(Introduction to) Logic Programming

D. Vermeir

September 23, 2009

Outline

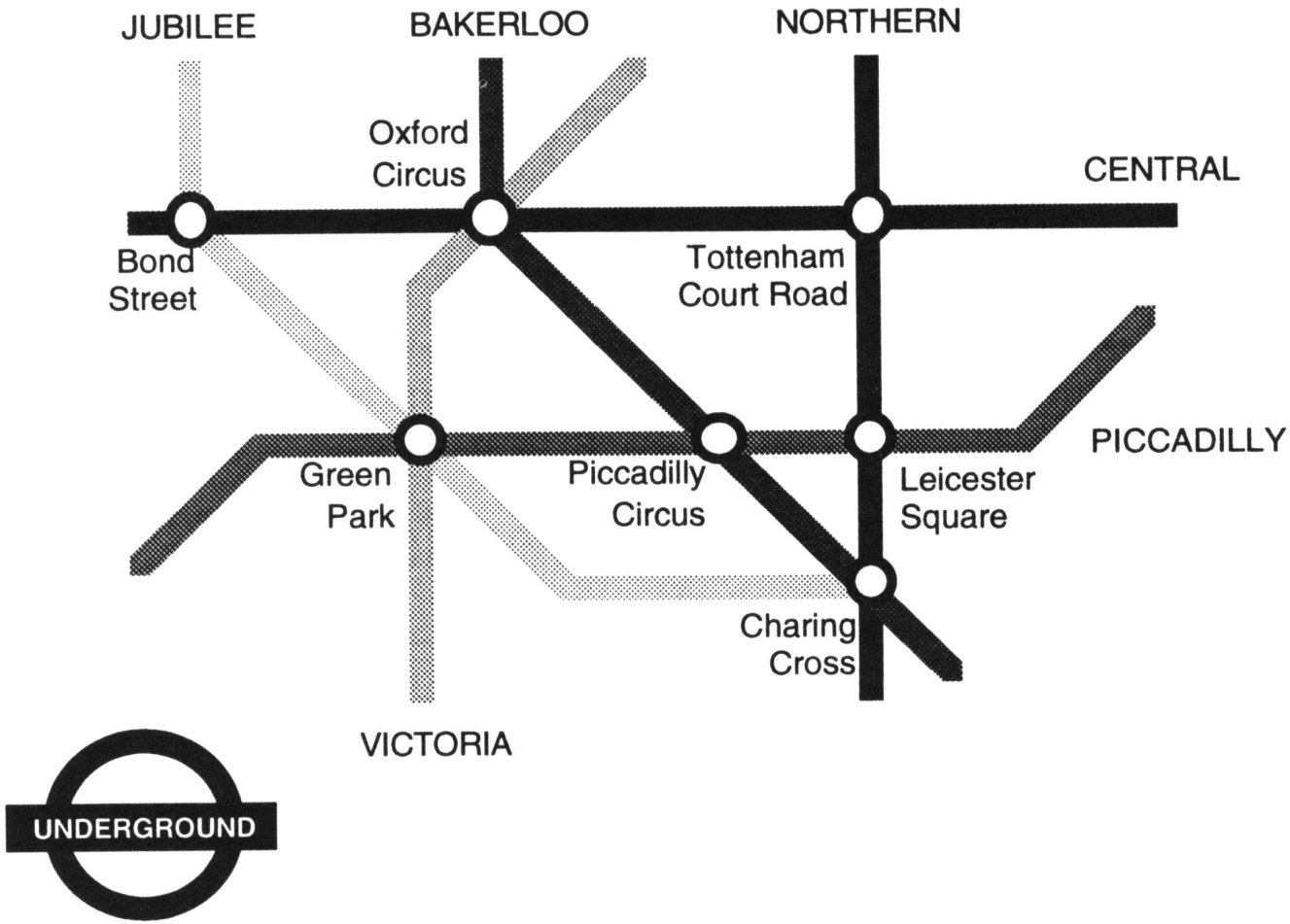
Organization

- Evaluation: 50% project, 50% (closed book) theory exam.
- Exercise sessions: first one on Wed. Sep. 30 2009, 15:00-17:00, IG.
- Book.
- Copies of transparencies.
- See website for further information.

Contents

Outline

introduction



Peter Flach, *Simply Logical: Intelligent Reasoning by Example*, Wiley, 1994.

logical representation of map

described by a series of logical **facts**:

```
connected(bond_street,oxford_circus,central)
connected(oxford_circus,tottenham_court_road,central)
connected(bond_street,green_park,jubilee)
connected(green_park,charing_cross,jubilee)
connected(green_park,piccadilly_circus,piccadilly)
connected(piccadilly_circus,leicester_square,piccadilly)
connected(green_park,oxford_circus,victoria)
connected(oxford_circus,piccadilly_circus,bakerloo)
connected(piccadilly_circus,charing_cross,bakerloo)
connected(tottenham_court_road,leicester_square,northern)
connected(leicester_square,charing_cross,northern)
```

derived information

"Two stations are *near* if they are on the **same line**, with at most one station in between"

```
near(bond_street,oxford_circus)
near(oxford_circus,tottenham_court_road)
near(bond_street,tottenham_court_road)
near(bond_street,green_park)
near(green_park,charing_cross)
near(bond_street,charing_cross)
% etc. (16 formulas)
```

The same effect can be obtained using 2 rules:

```
near(X,Y) :- connected(X,Y,L).
near(X,Y) :- connected(X,Z,L), connected(Z,Y,L).
```

the meaning of rules

The second rule

```
near(X,Y) :- connected(X,Z,L), connected(Z,Y,L)
```

reads:

> *"For any values of X, Y, Z and L, X is near Y* ***if*** *X is connected to Z via L,* ***and*** *Z is connected to Y via L."*

or

$$\forall X, Y, Z, L \cdot connected(X, Z, L) \wedge connected(Z, Y, L) \Rightarrow near(X, Y)$$

queries

```
?- connected(W,tottenham_court_road,L)
```

the answer can be found by matching it with facts:

```
{ W = oxford_circus , L = central}
```

```
?- near(tottenham_court_road,W)
```

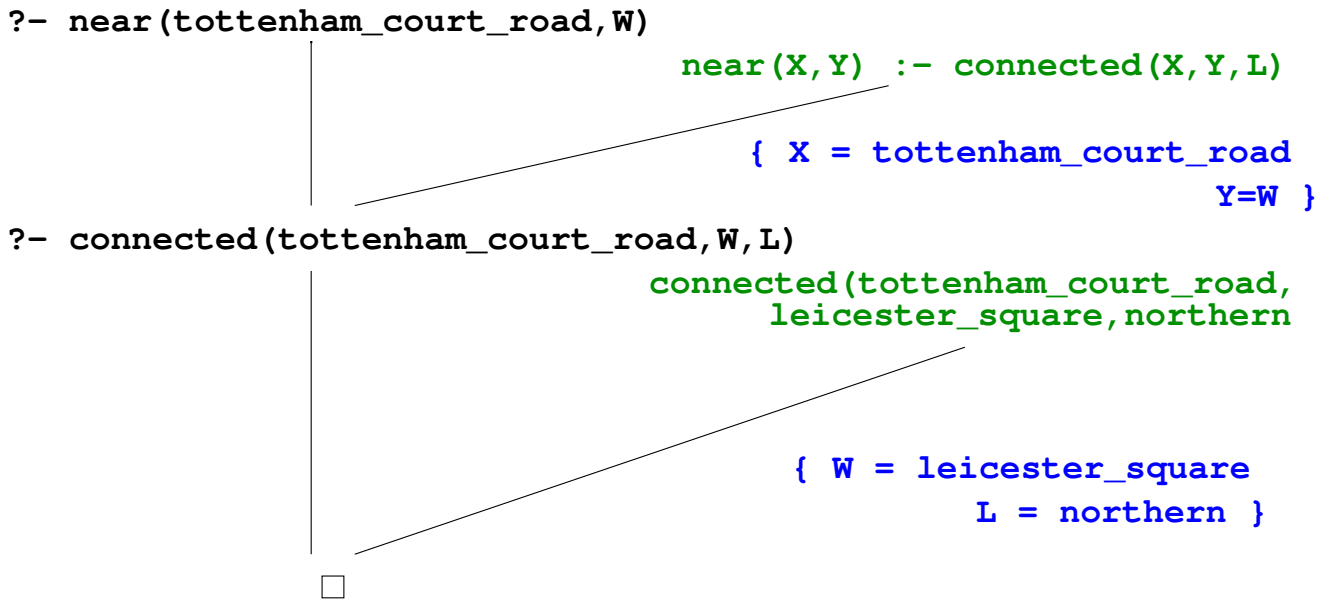
match it with the conclusion of `near(X,Y) :- connected(X,Y,L)`
yielding the substitution `{ X = tottenham_court_road, Y = W }`
and try to find an answer to the premises

```
?- connected(tottenham_court_road,W,L)
```

giving `{ W = leicester_square, L = northern }`
The final result is

```
{ X = tottenham_court_road, Y = W = leicester_square,
  L = northern }
```

solving a query = constructing a proof (tree)



To solve a query $?- Q_1, \ldots, Q_n$ find a rule $A :- B_1, \ldots, B_m$ where A matches Q_1 and solve

$$?- B_1, \ldots, B_m, Q_2, \ldots, Q_n$$

Resolution gives a procedural interpretation to logic.

proof by refutation

Note: the procedural interpretation $\neq$ the declarative semantics (e.g. because of looping).

The proof technique used is “reductio ad absurdum” or proof by refutation: assume that the formula (query) is false and deduce a contradiction:

```
?- near(tottenham_court_road,W)
```

stands for

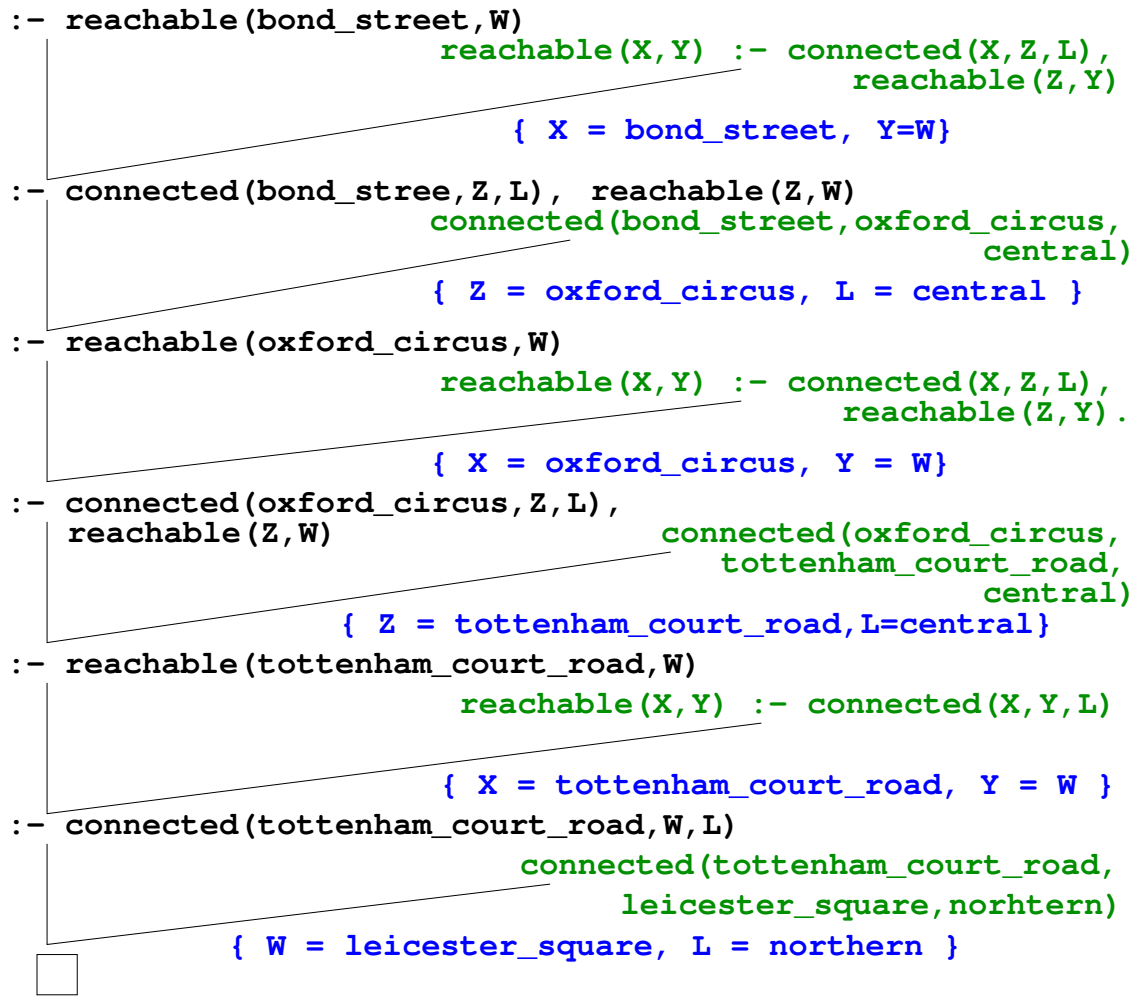
$$\forall W \cdot near(tottenham\_court\_road, W) \Rightarrow \textbf{false}$$

or

there are no stations near tottenham_court_road

recursive rules

```
reachable(X,Y) :- connected(X,Y,L).
reachable(X,Y) :- connected(X,Z,L), reachable(Z,Y).
?- reachable(bond_street,W)
```



prolog proof strategy

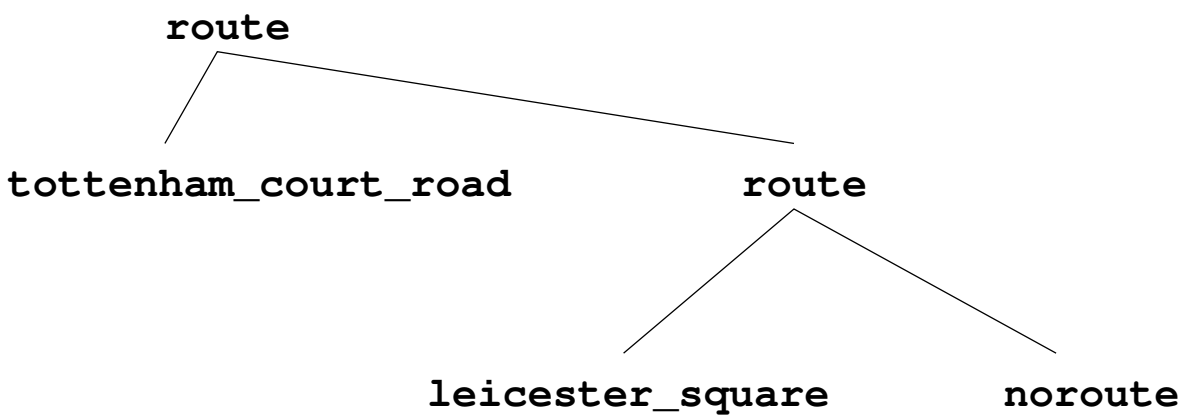
Prolog uses **depth-first** search when finding a proof, **backtracking** when it fails, until a solution is found or there are no more possibilities. It tries rules and facts in the given order, always trying to resolve the first subgoal.

functors

can be used to represent complex data structures; the term

```
route(tottenham_court_road, route(leicester_square, noroute) )
```

represents



using functors

```
reachable(X,Y,noroute) :- connected(X,Y,L).
reachable(X,Y,route(Z,R)) :-
                  connected(X,Z,L),
                  reachable(Z,Y,R).
?- reachable(oxford_circus,charing_cross,R)
{ R = route(tottenham_court_road,
            route(leicester_square,noroute))}
{ R = route(piccadilly_circus,noroute)}
{ R = route(piccadilly_circus,
            route(leicester_square,noroute))}
```

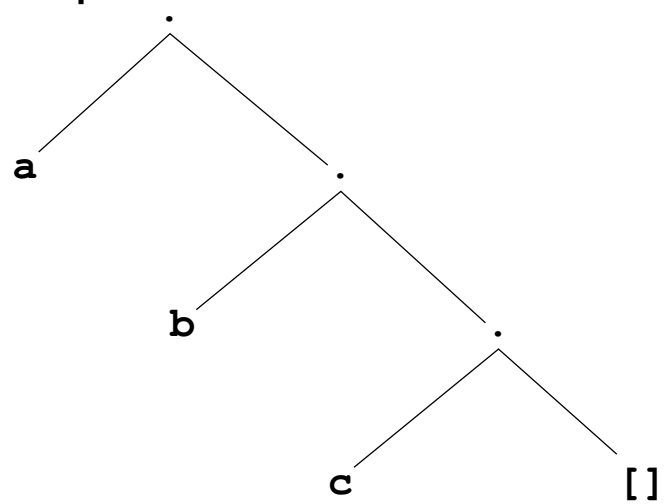
```
route(oxford_circus,
      route(leicester_square,noroute))
```

represents a route via ..

Note: functors are not evaluated in normal LP.

lists

Lists are also represented by a functor "." (compare "*cons*").
E.g. the list `[a,b,c]` is represented as



which can also be written as `.(a,.(b,.(c,[])))`
We also use `[Head | Tail ]` where `Tail` is a list, as a shorthand for `.(Head,Tail)`
We can also write e.g. `[ First, Second, Third | Rest ]`

using lists

A route can be represented by a list:

```
reachable(X,Y,[]) :- connected(X,Y,L).
reachable(X,Y,[Z|R]) :-
                  connected(X,Z,L),
                  reachable(Z,Y,R).
?- reachable(oxford_circus,charing_cross,R)
{R=[tottenham_court_road,leicester_square]}
{R=[piccadilly_circus]}
{R=[piccadilly_circus,leicester_square]}
```

To ask from which station we can reach *charing_cross* via 4 intermediate stations:

```
?- reachable(X,charing_cross,[A,B,C,D])
```

Outline

clausal logic

Any logic system has a:

- **syntax**: which “sentences” are legal.
- **semantics**: the meaning of sentences, i.e. what is the truth value of a sentence.
- **proof theory**: how to derive new sentences (theorems) from assumed ones (axioms) by means of inference rules.

A logic system is called

- **sound** if anything you can prove is true
- **complete** if anything that is true can be proven

propositional clausal logic syntax

Connectives

`:-`	"if"
`;`	"or"
`,`	"and"

$$
\begin{aligned}
\textit{clause} &: \textit{head}[:\!- \textit{body}] \\
\textit{head} &: [\textit{proposition}[; \textit{proposition}]^*] \\
\textit{body} &: \textit{proposition}[, \textit{proposition}]^* \\
\textit{proposition} &: \textit{atom} \\
\textit{atom} &: \text{single word starting with lower case}
\end{aligned}
$$

example:

```
married ; bachelor :- man, adult
```

Someone is married or a bachelor if he is a man and an adult

logic program

a **program** is a finite set of clauses, each terminated by a period; the clauses are to be read conjunctively

```
woman;man :- human.
human :- man.
human :- woman.
```

in traditional logic notation:

$$(human \Rightarrow (woman \vee man))$$
$$\wedge(man \Rightarrow human)$$
$$\wedge(woman \Rightarrow human)$$

Using $A \Rightarrow B \equiv \neg A \vee B$ we get

$$(\neg human \vee woman \vee man)$$
$$\wedge(\neg man \vee human)$$
$$\wedge(\neg woman \vee human)$$

clause

In general a clause

$$H_1 ; \ldots ; H_n :- B_1 , \ldots , B_m$$

is equivalent with

$$H_1 \vee \ldots \vee H_n \vee \neg B_1 \vee \ldots \vee \neg B_m$$

A clause can also be defined as

$$L_1 \vee L_2 \vee \ldots \vee L_n$$

where each L_i is a literal, i.e. $L_i = A_i$ or $L_i = \neg A_i$, with A_i a proposition.

special clauses

An empty head stands for **false**, an empty body stands for **true**:

```
man :- .          % usually written as ``man.''
:- impossible.
```

is the same as

$$(\textbf{true} \Rightarrow \mathit{man}) \wedge (\mathit{impossible} \Rightarrow \textbf{false})$$

i.e.

$$\mathit{man} \wedge \neg \mathit{impossible}$$

semantics

- The **Herbrand base** $\mathcal{B}_P$ of a program P is the set of all atoms occurring in P.
- A **Herbrand interpretation** of P is a mapping

$$i : \mathcal{B}_P \rightarrow \{\textbf{true}, \textbf{false}\}$$

  We will represent i by $I = i^{-1}(\textbf{true})$, the set of true propositions.
- An interpretation is a **model for a clause** if the clause is true under the interpretation, i.e. if either the head is true or the body is false.
- An interpretation is a **model for a program** if it is a model for each clause in the program.

example

Consider P:

```
woman;man :- human.
human :- man.
human :- woman.
```

Then:

$$\mathcal{B}_P = \{woman, man, human\}$$

and a possible interpretation is

$$i = \{(woman, \textbf{true}), (man, \textbf{false}), (human, \textbf{true})\}$$

or

$$I = \{woman, human\}$$

All clauses in P are true under I so it is a model of P.
The interpretation $J = \{woman\}$ is not a model of P.

logical consequence

A clause C is a **logical consequence** of a program P, denoted

$$P \models C$$

if every model of P is also a model of C.
E.g. for P:

```
woman.
woman;man :- human.
human :- man.
human :- woman.
```

we have that $P \models human$. Note that P has two models:

$$\begin{aligned} M_1 &= \{woman, human\} \\ M_2 &= \{woman, man, human\} \end{aligned}$$

Intuitively, M_1 is preferred since it only accepts what *must* be true.

minimal models

Thus we could define the best model to be the minimal one.
However, consider P':

```
woman;man :- human.
human.
```

P has 3 models

$$\begin{aligned} M_1 &= \{woman, human\} \\ M_2 &= \{man, human\} \\ M_3 &= \{woman, man, human\} \end{aligned}$$

and M_1 and M_2 are both minimal!
If we restrict (as in Prolog) to **definite** clauses, which have at most one atom in the head, then:

Theorem

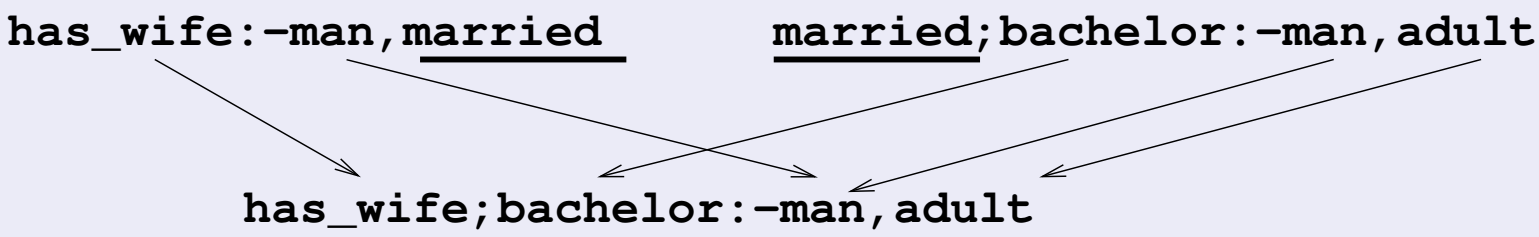
A definite logic program has a unique minimal model.

proof theory

How to compute logical consequences without checking all the models?

