Lecture 8
SLD-Resolution and PROLOG

13/12/2006

Overview

- SLD-Resolution
- PROLOG by example
- Lists,
- slowsort
- built-in arithmetic and other useful operators

- Next time: negation and cut

Resolution:

- Let C be a Horn clause

  $A \leftarrow A_1, \ldots A_n$

  and G be a goal

  $\leftarrow B_1, \ldots, B_m$

  where G and C have no variables in common.

  Let further θ be an mgu of A and B_i for some $1 \leq i \leq m$

  Then the goal

  $\leftarrow B_1\theta, \ldots B_{i-1}\theta, A_1\theta, \ldots A_n\theta, B_{i+1}\theta, B_m\theta$

  is called resolvent of G and C.

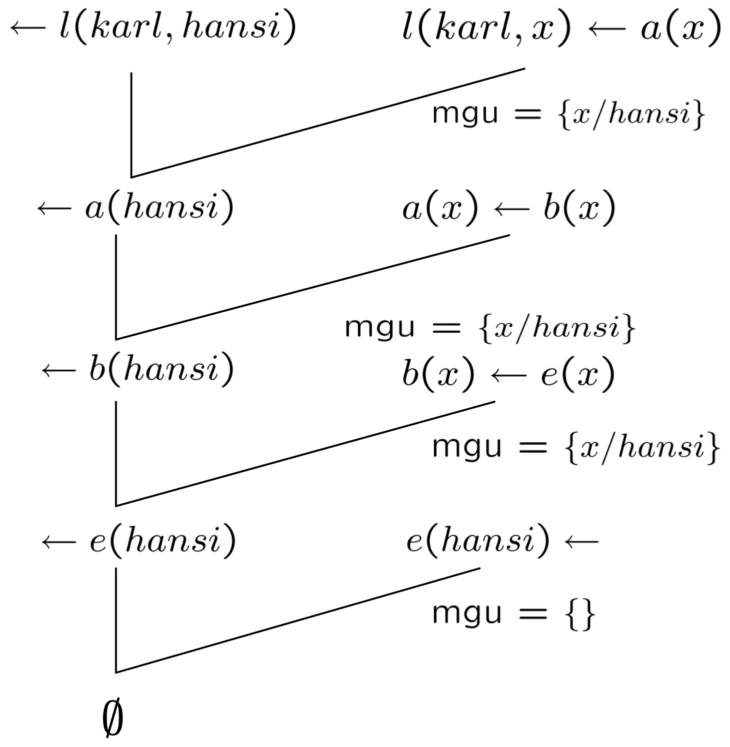
Remark: There is also a general resolution for full Clause logic, but for Horn programs the above is sufficient…

A refutation proof by resolution

$S = \{a(x) \leftarrow b(x),\ b(x) \leftarrow e(x),\ l(karl,x) \leftarrow a(x),\ b(franz) \leftarrow,\ e(hansi) \leftarrow\}$

- Does Karl love Hansi?

G:



Attention:

Why goal and clause may not have variables in common:

- try resolving:

$$\begin{tikzcd} G : \leftarrow p(x, f(x)) \arrow[rrd, dash] & & C : p(f(x), y) \leftarrow q(x, y) \arrow[d, dash] \\ & & \text{Not unifiable!} \end{tikzcd}$$

This can however always be solved by renaming the variables in the clause to resolve:

$$\begin{tikzcd} G : \leftarrow p(x, f(x)) \arrow[d, dash] & C : p(f(x_1), y_1) \leftarrow q(x_1, y_1) \arrow[dl, dash, "\text{mgu} = \{x/f(x_1), y_1/f(f(x_1))\}"] \\ \leftarrow q(x_1, f(f(x_1))) & \end{tikzcd}$$

Desirable properties: We want to show that resolution is a sound and complete refutation procedure!

- **Soundness:** Whenever a resolution refutation is found from a set of clauses S, then S is contradictory.
- **Completeness:** Whenever S is contradictory, there exists a resolution proof.

… for this we will use a declarative characterization of the set of all consequences by the so-called **least Herbrand Model**

The least Herbrand Model

- ***Model Intersection Property***:

  Let P be a Horn program and $\{M_i\}_{i \in I}$ be a non-empty set of Herbrand Models for P: Then $\bigcap_{i \in I} M_i$ is an Herbrand model of P

- Every Horn Program P which only consists of rules with exactly one head atom (short: definite program) has a Herbrand Model. Thus: There exists a so called **least Herbrand Model** M_P**.**

- *Let P be a definite program. Then* $\boldsymbol{M_P} = \{ A \in B_P : A$ is a logical consequence of $P\}$.

