuire a bazei teoretice a programarii logice. Bazată pe [Ben-Ari, 2001] și [Nilsson and Maluszynski, 2000].

Part 3: Programare logică avansată. Bazată pe [Ben-Ari, 2001] și [Nilsson and Maluszynski, 2000].

Organizare

- Isabela Drămnesc
- Pagina web unde găsiți cursurile: `http://staff.fmi.uvt.ro/~isabela.dramnesc`
  - 14 Cursuri
  - 7 Laboratoare: Prolog
- Notare:
  - 50% : activitate curs și laborator, teme
  - 50% : 1 examen scris (colocviu) (în ultimul curs)

Traditional programming paradigms

Recalling the Von Neumann machine

- The von Neumann machine (architecture) is characterized by:
  - large uniform store of memory,
  - processing unit with registers.
- A program for the von Neumann machine: a sequence of instructions for
  - moving data between memory and registers,
  - carrying out arithmetical-logical operations between registers,
  - control, etc.
- Most programming languages (like C, C++, Java, etc.) are influenced by and were designed for the von Neumann architecture.
  - In fact, such programming languages take into account the architecture of the machine they address and can be used to write efficient programs.
  - The above point is by no means trivial, and it leads to a separation of work ("the software crisis"):
  - finding solutions of problems (using reasoning),
  - implementation of the solutions (mundane and tedious).

Alternatives to the von Neumann approach

- How about making programming part of problem solving?
- i.e. write programs as you solve problems?
- "rapid prototyping"?
- Logic programming is derived from an abstract model (not a reorganization/abstraction of a von Neumann machine).
- In logic programming
  - program = set of axioms,
  - computation = constructive proof of a goal statement.

Logic programming: some history

- David Hilbert's program (early 20th century): formalize all mathematics using a finite, complete, consistent set of axioms.
- Kurt Gödel's incompleteness theorem (1931): any theory containing arithmetic cannot prove its own consistency.
- Alonzo Church and Alan Turing (independently, 1936): undecidability - no mechanical method to decide truth (in general).

Logic programming: some history

- Alan Robinson (1965): the resolution method for first order logic (i.e. machine reasoning in first order logic).

Logic programming: some history

- Alan Robinson (1965): the resolution method for first order logic (i.e. machine reasoning in first order logic).
- Robert Kowalski (1971): procedural interpretation of Horn clauses, i.e. computation in logic.

Logic programming: some history

- Alan Robinson (1965): the resolution method for first order logic (i.e. machine reasoning in first order logic).
- Robert Kowalski (1971): procedural interpretation of Horn clauses, i.e. computation in logic.
- Alan Colmerauer (1972): Prolog (PROgrammation en LOGique).prover.

Logic programming: some history

- Alan Robinson (1965): the resolution method for first order logic (i.e. machine reasoning in first order logic).
- Robert Kowalski (1971): procedural interpretation of Horn clauses, i.e. computation in logic.
- Alan Colmerauer (1972): Prolog (PROgrammation en LOGique).prover.
- David H.D. Warren (mid-late 1970's): efficient implementation of Prolog.

Logic programming: some history

- Alan Robinson (1965): the resolution method for first order logic (i.e. machine reasoning in first order logic).
- Robert Kowalski (1971): procedural interpretation of Horn clauses, i.e. computation in logic.
- Alan Colmerauer (1972): Prolog (PROgrammation en LOGique).prover.
- David H.D. Warren (mid-late 1970's): efficient implementation of Prolog.
- 1981 Japanese Fifth Generation Computer project: project to build the next generation computers with advanced AI capabilities (using a concurrent Prolog as the programming language).