Use **resolution** as an inference rule.

```
married;bachelor :- man,adult.
has_wife :- man,married.
```



Using resolution, we get

```
has_wife;bachelor :- man,adult.
```

which is a logical consequence of the program.

resolution: intuition

```
married;bachelor :- man,adult.
has_wife :- man,married.
```

$$\neg man \vee \neg adult \vee \underline{married} \vee bachelor$$

$$\neg man \vee \underline{\neg married} \vee has\_wife$$

either *married* and then $\neg man \vee has\_wife$

or $\neg$*married* and then $\neg man \vee \neg adult \vee bachelor$

thus

$$\neg man \vee \neg adult \vee bachelor \vee \neg man \vee has\_wife$$

```
has_wife;bachelor :- man,adult.
```

resolution in traditional notation

$$\frac{E_1 \vee E_2 \qquad \neg E_2 \vee E_3}{\Rightarrow E_1 \vee E_3}$$

special case: modus ponens

$$\frac{A \qquad A \Rightarrow B}{B}$$

In clause notation:

$$\frac{A \qquad \neg A \vee B}{B}$$

special case: modus tollens

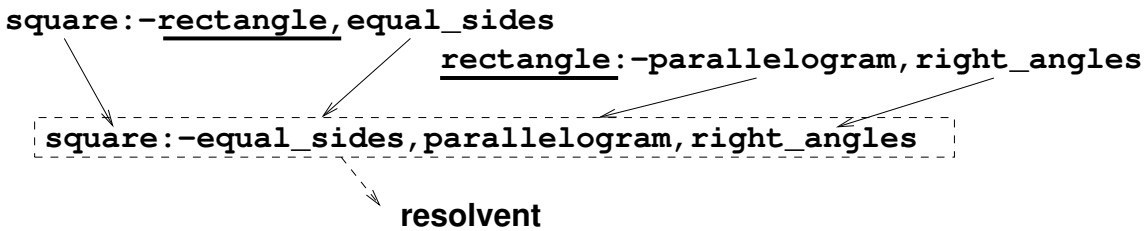
$$\frac{\neg B \qquad A \Rightarrow B}{\neg A}$$

In clause notation

$$\frac{\neg B \qquad \neg A \vee B}{\neg A}$$

resolution with definite clauses

```
square :- rectangle, equal_sides.
rectangle :- parallelogram, right_angles.
```



The resolvent can be used in further resolution steps...

Definition

A **proof** or **derivation** of a clause C from a program P is a sequence of clauses

$$C_0, \ldots, C_n = C$$

such that $\forall i \cdot = 0 \ldots n$: either $C_i \in P$ or C_i is the resolvent of C_{i_1} and C_{i_2} ($i_1 < i$, $i_2 < i$).

We write $P \vdash C$ if there is a proof of C from P.

soundness, completeness

Theorem

Resolution is sound for propositional clausal logic, i.e. if $P \vdash C$ then $P \models C$.

About completeness:

$P \models C$
iff each model of P is a model of C
iff no model of P is a model of $\neg C$ (1)

If $C \equiv L_1 \vee L_2 \vee \ldots \vee L_n$ then

$$\begin{aligned} \neg C &\equiv \neg L_1 \wedge \neg L_2 \wedge \ldots \wedge \neg L_n \\ &\equiv \{\neg L_1, \neg L_2, \ldots, \neg L_n\} \end{aligned}$$

i.e. $\neg C$ is a set of clauses and so
(1) $\equiv P \cup \neg C$ has no model
iff $P \cup \neg C$ is inconsistent (by definition)

completeness

Theorem

If Q is inconsistent then $Q \vdash \square$.

where $\square$ is the empty clause (**true** $\Rightarrow$ **false**) which has no model.

Theorem

Resolution is complete for propositional clausal logic, i.e. if

$$P \models C$$

then

$$P \cup \neg C \vdash \square$$

I.e. C is a logical consequence of P iff one can derive, using resolution, the empty clause ($\square$) from $P \cup \neg C$.

Example: consider P_2

```
happy :- has_friends.
friendly :- happy.
```

We show that $P_2 \models C$ where C is

```
friendly :- has_ friends
```

$P_2 \cup \neg C$ is

```
happy :- has_friends.   (1)
friendly :- happy.      (2)
has_friends.            (3)
:- friendly.            (4)
```

The proof:

```
(2+4) :- happy           (5)
(1+5) :- has_friends     (6)
(3+6) <empty-clause>
```

relational clausal logic

Add individual constants and variables; the syntax has the same connectives as in the propositional case.

constant	:	single word starting with lower case
variable	:	single word starting with upper case
term	:	*constant* $\mid$ *variable*
predicate	:	single word starting with lower case
atom	:	*predicate*[(*term*[, *term*]*)]
clause	:	*head*[:– *body*]
head	:	[*atom*[; *atom*]*]
body	:	*atom*[, *atom*]*

```
likes(peter,S) :- student_of(S,peter)
```

for any S: if S is a student of peter then peter likes S

arity, ground term, semantics

- A predicate has an **arity** to denote the number of **arguments**. E.g. `likes/2`. In Prolog, `p/2` is different from `p/3`.
- A term (atom) is **ground** if it does not contain any variables.

semantics:

- The **Herbrand universe** of a program P is the set of all ground terms occurring in it.
- The **Herbrand base** $\mathcal{B}_P$ of P is the set of all ground atoms that can be constructed using predicates in P and arguments in the Herbrand universe of P.
- A **Herbrand interpretation** is a subset $I \subseteq \mathcal{B}_P$ of ground atoms that are true.

Consider P_3

```
likes(peter,S) :- student_of(S,peter). % C1
student_of(maria,peter).
```

$\mathcal{B}_{P_3} =$

```
{ likes(peter,peter), likes(peter,maria), likes(maria,peter),
likes(maria,maria), student_of(peter,peter),
student_of(peter,maria), student_of(maria,peter),
student_of(maria,maria) }
```

An interpretation

$I_3 =$ `{likes(peter,maria), student_of(maria,peter)}`

Definition

A **substitution** is a mapping σ : **Var** $\rightarrow$ **Trm**. For a clause C, the result of σ on C, denoted $C\sigma$ is obtained by replacing all occurrences of $X \in$ **Var** in C by $\sigma(X)$. $C\sigma$ is an **instance** of C.

If $\sigma = \{S/maria\}$ then $C_1\sigma$ is

```
likes(peter,maria) :- student_of(maria,peter).
```

semantics

Definition

A **ground instance** of a clause C is the result $C\sigma$ of some substitution such that $C\sigma$ contains but ground atoms.

Ground clauses are like propositional clauses.

Definition

An interpretation I is a **model** of a clause C iff it is a model of every ground instance of C. An interpretation is a model of a program P iff it is a model of each clause $C \in P$.

All ground instances of clauses in P_3:

```
likes(peter,peter) :- student_of(peter,peter).
likes(peter,maria) :- student_of(maria,peter).
student_of(maria,peter).
```

Thus $M_3 =$ `{likes(peter,maria), student_of(maria,peter)}` is a model of P_3.

proof theory

Naive version: do (propositional) resolution with all ground instances of clauses in P.

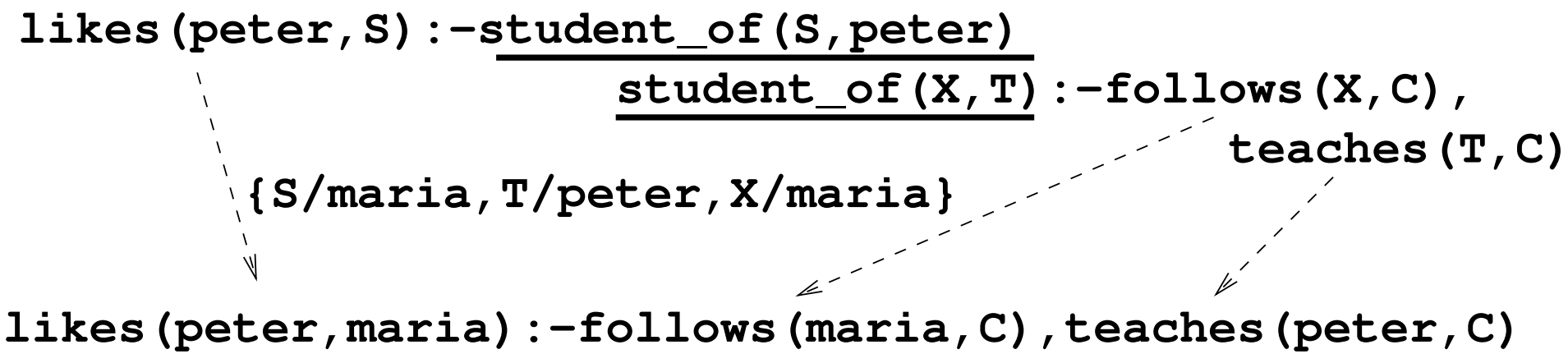
Definition

A substitution σ is a **unifier** of two atoms a_1 and a_2 iff $a_1\sigma = a_2\sigma$. A substitution σ_1 is more general than σ_2 if $\sigma_2 = \sigma_1\theta$ for some substitution θ.
A unifier θ of a_1 and a_2 is a most general unifier (**mgu**) of a_1 and a_2 iff it is more general than any other unifier of a_1 and a_2.

Theorem

If two atoms are unifiable then they their mgu is unique up to renaming.

proof theory using mgu: example



proof theory using mgu

"Do resolution on many clause-instances at once."

If

$$\begin{aligned} C_1 &= L_1^1 \vee \ldots L_{n_1}^1 \\ C_2 &= L_1^2 \vee \ldots L_{n_2}^2 \\ L_i^1\theta &= \neg L_j^2\theta \quad \text{for some } 1 \leq i \leq n_1,\ 1 \leq j \leq n_2 \end{aligned}$$

where $\theta = \mathbf{mgu}(L_i^1, L_j^2)$, then

$$L_1^1\theta \vee \ldots \vee L_{i-1}^1\theta \vee L_{i+1}^1\theta \vee \ldots \vee L_{n_1}^1\theta \vee L_1^2\theta \vee \ldots \vee L_{j-1}^2\theta \vee L_{j+1}^2\theta \vee \ldots \vee L_{n_2}^2\theta$$

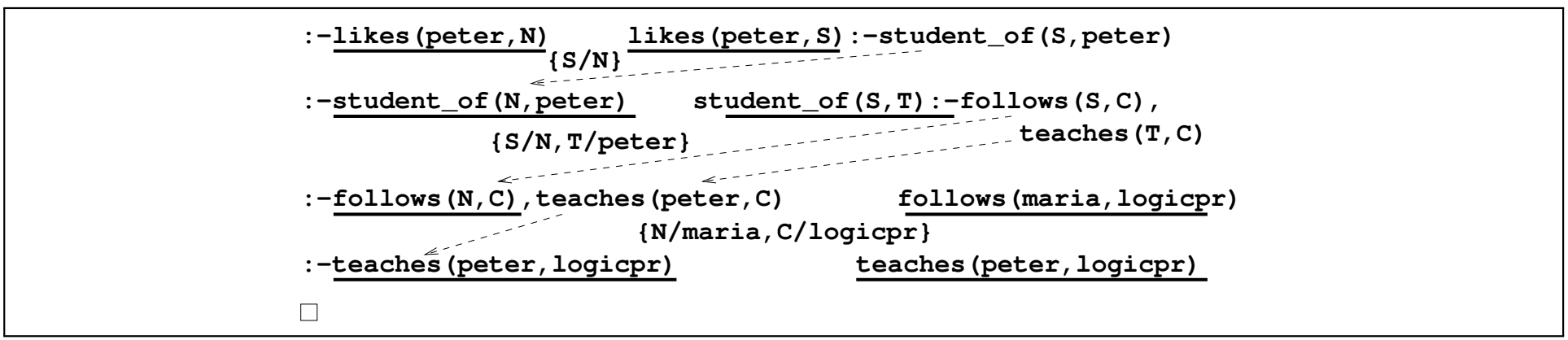
proof theory example

Consider P_4:

```
likes(peter,S) :- student_of(S,peter).
student_of(S,T) :- follows(S,C), teaches(T,C).
teaches(peter,logicpr).
follows(maria,logicpr).
```

? "is there anyone whom peter likes" (*query*)

$\Rightarrow$ add "peter likes nobody" (`:- likes(peter,N)`)



Thus (:- likes(peter,N)){N/maria} $\cup$ $P_4 \vdash \square$ and thus

$P_4 \models$ likes(peter,maria)

soundness, completeness

Theorem

Relational clausal logic is sound and (refutation) complete:

$$\begin{aligned} P \vdash C &\Rightarrow P \models C \\ P \cup \{C\} \text{ inconsistent} &\Rightarrow P \cup \{C\} \vdash \square \end{aligned}$$

New formulation is because: $\neg(\forall X \cdot p(X)) \equiv \exists X \cdot \neg p(X)$ but

$$\texttt{p(X).} \equiv \forall X \cdot p(X)$$

while

$$\texttt{:- p(X).} \equiv \forall X \cdot \neg p(X)$$

decidability relational clausal logic

Theorem

The question

$$P \models C$$

is decidable for relational clausal logic

Because the Herbrand base is finite.

full clausal logic

Add function symbols (functors), with an **arity**; constants are 0-ary functors.

functor	:	single word starting with lower case
variable	:	single word starting with upper case
term	:	*variable* | *functor*[(*term*[, *term*]*)]
predicate	:	single word starting with lower case
atom	:	*predicate*[(*term*[, *term*]*)]
clause	:	*head*[:– *body*]
head	:	[*atom*[; *atom*]*]
body	:	*atom*[, *atom*]*

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
% read s(X) as ``successor of X''
```

semantics

As for relational case; models may be (necessarily) infinite as in P_5:

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

$M_5 =$

```
{ plus(0,0,0), plus(s(0),0,s(0),
               plus(s(s(0)),0,s(s(0)), ...
plus(0,s(0),s(0)), plus(s(0),s(0),s(s(0))), ...
... }
```

computing the mgu

Definition

A set of equations

$$\{s_i = t_i \mid i = 1 \ldots n\}$$

between terms is in **solved form** if

- $\forall 1 \leq i \leq n \cdot s_i \in \mathbf{Var}$
- $\forall 1 \leq i \leq n \cdot t_i$does not contain any variable from $\{s_i \mid 1 \leq i \leq n\}$

A set of equations $\{X_i = t_i\}$ represents a substitution $\{X_i/t_i\}$.

Theorem

if $solve(\{t_1 = t_2\})$ *succeeds, it returns* $mgu(t_1, t_2)$.

proc *solve*(**var** $\mathcal{E}$: set of equations)

repeat
  select $s = t \in \mathcal{E}$
  case $s = t$ **of**
    $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ $(n \geq 0)$:
      replace $s = t$ **by** $\{s_1 = t_1, \ldots, s_n = t_n\}$
    $f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n)$ $(f/m \neq g/n)$:
      fail
    $X = X$:
      remove $X = X$ **from** $\mathcal{E}$
    $t = X$ $(t \notin$ **Var**$)$:
      replace $t = X$ **by** $X = t$
    $X = t$ $(X \in$ **Var** $\wedge\, X \neq t \wedge X$ occurs more than once in $\mathcal{E})$:
      if Xoccurs in t
      then fail
      else replace all occurrences of X in $\mathcal{E}$ (except in $X = t$) by t
  esac
until no change

examples

$$
\begin{aligned}
& \{\underline{f(X, g(Y)) = f(g(Z), Z)}\} \\
\Rightarrow \quad & \{X = g(Z), \underline{g(Y) = Z}\} \\
\Rightarrow \quad & \{X = g(Z), \underline{Z = g(Y)}\} \\
\Rightarrow \quad & \{X = g(g(Y)), Z = g(Y)\} \\
\Rightarrow \quad & \{X/g(g(Y)), Z/g(Y)\}
\end{aligned}
$$

$$
\begin{aligned}
& \{\underline{f(X, g(X), b) = f(a, g(Z), Z)}\} \\
\Rightarrow \quad & \{X = a, \underline{g(X) = g(Z)}, b = Z\} \\
\Rightarrow \quad & \{\underline{X = a}, X = Z, b = Z\} \\
\Rightarrow \quad & \{X = a, \underline{a = Z}, b = Z\} \\
\Rightarrow \quad & \{X = a, \underline{Z = a}, b = Z\} \\
\Rightarrow \quad & \{X = a, Z = a, \underline{b = a}\} \\
\Rightarrow \quad & \textbf{fail}
\end{aligned}
$$

occur check

$$
\begin{array}{ll}
 & \{\underline{l(Y,Y) = l(X,f(X))}\} \\
\Rightarrow & \{\underline{Y = X}, Y = f(X)\} \\
\Rightarrow & \{Y = X, \underline{X = f(X)}\} \\
\Rightarrow & \textbf{fail}
\end{array}
$$

The last example illustrates the need for the “occur check” (which is not done in most Prolog implementations)

soundness, completeness

Theorem

Full clausal logic is sound and (refutation) complete:

$$P \vdash C \quad \Rightarrow \quad P \models C$$
$$P \cup \{C\} \text{ inconsistent} \quad \Rightarrow \quad P \cup \{C\} \vdash \square$$

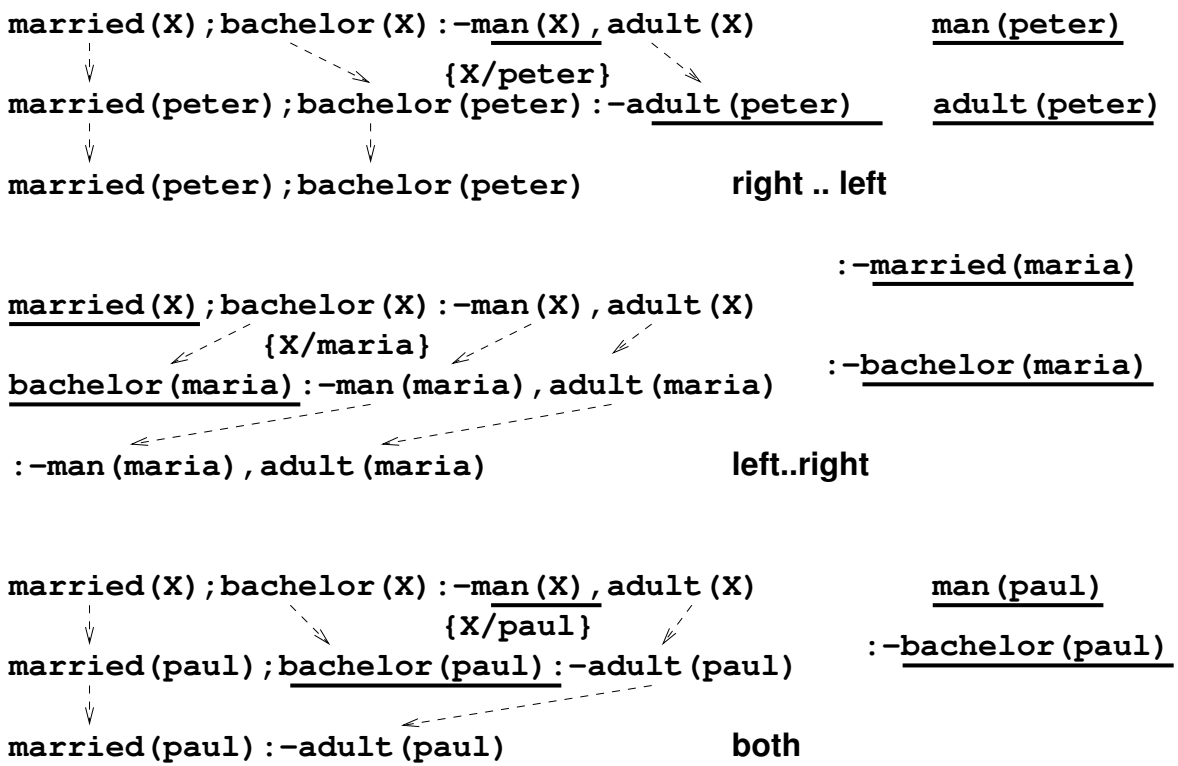
However the question

$$P \models C$$

is only semi-decidable, i.e. there is no algorithm that will always answer the question (with “yes” or “no”) in finite time; but there is an algorithm that, if $P \models C$, will answer “yes” in finite time but this algorithm may loop if $P \not\models C$.
This means that prolog may loop on certain queries.

definite clause logic