- **This least Herbrand Model can be characterized by a fixpoint operator !**

The immediate consequence operator:

Let I *be a Herbrand interpretation and P a definite program:*

$T_P(I)$ =

$\{A \in B_P : A \leftarrow A_1, \ldots A_n$ is a ground instance in of a rule in P such that $A_1, \ldots, A_n \in I\}$

is called the immediate consequence operator.

Clearly $T_P(I)$ is monotonic, i.e. $I_1 \subseteq I_2 \Rightarrow T_P(I_1) \subseteq T_P(I_2)$.

Fixpoint characterization of the Least Herbrand Model:

- The least Herbrand model can be characterized in terms of $T_{P:}$

  Let P be a definite program: Then

$$M_P = lfp(T_P) = T_P^{\infty}(\emptyset)$$

Remark: for a finite Herbrand universe, this is finitely computable!

Example:

- $P = \{a(x) \leftarrow b(x),\ b(x) \leftarrow e(x),\ l(karl,x) \leftarrow a(x),\ b(franz) \leftarrow\ ,\ e(hansi) \leftarrow\}$

$T_P(\emptyset) \quad = \{b(franz),\ e(hansi)\}$

$T_P^2(\emptyset) \quad = \{b(franz),\ e(hansi),\ a(franz),\ b(hansi)\}$

$T_P^3(\emptyset) \quad = \{b(franz),\ e(hansi),\ a(franz),\ b(hansi),\ l(karl,franz),\ a(hansi)\}$

$T_P^4(\emptyset) \quad = \{b(franz),\ e(hansi),\ a(franz),\ b(hansi),\ l(karl,franz),\ a(hansi),\ l(karl,hansi)\}$

$T_P^5(\emptyset) \quad = T_P^4(\emptyset)$

i.e. $lfp(T_P) = T_P^4(\emptyset) = M_P$

A simple example where the least Herbrand model is infinite:

- $P = \{p(a) \leftarrow,\ p(f(x)) \leftarrow p(x)\}$

$lfp(T_P)$ = {p(a), p(f(a), p(f(f(a)) … }

SLD resolution and its properties:

- We will now focus on resolution for a single Goal clause and a definite program:
- SLD – resolution: **L**inear resolution with **S**election function for **D**efinite clauses.

Soundness:

We want to show that whenever we can find a SLD-refutation from a definite program P plus a goal G, then G is a logical consequence of P, i.e. G is in the least Herbrand Model $\boldsymbol{M_{P}}$.

Completeness:

We want to show that for a definite program P plus a goal G, whenever G is a logical consequence if P, i.e. G is in the least Herbrand Model $\boldsymbol{M_P}$ then we can find a SLD-refutation.

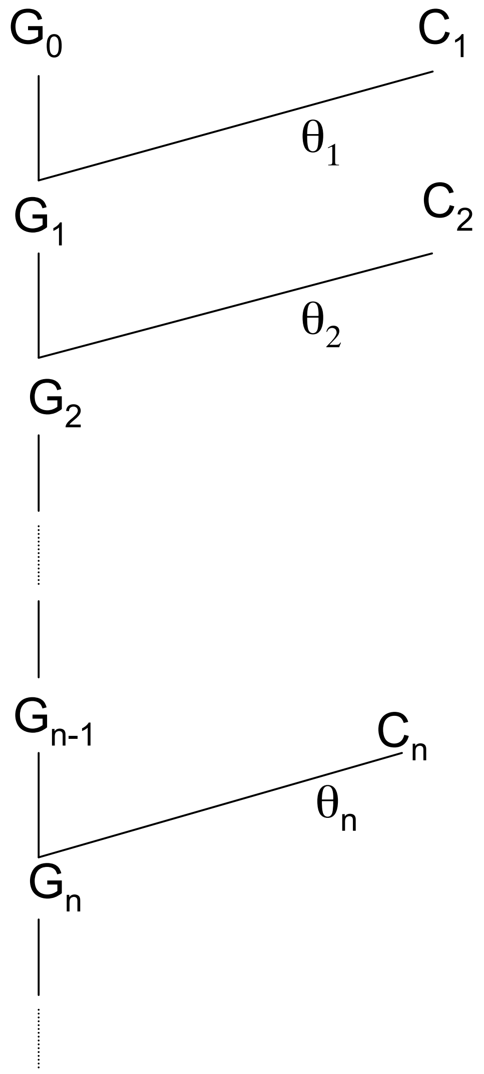
Definitions: SLD-Derivation and SLD-Refutation:

- **Definition:** Let P be a definite program and G_0 be a definite Goal. Then an **SLD*-derivation** of $P \cup \{G_0\}$ consists of a (finite or infinite) sequence $G_0, G_1, \ldots$ of goals, a sequence of clauses $C_1, C_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that G_{i+1} is the resolvent of G_i and C_{i+1} using θ_{i+1}.