Applications of logic programming

- Symbolic computation:

Applications of logic programming

- Symbolic computation:
  - relational databases,

Applications of logic programming

- Symbolic computation:
  - relational databases,
  - mathematical logic,

Applications of logic programming

- Symbolic computation:
  - relational databases,
  - mathematical logic,
  - abstract problem solving,

Applications of logic programming

- Symbolic computation:
  - relational databases,
  - mathematical logic,
  - abstract problem solving,
  - natural language understanding,

Applications of logic programming

- Symbolic computation:
  - relational databases,
  - mathematical logic,
  - abstract problem solving,
  - natural language understanding,
  - symbolic equation solving,

Applications of logic programming

- Symbolic computation:
  - relational databases,
  - mathematical logic,
  - abstract problem solving,
  - natural language understanding,
  - symbolic equation solving,
  - design automation,

Applications of logic programming

- Symbolic computation:
  - relational databases,
  - mathematical logic,
  - abstract problem solving,
  - natural language understanding,
  - symbolic equation solving,
  - design automation,
  - artificial intelligence,

Applications of logic programming

- Symbolic computation:
  - relational databases,
  - mathematical logic,
  - abstract problem solving,
  - natural language understanding,
  - symbolic equation solving,
  - design automation,
  - artificial intelligence,
  - biochemical structure analysis, etc.

Aplications of logic programming (cont'd)

- Industrial applications:

Aplications of logic programming (cont'd)

- Industrial applications:
  - aviation:

Aplications of logic programming (cont'd)

- Industrial applications:
  - aviation:
    - SCORE - a longterm aiport capacity management system for coordinated airports (20% of air traffic worldwide, according to `www.pdc-aviation.com`)

Aplications of logic programming (cont'd)

- Industrial applications:
  - aviation:
    - SCORE - a longterm aiport capacity management system for coordinated airports (20% of air traffic worldwide, according to `www.pdc-aviation.com`)
    - FleetWatch - operational control, used by 21 international air companies.

Aplications of logic programming (cont'd)

- Industrial applications:
  - aviation:
    - SCORE - a longterm aiport capacity management system for coordinated airports (20% of air traffic worldwide, according to `www.pdc-aviation.com`)
    - FleetWatch - operational control, used by 21 international air companies.
  - personnel planning: StaffPlan (airports in Barcelona, Madrid; Hovedstaden region in Denmark).

Aplications of logic programming (cont'd)

- Industrial applications:
  - aviation:
    - SCORE - a longterm aiport capacity management system for coordinated airports (20% of air traffic worldwide, according to `www.pdc-aviation.com`)
    - FleetWatch - operational control, used by 21 international air companies.
  - personnel planning: StaffPlan (airports in Barcelona, Madrid; Hovedstaden region in Denmark).
  - information management for disasters: ARGOS - crisis management in CBRN (chemical, biological, radiological and nuclear) incidents – used by Australia, Brasil, Canada, Ireland, Denmark, Sweden, Norway, Poland, Estonia, Lithuania and Montenegro.

Rezolvarea problemelor folosind Prolog

- Programarea în Prolog:
  - în loc de a scrie secvențe de pași pentru a rezolva o problemă,
  - se descriu fapte și relații cunoscute, apoi se pun întrebări.
- Utilizare Prolog pentru a rezolva probleme care includ obiecte și relații între obiecte.
- Exemple:
  - Obiecte: "John", "book", "jewel", etc.
  - Relații: "John are o carte", "Mariei ii place de Ion".
  - Reguli: "Două persoane sunt surori dacă amandouă sunt femei și dacă au aceiași părinți".

Rezolvarea problemelor folosind Prolog

- Cum se rezolvă problemele în Prolog:
  - se declară fapte despre obiecte și relatiile dintre ele,
  - se definesc reguli despre obiecte și relațiile dintre ele,
  - se pun întrebări despre obiecte și relațiile dintre ele.
- Programarea în Prolog: o conversație cu interpretorul Prolog.