```
married(X);bachelor(X) :- man(X), adult(X).
man(peter).                                   adult(peter).
:- married(maria).                            :- bachelor(maria).
man(paul).                                    :- bachelor(paul).
```



- For efficiency reasons: restriction to **definite** clauses where the head contains at most 1 atom.

$$A :- B_1, \ldots, B_n$$

"prove A by proving each of $B_1, \ldots, B_n$".
This is the **procedural interpretation** of definite clauses. It makes the search for a refutation much more efficient.

- Problem: how to represent

```
married(X);bachelor(X) :- man(X), adult(X).
```

⇒ *To prove* `married(X)` *: show* `man(X), adult(X)` *and not* `bachelor(X)` *.*

general clauses

A (pseudo-definite) **general** clause may contain negations in the body:

```
married(X) :- man(X), adult(X), not(bachelor(X)).
```

With `man(jim). adult(jim).` this will have

```
{ married(jim), adult(jim), man(jim) }
```

as a minimal model. Alternatively:

```
bachelor(X) :- man(X), adult(X), not(married(X)).
```

has, with `man(jim). adult(jim).`,

```
{ bachelor(jim), adult(jim), man(jim) }
```

as a minimal model.

clausal logic vs. predicate logic

Every set of clauses can be rewritten as an equivalent sentence in first order (predicate) logic.
Example:

```
married;bachelor :- man,adult.
haswife :- married
```

becomes

$$(man \wedge adult \Rightarrow married \vee bachelor) \wedge (married \Rightarrow haswife)$$

or, using $A \Rightarrow B \equiv \neg A \vee B$ and $\neg(A \wedge B) \equiv \neg A \vee \neg B$:

$$(\neg man \vee \neg adult \vee married \vee bachelor) \wedge (\neg married \vee haswife)$$

which is in **conjunctive normal form** (a conjunction of disjunctions of literals).

Variables in clauses are universally quantified:

```
reachable(X,Y,route(Z,R)):- connected(X,Z,L), reachable(Z,Y,R).
```

becomes

$$\forall X \forall Y \forall Z \forall R \forall L : \neg connected(X, Z, L) \vee \neg reachable(Z, Y, R) \vee reachable(X, Y, route(Z, R))$$

Note that `nonempty(X) :- contains(X,Y).` becomes

$$\begin{aligned} & \forall X \forall Y : nonempty(X) \vee \neg contains(X, Y) \\ \equiv \; & \forall X : (\forall Y : nonempty(X) \vee \neg contains(X, Y)) \\ \equiv \; & \forall X : (nonempty(X) \vee (\forall Y : \neg contains(X, Y))) \\ \equiv \; & \forall X : nonempty(X) \vee \neg(\exists Y : contains(X, Y)) \\ \equiv \; & \forall X : (\exists Y : contains(X, Y)) \Rightarrow nonempty(X) \end{aligned}$$

For each first order sentence, there exists an “almost equivalent” set of clauses.

algorithm

$$\forall X[brick(X) \Rightarrow (\exists Y[on(X,Y) \wedge \neg pyramid(Y)] \wedge \neg\exists Y[on(X,Y) \wedge on(Y,X)] \wedge \forall Y[\neg brick(Y) \Rightarrow \neg equal(X,Y)])]$$

Step 1: eliminate $\Rightarrow$ using $A \Rightarrow B \equiv \neg A \vee B$.

$$\forall X[\neg brick(X) \vee (\exists Y[on(X,Y) \wedge \neg pyramid(Y)] \wedge \neg\exists Y[on(X,Y) \wedge on(Y,X)] \wedge \forall Y[\neg(\neg brick(Y)) \vee \neg equal(X,Y)])]$$

Step 2: move $\neg$ inside using

$$\begin{aligned} \neg(A \wedge B) &\equiv \neg A \vee \neg B \\ \neg(A \vee B) &\equiv \neg A \wedge \neg B \\ \neg(\neg A) &\equiv A \\ \neg\forall X[p(X)] &\equiv \exists X[\neg p(X)] \\ \neg(\exists X[p(X)] &\equiv \forall X[\neg p(X)] \end{aligned}$$

$$\forall X[\neg brick(X) \vee (\exists Y[on(X,Y) \wedge \neg pyramid(Y)] \\ \wedge \forall Y[\neg on(X,Y) \vee \neg on(Y,X)] \\ \wedge \forall Y[brick(Y) \vee \neg equal(X,Y)])]$$

Step 3: replace $\exists$ using skolem functors
E.g.

$$\forall X \exists Y : likes(X,Y)$$

becomes

$$\forall X : likes(X, f(X))$$

where "f" is a new **Skolem** functor. All universally quantified variables in whose scope $\exists$ occurs become arguments of the Skolem term.
E.g. $\exists X : likes(peter, X)$ becomes $likes(peter, g)$.
In clausal logic, one is forced to use abstract names (using functors) for existentially quantified individuals.

In example:

$$\forall X[\neg brick(X) \vee (\exists Y[on(X,Y) \wedge \neg pyramid(Y)] \\ \wedge \forall Y[\neg on(X,Y) \vee \neg on(Y,X)] \\ \wedge \forall Y[brick(Y) \vee \neg equal(X,Y)])]$$

becomes

$$\forall X[\neg brick(X) \vee \\ ([on(X, sup(X)) \wedge \neg pyramid(sup(X))] \\ \wedge \forall Y[\neg on(X,Y) \vee \neg on(Y,X)] \\ \wedge \forall Y[brick(Y) \vee \neg equal(X,Y)])]$$

Step 4: rename variables (make unique)

$$\forall X[\neg brick(X) \vee \\ ([on(X, sup(X)) \wedge \neg pyramid(sup(X))] \\ \wedge \forall Y[\neg on(X,Y) \vee \neg on(Y,X)] \\ \wedge \forall Z[brick(Z) \vee \neg equal(X,Z)])]$$

$$\forall X[\neg brick(X) \vee$$
$$([on(X, sup(X)) \wedge \neg pyramid(sup(X))]$$
$$\wedge \forall Y[\neg on(X, Y) \vee \neg on(Y, X)]$$
$$\wedge \forall Z[brick(Z) \vee \neg equal(X, Z)])]$$

Step 5: bring $\forall$ to front

$$\forall X \forall Y \forall Z[\neg brick(X) \vee$$
$$([on(X, sup(X)) \wedge \neg pyramid(sup(X))]$$
$$\wedge [\neg on(X, Y) \vee \neg on(Y, X)]$$
$$\wedge [brick(Z) \vee \neg equal(X, Z)])]$$

Step 6

: bring $\vee$ inside using

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

$$\forall X \forall Y \forall Z [\underline{\neg brick(X) \vee} ([on(X, sup(X)) \wedge \neg pyramid(sup(X))] \underline{\wedge} [\neg on(X, Y) \vee \neg on(Y, X)] \underline{\wedge} [brick(Z) \vee \neg equal(X, Z)])]$$

becomes

$$\forall X \forall Y \forall Z [\underline{\neg brick(X) \vee} ([on(X, sup(X)) \underline{\wedge} \neg pyramid(sup(X))] \wedge [\neg brick(X) \vee \neg on(X, Y) \vee \neg on(Y, X)] \wedge [\neg brick(X) \vee brick(Z) \vee \neg equal(X, Z)])]$$

$$\forall X \forall Y \forall Z [\neg brick(X) \vee on(X, sup(X))] \wedge [\neg brick(X) \vee \neg pyramid(sup(X))] \wedge [\neg brick(X) \vee \neg on(X, Y) \vee \neg on(Y, X)] \wedge [\neg brick(X) \vee brick(Z) \vee \neg equal(X, Z)])]$$

Step 7

: eliminate $\wedge$ by splitting

$$\forall X[\neg brick(X) \vee on(X, sup(X))]$$
$$\forall X[\neg brick(X) \vee \neg pyramid(sup(X))]$$
$$\forall X \forall Y[\neg brick(X) \vee \neg on(X, Y) \vee \neg on(Y, X)]$$
$$\forall X \forall Z[\neg brick(X) \vee brick(Z) \vee \neg equal(X, Z)])]$$

Step 8: rename variables (make unique)

$$\forall X[\neg brick(X) \vee on(X, sup(X))]$$
$$\forall W[\neg brick(W) \vee \neg pyramid(sup(W))]$$
$$\forall U \forall Y[\neg brick(U) \vee \neg on(U, Y) \vee \neg on(Y, U)]$$
$$\forall V \forall Z[\neg brick(V) \vee brick(Z) \vee \neg equal(V, Z)])]$$

Step 9: make clauses

```
on(X,sup(X)) :- brick(X).
:- brick(W), pyramid(sup(W)).
:- brick(U), on(U,Y), on(Y,U).
brick(Z) :- brick(V), equal(V,Z).
```

definite clause programs are universal

Definite clause programs are as powerful as any other programming language:

Theorem

Let f be an n-ary partial recursive function. There exists a definite clause program P_f and an $n+1$-ary predicate symbol p_f such that the query

$$:- p_f(s^{k_1}(0), \ldots, s^{k_n}(0), X)$$

returns

$$\{X/s^k(0)\}$$

iff

$$f(k_1, \ldots, k_n) = k$$

Outline

logic programming

sentences in clausal logic:

- have a declarative meaning (e.g. order of atoms in the body is irrelevant)
- have a procedural meaning

Thus clausal logic can be used as a programming language:

1. write down knowledge in a (declarative) program that specifies *what* the problem is rather than *how* it should be solved
2. apply inference rules to find solution

$$\textit{algorithm} = \textit{logic} + \textit{control}$$

algorithm = logic + control

where

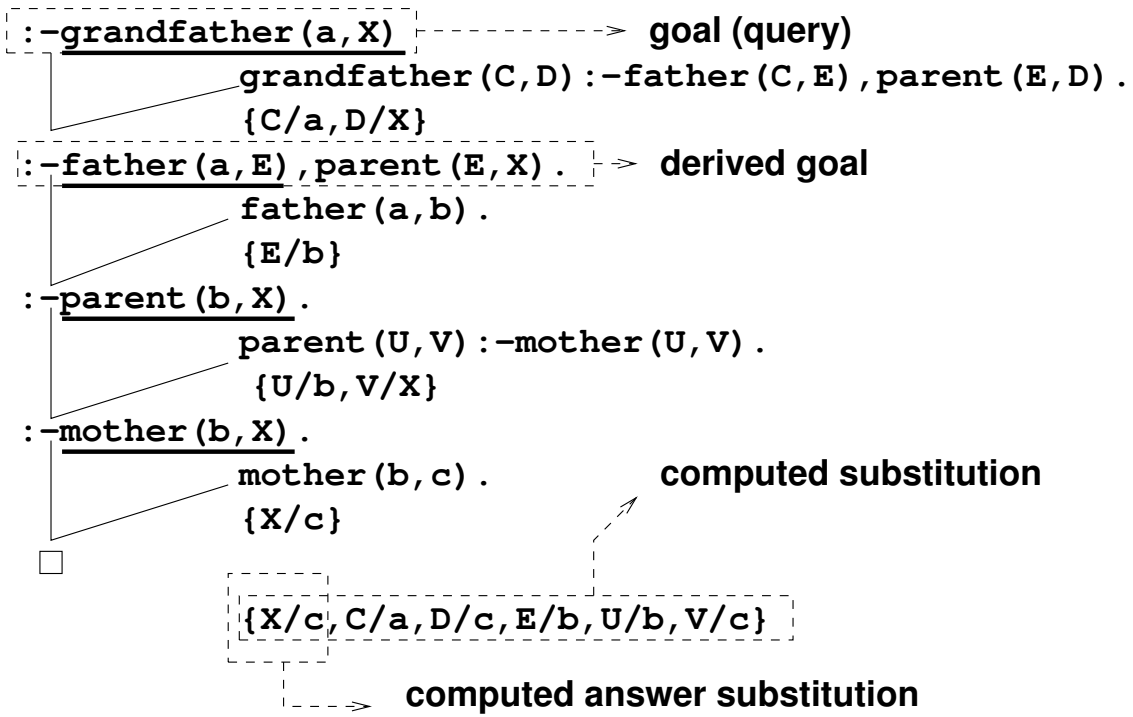
- *logic* is declarative knowledge and
- *control* is procedural knowledge

Prolog is not a purely declarative language since e.g. the order of the rules matters.
Prolog's proof procedure is based on resolution refutation in definite clause logic where a resolution strategy is fixed:

- which literal to resolve upon
- which clause to resolve with

sld refutation

```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).
```



SLD

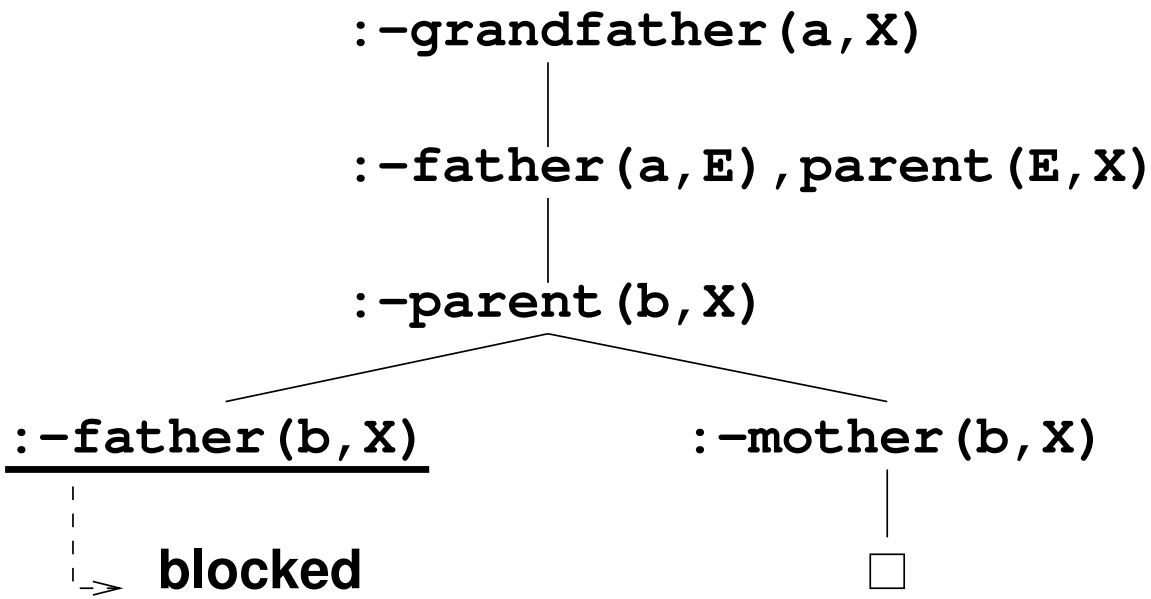
- Always resolve with (derived) goal: **L**inear proof trees.
- Use **S**election rule to determine literal in goal to resolve with.
- Programs with **D**efinite clauses only.

Prolog's selection rule:

- Consider goal literals left to right.
- (Try clauses in order of occurrence in program)

SLD tree: shows (only) alternative resolvents

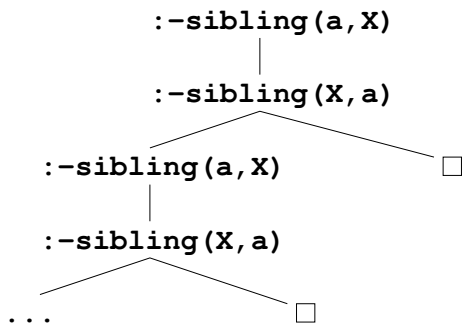
SLD trees



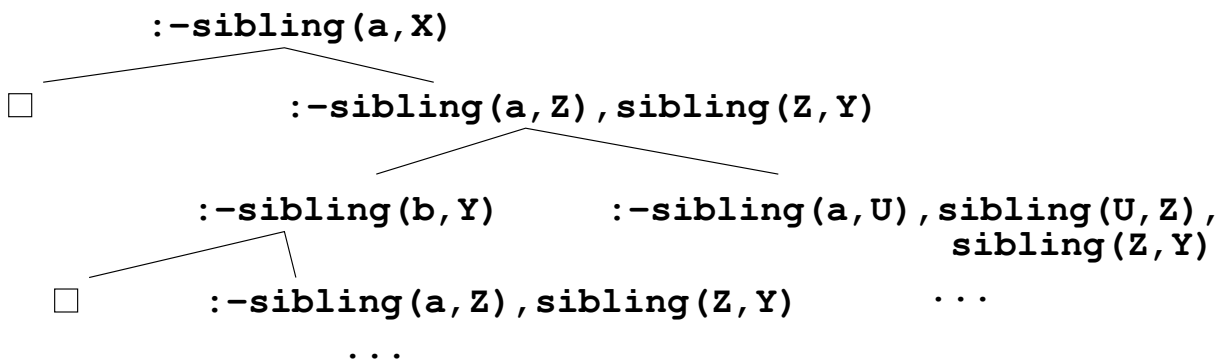
Every □ leaf corresponds to a successful refutation (a *success branch*). A blocked leaf corresponds to a *failed branch*.
Prolog does a depth-first traversal of an SLD tree.
What if an SLD tree has infinite branches?

infinite sld trees

```
sibling(X,Y) :- sibling(Y,X).
sibling(b,a).
```




```
sibling(a,b).
sibling(b,c).
sibling(X,Y) :- sibling(X,Z), sibling(Z,Y).
```



problems with SLD resolution

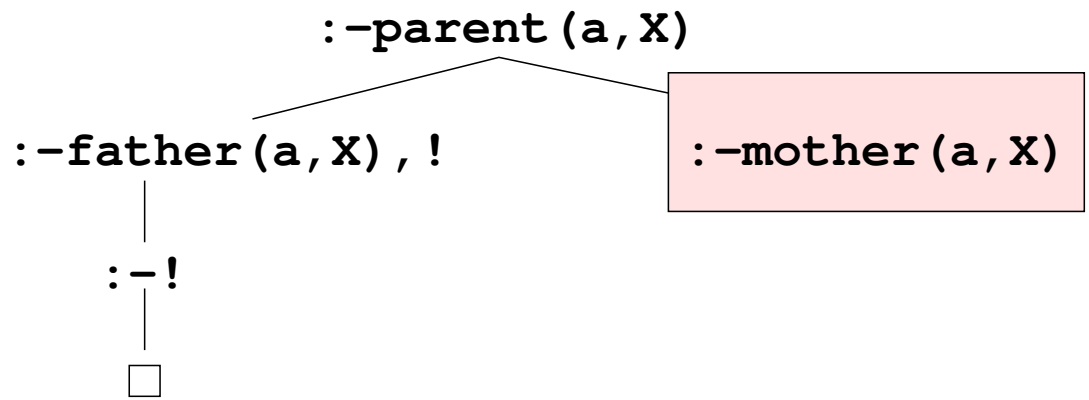
- Get trapped in infinite subtree: thus *Prolog is incomplete*. We could do e.g. breadth-first search but Prolog sacrifices completeness for efficiency (also in memory usage).
- Any infinite SLD tree may cause the interpreter to loop if no (more) answers are to be found: this is because clausal logic is only semi-decidable.

Thus one should be aware of the Prolog strategy (procedural knowledge), e.g.

- recursive clauses after non-recursive ones
- be careful with symmetric predicates: `p(X,Y) :- p(Y,X)`

pruning with “cut”

```
parent(X,Y) :- father(X,Y), !.
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).
```



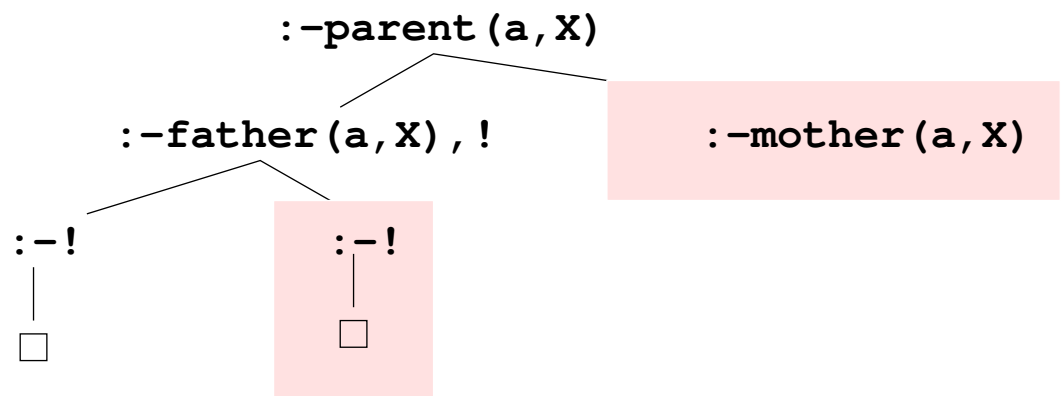
The meaning of “cut” (!): don’t try alternatives for

- the literals to the left of the cut
- the clause in which the cut is found

(A cut is always true.)

red & green cuts

```
parent(X,Y) :- father(X,Y), !.
parent(X,Y) :- mother(X,Y).
father(a,b).
father(a,c).
mother(m,b).
mother(m,c).
```



A **green** cut does not cut away any success branches. A **red** cut does, making the procedural meaning of the program different from the declarative meaning.

the dangers of “!”

```
max(M,N,M)  :- M>=N.
max(M,N,N)  :- M=<N.
```

More efficient using a red cut:

```
max(M,N,M)  :- M>=N,!.
max(M,N,N).
```

the dangers of “!”

```
max(M,N,M)  :- M>=N.
max(M,N,N)  :- M=<N.
```

More efficient using a red cut:

```
max(M,N,M)  :- M>=N,!.
max(M,N,N).
```

This is not equivalent! (try `max(5,3,3)`).
The program would be correct if only queries of the form

```
max(a,b,X)
```

were asked.

negation as failure

Cut can be used to ensure that the bodies of clauses are mutually exclusive.