- **Definition:** A finite SLD-derivation of $P \cup \{G_0\}$ which has the empty clause $\emptyset$ as the last goal G_n is called **SLD-refutation** of length *n.*

* ***Remark:*** The **S** in SLD-resolution refers to which atom/subgoal is selected for resolution in each step.

Notation:



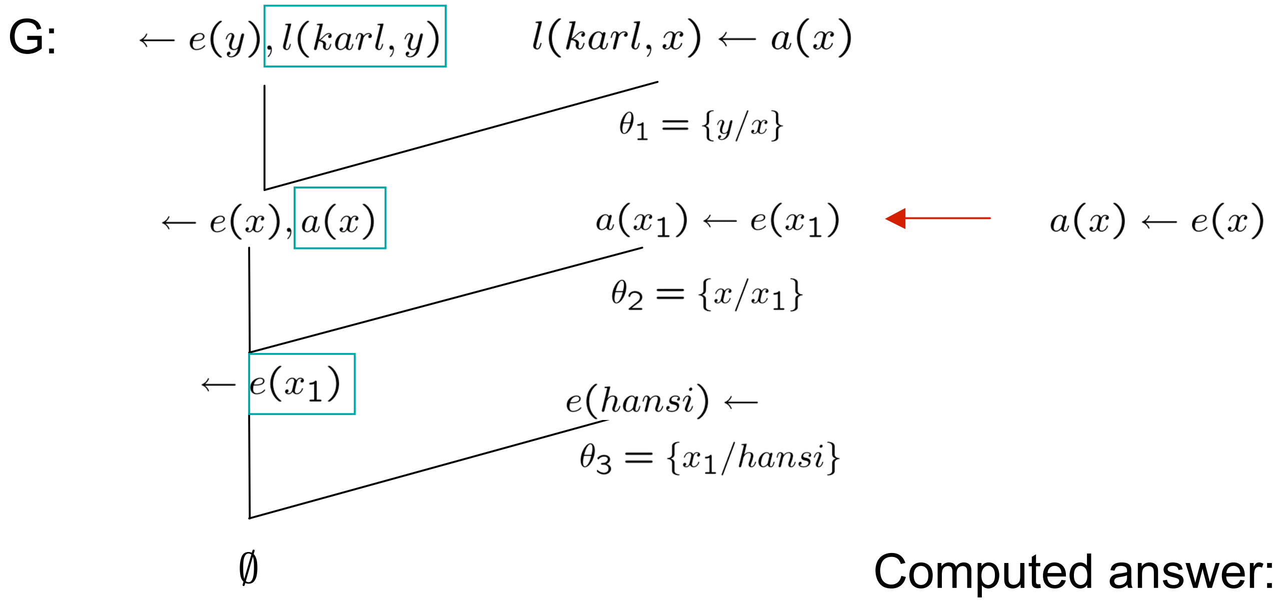
Finite, infinite, successful, failed SLD-derivations:

- SLD-derivations can be *finite* or *infinite*
- A *failed* SLD-derivation is on that ends in a non-empty goal such that the selected atom does not unify with the head of any program clause.

Computed answers:

- Definition: A **computed answer** θ for P ∪ {G} is the substitution obtained by restricting the composition of $\theta_1, \ldots, \theta_n$ from an SLD-refutation of P ∪ {G} to the variables of G.

Back to our example: Which Eagles does Karl love?
P = $\{a(x) \leftarrow b(x),\ b(x) \leftarrow e(x),\ l(karl,x) \leftarrow a(x),\ b(franz) \leftarrow,\ \ e(hansi) \leftarrow\}$
G = $\leftarrow e(y),\ l(karl,y)$


G:
$\leftarrow e(y), \boxed{l(karl,y)}$
$l(karl,x) \leftarrow a(x)$
$\theta_1 = \{y/x\}$
$\leftarrow e(x), \boxed{a(x)}$
$a(x_1) \leftarrow e(x_1)$
$a(x) \leftarrow e(x)$
$\theta_2 = \{x/x_1\}$
$\leftarrow \boxed{e(x_1)}$
$e(hansi) \leftarrow$
$\theta_3 = \{x_1/hansi\}$
$\emptyset$

Computed answer: $\theta = \{y/hansi\}$

There are possibly several valid refutations, each of which "contains" an answer substitution (e.g. add *e(seppl)*← to P.)

Soundness of SLD resolution:

- **Proposition 1:** Every computed answer for $P \cup \{G\}$ is a correct answer
  (i.e. for $G = \leftarrow A_1, \ldots A_k$, and and a refutation of length n with mgu's $\theta_1 \ldots \theta_n$
  $\forall(A_1 \wedge \ldots \wedge A_k)\theta_1 \ldots \theta_n$ is a logical consequence of P).

- **Proof:** By inductuction on length n of the refutation.