Fapte

- Scrierea unui fapt în Prolog:

```
likes(johnny, mary).
```

  - Relatiile (predicatele) și obiectele se scriu cu litere mici.
  - Prolog foloseste (mai mult) notatia prefix (dar exista si exceptii).
  - Faptele se încheie cu "." (full stop).
- Un model este construit in Prolog, iar faptele descriu modelul.
- Utilizatorul trebuie să fie con ctient că următoarele:

```
likes(john, mary).
likes(mary, john).
```

  nu sunt identice.
- Putem avea un număr arbitrar de argumente.
- Notatie: likes /2 denotă un predicat binar.
- Faptele sunt parte a bazei de cunoștințe Prolog (knowledge base).

Întrebari

- O intrebare in Prolog:

```
?- owns(mary, book).
```

  Prolog cauta in baza de cunostinte fapte care fac matching (se potrivesc) cu întrebarea:
- Prolog raspunde `true` daca :
  - predicatele sunt la fel,
  - argumentele sunt la fel,
- În caz contrat raspunsul este `false`:
  - doar ceea ce este cunoscut este true ("closed world assumption"),
  - Atenție: `false` nu inseamna ca raspunsul este false (mai degraba "nu e dedus din baza de cunostinte").

Variabile

- Ganditi-va la variabile in logica predicatelor.
- In loc de:

```
?- likes(john, mary).
?- likes(john, apples).
?- likes(john, candy).
```

  intrebati ceva de genul "What does John like?" (vrem tot ce ii place lui John).
- Variabilele sunt obiecte care urmeaza a fi determinate de Prolog.
- Variabilele pot fi:
  - instantiate
  - neinstantiate
- In Prolog variablele incep cu LITERE MARI:

```
?- likes(john, X).
```

Exemplu in Prolog

- Se considera urmatoarele fapte in baza de cunostinte Prolog:

```
...
likes(john, flowers).
likes(john, mary).
likes(paul, mary).
...
```

- La intrebarea

```
?-likes(john, X).
```

  Prolog va raspunde

```
X = flowers
```

  si asteapta urmatoarele instructiuni.

Raspunsul Prolog

- Prolog cauta in baza de cunostinte un fapt care potriveste cu intrebarea,

Raspunsul Prolog

- Prolog cauta in baza de cunostinte un fapt care potriveste cu intrebarea,
- cand o potrivire e gasita, aceasta se marcheaza.

Raspunsul Prolog

- Prolog cauta in baza de cunostinte un fapt care potriveste cu intrebarea,
- cand o potrivire e gasita, aceasta se marcheaza.
- daca utilizatorul apasa "`Enter`", atunci cautarea se termina,

Raspunsul Prolog

- Prolog cauta in baza de cunostinte un fapt care potriveste cu intrebarea,
- cand o potrivire e gasita, aceasta se marcheaza.
- daca utilizatorul apasa "`Enter`", atunci cautarea se termina,
- daca utilizatorul apasa ";" apoi "`Enter`", Prolog cauta dupa o noua potrivire, incepand din locul marcat, si cu variabile neinstantiate in intrebare.

Raspunsul Prolog

- Prolog cauta in baza de cunostinte un fapt care potriveste cu intrebarea,
- cand o potrivire e gasita, aceasta se marcheaza.
- daca utilizatorul apasa "`Enter`", atunci cautarea se termina,
- daca utilizatorul apasa ";" apoi "`Enter`", Prolog cauta dupa o noua potrivire, incepand din locul marcat, si cu variabile neinstantiate in intrebare.
- In exemplul precedent, inca doua "`; Enter`" vor determina Prolog sa raspunda:

```
X = mary.
false
```

Raspunsul Prolog

- Prolog cauta in baza de cunostinte un fapt care potriveste cu intrebarea,
- cand o potrivire e gasita, aceasta se marcheaza.
- daca utilizatorul apasa "`Enter`", atunci cautarea se termina,
- daca utilizatorul apasa ";" apoi "`Enter`", Prolog cauta dupa o noua potrivire, incepand din locul marcat, si cu variabile neinstantiate in intrebare.
- In exemplul precedent, inca doua "`; Enter`" vor determina Prolog sa raspunda:

```
X = mary.
false
```

- Cand nicio potrivire nu se mai gaseste in baza de cunostinte, Prolog raspunde `false`.