```
p :- q,!,r.
p :- s. % only if q fails
```

is equivalent to

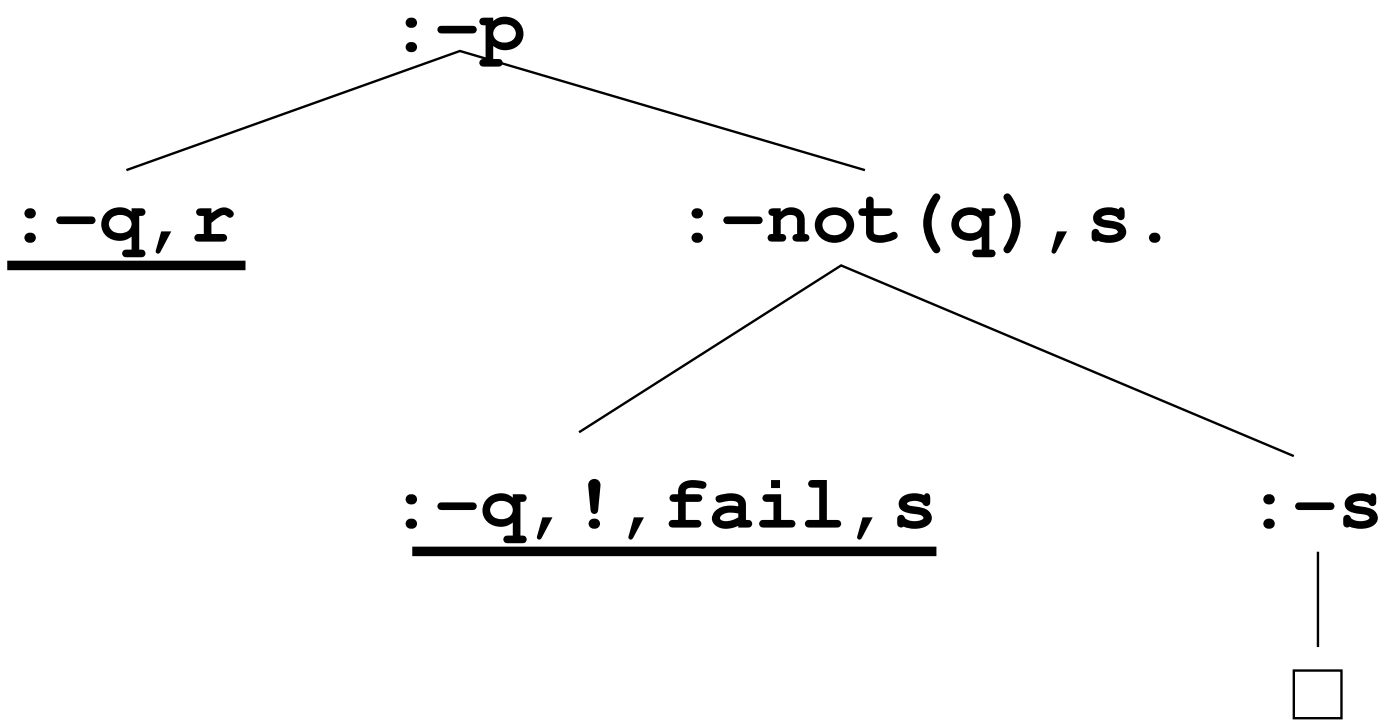
```
p :- q, r.
p :- not_q,s.
not_q :- q,!,fail.
not_q.
```

where **fail** is always false.
More general: meta-predicate `not/1` implementing negation by failure.

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

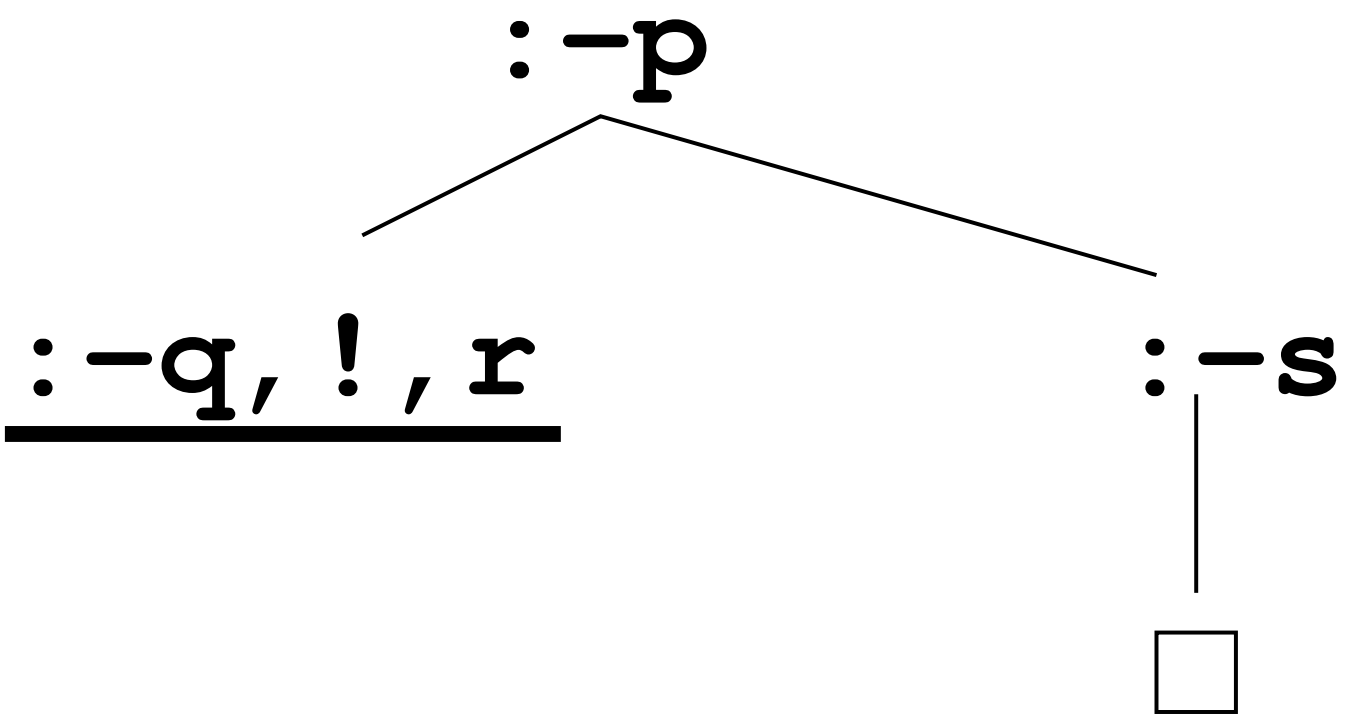
naf examples

```
p :- q, r.
p :- not(q), s.
s.
% not(q) :- q,!,fail.
% not(q).
```



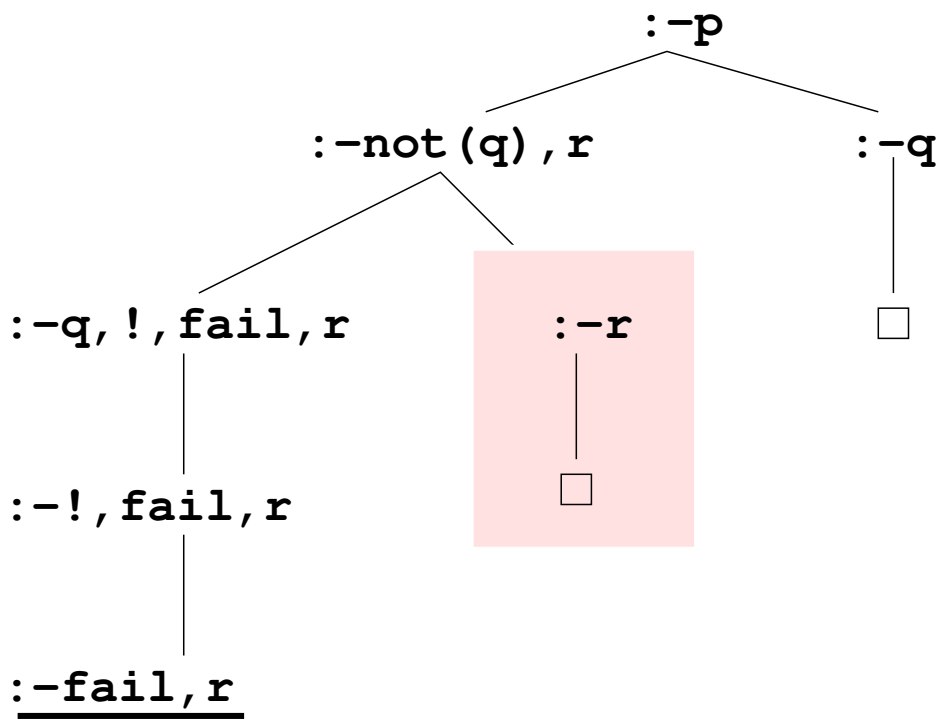
naf examples

```
p :- q,!,r.   % more efficient but less clear
p :- s.
s.
```



naf examples

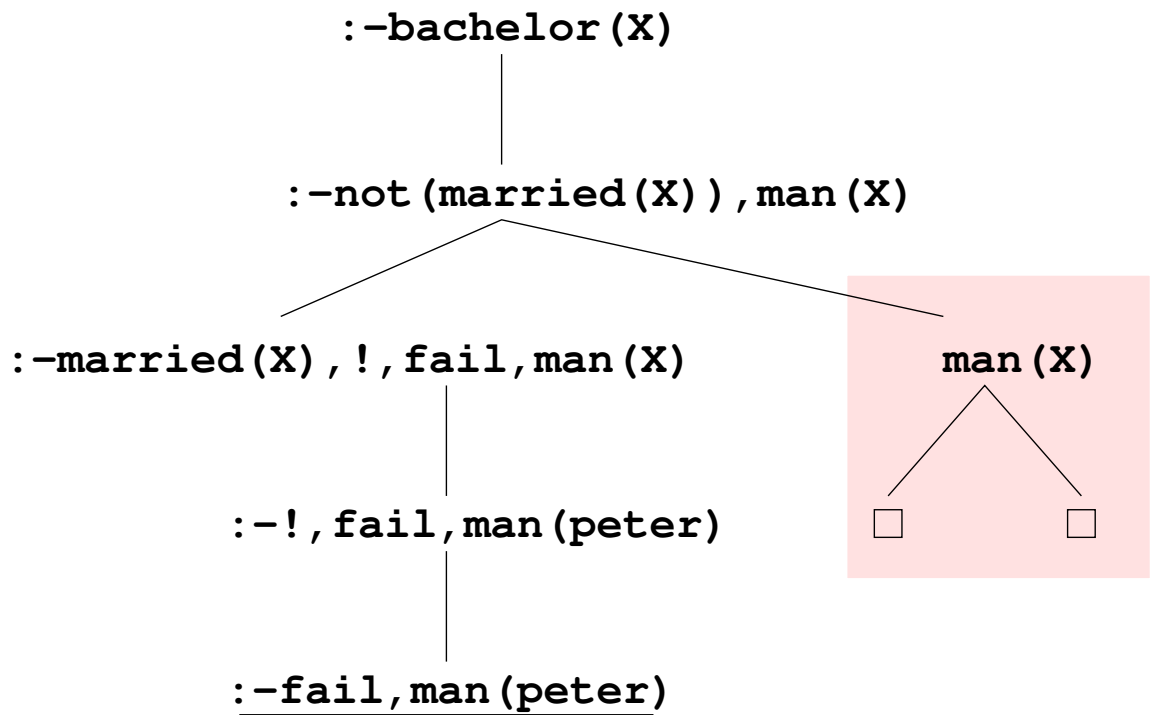
```
p :- not(q), r.
p :- q.
q.
r.
% not(q) :- q,!,fail.
% not(q).
```



floundering

occurs when the argument of `not/1` is not grounded.

```
bachelor(X) :- not(married(X)), man(X).
man(fred).
man(peter).
married(peter).
```



X is not a bachelor if *anyone* is married,..

sldnf resolution

- SLDNF resolution says that *not*(*Goal*) fails only if *Goal* has a refutation with an *empty* answer substitution (in the example: if `married(X).`).
- Prolog does not check this; hence Prolog is not sound w.r.t. negation by failure.
- If *Goal* is ground, only empty answer substitutions are possible...
- The example can be fixed by changing the order of the body of the rule:

```
bachelor(X) :- man(X), not(married(X)).
```

You can also read the original as

$$\forall X \cdot \neg(\exists Y : married(Y)) \wedge man(X) \Rightarrow bachelor(X)$$

if .. then .. else ..

```
p:- q,r,s,!,t.
p:- q,r,u.
q.
r.
u.
```

`q` , `r` are evaluated twice.

```
p :-  q,r,if_then_else(s,t,u).
if_then_else(S,T,U):- S,!,T.
if_then_else(S,T,U):- U.
```

In most prologs:

```
diagnosis(P,C):-  % C: condition, P: patient
    temperature(P,T),
    (T=<37          -> blood_pressure(P,C)
    ;T>37,T<38      -> Condition = ok
    ;otherwise      -> fever(P,C)
    ).
```

tail recursion and “!”

```
play(Board,Player):-
  lost(Board,Player).
play(Board,Player):-
  find_move(Board,Player,Move),
  make_move(Board,Move,NewBoard),
  next_player(Player,Next),!,
  play(NewBoard,Next).
```

Cut ensures that no previous moves are reconsidered and optimizes tail recursion to iteration.

arithmetic in prolog

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

add(0,X,X).
add(s(X),Y,s(Z)):- add(X,Y,Z).

mul(0,X,0).
mul(s(X),Y,Z):-
    mul(X,Y,Z1), add(Y,Z1,Z).
```

Not efficient!

`is(Result,expression)` is true iff `expression` can be evaluated as an expression and its resulting value unified with `Result`

```
?- X is 5+7-3
   X = 9
?- X is 5*3+7/2
   X = 18.5
```

`is/2` is different from `=/2` ; the latter succeeds if its arguments can be unified.

```
?- X = 5+7-3
   X = 5+7-3
?- 9 = 5+7-3
   no
?- X = Y+3
   X = _947+3
   Y = _947
?- X = f(X)
   X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f( ..
   error: term being written is too deep
```

The last example illustrates that Prolog does not implement the occur check.
Prolog also has other built-in arithmetic predicates: `<`, `>`, `=<`, `>=`.
`\=/2` succeeds if its arguments are **not** unifiable.

accumulators

Tail-recursive clauses are more efficient.

```
length([],0).
length([H|T],N) :-  length(T,N1), N is N1+1.
```

The program is not tail-recursive.
It can be made tail-recursive by introducing an accumulator:

Read `length_acc(L,M,N)` as $N = M + length(L)$.

```
length(L,N) :-  length_acc(L,0,N).
length_acc([],N,N).
length_acc([H|T],N0,N) :-
    % N0 is "length so far"
    N1 is N0+1, length_acc(T,N1,N).
```

reverse/2

```
naive_reverse([],[]).
naive_reverse([H|T],R) :-
    naive_reverse(T,R1),
    append(R1,[H],R).
append([],Y,Y).
append([H|T],Y,[H|Z]) :-
    append(T,Y,Z).
```

efficient reverse using accumulator

Define

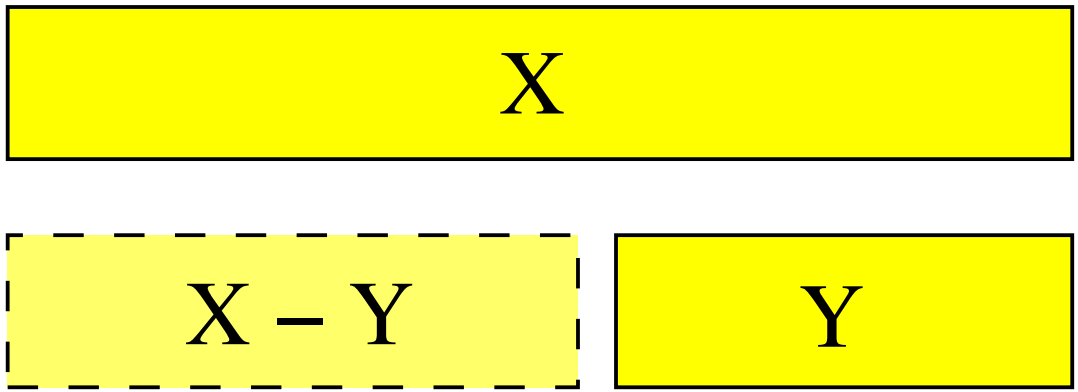
$$\textbf{reverse}(X, Y, Z) \Leftrightarrow Z = \mathit{reverse}(X) + Y$$

Then:

$$\begin{aligned}
\textbf{reverse}(X, [], Z) &\Leftrightarrow Z = \mathit{reverse}(X) \\
\\
\textbf{reverse}([H|T], Y, Z) &\Leftrightarrow Z = \mathit{reverse}([H|T]) + Y \\
&\Leftrightarrow Z = \mathit{reverse}(T) + [H] + Y \\
&\Leftrightarrow Z = \mathit{reverse}(T) + [H|Y] \\
&\Leftrightarrow \textbf{reverse}(T, [H|Y], Z)
\end{aligned}$$

```
reverse(X,Z) :-  reverse(X,[],Z).
reverse([],Z,Z).
reverse([H|T],Y,Z) :-
    % Y is "reversed so far"
    reverse(T,[H|Y],Z).
```

difference lists



Represent a list by a term `L1-L2`.

```
[a,b,c,d]-[d]       [a,b,c]
[a,b,c,1,2]-[1,2]   [a,b,c]
[a,b,c|X]-X         [a,b,c]
```

difference lists

$$\begin{aligned} \textbf{reverse}(X, Y, Z) &\Leftrightarrow Z = reverse(X) + Y \\ &\Leftrightarrow reverse(X) = Z - Y \end{aligned}$$

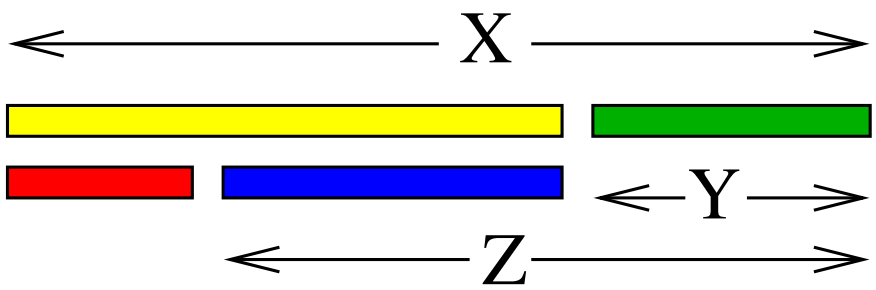
and

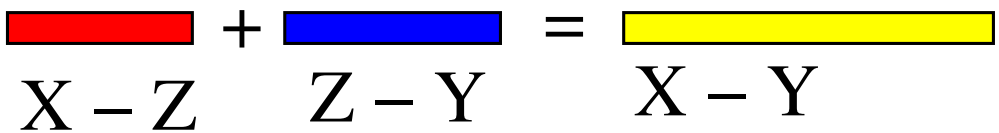
$$\begin{aligned} \textbf{reverse}([H|T], Y, Z) &\Leftrightarrow Z = reverse([H|T]) + Y \\ &\Leftrightarrow Z = reverse(T) + [H|Y] \\ &\Leftrightarrow reverse(T) = Z - [H|Y] \end{aligned}$$

```
reverse(X,Z) :-  reverse_dl(X,Z-[]).
reverse_dl([],Z-Z).
reverse_dl([H|T],Z-Y) :-
    reverse_dl(T,Z-[H|Y]).
```

appending difference lists

Difference lists can be appended in constant time:





append(X–Z , Z–Y, X–Y)

difference lists example

```
append_dl(X-Z, Z-Y, X-Y).

:-  append_dl([abc|A] - A, [de|B] - B, D)
% unify with append_dl(X-Z, Z-Y, X-Y)
X = [abc|A]
Z = A = [de|B]
Y = B
D = X - Y = [abc|A] - B = [abcde|B] - B
```

example: flatten/2

`atomic/1` succeeds if its argument is a simple constant.

```
flatten([X|Xs],Y) :-
    flatten(X,Y1), flatten(Xs,Y2),
    append(Y1,Y2,Y).
flatten(X,[X]) :-  atomic(X), X\=[].
flatten([],[]).
```

with difference lists:

```
flatten(X,Y) :-  flatten_dl(X,Y-[]).
flatten_dl([X|Xs],Y-Z) :-  % append flat(X), flat(Xs)
    flatten_dl(X,Y-Y1), flatten_dl(Xs,Y1-Z).
flatten_dl(X,[X|Xs]-Xs) :-
    atomic(X), X\=[].
flatten_dl([],U-U).
```

other incomplete data structures

```
lookup(Key,[(Key,Value)|Dict],Value).
lookup(Key,[(Key1,Value1)|Dict],Value) :-
    Key \= Key1,
    lookup(Key,Dict,Value).
```

Example: suppose `D = [(a,b),(c,d)|X]`

```
?- lookup(a,D,V)
V = b
?- lookup(c,D,e)
no
?- lookup(e,D,f)
yes
% D = [(a,b),(c,d),(e,f)|X]
```

second order predicates

```
map(R,[],[]).
map(R,[X|Xs],[Y|Ys]):-
    R(X,Y), map(R,Xs,Ys).
?-map(parent,[a,b,c],X)
```

Most systems do not allow `R(X,Y)` in the body.