Assume n = 1, i.e. $G = \leftarrow A_1$ and there is a unit clause $A \leftarrow$ and in P such that $A_1\theta_1 = A\theta_1$. Since
$A_1\theta_1 \leftarrow$ is an instance of a unit clause in P it follows that $\forall(A_1\theta_1)$ is a logical consequ.of P.
Next, assume that the result holds for refutations of length n-1 for goal $G_0 = \leftarrow A_1, \ldots, A_k$.
Suppose $\theta_1 \ldots \theta_n$ is the set of mgu's of a refutation of length n. Let $C_1 = A \leftarrow B_1, \ldots, B_q$
be the **first** input clause for the refutation and A_m be the selected clause in this **first**
resolution step.
Then, by induction hypothesis,

$$\forall (A_1 \ldots \wedge A_{m-1} \wedge B_1 \wedge \ \ldots \wedge B_q \wedge A_{m-1} \ldots \wedge A_k)\, \theta_1 \ldots \theta_n$$

is a logical consequence of P. Thus, if q > 0, $\forall ((B_1 \wedge \ \ldots \wedge B_q)\, \theta_1 \ldots \theta_n)$ is a log. consequence of P, which further means that also $\forall (A\, \theta_1 \ldots \theta_n) = \forall (A_m\, \theta_1 \ldots \theta_n)$ is a consequence of P and therefore also: $\forall (A_1 \ldots \wedge A_k)\, \theta_1 \ldots \theta_n$ □

Completeness of SLD-Resolution:

First for ground atoms:

Proposition 2: The set SLD-provable ground atoms for a definite program is equal to its least Herbrand model.

Proof (sketch):

$\subseteq$: follows by soundness already.

$\supseteq$: (i.e. for every atom A in the least Herbrand model we can find an SLD-refutation) this is the actual completeness result:

Suppose A is in the least Herbrand model, then there is some n such that $A \in T_P^n(\emptyset)$.

We can then show inductively that $A \in T_P^n(\emptyset)$ implies that A has a refutation of length n.

Now for definite goals:

Proposition 3: Let P be a definite program and G be a definite goal. Suppose that $P \cup \{G\}$ is unsatisfiable. Then there exists an SLD-refutation of $P \cup \{G\}$.

Proof (sketch):

Let $G = \leftarrow A_1, \ldots, A_k$. Since $P \cup \{G\}$ is unsatisfiable, G is false wrt. M_P. So some ground instance $G\theta$ is false wrt. M_P. Thus $\{A_1\theta, A_k\theta\} \in M_P$.By the above proposition, there is a SLD-refutation of $P \cup \leftarrow A_i\theta$, for each i= 1,...,k. Since each $A_i\theta$ is ground, we can combine these refutations into a refutation for $P \cup \{G\theta\}$. Thus, by the so-called lifting lemma, there also exists a refutation for $P \cup \{G\}$.

Remark: For the proof details: check [Lloyd, 1987]

Further properties of SLD resolution:

- Soundness and completeness do not depend on the selection function S (i.e. which subgoal is selected first).

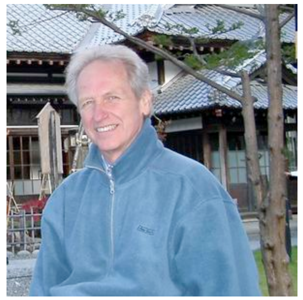
- What does this mean?

  We could (naively) implement a Breath-First search algorithm for finding a refutation proofs, which uses backtracking for finding all computed answers.

PROLOG:

- In the **early 70's** A. Colmerauer and R.A. Kowalski more or less at the same time invented PROLOG as a recursive programming language which uses the SLD-resolution principle in a slightly more procedural way:

Alain Colmerauer


Robert Kowalski

- Colmerauer and his group at the University of Marseille-Aix developed a specialized theorem prover, which they called PROLOG (for “Programacion en Logique” or “Programming in Logic”), embodied Kowalski's procedural interpretation

PROLOG

- For efficiency reasons, PROLOG systems make the following simplifications wrt. Full SLD resolution:
  1) No occur check
  2) Depth-first search

- PROLOG is rather a procedural implementation of the resolution principle than really declarative: Efficient, procedural implementation vs. declarativity and completeness

- In the following: Some examples and problems

- Next time:
  – Programming in PROLOG,
  – Termination, Negation as finite failure
  – Cut, …

The problem with the occur-check:

- Occur-check can be very expensive, that's why it is ommitted in most PROLOG implementations.

- Problem: wrong answers or infinite loops

Examples:

1) P = testme $\leftarrow$ p(x,x)
   p(x,f(x)) $\leftarrow$
   G = $\leftarrow$ testme

"mgu" = {x/f(x)} … circular binding!

PROLOG will wrongly answer "yes" here!
➔ SOUNDNESS OF SLD-Resolution is sacrificed for efficiency!