Conjunctii: intrebari mai complexe

- Consideram urmatoarele fapte:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

Conjunctii: intrebari mai complexe

- Consideram urmatoarele fapte:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

- Si intrebarea:

```
?- likes(john, mary), likes(mary, john).
```

Conjunctii: intrebari mai complexe

- Consideram urmatoarele fapte:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

- Si intrebarea:

```
?- likes(john, mary), likes(mary, john).
```

  - citim "does john like mary and does mary like john?"

Conjunctii: intrebari mai complexe

- Consideram urmatoarele fapte:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

- Si intrebarea:

```
?- likes(john, mary), likes(mary, john).
```

  - citim "does john like mary and does mary like john?"
  - Prolog va raspunde false: cauta pentru fiecare goal (toate goal-urile trebuie sa fie satisfacute, iar daca nu, atunci nu va reusi, deci raspunsul va fi false).

Conjunctii: intrebari mai complexe

- La intrebarea:

```
?- likes(mary, X), likes(john, X).
```

Conjunctii: intrebari mai complexe

- La intrebarea:

```
?- likes(mary, X), likes(john, X).
```

  - Prolog: incearca sa satisfaca prima conditie (daca e satisfacuta, atunci marcheaza valoarea), apoi incearca sa satisfaca a doua conditie (daca e satisfacuta, atunci marcheaza valoarea).

Conjunctii: intrebari mai complexe

- La intrebarea:

```
?- likes(mary, X), likes(john, X).
```

  - Prolog: incearca sa satisfaca prima conditie (daca e satisfacuta, atunci marcheaza valoarea), apoi incearca sa satisfaca a doua conditie (daca e satisfacuta, atunci marcheaza valoarea).
  - Daca intr-un punct esueaza, atunci se intoarce la ultima valoare marcata si incearca alternative.

Exemple: conjunctiii, backtracking

Modul in care Prolog calculeaza raspunsul la intrebarea urmatoare este reprezentat astfel:

- In Figura 1, prima conditie e satisfacuta, Prolog urmeaza sa gaseasca o potrivire pentru a doua conditie (cu variabila instantiata).
- Daca esueaza sa gaseasca in baza de cunostinte o potrivire atunci incepe procesul de backtracking, in Figura 2.
- Noua alternativa incercata reuseste pentru ambele conditii, vezi Figura 3.

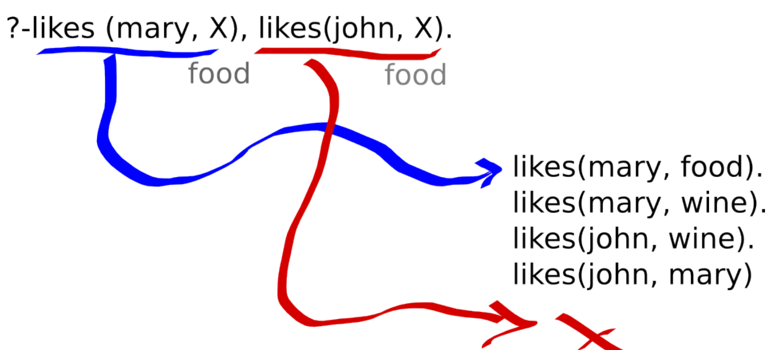

Figure: Esecul pentru a doua conditie cauzeaza backtracking.

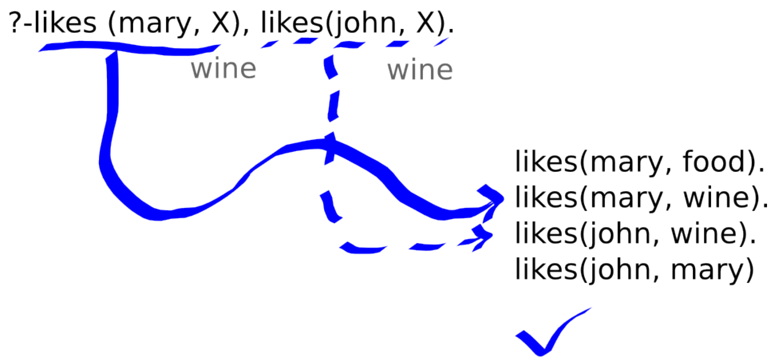

Figure: Success cu o alternativa instantiata.