`Term=..List` is true iff

- `Term` is a constant and `List` is the list `[Term]`
- `Term` is a compound term `f(A1,..,An)` and `List` is a list with head `f` and whose tail unifies with `[A1,..,An]`

```
map(R,[],[]).
map(R,[X|Xs],[Y|Ys]):-
    Goal =.. [R,X,Y],
    call(Goal), map(R,Xs,Ys).
```

findall/3

`findall(Term,Goal,Bag)` is true iff `Bag` unifies with the list of values to which a variable `X` not occurring in `Term` or `Goal` would be bound by successive resatisfactions of `(call(Goal), X=Term)` after systematic replacement of all variables in `X` by new variables.

```
parent(a,b).
parent(a,c).
parent(a,d).
parent(e,f).
parent(e,g).
children(Parent,Children):-
    findall(C,parent(Parent,C),Children).
?-children(a,Children)
  Children = [b,c,d]
```

bagof/3, setof/3

```
parent(a,b).  parent(a,c).  parent(a,d).
parent(e,f).  parent(e,g).
?-bagof(C,parent(P,C),L)
  C = _951
  P = a
  L = [b,c,d]
;
  C = _951
  P = e
  L = [f,g]
?-bagof(C,P^parent(P,C),L)
  C = _957
  P = _958
  L = [b,c,d,f,g]
```

`(P ^ parent(P,C))` reads like $\exists P : parent(P, C)$

`setof/3` is like `bagof/3` with duplicates removed.

assert/1, retract/1

Variables in Prolog are local to the clause. Global variables can be simulated using:

- `asserta(Clause)` adds `Clause` at the beginning of the Prolog database.
- `assertz(Clause)` adds `Clause` at the end of the Prolog database.
- `retract(Clause)` removes first clause that unifies with `Clause` from the Prolog database.

Note that backtracking does **not** undo the modifications.

```
% retract all clauses whose head unifies with ``Term''
retractall(Term):-
  retract(Term), fail.
retractall(Term):-
  retract((Term:- Body)), fail.
retractall(Term).
```

operators

In Prolog, functors and predicates are called *operators*. Operators can be declared using

```
:-  op(Priority,Type,Name)
```

where

- *Priority* is a number between 0 and 1200 (lower priority binds stronger)
- *Type* is *fx* or *fy* (prefix), *xfx*,*xfy* or *yfx* (infix), and *xf* or *yf* (postfix)
- The *x* and *y* determine associativity:

associative	no	right	left
	xfx	xfy	yfx
X op Y op Z	no	op(X,op(Y,Z))	op(op(X,Y),Z)

meta-programs

Clauses are represented as terms `:- (Head,Body)` where `:-` can be treated as a functor (meta-level) or as a predicate (object-level).

```
% if A and B then C = if(then(and(A,B),C))
:- op(900,fx,if).
:- op(800,xfx,then).
:- op(700,yfx,and).
% object-level rules
if has_feathers and lays_eggs then is_bird.
if has_gills and lays_eggs then is_fish.
if tweety then has_feathers.
if tweety then lays_eggs.
```

it should be possible to show that

```
if tweety then is_bird
```

follows from the object-level rules.

meta-program

```
derive(if Assumptions then Goal):-
  if Body then Goal,
  derive(if Assumptions then Body).
derive(if Assumptions then G1 and G2):-
  derive(if Assumptions then G1),
  derive(if Assumptions then G2).
derive(if Assumptions then Goal):-
  assumed(Goal,Assumptions). % Goal is one of the assumptions
assumed(A,A).
assumed(A,A and As).
assumed(A,B and As):-
  assumed(A,As).
```

prolog meta-interpreter

```
prove(Goal):-
     clause(Goal,Body),
     prove(Body).
prove((Goal1,Goal2)):-
     prove(Goal1),
     prove(Goal2).
prove(true).
```

or, more conventionally, and adding negation as failure:

```
prove(true):- !.
prove((A,B)):- !,
     prove(A), prove(B).
prove(not(Goal)):- !,
   not(prove(Goal)).
prove(A):-
     % not (A=true; A=(X,Y); A=not(G))
     clause(A,B), prove(B).
```

quicksort

```
% partition(l,n,Smalls,Bigs):
%   Smalls contains numbers in l
%   smaller than n, Bigs the rest.
partition([],N,[],[]).
partition([H|T],N,[H|Small],Big):-
    H<N,partition(T,N,Small,Big).
partition([H|T],N,Small,[H|Big]):-
    H>=N,partition(T,N,Small,Big).

quicksort([],[]).
quicksort([X|Xs],Sorted):-
    partition(Xs,X,Small,Big),
    quicksort(Small,S_Small),
    quicksort(Big,S_Big),
    append(S_Small,[X|S_Big],Sorted).
```

towers of hanoi

```
:- op(900,xfx,to).
% hanoi(N,A,B,C,Moves): Moves is the list of moves to
% move N disks from peg A to peg C, using peg B as
% an intermediary.
hanoi(0,A,B,C,[]).
hanoi(N,A,B,C,Moves):-
  N1 is N-1, % assume solved for N-1 disks
  hanoi(N1,A,C,B,Moves1),
  hanoi(N1,B,A,C,Moves2),
  append(Moves1,[A to C|Moves2],Moves).

?- hanoi(3,left,middle,right,M)
   M = [ left to right, left to middle,
       right to middle, left to right,
       middle to left, middle to right,
       left to right ]
```

Outline

Representing structured knowledge

- Knowledge is structured if its components have certain logical relationships.
- Explicit relationships are represented directly (as facts).
- Implicit relationships are found by reasoning.
- Reasoning is often done by searching, e.g. in a graph, a term etc.

tree as terms adt

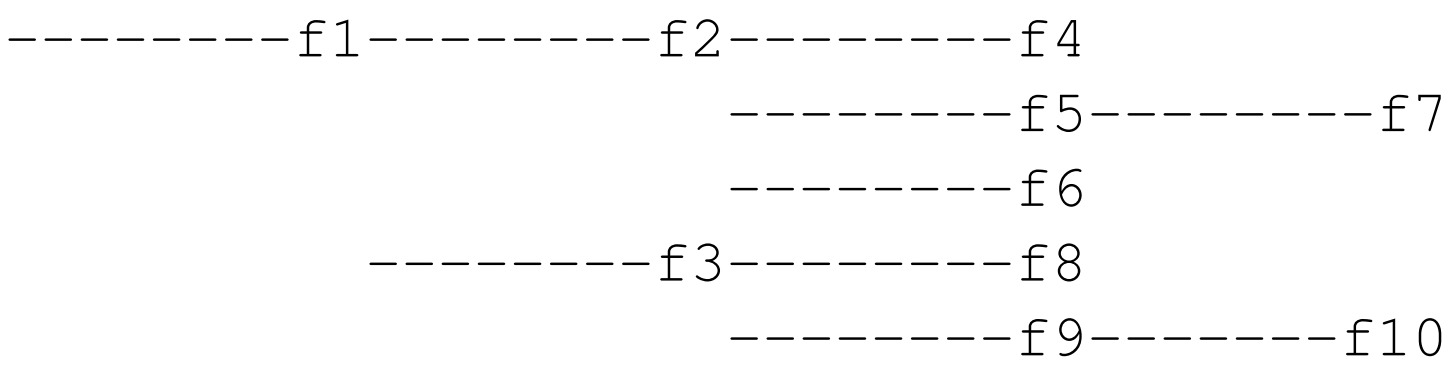
```
% term_tree(Tree,R,S): term Tree represents
% a tree with root R, list of subtrees S
term_tree(Tree, Root, Subtrees):-
    Tree =.. [Root|Subtrees].
% term_root(Tree, Root): R is root of T
term_root(Tree, Root):-
    term_tree(Tree, Root, Subtrees).
% term_subtree(Tree, Subtree): Subtree is a subtree
term_subtree(Tree, Subtree):-
    term_tree(Tree, Root, Subtrees),
    element(Subtree, Subtrees).
% element(X,Ys): X is element in list
element(X,[X|Ys]).
element(X,[Y|Ys]):-
    element(X,Ys).
```

```
% term_arc(Tree, Arc): A is arc in Tree
term_arc(Tree, [Root, SubRoot]):-  % from root
  term_root(Tree, Root),
  term_subtree(Tree, SubTree),
  term_root(Subtree, SubRoot).
term_arc(Tree, Arc):-  % in subtree
  term_subtree(Tree, Subtree),
  term_arc(Subtree, Arc).
% term_path(Tree, Path): Path is path in Tree
term_path(Tree, Path):-  % an arc is a path
  term_arc(Tree, Path).
term_path(Tree, [Node1,Node2|Nodes]):-
  term_arc(Tree,[Node1,Node2]),
  term_path(Tree,[Node2|Nodes]).
```

writing terms as trees

```
term_write(Tree):-
  term_write(0,Tree), nl.
term_write(Indent,Tree):-
  term_tree(Tree, Root, Subtrees),
  term_write_node(Indent, NewIndent, Root),
  term_write_subtrees(NewIndent, Subtrees).
term_write_subtrees(Indent,[]).
term_write_subtrees(Indent,[Tree]):- !,
  term_write(Indent,Tree).
term_write_subtrees(Indent,[Tree|Subtrees]):-
  term_write(Indent,Tree),
  nl,tabs(Indent), term_write_subtrees(Indent, Subtrees).
term_write_node(Begin,End,Node):-
  name(Node,L), length(L,N),
  End is Begin+10, N1 is End-Begin-N,
  write_line(N1), write(Node).
```

```
write_line(0).
write_line(N):-
  N>0, N1 is N-1,
  write('-'), write_line(N1).


:-  term_write(f1(f2(f4,f5(f7),f6),f3(f8,f9(f10))))
```



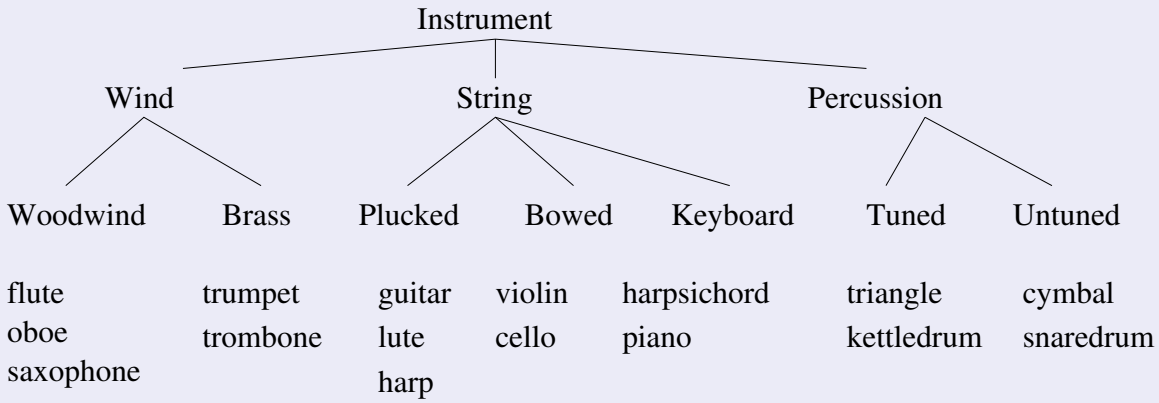
graphs generated by a predicate

```
% path(P): P is a list of nodes representing a path
% in graph defined by arc/2.
path([N1, N2]):-
  arc(N1, N2).
path([N1, N2|Nodes]):-
  arc(N1, N2), path([N2|Nodes]).
% path_leaf(N, Path): Path is a path starting at N, ending
% in a leaf in graph generated by arc/2.
path_leaf(Leaf, [Leaf]):-
  leaf(Leaf).
path_leaf(N1, [N1|Nodes]):-
  arc(N1, N2),
  path_leaf(N2, Nodes).
%
leaf(Leaf):-  % no outgoing arcs
  not(arc(Leaf, SomeNode)).
```

sld trees

```
% resolve(goal, clause, newGoal):
%   newGoal is resolvent of goal using clause
resolve([Literal|Literals], (Head:- Body), NewGoal):-
  Literal = Head, % unify with head
  append(Body, Literals, NewGoal).
% an arc in an SLD tree
arc(Goal1, Goal2):-
  clause(Head, Body),
  resolve(Goal1, (Head:- Body), Goal2).
prove(Goal):-
  path(Goal, []).

% where
path(N1, N2) :-
  arc(N1, N2).
path(N1, N3) :-
  arc(N1, N2), path(N2, N3).
```

inheritance hierarchies



A class is represented by a unary predicate, an object by a constant.

```
instrument(X) :- wind(X).
instrument(X) :- string(X).
instrument(X) :- percussion(X).
wind(X) :- woodwind(X).
wind(X) :- brass(X).
string(X) :- plucked(X).
string(X) :- bowed(X).
string(X) :- keyboard(X).
percussion(X) :- tuned(X).
percussion(X) :- untuned(X).
```

```
woodwind(flute).          brass(trumpet).
plucked(guitar).          bowed(violin).
keyboard(piano).          tuned(triangle).
untuned(cymbal).
```

Properties:

```
material(flute, metal).
% string instruments are made of wood
material(X, wood) :-  woodwind(X).
material(X, wood) :-  string(X).
material(X, metal) :-  brass(X).
material(X, metal) :-  percussion(X).
?- material(piano, X)
   X = wood
?- material(flute, X)
   X = metal;
   X = wood
```

Putting most specific clauses first ensures that the first answer is correct.

```
function(X, musical) :-  instrument(X).
action(oboe, reed(double)).
action(saxophone, reed(single)).
action(piano, hammered).
action(X, hammered) :-  percussion(X).
```

What are the properties of an object *I*?

```
attributes([material,action,function]).
properties(I, Props):-  attributes(Attrs),
                        properties(Attrs, I, Props).
properties([], Instance, []).
properties([Attribute|Attributes],
    Instance, [Attribute=Val|Props]):-
  get_value(Attribute, Instance, Val) ,!, % first only
  properties(Attributes, Instance, Props).
get_value(Attribute, Instance, Value):-
  Goal =.. [Attribute, Instance, Value], call(Goal).
```

```
?- properties(saxophone,P)
   P = [ material = metal,
         action=reed(single),
         function = musical]
```

- Questions about classes are not easy to answer since one must resort to second-order programming.

$\Rightarrow$ Design alternative representation where both classes and instances are represented by terms.

semantic networks

Represent hierarchy as set of facts.

```
isa(wind,instrument).
isa(woodwind,wind).
isa(brass,wind).
% etc.
inst(oboe,woodwind).
inst(flute,woodwind).
inst(trumpet,brass).
% etc.
% class properties:
prop(instrument,function,musical).
prop(woodwind,material,wood).
prop(brass,material,metal).
prop(brass,action,reed(lip)).
% instance properties
prop(flute,material,metal).
prop(oboe,action,reed(double)).
% ..
```

properties in semantic networks

```
properties(Instance, Properties):-
  direct_properties(Instance, InstanceProperties),
  inst(Instance, Class), % inherit rest
  inherit(Class, InstanceProps, Properties).
direct_properties(Instance, InstanceProperties):-
  findall(Attribute=Value,
          prop(Instance,Attribute,Value),
          InstanceProperties).
%
isa(instrument,top).

inherit(top, Properties, Properties),
inherit(Class, Properties, AllProperties):-
  direct_properties(Class, ClassProperties),
  override(Properties, ClassProperties,
           ExtendedProperties),
  isa(Class, SuperClass),
  inherit(SuperClass, ExtendedProperties, AllProperties).
```

```
% override(SpecificProps,ClassProps,Ps): Ps contains all
% SpecificProps and those ClassProps that are not
% overridden by SpecificProps.
override(Properties, [], Properties).
override(Properties, [Attr=AnyValue|ClassProperties],
                     ExtendedProperties):-
  element(Attr=Value, Properties),
  override(Properties, ClassProperties,
                       ExtendedProperties).
override(Properties, [Attr=Value|ClassProperties],
         [Attr=Value|ExtendedProperties]):-
  not(element(Attr=AnyValue, Properties)),
  override(Properties, ClassProperties,
                       ExtendedProperties).
```

frame-based inheritance

Add property list to each arc in the network.

```
isa(instrument, top, [function=musical]).
isa(wind, instrument, []).
isa(woodwind, wind, [material=wood]).
isa(brass, wind, [material=metal,action=reed(lip)]).
%
instance(flute, woodwind, [material=metal]).
instance(oboe, woodwind, [action=reed(double]).
instance(trumpet, brass, []).
```

frame-based inheritance 2

```
properties(Instance, Properties):-
  instance(Instance, Class, InstanceProperties),
  inherit(Class, InstanceProperties, Properties).
inherit(top, Properties, Properties).
inherit(Class, Properties, AllProperties):-
  class(Class, SuperClass, ClassProperties),
  override(Properties, ClassProperties,
                       ExtendedProperties),
  inherit(SuperClass, ExtendedProperties, AllProperties).
```

Outline

searching graphs

A *search space* is a graph with one or more *starting nodes* and one or more *goal nodes*. A *solution* is a path from a start node to a goal node. A *cost function* assigns a cost to each arc. An *optimal solution* is a solution with minimal cost.
Search algorithms differ w.r.t.

- completeness: is a solution always found?
- optimality: will shorter paths be found before longer ones?
- efficiency of the algorithm.

A general framework

```
% search(agenda,Goal): agenda contains reached but
% untried nodes.
% succeeds if a node Goal, for which goal(Goal), can
% be reached from a node in Agenda.
search(Agenda, Goal):-
  % select/3 selects a node from the Agenda
  select(Agenda, Goal, RestOfAgenda),
  goal(Goal).
search(Agenda, Goal):-
  select(Agenda, CurrentNode, RestOfAgenda),
  children(CurrentNode, Children),
  add(Children, RestOfAgenda, NewAgenda),
  search(NewAgenda, Goal).
```

Different algorithms result from different implementations of `select/3` and `add/3` .

depth-first search

- Agenda is a list (stack).
- `select/3` selects the first node of the list
- `add/3` puts children in front of the new agenda

```
search_df([Goal|RestOfAgenda], Goal):-
  goal(Goal).
search_df([CurrentNode|RestOfAgenda], Goal):-
  children(CurrentNode, Children),
  append(Children, RestOfAgenda, NewAgenda),
  search_df(NewAgenda, Goal).

children(Node, Children):-
  findall(Child, arc(Node,Child), Children).
```

depth-first search with paths

Return path to goal: keep paths instead of nodes in agenda.

```
children([Node|RestOfPath], Children):-
  findall([Child,Node|RestOfPath],
          arc(Node,Child),
          Children).
?- search_df([[initial_node]], PathToGoal).
```

depth-first search with loop detection

Loop detection: keep list of visited nodes.

```
search_df([Goal|RestOfAgenda], VisitedNodes, Goal):-
  goal(Goal).
search_df([Node|RestOfAgenda], VisitedNodes, Goal):-
  children(Node ,Children),
  add_df(Children, RestOfAgenda,
         [Node|VisitedNodes], NewAgenda),
  search_df(NewAgenda, [Node|VisitedNodes], Goal).
% add_df(Nodes, Agenda, VisitedNodes, NewAgenda)
add_df([], Agenda, VisitedNodes, Agenda).
add_df([Node|Nodes], Agenda, VisitedNodes, [Node|NewAgenda]):-
  not(element(Node, Agenda)),
  not(element(Node, VisitedNodes),
  add_df(Nodes, Agenda, VisitedNodes, NewAgenda).
add_df([Node|Nodes], Agenda, VisitedNodes, NewAgenda):-
  element(Node, Agenda),
  add_df(Nodes, Agenda, VisitedNodes, NewAgenda).
add_df([Node|Nodes], Agenda, VisitedNodes, NewAgenda):-
  element(Node, VisitedNodes),
  add_df(Nodes, Agenda, VisitedNodes, NewAgenda).
```

depth-first search with agenda on prolog stack

Using Prolog's goal stack to keep agenda, but without loop detection:

```
search_df(Goal,Goal):-
  goal(Goal).
search_df(CurrentNode, Goal):-
  arc(CurrentNode, Child),
  search_df(Child, Goal).
```

depth-first search with depth bound

An incomplete version with a depth bound:

```
search_bd(Depth, Goal, Goal):-
  goal(Goal).
search_bd(Depth, CurrentNode, Goal):-
  Depth>0, NewDepth is Depth-1,
  arc(CurrentNode, Child),
  search_bd(NewDepth, Child, Goal).
?- search_df(10,initial_node,Goal).
```

depth-first search with iterative deepening

Iterative deepening (e.g. in chess):

```
search_id(CurrentNode, Goal):-
  search_id(1, CurrentNode, Goal).
search_id(Depth, CurrentNode, Goal):-
  search_bd(Depth, CurrentNode, Goal).
search_id(Depth, CurrentNode, Goal):-
  NewDepth is Depth+1,
  search_id(NewDepth, CurrentNode, Goal).
```

breadth-first search

- Agenda is a list (queue).
- `select/3` selects the first node of the list
- `add/3` puts children at the back of the new agenda

```
search_bf([Goal|RestOfAgenda], Goal):-
  goal(Goal).
search_bf([CurrentNode|RestOfAgenda], Goal):-
  children(CurrentNode, Children),
  append(RestOfAgenda ,Children, NewAgenda),
  search_bf(NewAgenda, Goal).

children(Node, Children):-
  findall(Child, arc(Node,Child), Children).
```

a full clause refutation engine

- Use breadth-first search.
- Clause representation:

```
clause(([bach(X),married(X)] :-  [man(X),adult(X)])).
clause(([] :-  [has_wife(paul)])). % empty head
clause(([] :-  [])). % empty clause
```