2) P = testme $\leftarrow$ p(x,x)
   p(x,f(x)) $\leftarrow$ p(x,x)
   G = $\leftarrow$ testme

PROLOG will run into an infinite loop here, not finding the correct answer "no"!
➔ COMPLETENESS OF SLD-Resolution is sacrificed for efficiency!

- See exercises!

The problem with Depth-First-Search

- PROLOG systems always select the first subgoal, and try to unify with the clauses of the program in a first-to-last, depth-first fashion. Backtracking is used, when no solution has been found.
- Problem: Termination!
- Simple example:

$$P = \boxed{\begin{array}{l} test \leftarrow p(x) \\ p(a) \leftarrow \\ p(x) \leftarrow p(f(x)) \end{array}} \quad \text{vs.} \quad P = \boxed{\begin{array}{l} test \leftarrow p(x) \\ p(x) \leftarrow p(f(x)) \\ p(a) \leftarrow \end{array}}$$

$$G = \leftarrow test \qquad\qquad G = \leftarrow test$$

Overview

- SLD-Resolution
- **PROLOG by example**
- Lists and built-ins
- Cut and Negation

PROLOG +/-

- The Log.Prog.Paradigm of Prolog has advantages and drawbacks against "pure" Logic Programming:

Prolog:

+ simplicity, conciseness (simple backtracking, easy to understand)

+ useful extensions by arithmetics, datatypes, etc.

- restrictions of simple control mechanism (not fully declarative)

cf. [Apt ,2001]

Basic Syntax and Programs:

- As in clausal notation we distinguish different kinds of clauses:
  - Facts

    b(hansi)$\leftarrow$ `b(hansi).`
  - Rules

    parrot(x) $\leftarrow$ *bird(x),colorful(x).* `parrot(X) :-bird(X),colorful(X).`
  - Queries (Goals)

    $\leftarrow$ *l(karl,hansi)* `?- l(karl,hansi).`

- A program is a set of facts and rules.

- You can store a program in a file
  (usually with extension ".pl" (SWI-Prolog), or ".P" (XSB-Prolog))

More on PROLOG syntax

- Variables start with a capital letter or with '_' (more on that later), all other combinations of letters, numbers are constants.
- Constants start with a lower case letter or can alternatively be included in single quotes `axel,'Axel'.`
- Single quotes and double quotes are something different!!!
- Body elements of rules are separated by ','
- Each rule/fact/query ends with a fullstop '.'
- $\leftarrow$ is written '`:-`' in rules, '`?-`' in queries and can be skipped for rules with empty body (facts).
- Prolog does not make a distinction between function symbols and predicate symbols on a syntactical level! (more on that later), see also slides 17/18.
- Comments start with '`%`' (the rest of the line will be ignored).

A small program…

… which we will subsequently extend:

```
% Some relations between Austrian emporers
% in the 17th and 18th century:

child_of(joseph_I, leopold_I).
child_of(karl_VI, leopold_I).
child_of(maria_theresia,karl_VI).
child_of(joseph_II, maria_theresia).
child_of(joseph_II,franz_I).
child_of(leopold_II,maria_theresia).
child_of(leopold_II,franz_I).
child_of(marie_antoinette,maria_theresia).
child_of(franz_II,leopold_II).

male(franz_I).
male(franz_II).
male(joseph_I).
male(joseph_II).
male(karl_VI).
male(leopold_I).
male(leopold_II).

female(maria_theresia).
female(maria_antoinette).
```

emporers.pl

A small program…

Our program is currently simply a set of facts.

This can be viewed as a relational database,

with tables:

`child_of`	
Child	**Parent**
Joseph I	Leopold I
Karl VI	Leopold I
Maria Theresia	Karl VI
Joseph II	Maria Theresia
Joseph II	Franz I
Leopold II	Maria Theresia
Leopold II	Franz I
Marie Antoinette	Maria Theresia
Franz II	Leopold II

`female`
Name
Maria Theresia
Marie Antoinette

`male`
Name
Joseph I
Joseph II
Franz I
Franz II
Karl VI
Leopold I
Ferdinand I

- Now we want to query this database, therefore we use queries (i.e. goals).

Start PROLOG and load Program

- Download/install: http://www.swi-prolog.org/

- From the command line:

```
$ swipl
Welcome to SWI-Prolog (Multi-threaded, Version 5.4.2)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

- Load a Program with

```
    ?- [name].
```

  or

```
    ?- compile('name.pl').
```

  or

```
    ?- consult('name.pl').
```

- Now you can ask queries on the program

- Leave PROLOG with <Ctrl>-<D> or

```
    ?- halt.
```

Queries

- Simple queries: e.g.
  "Who are the children of Maria Theresia?"

```
?- child_of(X, maria_theresia).
```

- Composed queries: e.g.
  "Who are the sons (i.e., male children) of Maria Theresia?"