Reguli

- "John likes all people" se poate reprezenta:

Reguli

- "John likes all people" se poate reprezenta:

```
likes(john, alfred).
likes(john, bertrand).
likes(john, charles).
likes(john, david).
...
```

dar doar atat?!!!

Reguli

- "John likes all people" se poate reprezenta:

```
likes(john, alfred).
likes(john, bertrand).
likes(john, charles).
likes(john, david).
...
```

dar doar atat?!!!

```
likes(john, X).
```

dar ar trebui sa fie doar pentru oameni!!!

Reguli

- "John likes all people" se poate reprezenta:

```
likes(john, alfred).
likes(john, bertrand).
likes(john, charles).
likes(john, david).
...
```

  dar doar atat?!!!

```
likes(john, X).
```

  dar ar trebui sa fie doar pentru oameni!!!

- Introducem reguli: "John likes any object, but only that which is a person" este o regula despre ceva (cineva) pe care John place.

Reguli

- "John likes all people" se poate reprezenta:

```
likes(john, alfred).
likes(john, bertrand).
likes(john, charles).
likes(john, david).
...
```

  dar doar atat?!!!

```
likes(john, X).
```

  dar ar trebui sa fie doar pentru oameni!!!

- Introducem reguli: "John likes any object, but only that which is a person" este o regula despre ceva (cineva) pe care John place.
- Regulile exprima: un fapt depinde de alt fapt.

Reguli ca definitii

- Regulile pot fi folosite pentru a exprima “definitii”.
- Exemplu:

  “X is a bird if X is an animal and X has feathers.”
- Exemplu:

  “X is a sister of Y if X is female and X and Y have the same parents.”
- Atentie! Notiunea de “definitie” aici nu e aceeasi cu notiunea de definitie din logica:
  - asemenea definitii permit detectarea predicatelor in capul unei reguli,
  - dar pot fi si alte modalitati (ex. alte reguli cu acelasi cap) pentru a detecta asemenea predicate,
  - pentru a avea o definitie completa ar trebui sa avem “iff” in loc de “if”.
- Regulile sunt afirmatii generale despre obiecte si relatiile dintre ele (in general variabilele apar in reguli, dar nu intotdeauna).

Reguli in Prolog

- Regulile in Prolog au head si body.
- Corpul (the body) unei reguli descrie conditiile ce trebuie sa fie satisfacute pentru ca head-ul sa fie true.
- Exemplu:

```
likes(john, X) :-
        likes(X, wine).
likes(john, X) :-
        likes(X, wine), likes(X, food).
likes(john, X) :-
        female(X), likes(X, wine).
```

- Atentie! Variabilele in reguli diferite sunt diferite.

Exemplu (royals)

- Baza de cunoștințe:

```
male(albert).
male(edward).
female(alice).
female(victoria).
parents(alice, albert, victoria).
parents(edward, albert, victoria).
sister_of(X, Y):-
                female(X),
                parents(X, M, F).
                parents(Y, M, F).
```

- Întrebări:

```
?-sister_of(alice, edward).
?-sister_of(alice, X).
```

Exercițiu (thieves)

- Se consideră următoarele:

```
/*1*/  thief(john).

/*2*/  likes(mary, food).
/*3*/  likes(mary, wine).
/*4*/  likes(john, X):- likes(X, wine).

/*5*/  may_steal(X, Y) :-
                thief(X), likes(X, Y).
```

- Explicați pentru întrebarea următoare

```
?- may_steal(john, X).
```

cum e executată în Prolog.

- Prolog programs are built from terms (written as strings of characters).
- The following are terms:
  - constants,
  - variables,
  - structures.