- Because `findall(X,G,L)` creates new variables for the unbound variables in X before putting it in L, we keep a copy of the original goal in order to be able to retrieve the computed substitution.
- The agenda is a list of pairs

```
agenda_item(SubGoalClause, OriginalGoal)
```

```
refute(GoalClause):-
  refute([agenda_item(GoalClause,GoalClause)], GoalClause).

% The following clause unifies two versions of the
% original clause where the first version contains
% the answer substitution.
refute([ agenda_item(([]:- []), GoalClause) | _ ], GoalClause).

refute([agenda_item(InputClause,GoalClause)|RestOfAgenda],
       OriginalGoalClause):-
  findall( agenda_item(Resolvent, GoalClause),
           (  clause(Resolver),
              resolve(InputClause, Resolver, Resolvent) ),
           Children),
  append(RestOfAgenda, Children, NewAgenda), % breadth-first
  refute(NewAgenda, OriginalGoalClause).
```

```
% resolve(Clause1,Clause2,R) iff R is a resolvent
% of Clause1 and Clause2.
resolve((H1:-B1), (H2:-B2), (ResHead:-ResBody)):-
  % remove common literals from H1, B2, yielding R1, R2
  remove_common_element(H1, B2, R1, R2),
  append(R1, H2, ResHead),
  append(B1, R2, ResBody).

resolve((H1:- B1), (H2:-B2), (ResHead:-ResBody)):-
  remove_common_element(H2, B1, R2, R1),
  append(H1, R2, ResHead),
  append(R1, B2, ResBody).
```

```
% remove_common_element(+L1, +L2, -R1, -R2)
%  iff (roughly) exists X such that els(Li) = els(Ri) + {X}
%  (note that, necessarily, els(Li) not empty
remove_common_element([A|B], C, B, E) :-
  remove_element(A, C, E).
remove_common_element([A|B], C, [A|D], E) :-
  remove_common_element(B, C, D, E).

% remove_element(+A,+L,-R)
%  iff (roughly) els(L) = els(R) + {A}
remove_element(A, [A|B], B).
remove_element(A, [C|B], [C|D]) :-
 A\=C, remove_element(A, B, D).
```

```
clause(([bachelor(X),married(X)]:- [man(X),adult(X)])).
clause(([has_wife(X)]:- [man(X),married(X)])).
clause(([]:- [has_wife(paul)])).
clause(([man(paul)]:- [])).
clause(([adult(paul)]:- [])).

?- refute(([] :- [bach(X)]))
   X = paul
```

The resolution strategy ("**input resolution**": every resolvent has at least one program clause as its parent) used is incomplete for general clauses (but complete for definite ones).

forward chaining

```
% model(-M) iff M is a model of the clauses defined by cl/1
model(M):-
    model([], M).

% model(+M0,-M) iff M0 can be extended to a model M
% of the cl/1 clauses.
model(M0, M):-
    clause((H:- B)),
    % find violated clause instance
    is_violated((H:- B), M0),!,
    element(L, H),   % select ground literal from the head
    model([L|M0], M).   % and add it to the model
model(M, M). % no violated clauses

% is_violated((H:- B),+M) iff instance of H:-B
% is violated by M
is_violated((H:- B), M):-
    satisfied_body(B, M), % this will ground the variables
    not(satisfied_head(H, M)).
```

```
% satisfied_body(L,+M) iff M |= A for all A in L,
% may bind vars in L
satisfied_body([], M).
satisfied_body([A|B], M) :-
  element(A, M),
  satisfied_body(B, M).

% satisfied_head(+L,+M) iff exists A in els(L)
% with  M |= A
satisfied_head(L,M):-
   element(A, L), element(A, M).

element(A, [A|_]).
element(A, [_|B]) :-
  element(A, B).
```

```
clause(([bach(X),married(X)]:- [man(X),adult(X)])).
clause(([has_wife(X)]:- [man(X),married(X)])).
clause(([man(paul)]:- [])).
clause(([adult(paul)]:- [])).

?- model(M)
   M = [has_wife(paul), married(paul),
        adult(paul), man(paul)];
   M = [bach(paul), adult(paul), man(paul)]
```

The program works correctly only for clauses for which grounding the body also grounds the head.

```
clause(([man(X),woman(X)]:- [])).
clause(([]:- [man(jane)])). % jane is not a man
clause(([]:- [woman(peter)])). % peter is not a woman
```

range-restricted clauses

This can be fixed:

```
clause(([man(X),woman(X)]:- [person(X)])).
clause(([person(jane)]:- [])).
clause(([person(peter)]:- [])).
clause(([]:- [man(jane)])).
clause(([]:- [woman(peter)])).
```

Range-restricted clauses: where all variables in the head also occur in the body.
Any program can be transformed into an equivalent one using only range-restricted clauses.

Outline

best-first search

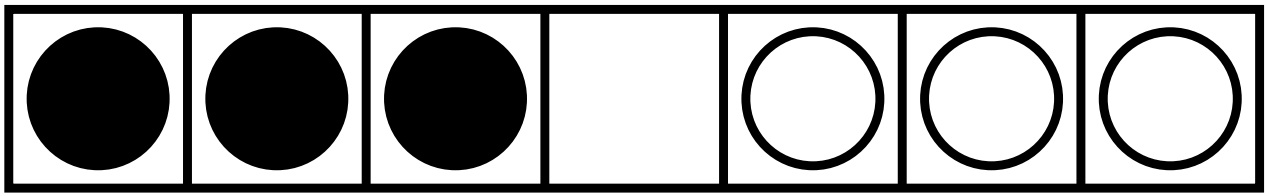
Informed search uses an heuristic estimate of the distance from a node to a goal.

```
% eval(Node,Value) estimates distance from node to goal

search_best([Goal|RestAgenda], Goal):-
  goal(Goal).
search_best([CurrentNode|RestAgenda], Goal):-
  children(CurrentNode, Children),
  add_best(Children, RestAgenda, NewAgenda),
  search_best(NewAgenda, Goal).
% add_best(A,B,C): C contains
%  els from A,B sorted according to eval/2
add_best([], Agenda, Agenda).
add_best([Node|Nodes], Agenda, NewAgenda):-
  insert(Node, Agenda, TmpAgenda),
  add_best(Nodes, TmpAgenda, NewAgenda).
```

```
insert(Node, Agenda, NewAgenda):-
  eval(Node, Value),
  insert(Value, Node, Agenda, NewAgenda).

insert(Value, Node, [], [Node]).
insert(Value, Node, [FirstNode|RestOfAgenda],
       [Node, FirstNode|RestOfAgenda]):-
  eval(FirstNode, FirstNodeValue),
  Value < FirstNodeValue.
insert(Value, Node, [FirstNode|RestOfAgenda],
       [FirstNode|NewRestOfAgenda]):-
  eval(FirstNode, FirstNodeValue),
  Value >= FirstNodeValue,
  insert(Value, Node, RestOfAgenda, NewRestOfAgenda).
```

Best-first search is not complete since, with certain estimate functions, it may get lost in an infinite subgraph.

example puzzle



- A tile may be moved to the empty spot if there are at most 2 tiles between it and the empty spot.
- Find a series of moves that bring all the black tiles to the right of all the white tiles.

representing and manipulating the board

A **position** of the board is represented using a list, e.g.

`[b,b,b,e,w,w,w]`

```
% get_tile(position,n,Tile): position[n]=Tile
get_tile(Position, N, Tile) :-
  get_tile(Position, 1, N, Tile).
get_tile([Tile|Tiles], N, N, Tile).
get_tile([Tile|Tiles], N0, N, FoundTile) :-
  N1 is N0+1,
  get_tile(Tiles, N1, N, FoundTile).

% replace(position,n,t,B): B is position with board[n]=t
replace([Tile|Tiles], 1, ReplacementTile,
        [ReplacementTile|Tiles]).
replace([Tile|Tiles], N, ReplacementTile,
        [Tile|RestOfTiles]):-
  N>1, N1 is N-1,
  replace(Tiles, N1, ReplacementTile, RestOfTiles).
```

representing the agenda

- A move is represented by a term

  `move(FromPosition, ToPosition, Cost).`

- The start move:

```
start_move( move(noparent, [b,b,b,e,w,w,w], 0) ).
```

- showing a move (in a sequence)

```
show_move( move(OldPosition, NewPosition, Cost), Value):-
  write(NewPosition-Value), nl.
```

- An agenda is list of terms `move_value(Move, Value)` where `Value` is the heuristic evaluation of the position reached by `Move`.

tiles/2

```
tiles(ListOfPositions, TotalCost):-
  start_move(StartMove),
  % Value is heuristic of distance to goal
  eval(StartMove, Value),

  % best-first search accumulating moves
  tiles([move_value(StartMove, Value)], FinalMove,
    [], VisitedMoves), % accumulator

  % find (and print) a backward path and its cost in
  % VisitedMoves from the final move to the start move
  order_moves(FinalMove, VisitedMoves,
    [], ListOfPositions, % accumulator
    0, TotalCost). % accumulator
```

tiles/4

```
% tiles(Agenda, LastMove, V0, V): goal can be
% reached from a move in Agenda where LastMove
% is the last move leading to the goal,
% and V is V0 + the set of moves tried.
tiles([move_value(LastMove,Value)|RestAgenda], LastMove,
      VisitedMoves, VisitedMoves):-
  goal(LastMove). % eval(LastMove, 0), i.e. goal reached

tiles([move_value(Move,Value)|RestAgenda], Goal,
                          VisitedMoves, FinalVisitedMoves):-
  show_move(Move, Value), % show move ``closest to goal''
  % find and evaluate possible next moves from M
  setof0( move_value(NextMove, NextValue),
          ( next_move(Move, NextMove),
            eval(NextMove, NextValue) ),
          Children),
  merge(Children, RestAgenda, NewAgenda),
  tiles(NewAgenda, Goal,
        [Move|VisitedMoves], FinalVisitedMoves).
```

next_move/2

```
next_move( move(Position, LastPosition, LastCost),
           move(LastPosition, NewPosition, Cost) ) :-
  % consecutive moves: NewPosition can be reached from
  %   LastPosition in 1 move at cost Cost
  % Ne = index of empty spot
  get_tile(LastPosition, Ne, e),
  % Nbw = index of nonempty spot
  get_tile(LastPosition, Nbw, BW), not(BW=e),
  Diff is abs(Ne-Nbw), Diff<4, % not too far from Ne
  replace(LastPosition, Ne, BW, IntermediatePosition),
  replace(IntermediatePosition, Nbw, e, NewPosition),
  (
    Diff=1 -> Cost=1
  ; otherwise -> Cost is Diff-1
  ).
```

```
% order_moves(FinalMove, VisitedMoves,
%              Positions, FinalPositions,
%              TotalCost,FinalTotalCost):
% FinalPositions = Positions + connecting sequence of
%  target positions from VisitedMoves ending in
%  FinalMove's target position.
% FinalTotalCost = TotalCost + total cost of moves
%  added to Positions to obtain FinalPositions.
order_moves(move(noparent,StartPosition,0), VisitedMoves,
            Positions, [StartPosition|Positions],
            TotalCost, TotalCost).
order_moves(move(FromPosition, ToPosition, Cost),
            VisitedMoves,
            Positions, FinalPositions,
            TotalCost, FinalTotalCost):-
  element( PreviousMove, VisitedMoves),
  PreviousMove = move(PreviousPosition, FromPosition,
                      CostOfPreviousMove),
  NewTotalCost is TotalCost + Cost,
  order_moves(PreviousMove, VisitedMoves,
         [ToPosition|Positions], FinalPositions,
         NewTotalCost, FinalTotalCost).
```

utilities

```
% setof0/3: variant of setof/3
%  which succeeds with empty list if no solutions are found
setof0(X, G, L):-
  setof(X, G, L), !.
setof0(X, G, []).

merge([], Agenda, Agenda).
% avoid succeeding twice on merge([],[],L).
merge([C|Cs],[],[C|Cs]).
merge([C|Cs],[N|Ag],[C|NewAg]):-
  eval(C,CVal),
  eval(N,NVal),
  CVal<NVal,
  merge(Cs,[N|Ag],NewAg]).
merge([C|Cs],[N|Ag],[N|NewAg]):-
  eval(C,CVal),
  eval(N,NVal),
  CVal>=NVal,
  merge([C|Cs],Ag,NewAg]).
```

eval/1

```
goal(Move):-
  eval(Move,0).

eval(move(OldPosition,Position,C),Value):-
  bLeftOfw(Position,Value).

% Val is the sum of the number of black tiles
% to the left of each white tile
bLeftOfw(Pos,Val):-
  findall((Nb,Nw),
          ( get_tile(Pos,Nb,b),
            get_tile(Pos,Nw,w), Nb<Nw),
          L),
  length(L,Val).
```

example run

```
?- tiles(M,C).
[b,b,b,e,w,w,w]-9
[b,b,b,w,e,w,w]-9
[b,b,e,w,b,w,w]-8
[b,b,w,w,b,e,w]-7
[b,b,w,w,b,w,e]-7
[b,b,w,w,e,w,b]-6
[b,e,w,w,b,w,b]-4
[b,w,e,w,b,w,b]-4
[e,w,b,w,b,w,b]-3
[w,w,b,e,b,w,b]-2
[w,w,b,w,b,e,b]-1
M = [[b,b,b,e,w,w,w],[b,b,b,w,e,w,w],
     [b,b,e,w,b,w,w],[b,b,w,w,b,e,w],
     [b,b,w,w,b,w,e],[b,b,w,w,e,w,b],
     [b,e,w,w,b,w,b],[b,w,e,w,b,w,b],
     [e,w,b,w,b,w,b],[w,w,b,e,b,w,b],
     [w,w,b,w,b,e,b],[w,w,e,w,b,b,b]]
C = 15
```

optimal best-first search

- Best-first search can be made complete by using

$$f(n) = g(n) + h(n)$$

  where $g(n)$ is actual cost so far and $h(n)$ is estimate on further cost to reach goal. Such an algorithm is called an A-algorithm.
- $g(n)$ will prevent getting lost in an infinite subgraph: adds a breadth-first flavor.
- If $h(n)$ is optimistic, i.e. it underestimates the cost, then the algorithm always finds an optimal path. Such an algorithm is called an A^*-algorithm.
- In an extreme case, if $h(n) = 0$, the algorithm degenerates to breadth-first (the heuristic in the previous example is optimistic).

Outline

language processing

- Syntax: definite clause grammars
- Semantics: terms instead of nonterminals
- Language generation
- An example interpreter

definite clause grammars (dcg)

- Context-free grammars in prolog.
- A sentence is a list of *terminals*:

```
[socrates, is, human]
```

- *Non-terminals* are defined by rules

```
sentence --> noun_phrase, verb_phrase
verb_phrase --> [is], property
noun_phrase -->proper_noun
proper_noun --> [socrates]
property --> [mortal]
property --> [human]
```

rules and prolog

- A rule

```
sentence --> noun_phrase, verb_phrase
```

can be read as:

```
sentence(S) :-
    noun_phrase(NP),
    verb_phrase(VP),
    append(NP,VP,S).
```

- A rule

```
property --> [mortal]
```

can be read as:

```
property([mortal]).
```

- thus: sentence([socrates, is , mortal]) parses the sentence

without append/3

- A rule

```
sentence --> noun_phrase, verb_phrase
```

corresponds to

```
sentence(L,L0) :-
   noun_phrase(L,L1),
   verb_phrase(L1,L0).
```

reading sentence(L,L0) as

" L consists of a sentence followed by L0 ".

- The conversion of rules to clauses is often built into prolog, as is the meta-predicate phrase/2 where

$$phrase(sentence,L) \equiv sentence(L,[])$$

DCGs vs. context-free grammars

- non-terminals can have arguments
- goals can be put into the rules
- no need for deterministic (LL(k), LR(k)) grammars!
- a single formalism for specifying syntax, semantics

example: adding plurality constraints

```
sentence --> noun_phrase(N), verb_phrase(N)
noun_phrase(N) --> article(N), noun(N)
verb_phrase(N) --> intransitive_verb(N)
article(singular) --> [a]
article(singular) --> [the]
article(plural) --> [the]
noun(singular) --> [student]
noun(plural) --> [students]
intransitive_verb(singular) --> [sleeps]
intransitive_verb(plural) --> [sleep]
```

```
phrase(sentence,[a,student,sleeps]). % yes
phrase(sentence,[the,students,sleep]). % yes
phrase(sentence,[the,students,sleeps]). % no
```

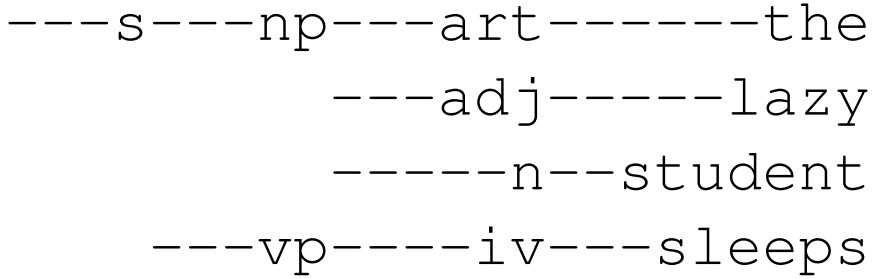
example: explicit parse trees

A parse tree is represented by a term.

```
sentence(s(NP,VP)) -->
     noun_phrase(NP), verb_phrase(VP)
noun_phrase(np(Art,Adj,N)) -->
     article(Art), adjective(Adj), noun(N)
noun_phrase(np(Art,N)) -->
     article(Art), noun(N)
verb_phrase(vp(IV)) -->
     intransitive_verb(IV)
article(art(the)) --> [the]
adjective(adj(lazy)) --> [lazy]
noun(n(student)) --> [student]
intransitive_verb(iv(sleeps)) --> [sleeps]
```

example: explicit parse trees

```
?- phrase(sentence(T),
          [the,lazy,student,sleeps])
T = s(np(art(the),
         adj(lazy),
         n(student)),
      vp(iv(sleeps)))
?- phrase(sentence(T),
      [the,lazy,student,sleeps]),
      term_write(T)
```



example: number parsing

- nX_Y(N) if N is a number in [X..Y].
- The grammar:

```
num(N) --> n1_999(N).
num(N) --> n1_9(N1),[thousand],n1_999(N2),
                {N is N1*1000+N2}.
n1_999(N) --> n1_99(N).
n1_999(N) --> n1_9(N1),[hundred],n1_99(N2),
                {N is N1*100+N2}.
n1_99(N) --> n0_9(N).
n1_99(N) --> n10_19(N).
n1_99(N) --> tens(N).
n1_99(N) --> tens(N1),n1_9(N2),{N is N1+N2}.
n0_9(0) --> [].
n0_9(N) --> n1_9(N).
n1_9(1) --> [one]. % two, .. , nine
n10_19(10) --> [ten]. % eleven,.., nineteen
tens(20) --> [twenty]. % thirty,.., ninety
```

The rule

```
n1_99(N) --> tens(N1), n1_9(N2), {N is N1+N2}.
```

corresponds to the clause

```
n1_99(N,L,L0) :-
  tens(N1,L,L1),
  n1_9(N2,L1,L0),
  N is N1 + N2.
```

number parsing example

```
?- phrase(num(N),
          [two,thousand,two,hunderd,eleven])
N = 2211
```

interpretation of natural language

- Syntax:

```
sentence --> determiner, noun, verb_phrase
sentence --> proper_noun, verb_phrase
verb_phrase --> [is], property
property --> [a], noun
property --> [mortal]
determiner --> every
proper_noun --> [socrates]
noun --> [human]
```

- Semantics: convert sentences to clauses, e.g.

  “every human is mortal”

  becomes

```
mortal(X):-  human(X)
```

- A proper noun is interpreted as a constant.

```
proper_noun(socrates) --> [socrates]
```

- A verb phrase is interpreted as a mapping from terms to literals X=>L:

```
verb_phrase(M) --> [is], property(M).
property(X=>mortal(X)) --> [mortal].
sentence((L:- true)) --> proper_noun(X),
                         verb_phrase(X=>L).

?-phrase(sentence(C),[socrates,is,mortal]).
  C = (mortal(socrates):- true)

sentence(C) --> determiner(M1,M2,C), noun(M1),
                verb_phrase(M2).
determiner(X=>B, X=>H, (H:- B)) --> [every].
noun(X=>human(X)) --> [human].

?-phrase(sentence(C), [every human is mortal])
  C = (mortal(X):- human(X))
```

grammar

```
:- op(600,xfy,'=>').
sentence(C) --> determiner(N,M1,M2,C), noun(N,M1),
                verb_phrase(N,M2).
sentence([(L:- true)]) --> proper_noun(N,X),
                           verb_phrase(N,X=>L).
verb_phrase(s,M) --> [is], property(s,M).
verb_phrase(p,M) --> [are], property(p,M).
property(N,X=>mortal(X)) --> [mortal].
property(s,M) --> noun(s,M).
property(p,M) --> noun(p,M).
determiner(s, X=>B , X=>H, [(H:- B)]) --> [every].
determiner(p, sk=>H1, sk=>H2,
  [(H1 :- true),(H2 :- true)]) -->[some].
proper_noun(s,socrates) --> [socrates].
noun(s,X=>human(X)) --> [human].
noun(p,X=>human(X)) --> [humans].
noun(s,X=>living_being(X)) --> [living],[being].
noun(p,X=>living_being(X)) --> [living],[beings].
```

questions

```
question(Q) -->
  [who], [is], property(s,X=>Q)
question(Q) -->
  [is], proper_noun(N,X), property(N,X=>Q)
question((Q1,Q2)) -->
  [are], [some], noun(p,sk=>Q1), property(p,sk=>Q2)
```

the interpreter: handle_input/2

```
% RB = rule base
nl_shell(RB) :-
  get_input(Input), handle_input(Input,RB).
handle_input(stop,RB) :- !.
handle_input(show,RB) :- !,
  show_rules(RB), nl_shell(RB).
handle_input(Sentence,RB) :-
  phrase(sentence(Rule),Sentence),
  nl_shell([Rule|RB]).
handle_input(Question,RB) :-
  phrase(question(Query),Question),
  prove(Query,RB),
  transform(Query,Clauses),
  phrase(sentence(Clauses),Answer),
  show_answer(Answer),
  nl_shell(RB).
handle_input(Error,RB) :-
  show_answer('no'), nl_shell(RB).
```

auxiliary clauses

```
show_rules([]).
show_rules([R|Rs]) :-
  phrase(sentence(R),Sentence),
  show_answer(Sentence),
  show_rules(Rs).

get_input(Input) :-
  write('? '),read(Input).
show_answer(Answer) :-
  write('! '),write(Answer), nl.
```

answering questions

```
prove(true,RB) :- !.
prove((A,B),RB) :- !,
  prove(A,RB),prove(B,RB).
prove(A,RB) :-
  find_clause((A:- B),RB), prove(B,RB).
%
find_clause(C,[R|Rs]) :-
  % don't instantiate rule
  copy_element(C,R).
find_clause(C,[R|Rs]) :-
  find_clause(C,Rs).
copy_element(X,Ys) :-
  element(X1,Ys),
  % copy with fresh variables
  copy_term(X1,X).

transform((A,B),[(A:- true)|Rest]) :-
  transform(B,Rest).
transform(A,(A:- true)).
```

example session

```
? [every,human,is,mortal]
? [socrates,is,a,human]
? [who,is,mortal]
! [socrates,is,mortal]
? [some,living,beings,are,humans]
? [are,some,living,beings,mortal]
! [some,living,beings,are,mortal]
```

Outline

reasoning with incomplete information

Forms of reasoning where conclusions are plausible but not guaranteed to be true:

- default reasoning: when a “normal” state of affairs is assumed (“birds fly”).

reasoning with incomplete information

Forms of reasoning where conclusions are plausible but not guaranteed to be true:

- default reasoning: when a “normal” state of affairs is assumed (“birds fly”).
- abductive resoning when there is a choice between several explanations that explain observations, e.g. in a diagnosis

reasoning with incomplete information

Forms of reasoning where conclusions are plausible but not guaranteed to be true:

- default reasoning: when a “normal” state of affairs is assumed (“birds fly”).
- abductive resoning when there is a choice between several explanations that explain observations, e.g. in a diagnosis
- inductive reasoning when a general rule is learned from examples.

reasoning with incomplete information

Forms of reasoning where conclusions are plausible but not guaranteed to be true:

- default reasoning: when a “normal” state of affairs is assumed (“birds fly”).
- abductive resoning when there is a choice between several explanations that explain observations, e.g. in a diagnosis
- inductive reasoning when a general rule is learned from examples.