```
?- child_of(X, maria_theresia), male(X).
```

  Alternative:

```
?- child_of(X, maria_theresia), male(Y), X=Y.
```

In SWI-PROLOG: Hit <n> or <;> to get the next answer, Hit <RETURN> to get not further answers.

Simple Queries vs. SQL

- A program consisting only of a set of facts can be viewed as a database.
- We can ask simple conjunctive queries on this database.
  e.g. "Who are the grandchildren of Maria Theresia?":
  ```
  ? - child_of(X,Y), child_of(Y,maria_theresia).
  ```
  This can be similarly expressed in SQL:
  ```
   SELECT c2.child FROM child_of c1, child_of c2
   WHERE c1.parent = 'Maria  Theresia'
      AND c1.child = c2.parent;
  ```

- Note that these queries are exactly what is expressible in databases by so called SPJ-queries (select-project-join)…

- … **Moment**! We don't have projection (yet)! Our query is more something like:
-
  ```
   SELECT c2.child, c2.parent FROM child_of c1, child_of c2
   WHERE c1.parent = 'Maria  Theresia'
      AND c1.child = c2.parent;
  ```
  i.e., we have to output all bindings used in the query.

The answer 'No' and the Closed World Assumption (CWA):

- Bear in mind… PROLOG only answers 'yes' to what can be proven from the program.
- The answer 'no' means: "No answers could be found (in finite time)"

- So, a negative answer to:

```
?- child_of(D, franz_I), male(D).
```

  can have several reasons:

  1. The program describes reality incompletely, the result 'no' is not necessarily true. (In fact Marie Antoinette was a daughter of Franz I, which is not represented in our facts!)

  2. The program describes reality completely, then result 'no' is correct.

  … by answering 'no' Prolog adopts the so called **Closed World Assumption**, i.e. complete knowledge in the facts is assumed we will speak about this in more detail in later lectures.

Rules

- As mentioned above, the current "program" is only a set of facts similar to tables in a database…
- …but in databases we had something like "views", right?

- For more interesting programs and queries we need rules defining new predicates!

Examples for rules:

- Grandchildren from before, add a rule to the program:

  ```
  grandchild_of(X,Z) :- child_of(X,Y), child_of(Y,Z).
  ```

  The set of rules with a certain predicate p in its head are also called the definition of p.

- You can now define grandsons or grandfathers in terms of this new predicate:

  ```
  grandson(X) :- grandchild_of(X,Y), male(X).
  ```

  etc.

- A predicate can be defined by several rules, e.g.:

  ```
  person(X) :- male(X).
  person(X) :- female(X).
  person(X) :- child_of(X,Y).
  person(X) :- child_of(Y,X).
  ```

  e.g. in our database, it is not listed whether ferdinand I is a child of any of the others, or for Leopld it it is not listed whether he was a man or a woman.

Rules as views:

```
person(X) :- male(X).
person(X) :- female(X).

CREATE VIEW person AS
SELECT name from male
UNION SELECT name from female;
```

Since recursion is allowed, this goes beyond SQL-2!

Recursive Rules

- Interesting rules are recursive ones, with recursion we can, for instance define a predicate `ancestor`:

```
ancestor(X,Y) :- child_of(X,Y).
ancestor(X,Z) :- child_of(X,_Y), ancestor(_Y,Z).
```

When using recursion, remember the evaluation of Prolog Programs: Be aware of the rule order and the order of goals!

e.g.:

```
  ancestor(X,Y) :- child_of(X,Y).
  ancestor(X,Z) :- ancestor(_Y,Z), child_of(X,_Y).
```

or

```
  ancestor(X,Z) :- ancestor(_Y,Z), child_of(X,_Y).
  ancestor(X,Y) :- child_of(X,Y).
```

cause problems (similar to what you already know from programming recursively in procedural languages).

Tracing of queries!

```
1 ?- trace.

Yes
[trace] 1 ?-
 Call: (7) ancestor(_G523, maria_theresia) ? creep
   Call: (8) child_of(_G523, maria_theresia) ? creep
   Exit: (8) child_of(leopold_II, maria_theresia) ? creep
   Exit: (7) ancestor(leopold_II, maria_theresia) ? Creep

X = leopold_II ;
   Redo: (8) child_of(_G523, maria_theresia) ? creep
   Exit: (8) child_of(joseph_II, maria_theresia) ? creep
   Exit: (7) ancestor(joseph_II, maria_theresia) ? creep

X = joseph_II ;
   Redo: (8) child_of(_G523, maria_theresia) ? creep
   Exit: (8) child_of(marie_antoinette, maria_theresia) ? creep
   Exit: (7) ancestor(marie_antoinette, maria_theresia) ? creep