Constants

- Constants are simple (basic) terms.
- They name specific things or predicates (no functions in Prolog).
- Constants are of 2 types:
  - atoms,
  - numbers: integers, rationals (with special libraries), reals (floating point representation).

Examples of atoms

- atoms:
  - likes ,
  - a (lowercase letters),
  - =,
  - −−>,
  - 'Void' (anything between single quotes),
  - george_smith (constants may contain underscore),
- not atoms:
  - 314a5 (cannot start with a number),
  - george−smith (cannot contain a hyphen),
  - George (cannot start with a capital letter),
  - _something (cannot start with underscore).

Variables

- Variables are simple (basic) terms,
- written in uppercase or starting with underscore _,
- Examples: X, Input, _something, _ (the last one called anonymous variable).
- Anonymous variables need not have consistent interpretations (they need not be bound to the same value):

```
?-likes(_,john). % does anybody like John?
?-likes(_,_).    % does anybody like anybody?
```

Structures

- Structure are compound terms, single objects consisting of collections of objects (terms),
- they are used to organize the data.
- A structure is specified by its functor (name) and its components

```
owns(john,book(wuthering_heights,bronte)).
book(wuthering_heights,author(emily,bronte)).

?-owns(john,book(X,author(Y,bronte))).
% does John own a book (X) by Bronte(Y,bronte)?
```

Characters in Prolog

- Characters:
  - A-Z
  - a-z
  - 0-9
  - + - * / \ ~ ^ < > : .
  - ? @ # $ &
- Characters are ASCII (printing) characters with codes greater than 32.
- Remark: ' ' allows the use of any character.

Arithmetic operators

- Arithmetic operators:
  - $+$,
  - $-$,
  - $*$,
  - $/$,
- $+(x, *(y, z))$ is equivalent with $x + (y \cdot z)$
- Operators do not cause evaluation in Prolog.
- Example: 3+4 (structure) does not have the same meaning with 7 (term).
- X is 3+4 causes evaluation (is represents the evaluator in Prolog).
- The result of the evaluation is that X is assigned the value 7.

Parsing arithmetic expressions

- To parse an arithmetic expression you need to know:
  - The position:
    - infix: $x + y$, $x * y$
    - prefix $-x$
    - postfix $x!$
  - Precedence: $x + y * z$?
  - Associativity: What is $x + y + z$? $x + (y + z)$ or $(x + y) + z$?

- Each operator has a precedence class:
  - 1 - highest
  - 2 - lower
  - ...
  - lowest
- $* \, /$ have higher precedence than $+ \, -$
- $8/2/2$ evaluates to:
  - 8 $(8/(2/2))$ - right associative?
  - or 2 $((8/2)/2)$ - left associative?
- Arithmetic operators are left associative.

The unification predicate '='

- = - infix built-in predicate.

```
?−X = Y.
```

  Prolog will try to match(unify) X and Y, and will answer true if successful.
- In general, we try to unify 2 terms (which can be any of constants, variables, structures):

```
?−T1 = T2.
```

- Remark on terminology: while in some Prolog sources the term "matching" is used, note that in the (logic) literature matching is used for the situation where one of the terms is ground (i.e. contains no variables). What = does is unification.

The unification procedure

Summary of the unification procedure ?− T1 = T2:

- If T1 and T2 are identical constants, success (Prolog answers true);
- If T1 and T2 are uninstantiated variable, success (variable renaming);
- If T1 is an uninstantiated variable and T2 is a constant or a structure, success, and T1 is instantiated with T2;
- If T1 and T2 are instantiated variables, then decide according to their value (they unify - if they have the same value, otherwise not);
- If T1 is a structure: $f(X_1, X_2, ..., X_n)$ and T2 has the same functor (name): $f(Y_1, Y_2, ..., Y_n)$ and the same number of arguments, then unify these arguments recursively ($X_1 = Y_1$, $X_2 = Y_2$, etc.). If all the arguments unify, then the answer is true, otherwise the answer is false (unification fails);
- In any other case, unification fails.