Such reasoning is **unsound**. Sound reasoning is called **deduction**. Deduction only makes implicit information explicit.

Outline

default reasoning

Tweety is a bird. Normally, birds fly. Therefore, Tweety flies.

```
bird(tweety).
flies(X) :- bird(X), normal(X).
```

has 3 models:

```
{bird(tweety)}
{bird(tweety), flies(tweety)}
{bird(tweety), flies(tweety), normal(tweety)}
```

In default reasoning, it is more natural to use `abnormal/1` instead of `normal/1`:

```
flies(X) ; abnormal(X) :- bird(X).
```

can be transformed to a general definite clause:

```
flies(X) :- bird(X) , not(abnormal(X)).
```

Using negation as failure, we can now prove that Tweety flies.

```
bird(X) :- ostrich(X).
ostrich(tweety).
abnormal(X) :- ostrich(X).
```

Here the default rule

```
flies(X) :- bird(X), not(abnormal(X)).
```

is cancelled by the more specific rule about ostriches.

non-monotonic reasoning

In the example, new information invalidates previous conclusions. This is not the case for deductive reasoning where

$$Th \vdash p \quad \Rightarrow \quad Th \cup \{q\} \vdash p$$

for any q or, defining $Closure(Th) = \{p \mid Th \vdash p\}$ we get that

deduction is monotonic

$$Th_1 \subseteq Th_2 \Rightarrow Closure(Th_1) \subseteq Closure(Th_2)$$

non-monotonic reasoning

In the example, new information invalidates previous conclusions. This is not the case for deductive reasoning where

$$Th \vdash p \quad \Rightarrow \quad Th \cup \{q\} \vdash p$$

for any q or, defining $Closure(Th) = \{p \mid Th \vdash p\}$ we get that

deduction is monotonic

$$Th_1 \subseteq Th_2 \Rightarrow Closure(Th_1) \subseteq Closure(Th_2)$$

Default reasoning using *not/1* is problematic because *not/1* has no declarative semantics (but see later).

non-monotonic reasoning

In the example, new information invalidates previous conclusions. This is not the case for deductive reasoning where

$$Th \vdash p \quad \Rightarrow \quad Th \cup \{q\} \vdash p$$

for any q or, defining $Closure(Th) = \{p \mid Th \vdash p\}$ we get that

deduction is monotonic

$$Th_1 \subseteq Th_2 \Rightarrow Closure(Th_1) \subseteq Closure(Th_2)$$

Default reasoning using *not/1* is problematic because *not/1* has no declarative semantics (but see later).
Alternatively we can distinguish between rules with exceptions (default rules) and rules without exceptions. Rules are applied whenever possible, but default rules are only applied when they do not lead to an inconsistency.

an interpreter for default reasoning

Example

```
default((flies(X) :- bird(X))).
rule((not(flies(X)) :- penguin(X))).
rule((bird(X) :- penguin(X))).
rule((penguin(tweety) :- true)).
rule((bird(opus) :- true)).
```

the interpreter 1/2

```
% E explains F from rules, defaults
explain(F,E):-
  explain(F,[],E).
explain(true,E,E) :- !.
explain((A,B),E0,E) :- !,
  explain(A,E0,E1), explain(B,E1,E).
explain(A,E0,E):-
  prove(A,E0,E).
explain(A,E0,[default((A:-B))|E]):-
  default((A:-B)),
  explain(B,E0,E),
  not(contradiction(A,E)).
```

the interpreter 2/2

```
% prove using non-defaults
prove(true,E,E) :- !.
prove((A,B),E0,E) :- !,
  prove(A,E0,E1), prove(B,E1,E).
prove(A,E0,[rule((A:-B))|E]):-
  rule((A:-B)), prove(B,E0,E).

contradiction(not(A),E) :- !,
  prove(A,E,E1).
contradiction(A,E):-
  prove(not(A),E,E1).
```

Example

```
?- explain(flies(X),E)
X=opus
E=[default((flies(opus) :- bird(opus))),
   rule((bird(opus) :- true))]

?- explain(not(flies(X)),E)
X=tweety
E=[rule((not(flies(tweety)) :- penguin(tweety))),
   rule((penguin(tweety) :- true))]
```

Example

```
default((not(flies(X)) :- mammal(X))).
default((flies(X) :- bat(X))).
default((not(flies(X)) :- dead(X))).
rule((mammal(X) :- bat(X))).
rule((bat(a) :- true)).
rule((dead(a) :- true)).

?-explain(flies(a),E)
E=[default((flies(a) :- bat(a))),
   rule((bat(a) :- true))]

?-explain(not(flies(a)),E)
E=[default((not(flies(a)) :- mammal(a)))
   rule((mammal(a) :- bat(a))),
   rule((bat(a) :- true))]
E=[default((not(flies(a)) :- dead(a)))
   rule((dead(a) :- true))]
```

Only the third explanation seems acceptable.

We can refine by *naming* defaults and allow rules to cancel a specific default by name.

Example

```
default(mammals_dont_fly(X), (not(flies(X)):- mammal(X))).
default(bats_fly(X),(flies(X):- bat(X))).
default(dead_things_dont_fly(X), (not(flies(X)):- dead(X))).
rule((mammal(X):- bat(X))).
rule((bat(a):- true)).
rule((dead(a):- true)).
rule((not(mammals_dont_fly(X)):- bat(X))).
  % cancels mammals_dont_fly
rule((not(bats_fly(X)):- dead(X))). % cancels bats_fly
```

Change the interpreter:

```
explain(A,E0,[default(Name)|E]):-
  default(Name,(A:- B)), explain(B,E0,E),
  % default not cancelled
  not(contradiction(Name,E)),
  not(contradiction(A,E)).
```

Example

```
?-explain(flies(a),E)
no

?-explain(not(flies(a)),E)
E=[default(dead_things_dont_fly(a)),
   rule((dead(a):- true))]
```

Outline

semantics of negation

A program P is "complete" if for every (ground) fact f, we have that $P \models f$ or $P \models \neg f$

We consider two methods to "complete" programs:

- The closed world assumption works for definite clauses.

semantics of negation

A program P is "complete" if for every (ground) fact f, we have that $P \models f$ or $P \models \neg f$

We consider two methods to "complete" programs:

- The closed world assumption works for definite clauses.
- Predicate completion works for general clauses (with negation in the body) but leads to inconsistencies for some programs.

semantics of negation

A program P is "complete" if for every (ground) fact f, we have that $P \models f$ or $P \models \neg f$

We consider two methods to "complete" programs:

- The closed world assumption works for definite clauses.
- Predicate completion works for general clauses (with negation in the body) but leads to inconsistencies for some programs.

Alternatively, the stable model semantics introduces nondeterminism.

the closed world assumption

CWA: "*everything that is not known to be true must be false*" (e.g. databases).

$$CWA(P) = P \cup \{:-A \mid A \in \mathcal{B}_P \wedge P \not\models A\}$$

$CWA(P)$ is the intended program of P, according to the CWA.

Example

```
likes(peter,S) :- student_of(S,peter).
student_of(paul,peter).
```

CWA(P):

```
likes(peter,S) :- student_of(S,peter).
student_of(paul,peter).
:- student(paul,paul).          :- student(peter,paul).
:- student(peter,peter).
:- likes(paul,paul).            :- likes(paul,peter).
:- likes(peter,peter).
```

CWA(P) has only one model:

```
{student_of(paul,peter), likes(peter,paul)}
```

This intended model is the intersection of all (Herbrand) models.

Example

```
bird(tweety).
flies(X);abnormal(X) :- bird(X).
```

CWA(P):

```
bird(tweety).
flies(X);abnormal(X) :- bird(X).
:- flies(tweety).
:- abnormal(tweety).
```

which is inconsistent: CWA is unable to handle indefinite (or general, pseudo-definite) clauses.

predicate completion

Regard a clause as part of the *definition* of a predicate. E.g. if

```
likes(peter,S) :- student(S,peter).
```

is the only clause with head `likes/2`, its completion is

$$\forall X \cdot \forall S \cdot likes(X, S) \leftrightarrow X = peter \wedge student(S, peter)$$

which can be translated back to clausal form:

```
likes(peter,S) :- student(S,peter).
X=peter :- likes(X,S).
student(S,peter) :- likes(X,S)
```

the completion algorithm 1/3

```
likes(peter,S) :-  student(S,peter).
likes(X,Y) :-  friend(X,Y).
```

1. Ensure that each argument of the head of each clause is a distinct variable by adding literals of the form $Var = Term$ to the body.

```
likes(X,S) :-  X=peter, student(S,peter).
likes(X,Y) :-  friend(X,Y).
```

the completion algorithm 2/3

```
likes(X,S) :-  X=peter, student(S,peter).
likes(X,Y) :-  friend(X,Y).
```

2. If there are several clauses for the same predicate (in the head), combine them into a single formula with a disjunctive "body".

$$\forall X \cdot \forall Y \cdot likes(X, Y) \leftarrow \\ (X = peter \wedge student(Y, peter)) \\ \vee \\ friend(X, Y)$$

the completion algorithm 3/3

3. Replace the implication by an equivalence.

$$\forall X \cdot \forall Y \cdot likes(X, Y) \leftrightarrow \\ (X = peter \wedge student(Y, peter)) \\ \vee friend(X, Y)$$

Note: predicates that have no clauses, e.g. *p/1* becomes $\forall X \cdot \neg p(X)$

Clarke completion semantics

The intended model of P is the classical model of the predicate completion of $Comp(P)$.

Be careful with variables that do not occur in the head:

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

is equivalent to

$$\forall X \cdot \forall Y \cdot \forall Z \cdot ancestor(X, Y) \leftarrow parent(X, Z) \wedge ancestor(Z, Y)$$

but also with

$$\forall X \cdot \forall Y \cdot ancestor(X, Y) \leftarrow (\exists Z \cdot parent(X, Z) \wedge ancestor(Z, Y))$$

Example

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

becomes

$$\forall X \cdot \forall Y \cdot ancestor(X, Y) \leftarrow parent(X, Y) \vee (\exists Z \cdot parent(X, Z) \wedge ancestor(Z, Y))$$

in step 2, and

$$\forall X \cdot \forall Y \cdot ancestor(X, Y) \leftrightarrow parent(X, Y) \vee (\exists Z \cdot parent(X, Z) \wedge ancestor(Z, Y))$$

in step 3.

Example

```
bird(tweety).
flies(X) :- bird(X), not(abnormal(X)).
```

Comp(P) becomes:

$$\forall X \cdot bird(X) \leftrightarrow X = tweety$$
$$\forall X \cdot flies(X) \leftrightarrow (bird(X) \wedge \neg abnormal(X))$$
$$\forall X \cdot \neg abnormal(X)$$

which has a single model

```
{ bird(tweety), flies(tweety) }
```

Predicate completion gives inconsistent results in some cases:

Example

```
wise(X) :- not(teacher(X)).
teacher(peter) :- wise(peter).
```

becomes

$$\forall X \cdot wise(X) \leftrightarrow \neg teacher(X)$$
$$\forall X \cdot teacher(X) \leftrightarrow X = peter \wedge wise(peter)$$

which is inconsistent.

stratified programs

Definition

A program P is stratified if its predicate symbols can be partitioned into disjoint sets $S_0, \ldots, S_n$ such that for each clause

$$p(\ldots) \leftarrow L_1, \ldots, L_j$$

where $p \in S_k$, any literal L_j is such that

- if $L_j = q(\ldots)$ then $q \in S_0 \cup \ldots \cup S_k$

stratified programs

Definition

A program P is stratified if its predicate symbols can be partitioned into disjoint sets $S_0, \ldots, S_n$ such that for each clause

$$p(\ldots) \leftarrow L_1, \ldots, L_j$$

where $p \in S_k$, any literal L_j is such that

- if $L_j = q(\ldots)$ then $q \in S_0 \cup \ldots \cup S_k$
- if $L_j = \neg q(\ldots)$ then $q \in S_0 \cup \ldots \cup S_{k-1}$

stratified programs

Definition

A program P is stratified if its predicate symbols can be partitioned into disjoint sets $S_0, \ldots, S_n$ such that for each clause

$$p(\ldots) \leftarrow L_1, \ldots, L_j$$

where $p \in S_k$, any literal L_j is such that

- if $L_j = q(\ldots)$ then $q \in S_0 \cup \ldots \cup S_k$
- if $L_j = \neg q(\ldots)$ then $q \in S_0 \cup \ldots \cup S_{k-1}$

Theorem

If P is stratified then $Comp(P)$ is consistent.

Theorem

If P is stratified then $Comp(P)$ is consistent.

The condition is sufficient but not necessary:

Example

$$win(X) \leftarrow \neg loose(X).$$
$$loose(X) \leftarrow \neg win(X).$$

is not stratified but its completion is consistent.

the stable model semantics

Intuition: Guess a model and verify that it can be reconstructed from the program ("stability").

Example

```
win:- not(loose).
loose :- not(win).
```

Guess $M = \{win\}$: first rule becomes

```
win.
```

while the second rule is not applicable (since its body is false).

the Gelfond-Lifschitz transformation

For a program P and an interpretation I, the GL transform P_I is defined by

1. Removing all (true) negations $not(b)$, where $b \notin I$ from the bodies of the rules.
2. Remove all ("blocked") rules that still contain negations after the previous step.

The result is a positive program.

Example

```
win:- not(loose).
loose :- not(win).
```

$P_{\{win\}}$ contains just the clause

```
win.
```

stable model definition

Definition

M is a stable model of *P* iff *M* is the (unique) minimal model of P_M.

Example

```
win:- not(loose).
loose :- not(win).
```

Has two stable models: $\{win\}$ and $\{loose\}$.

graph colorability with stable models

```
% graph defined by node/1, arc/2
%
% a node must have a color
color(N,red) :-  node(N),
  not(color(N,green)), not(color(N,blue)).
color(N,green) :-  node(N),
  not(color(N,blue)), not(color(N,red)).
color(N,blue) :-  node(N),
  not(color(N,red)), not(color(N,green)).
%
% no two adjacent nodes have the same color
:-  arc(X,Y), color(X,C), color(Y,C).
```

Any stable model of the above program represents a solution to this NP-complete problem.

answer set programming

- Extension of logic programming based on the stable model semantics for datalog programs (finite universe).

answer set programming

- Extension of logic programming based on the stable model semantics for datalog programs (finite universe).
- Without disjunction in the head, NP problems (e.g. satisfiability of a propositional formula) can be represented.

answer set programming

- Extension of logic programming based on the stable model semantics for datalog programs (finite universe).
- Without disjunction in the head, NP problems (e.g. satisfiability of a propositional formula) can be represented.
- The stable model semantics can be extended to disjunctive (datalog) programs, which increases the expressiveness to $\Sigma_2 P$ (NP using an NP ($\Sigma_1 P$) oracle, e.g. deciding whether $\exists x \cdot \forall y \cdot \phi(x, y)$ is valid).

answer set programming

- Extension of logic programming based on the stable model semantics for datalog programs (finite universe).
- Without disjunction in the head, NP problems (e.g. satisfiability of a propositional formula) can be represented.
- The stable model semantics can be extended to disjunctive (datalog) programs, which increases the expressiveness to $\Sigma_2 P$ (NP using an NP ($\Sigma_1 P$) oracle, e.g. deciding whether $\exists x \cdot \forall y \cdot \phi(x, y)$ is valid).
- Efficient (sic) implementations are available: e.g. the **smodels** or the **dlv** systems.

answer set programming

- Extension of logic programming based on the stable model semantics for datalog programs (finite universe).
- Without disjunction in the head, NP problems (e.g. satisfiability of a propositional formula) can be represented.
- The stable model semantics can be extended to disjunctive (datalog) programs, which increases the expressiveness to $\Sigma_2 P$ (NP using an NP ($\Sigma_1 P$) oracle, e.g. deciding whether $\exists x \cdot \forall y \cdot \phi(x, y)$ is valid).
- Efficient (sic) implementations are available: e.g. the **smodels** or the **dlv** systems.
- Applications in configuration (space shuttle), planning, diagnostics, ...

sudoku using answer set programming

(author: Kim Bauters)

```
size(0..8). % like type declaration
1{p(X, Y, Value) : size(Value)}1 :- size(X), size(Y).
% A value may not appear more than once in any row.
:- p(X, Y1, Value), p(X, Y2, Value), size(X;Y1;Y2;Value), Y1!=Y2.
% A value may not appear more than once in any column.
:- p(X1, Y, Value), p(X2, Y, Value), size(X1;X2;Y;Value), X1!=X2.
% A value may not appear more than once in any subgrid.
:- p(X1, Y1, Value), p(X2, Y2, Value), size(X1;X2;Y1;Y2;Value),
    (X1 != X2 | Y1 != Y2), X1 / 3 == X2 / 3, Y1 / 3 == Y2 / 3.
hide size(_).
```

Note: smodels extension (syntax sugar):

- `2{p,q,r}3` is true in M iff it contains between 2 and 3 elements of $\{p, q, r\}$.
- `{p(a,X) : q(X)}` is shorthand for the set $\{p(a, X) \mid q(X)\}$.