X = marie_antoinette ;
   Redo: (7) ancestor(_G523, maria_theresia) ? creep
   Call: (8) child_of(_G523, _L186) ? creep
   Exit: (8) child_of(leopold_II, maria_theresia) ? creep
   Call: (8) ancestor(maria_theresia, maria_theresia) ? creep
   …
   Exit: (8) ancestor(leopold_II, maria_theresia) ? creep
   Exit: (7) ancestor(franz_II, maria_theresia) ? creep

X = franz_II ;
```

Rules as procedures

- Rules can be read as procedures, due to the working of PROLOG:

```
    ancestor(X,Z) :- child_of(X,_Y), ancestor(_Y,Z).

procedure ancestor(X,Z)
    call   child_of(X,Y);
    call   ancestor(Y,Z);
end
```

- In Logic Programming data-manipulation is achieved exclusively by the unification algorithm. Unification subsumes:
  - Single assignment
  - Parameter passing
  - Allocation of memory for (dynamic) data-structures like terms and lists
  - Read/(single-)write to these data-structures

Terms in PROLOG

Example:

```
natNumber(0).
natNumber(s(X)) :- natNumber(X).

% 0+X=X
% s(X)+Y=s(X+Y)
plus(0,X,X) :- natNumber(X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

Note:

- Prolog does not restrict function symbols and predicate symbols to be distinct (all are called functors), i.e. terms and predicates are not distinguished in PROLOG.
- Prolog does allow functors to appear with different arities.

Functions and Terms

- Terms in Prolog:
  - Constants: sequence of alphanumeric characters and '_' beginning lower case letter.

- Examples:

```
f(X,Y) :- p(f(X)), q(f(Y)).
```

is valid in PROLOG!

Note:

```
f(X,Y): f/2     f(X):f/1     f/1≠f/2
```

Functions and Lists

- We seen in the last exercises a very verbose encoding for lists with constant symbol `nil` and function symbol `l/2`.
- Prolog provides a more convenient notation of lists:
  - Empty list: `[]`
  - List notation `[Head|Tail]`
  - Short notation for `[El_1 |[El_2 [El_3|[]]]`
    `[El_1, El_2, El_3]`
  - Examples:

```
l(a,nil)                              [a]
l(1,X)                                [1|X]
l(1,l(2,l(3,nil)))                       [1,2,3]
l(l(a,nil), l(l(b,nil),nil)              [[a],[b]]
```

This can be used to encode Matrices!

Recursive Programming on Lists

- Several predicates such as `append/3`, `member/2`, etc. are predifined for lists, let us try to emulate the predicate appending two lists:

```
    app([],X,X).
    app([X|Xs],Y,[X|Z]) :- app(Xs,Y,Z).
```

Let us check (in `append.pl`) some other programs for reversing lists, deleting elements from lists, etc. …

```
% reverse a list, inefficient:
bad_rev([],[]).
bad_rev([X|Y],Z) :- bad_rev(Y,Y1), app(Y1,[X],Z).

% better version:
rev(List, Reversed) :- rev(List, [], Reversed).
rev([], Reversed, Reversed).
rev([Head|Tail], SoFar, Reversed) :- rev(Tail, [Head|SoFar], Reversed).
```

Slowsort in PROLOG! Version 1

```
slowsort(X,Y) :- perm(X,Y), sorted(Y).

sorted(nil).
sorted(l(X,nil)).
sorted(l(X,l(Y,Z))) :- leq(X,Y), sorted(l(Y,Z)).

perm(nil,nil).
perm(l(X,Y),l(U,V)) :- delete(U,l(X,Y),Z), perm(Z,V).
delete(X,l(X,Y),Y).
delete(X,l(Y,Z),l(Y,W)) :- delete(X,Z,W).

leq(0,X).
leq(s(X),s(Y)) :- leq(x,y).
```

Slowsort in PROLOG! Version 2

```
slowsort(X,Y) :- perm(X,Y), sorted(Y).

sorted([]).
sorted([X|[]]).
sorted([X|[Y|Z]]) :- X @=< Y, sorted([Y|Z]).

perm([],[]).
perm([X|Y],[U|V]) :- delete(U,[X|Y],Z), perm(Z,V).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|W]) :- delete(X,Z,W).
```

Uses PROLOG’s list notation and built-in arithmetic!

Built-in Predicates

- Prolog supports a variety of system predicates, which are implemented in the system, and evaluated directly, not by the normal resolution of pure PROLOG.
  - Arithmetic for numbers
  - Built-ins for lists
  - etc.

- Most systems also provide APIs to extend prolog by external function calls.

Equality and comparison operators:

- All these predicates are binary and can be used in infix notation.