Occurs check

- Consider the following unification problem:

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

Occurs check

- Consider the following unification problem:

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

- Answer of Prolog:

```
X = f(**).
```

Occurs check

- Consider the following unification problem:

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

- Answer of Prolog:

```
X = f(**).
```

```
X = f(X).
```

Occurs check

- Consider the following unification problem:

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

- Answer of Prolog:

```
X = f(**).

X = f(X).
```

- In fact this is due to the fact that according to the unification procedure, the result is $X = f(X) = f(f(X)) = ...= f(f(...(f(X)...)))$ - an infinite loop would be generated.

- Unification should fail in such situations.
- To avoid them, perform an occurs check: If T1 is a variable and T2 a structure, in an expression like T1 = T2 make sure that T1 does not occur in T2.
- Occurrence check is deactivated by default in most Prolog implementations (is computationally very expensive) - Prolog trades correctness for speed.
- A predicate complementary to unification:
  - \= succeeds only when = fails,
  - i.e. T1 \= T2 cannot be unified.

Built-in predicates for arithmetic

- Prolog has built-in numbers.
- Built-in predicates on numbers include:

  $X = Y$,
  $X \backslash= Y$,
  $X < Y$,
  $X > Y$,
  $X =< Y$,
  $X >= Y$,

  with the expected behaviour.
- Note that variables have to be instantiated in most cases (with the exception of the first two above, where unification is performed in the case of uninstantiation).

The arithmetic evaluator `is`

- Prolog also provides arithmetic operators (functions), e.g.: $+$, $-$, $*$, $/$, mod, rem, abs, max, min, random, floor, ceiling etc, but these cannot be used directly for computation(2+3 means 2+3, not 5) - expressions involving operators are not evaluated by default.
- The Prolog evaluator `is` has the form:

```
X is Expr.
```

  where X is an uninstantiated variable, and Expr is an arithmetic expression, where all variables must be instantiated (Prolog has no equation solver).

Example (with arithmetic(1))

```
reigns(rhondri, 844, 878).
reigns(anarawd, 878, 916).
reigns(hywel_dda, 916, 950).
reigns(lago_ap_idwal, 950, 979).
reigns(hywel_ap_ieuaf, 979, 985).
reigns(cadwallon, 985, 986).
reigns(maredudd, 986, 999).
prince(X, Y):-
        reigns(X, A, B),
        Y >= A,
        Y =< B.

?- prince(cadwallon, 986).
true
?- prince(X, 979).
X = lago_ap_idwal ;
X = hywel_ap_ieuaf
```

Example (with arithmetic(2))

```
pop(place1, 203).
pop(place2, 548).
pop(place3, 800).
pop(place4, 108).

area(place1, 3).
area(place2, 1).
area(place3, 4).
area(place4, 3).
density(X, Y):-
      pop(X, P),
      area(X, A),
      Y is P/A.

?-density(place3, X).
X = 200
true
```

- In this lecture the following were discussed:
  - asserting facts about objects,
  - asking questions about facts,
  - using variables, scopes of variables,
  - conjunctions,
  - an introduction to backtracking (in examples),
  - Prolog syntax: terms (constants, variables, structures),
  - Arithmetic in Prolog,
  - Unification procedure,
  - Subtle point: occurs check.

- All things SWIProlog can be found at `http://www.swi-prolog.org`.
- Install SWI-Prolog and try out the examples in the lecture.
- Read: Chapter 1 and Chapter 2 (including exercises section) of [Clocksin and Mellish, 2003].

Ben-Ari, M. (2001).
Mathematical Logic for Computer Science.
Springer Verlag, London, 2nd edition.

Clocksin, W. F. and Mellish, C. S. (2003).
Programming in Prolog.
Springer, Berlin, 5th edition.

Nilsson, U. and Maluszynski, J. (2000).
Logic, Programming and Prolog.
copyright Ulf Nilsson and Jan Maluszynski, 2nd edition.