=	Unifies, no evaluation.	X = Y
is	evaluates the term on the right-hand-side (rhs) and unifies with the left-hand-side (lhs), no uninstantiated variables allowed on the rhs.	X is 4+3
=:=	evaluates the term on both sides, no uninstantiated variables allowed.	3+2 =:= 4+1
==	Checks equivalence of terms on lhs and rhs.	1+1 == 2
=@=	Structural equivalence, wrt. To variable structure (weaker than == but stronger than =)	A =@= B
\=	Opposite of =	
\==	Opposite of ==	X \== 2
=\=	Opposite of =:=	
\=@=	Opposite of =@=	
>, < =<, >=	Comparison operators, evaluate the terms on both sides, no uninstantiated variables allowed.	4 < 5 4 =< 4

Arithmetic functions:

- Arithmetic expressions (can be nested):

+	…	addition
-	…	subtraction (binary) or unary minus
mod	…	modulo
rem	…	remainder
/	…	division
//	…	integer division
min, max	…	minimum, maximum
abs	…	absolute

etc.

For a complete list check SWI-Prolog manual!

Some examples:

Arithmetic Operators: `+, -, -, *, /, mod`, etc. in predicate or infix notation.

```
?-  X is Y.                   error
?-  3+1 < 4*5.                succeeds
?-  <(3+1,*(4,5)).            succeeds
?-  N < N+1.                  error
?-  N=4, N < 2+1.             fails
?- 1+1 =\= 2.                 fails
?- 1+1 \== 2.                 succeeds
?- 1+1 =:= 2.                 succeeds
?- 4 >= min(5,3).             succeeds
```

Useful built-ins for lists:

- member/2, reverse/2, sort/2, msort/2, merge/3, append/3, length/2
- etc.

- We've seen already ways to implement these, but the builtins are more efficient.

Meta-logical and structure predicates

Example: check the program natNumbers.pl

```
?- add(s(s(0)),X, s(s(s(s(0))))).
```

Not so, if we use arithmetics and the following simple definition:

```
plus(X,Y,Z) :- Z is X + Y.
```

- We need to analyze `X`, `Y`, and `Z` and distinguish cases where they are variables, numbers, etc.
- Prolog supports a number of meta-logical built-in predicates for that and for several other uses.

Structure predicates & meta-logical Predicates:

Used to check the structure of a term or predicate:

- compound/1
- is_list/1
- atom/1
- integer/1, float/1, number/1
- atomic/1
- var/1, nonvar/1,
- functor/3
- arg/3
- =..

Checking types and structure of terms:

- Check whether some term is of a certain type or structure:

var/1	… suceeds if argument is bound to a variable.
compound/1	… succeds if argument is bound to a compound term.
atom/1	… succeeds if argument is bound to an atom (note that by atom we only mean an identifier here, not an atom in FOL sense!).
integer/1,	
float/1,	
number/1	… succeed if argument is bound to a number
atomic/1	… succeeds for variables and atoms
is_list/1	… succeeds if argument is bound to a list.
ground/1	… succeeds if argument is bound to a ground term.

etc.

Examples:

- `?- X is 3, var(X).      fails`
- `?- var(X), X is 3.     succeeds`
- `?- is_list(X).                   fails`
- `?- atom(maria_theresia)suceeds`
- `?- atom(s)                suceeds`
- `?- atom(s(s(0))).      Fails`

More Meta-Logical predicates for analyzing and constructing terms:

• functor/3	…	functor(X,Y,Z) succeeds if X is a term with functor name Y and arity Z.
• arg/3	…	arg(X,Y,Z) succeeds if Z is the argument of term Y at position X.
• =../2	…	converts a term into a list and vice versa.

Examples

```
?- functor(a(b,_),N,A).          succeeds
?- arg(2, a(b,c),X).             succeeds
?- X =.. [a,b,c].                succeeds
?- a(b,c) =.. X.                 succeeds
?- X =.. a(b,c).                 error
```

Example: Redefine is_list/1 and ground/1:

- Meta-predicates can be used to define further predicates, e.g. we can model some of them as definitions using the other ones:

ground/1 acts as if defined by the following definition:

```
ground(X) :- nonvar(X), atomic(X).
ground(X) :- nonvar(X), compound(X), functor(X,F,N),ground(N,X).
ground(N,X) :- N>0, arg(N,X,Arg), ground(Arg), N1 is N-1, ground(N1,X).
ground(0,X).
```

is_list/1 acts as if defined by the following definition:

```
          is_list(X) :- var(X), !, fail.
          is_list([]).
          is_list([_|T]) :- is_list(T).
```

For the meaning of '!' and 'fail', see below.

Example: Redefine plus/3:

```
plus(X,Y,Z) :- nonvar(X), nonvar(Y), Z is X + Y.
plus(X,Y,Z) :- nonvar(X), nonvar(Z), Y is Z - X.
plus(X,Y,Z) :- nonvar(Y), nonvar(Z), X is Z - Y.
```