Outline

abduction

Given a theory T and an observation O, find an explanation E such that

$$T \cup E \vdash O$$

E.g. given the theory

```
likes(peter,S) :-  student_of(S,peter).
likes(X,Y) :-  friend(X,Y).
```

and the observation `likes(peter,paul)`, possible explanations are `{student_of(paul,peter)}` or `{friend(peter,paul)}`
Another possible explanation is
`{(likes(X,Y) :- friendly(Y)), friendly(paul)}` but abductive explanations are usually restricted to ground literals with predicates that are undefined in the theory (abducibles).

an abduction algorithm

Try to prove observation from theory; when an abducible literal is encountered that cannot be resolved, add it to the explanation.

```
abduce(O,E):-   % P+E |- O
  abduce(O,[],E).

abduce(true,E,E) :- !.
abduce((A,B),E0,E) :- !,
  abduce(A,E0,E1),
  abduce(B,E1,E).
abduce(A,E0,E):-
  clause(A,B),
  abduce(B,E0,E).
abduce(A,E,E):-  element(A,E).
abduce(A,E,[A|E]):-  not(element(A,E)), abducible(A).
abducible(A):-  not(clause(A,B)).
  % clauses are assumed to be definitions
```

Example

```
likes(peter,S) :-  student_of(S,peter).
likes(X,Y) :-  friend(X,Y).

?-abduce(likes(peter,paul),E)
E = [student_of(paul,peter)];
E = [friend(paul,peter)]
```

Problems with general clauses:

Example

```
flies(X) :-  bird(X), not(abnormal(X)).
abnormal(X) :-  penguin(X).
bird(X) :-  penguin(X).
bird(X) :-  sparrow(X).

?-abduce(flies(tweety),E)
E = [not(abnormal(tweety)),penguin(tweety)];
E = [not(abnormal(tweety)),sparrow(tweety)];
```

adding negation as failure

```
% E explains not(A) if E does not explain A
abduce(not(A),E,E):-
  not(abduce(A,E,E)).
..
abducible(A):-
  A \= not(X), not(clause(A,B)).

?-abduce(flies(tweety),E)
E = [sparrow(tweety)]
```

Still problems because `abduce(not(A),E,E)` assumes `E` is "complete". E.g.

```
abduce(not(abnormal(X)),[],[])
```

succeeds and thus, if `flies(X):- not(abnormal(X)),bird(X)` any explanation of `bird(X)` will explain `flies(X)`.
Thus we need a special `abduce_not/3` that provides evidence for accepting `not(..)`.

new interpreter 1/2

```
abduce(true,E,E) :- !.
abduce((A,B),E0,E) :- !,
  abduce(A,E0,E1),
  abduce(B,E1,E).
abduce(A,E0,E):-
  clause(A,B),
  abduce(B,E0,E).
abduce(A,E,E):-
  element(A,E).
abduce(A,E,[A|E]):-
  not(element(A,E)),
  abducible(A),
  not(abduce_not(A,E,E)).
   % only if E does not explain not(A)
abduce(not(A),E0,E):-
  not(element(A,E0)),
  abduce_not(A,E0,E).
abducible(A):-
  A \= not(X), not(clause(A,B)).
```

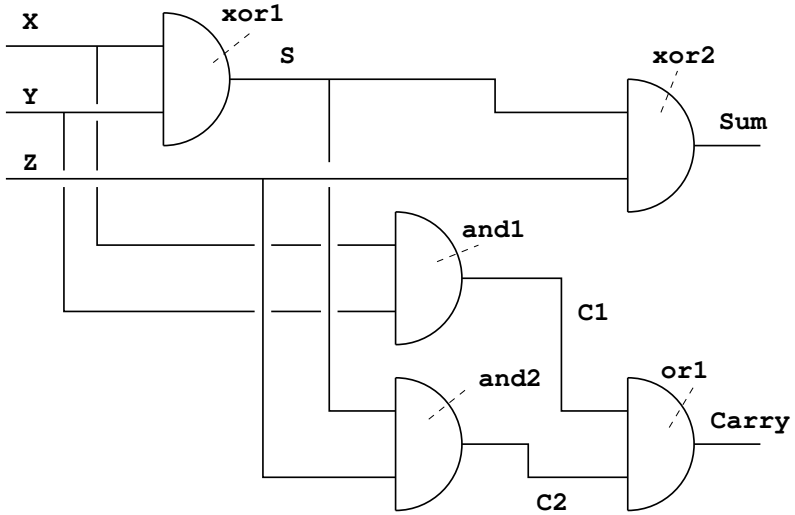
new interpreter 2/2

```
abduce_not((A,B),E0,E):-  % disjunction!
  abduce_not(A,E0,E) ; abduce_not(B,E0,E).
abduce_not(A,E0,E):-
  setof(B,clause(A,B),L), % abduce_not(B) for each body B
  abduce_not_list(L,E0,E).
abduce_not(A,E,E):-
  element(not(A),E). % not(A) already assumed
abduce_not(A,E,[not(A)|E]):-  % assume not(A) if
  not(element(not(A),E)), % not already there, and
  abducible(A), % it is abducible, and
  not(abduce(A,E,E)). % E does not explain A
abduce_not(not(A),E0,E):-
  not(element(not(A),E0)),
  abduce(A,E0,E).

abduce_not_list([],E,E).
abduce_not_list([B|Bs],E0,E):-
  abduce_not(B,E0,E1), % body cannot be used
  abduce_not_list(Bs,E1,E).
```

Example

```
flies(X) :- bird(X),not(abnormal(X)).
flies1(X) :- not(abnormal(X)),bird(X).
abnormal(X) :- penguin(X).
abnormal(X) :- dead(X).
bird(X) :- penguin(X).
bird(X) :- sparrow(X).

?- abduce(flies(tweety),E).
E = [not(penguin(tweety)),not(dead(tweety)),
     sparrow(tweety)]
?- abduce(flies1(tweety),E).
E = [sparrow(tweety),
     not(penguin(tweety)),not(dead(tweety))]
```

diagnosis using abduction




```
adder(X,Y,Z,Sum,Carry) :-
  xor(X,Y,S),xor(Z,S,Sum),
  and(X,Y,C1),and(Z,S,C2),
  or(C1,C2,Carry).

xor(0,0,0).      and(0,0,0).     or(0,0,0).
xor(0,1,1).      and(0,1,0).     or(0,1,1).
xor(1,0,1).      and(1,0,0).     or(1,0,1).
xor(1,1,0).      and(1,1,1).     or(1,1,1).
```

describes normal operation

The fault model of a system describes the behavior of each component when in a faulty state.
We distinguish 2 such states: `s0` (“stuck at 0”) and `s1` (“stuck at 1”). A faulty component is described by a literal `fault(NameComponent=State)`
Names of components can be nested as in

`nameSubSystem-nameComponent`

```
adder(N,X,Y,Z,Sum,Carry):-
  xorg(N-xor1,X,Y,S), xorg(N-xor2,Z,S,Sum),
  andg(N-and1,X,Y,C1), andg(N-and2,X,S,C2),
  org(N-or1,C1,C2,Carry).

xorg(N,X,Y,Z) :-  xor(X,Y,Z).
xorg(N,0,0,1) :-  fault(N=s1).
xorg(N,0,1,0) :-  fault(N=s0).
xorg(N,1,0,0) :-  fault(N=s0).
xorg(N,1,1,1) :-  fault(N=s1).
```

```
xandg(N,X,Y,Z):- and(X,Y,Z).
xandg(N,0,0,1):- fault(N=s1).   xandg(N,0,1,1) :- fault(N=s1).
xandg(N,1,0,1):- fault(N=s1).   xandg(N,1,1,0) :- fault(N=s0).

org(N,X,Y,Z):- or(X,Y,Z).
org(N,0,0,1):- fault(N=s1).   org(N,0,1,0) :- fault(N=s0).
org(N,1,0,0):- fault(N=s0).   org(N,1,1,0) :- fault(N=s0).

diagnosis(Observation,Diagnosis):-
  abduce(Observation,Diagnosis).

?-diagnosis(adder(a,0,0,1,0,1),D).
D = [fault(a-or1=s1),fault(a-xor2=s0)];
D = [fault(a-and2=s1),fault(a-xor2=s0)];
D = [fault(a-and1=s1),fault(a-xor2=s0)];
D = [fault(a-and2=s1),fault(a-and1=s1), fault(a-xor2=s0)];
D = [fault(a-xor1=s1)];
```

```
D = [fault(a-or1=s1),fault(a-and2=s0), fault(a-xor1=s1)];
D = [fault(a-and1=s1),fault(a-xor1=s1)];
D = [fault(a-and2=s0),fault(a-and1=s1), fault(a-xor1=s1)];
```

Minimal diagnoses are more plausible:

```
min_diagnosis(O,D) :-
  diagnosis(O,D),
  not(diagnosis(O,D1),proper_subset(D1,D)).

?-min_diagnosis(adder(a,0,0,1,0,1),D).
D = [fault(a-or1=s1),fault(a-xor2=s0)];
D = [fault(a-and2=s1),fault(a-xor2=s0)];
D = [fault(a-and1=s1),fault(a-xor2=s0)];
D = [fault(a-xor1=s1)];
```

Outline

Inductive logic programming (ILP)

Problem: Given

- B background knowledge (theory, i.e. LP)
- E^+ positive examples (set of facts),
- E^- negative examples (set of facts),

Find a theory H (*hypothesis*) such that

$$\forall p \in E^+ \cdot B \cup H \models p$$
$$\forall n \in E^- \cdot B \cup H \not\models n$$

Of course, we assume that $\forall e \in E^+ \cup E^- \cdot B \not\models e$
Difference with abduction: H is a *theory* instead of a set of facts.

Relationship with learning

- Concept learning tries to find a suitable concept in a description space where descriptions are related via generalization/specialization relationships. Examples are at the "bottom" of the generalization hierarchy.
- A concept is suitable if it covers (generalizes) all positive and none of the negative examples.
- Learning capabilities depend on the characteristics of the description space: too rough makes learning impossible, too fine leads to trivial concepts (e.g. when the description space supports disjunction).
- A well-known algorithm is Mitchell's candidate elimination algorithm where upper and lower bounds of possible solutions are updated according to input examples.

ILP as concept learning

- ILP as discussed here can be seen as concept learning where the description space consists of LP's. The generalization relationship may be based on subsumption between clauses.

Example: learning append/3

```
?- induce_rlgg([
              +append([1,2],[3,4],[1,2,3,4]),
              +append([a],[],[a]),
              +append([],[],[]),
              +append([],[1,2,3],[1,2,3]),
              +append([2],[3,4],[2,3,4]),
              +append([],[3,4],[3,4]),
              -append([a],[b],[b]),
              -append([c],[b],[c,a]),
              -append([1,2],[],[1,3])
             ], Clauses).
```

Example: learning append/3

```
RLGG of append([1,2],[3,4],[1,2,3,4]) and
        append([a],[],[a]) is
     append([X|Y],Z,[X|U]) :-  [append(Y,Z,U)]
Covered example: append([1,2],[3,4],[1,2,3,4])
Covered example: append([a],[],[a])
Covered example: append([2],[3,4],[2,3,4])
RLGG of append([],[],[]) and append([],[1,2,3],[1,2,3]) is
     append([],X,X) :-  []
Covered example: append([],[],[])
Covered example: append([],[1,2,3],[1,2,3])
Covered example: append([],[3,4],[3,4])

Clauses = [(append([],X,X) :-  []),
           (append([X|Y],Z,[X|U]) :-  [append(Y,Z,U)])]
```

generalizing clauses: θ-subsumption

Definition

A clause c_1 θ-subsumes a clause c_2 iff there exists a substitution θ such that $\theta c_1 \subseteq c_2$ (c_1 is "more general" or "more widely applicable" thant c_2).

Here clauses are seen as sets of (positive and negative) literals (disjunctions).

θ-subsumption examples

- The clause

```
element(X,V) :- element(X,Z)
```

  θ-subsumes, using $\theta = \{\mathrm{V} \rightarrow [\mathrm{Y}|\mathrm{Z}]\}$,

```
element(X,[Y|Z]) :- element(X,Z)
```

  (i.e. θ "specializes" `element(X,V) :- element(X,Z)`).

- The clause

```
a(X) :- b(X).
```

  θ-subsumes (with θ identity)

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

θ-subsumption implementation

```
% (H1:- B1) subsumes (H2 :- B2)
theta_subsumes((H1:- B1),(H2 :- B2)):-
  verify((grounded((H2:- B2)), H1=H2,subset(B1,B2))).
  % H1=H2 creates substitution, note that H2 has no vars
grounded(Term):-
  % instantiate vars in Term to terms of the form
  % '$VAR'(i) where i is different for each distinct
  % var, first i=0, last = N-1
  numbervars(Term,0,N).

verify(Goal) :-          % prove without binding
  not(not(call(Goal))).
```

θ-subsumption implementation

Example

```
?-  theta_subsumes( (element(X,V):- []),
        (element(X,V):- [element(X,Z)])).
yes.
?-  theta_subsumes((element(X,a):- []),
        (element(X,V):- [])).
no.
```

θ-subsumption vs. logical consequence

Theorem

If c_1 θ-subsumes c_2 then $c_1 \models c_2$

The reverse is not true:

```
a(X) :- b(X). % c1
p(X) :- p(X). % c2, tautology.
```

Here $c_1 \models c_2$ but there is no substitution θ such that $\theta c_1 \subseteq c_2$

Theorem

The set of (reduced) clauses form a lattice, i.e. a unique least general generalization $lgg(c_1, c_2)$ exists for any two clauses c_1 and c_2.

(a clause is reduced if it is minimal in the collection of equivalent clauses)

generalizing 2 terms

Consider the terms

```
element(1,[1]). %a1
element(z,[z,y,x]). %a2
element(X,[X|Y]). % a3
```

- a_3 subsumes a_1 using `{X/1, Y/[]}` and
- a_3 subsumes a_2 using `{X/z, Y/[y,x]}`
- Moreover, a_3 is the **least** generalization, i.e. every other term that θ-subsumes a_1 and a_2 also θ-subsumes a_3.

anti_unify

```
:- op(600,xfx,'<-'). % to record (inverse) substitutions

anti_unify(Term1,Term2,Term):-   % use accumulators S1, S2
  anti_unify(Term1,Term2,Term,[],S1,[],S2).

anti_unify(Term1,Term2,Term1,S1,S1,S2,S2):-
  Term1 == Term2,!.
anti_unify(Term1,Term2,V,S1,S1,S2,S2):-
  subs_lookup(S1,S2,Term1,Term2,V), !. % already substituted
anti_unify(Term1,Term2,Term,S10,S1,S20,S2):-
  nonvar(Term1), nonvar(Term2),
  functor(Term1,F,N), functor(Term2,F,N),!,
  functor(Term,F,N), % create F(X1,..,Xn)
  anti_unify_args(N,Term1,Term2,Term,S10,S1,S20,S2).
% Create new variable V and substitutions V->Term1, V->Term2
anti_unify(Term1,Term2,
  V, S10,[Term1<-V|S10], S20,[Term2<-V|S20]).
```

```
% anti_unify_args(N,T1,T2,T,..):
%   anti-unify first N arguments of T1, T2
anti_unify_args(0,Term1,Term2,Term,S1,S1,S2,S2).
anti_unify_args(N,Term1,Term2,Term,S10,S1,S20,S2):-
  N>0,N1 is N-1,
  arg(N,Term1,Arg1), arg(N,Term2,Arg2), arg(N,Term,ArgN),
  anti_unify(Arg1,Arg2,ArgN,S10,S11,S20,S21),
  anti_unify_args(N1,Term1,Term2,Term,S11,S1,S21,S2).

% subs_lookup(+subst1,+subst2,+term1,+term2,-var)
%       subst1(V) = term1, subst2(V) = term2
subs_lookup([T1<-V|Subs1],[T2<-V|Subs2],Term1,Term2,V):-
  T1 == Term1, T2 == Term2,!.
subs_lookup([S1|Subs1],[S2|Subs2],Term1,Term2,V):-
  subs_lookup(Subs1,Subs2,Term1,Term2,V).
```

Example

```
?-  anti_unify(2*2=2+2,2*3=3+3,T,[],S1,[],S2).
T = 2 * _G191 = _G191 + _G191
S1 = [2 <- _G191]
S2 = [3 <- _G191]
```

generalizing 2 clauses (1/2)

```
theta_lgg((H1:-B1),(H2:-B2),(H:-B)):-
  anti_unify(H1,H2,H,[],S10,[],S20),
  theta_lgg_bodies(B1,B2, [],B, S10,S1, S20,S2).
% theta_lgg_bodies considers each pair of literals
% from both bodies
theta_lgg_bodies([],B2,B,B,S1,S1,S2,S2).
theta_lgg_bodies([Lit|B1],B2, B0,B, S10,S1, S20,S2):-
  theta_lgg_literal(Lit,B2, B0,B00, S10,S11, S20,S21),
  theta_lgg_bodies(B1,B2, B00,B, S11,S1, S21,S2).
```

generalizing 2 clauses (2/2)

```
% theta_lgg_literal anti-unifies Lit1 with each
% literal in 2nd arg
theta_lgg_literal(Lit1,[], B,B, S1,S1, S2,S2).
theta_lgg_literal(Lit1,[Lit2|B2], B0,B, S10,S1, S20,S2):-
  same_predicate(Lit1,Lit2),
  anti_unify(Lit1,Lit2,Lit,S10,S11,S20,S21),
  theta_lgg_literal(Lit1,B2,[Lit|B0],B,S11,S1,S21,S2).
theta_lgg_literal(Lit1,[Lit2|B2],B0,B,S10,S1,S20,S2):-
  not(same_predicate(Lit1,Lit2)),
  theta_lgg_literal(Lit1,B2,B0,B,S10,S1,S20,S2).
same_predicate(Lit1,Lit2) :-
  functor(Lit1,P,N),
  functor(Lit2,P,N).
```

theta_lgg example

Example

```
?- theta_lgg(
     (element(c,[b,c]):- [
        element(c,[c])
        ]),
     (element(d,[b,c,d]):- [
       element(d,[c,d]),
       element(d,[d])
       ]),
     C).
C = element(X, [b, c|Y]):- [element(X,[X]), element(X,[c|Y])]
```

theta_lgg example

Example

```
?- theta_lgg(
     (reverse([2,1],[3],[1,2,3]):- [
        reverse([1],[2,3],[1,2,3])
        ]),
     (reverse([a],[],[a]) :-  [
        reverse([],[a],[a])
        ]),
     C).
C = reverse([X|Y], Z, [U|V]) :-  [reverse(Y, [X|Z], [U|V])]
```

a bottom-up induction algorithm

Definition

The relative least general generalization $rlgg(e_1, e_2, M)$ of two positive examples relative to a (partial) model M is defined by

$$rlgg(e_1, e_2, M) = lgg((e_1 :- M_\wedge), (e_2 :- M_\wedge))$$

a bottom-up induction algorithm: example

```
append([1,2],[3,4],[1,2,3,4]).   append([a],[],[a]).
append([],[],[]).                append([2],[3,4],[2,3,4]).
```

the *rlgg* on the first 2 examples is determined using

```
?- theta_lgg(
    (append([1,2],[3,4],[1,2,3,4]) :-  [
                          append([1,2],[3,4],[1,2,3,4]),
                          append([a],[],[a]),
                          append([],[],[]),
                          append([2],[3,4],[2,3,4])
                          ]),
    (append([a],[],[a]):-  [
                          append([1,2],[3,4],[1,2,3,4]),
                          append([a],[],[a]),
                          append([],[],[]),
                          append([2],[3,4],[2,3,4])
                          ]),
    C), write_ln(C).
```

example: result

```
append([X|Y], Z, [X|U]) :-  [
  append([2], [3, 4], [2, 3, 4]),
  append(Y, Z, U),
  append([V], Z, [V|Z]),
  append([K|L], [3, 4], [K, M, N|O]),
  append(L, P, Q),
  append([], [], []),
  append(R, [], R),
  append(S, P, T),
  append([A], P, [A|P]),
  append(B, [], B),
  append([a], [], [a]),
  append([C|L], P, [C|Q]),
  append([D|Y], [3, 4], [D, E, F|G]),
  append(H, Z, I),
  append([X|Y], Z, [X|U]),
  append([1, 2], [3, 4], [1, 2, 3, 4])
  ]
```

too complex!

constrained clauses

We remove:

- ground facts (examples) are redundant
- literals involving variables not occurring in the head: i.e. we restrict to constrained clauses.

The example result then becomes:

```
append([X|Y], Z, [X|U]) :-
  append(Y, Z, U), append([X|Y], Z, [X|U]).
```

The head is part of the body: it can also be removed if we restrict to strictly constrained clauses where the variables in the body are a strict subset of the variables in the head.

computing the rlgg

```
% rlgg(E1,E2,M,C): C is RLGG of E1 and E2 relative to M
rlgg(E1,E2,M,(H:- B)):-
  anti_unify(E1,E2,H,[],S10,[],S20),
  varsin(H,V), % determine variables in head of clause
  rlgg_bodies(M,M,[],B,S10,S1,S20,S2,V).
% rlgg_bodies(B0,B1,BR0,BR,S10,S1,S20,S2,V): rlgg all
% literals in B0 with all literals in B1, yielding BR
% containing only vars in V
rlgg_bodies([],B2,B,B,S1,S1,S2,S2,V).
rlgg_bodies([L|B1],B2,B0,B,S10,S1,S20,S2,V):-
  rlgg_literal(L,B2,B0,B00,S10,S11,S20,S21,V),
  rlgg_bodies(B1,B2,B00,B,S11,S1,S21,S2,V).
```

```
rlgg_literal(L1,[],B,B,S1,S1,S2,S2,V).
rlgg_literal(L1,[L2|B2],B0,B,S10,S1,S20,S2,V):-
  same_predicate(L1,L2),
  anti_unify(L1,L2,L,S10,S11,S20,S21),
  varsin(L,Vars),
  var_proper_subset(Vars,V), % no new variables in literal
  !,
  rlgg_literal(L1,B2,[L|B0],B,S11,S1,S21,S2,V).
rlgg_literal(L1,[L2|B2],B0,B,S10,S1,S20,S2,V):-
  rlgg_literal(L1,B2,B0,B,S10,S1,S20,S2,V).
```

varsin/2

```
% varsin(+term,-list) list is list of
%   variables occurring in term
varsin(Term,Vars):-
  varsin(Term,[],V), sort(V,Vars).
varsin(V,Vars,[V|Vars]):-
  var(V).
varsin(Term,V0,V):-
  functor(Term,F,N),
  varsin_args(N,Term,V0,V).

% varsin_args(N,T,V0,V) add vars in first
% N args of T to V0, yielding V
varsin_args(0,Term,Vars,Vars).
varsin_args(N,Term,V0,V):-
  N>0, N1 is N-1,
  arg(N,Term,ArgN),
  varsin(ArgN,V0,V1),
  varsin_args(N1,Term,V1,V).
```

var_remove_one/3, var_proper_subset/2

```
var_remove_one(X,[Y|Ys],Ys) :-
  X == Y.
var_remove_one(X,[Y|Ys],[Y|Zs) :-
  var_remove_one(X,Ys,Zs).

var_proper_subset([],Ys) :-
  Ys \= [].
var_proper_subset([X|Xs],Ys) :-
  var_remove_one(X,Ys,Zs),
  var_proper_subset(Xs,Zs).
```

rlgg

Example

```
?-  rlgg(
     append([1,2],[3,4],[1,2,3,4]),
     append([a],[],[a]),
     [
      append([1,2],[3,4],[1,2,3,4]),
      append([a],[],[a]),
      append([],[],[]),
      append([2],[3,4],[2,3,4])
     ],
     (H:- B)).
append([X|Y], Z, [X|U]) :-  [
   append([2], [3, 4], [2, 3, 4]),
   append(Y, Z, U),
   append([], [], []),
   append([a], [], [a]),
   append([1, 2], [3, 4], [1, 2, 3, 4])
   ]
```

main algorithm

- construct rlgg of two examples
- remove positive examples that are covered by the resulting clause
- remove further literals (generalizing the clause) as long as the clause does not cover any negative examples
- based on GOLEM system (Muggleton & Feng, 1990)

induce_rlgg implementation

```
induce_rlgg(Exs,Clauses):-
  pos_neg(Exs,Poss,Negs),
  bg_model(BG), append(Poss,BG,Model),
  induce_rlgg(Poss,Negs,Model,Clauses).

% induce_rlgg(+pos_exs,+neg_exs,+model,-clauses)
induce_rlgg(Poss,Negs,Model,Clauses):-
  covering(Poss,Negs,Model,[],Clauses).

% pos_neg(+exs,-poss,-negs) split
%  positive and negative examples
pos_neg([],[],[]).
pos_neg([+E|Exs],[E|Poss],Negs):-
  pos_neg(Exs,Poss,Negs).
pos_neg([-E|Exs],Poss,[E|Negs]):-
  pos_neg(Exs,Poss,Negs).
```

```
% covering(+pos_exs, +neg_exs, +model,+old_hypothesis,
%   -new_hypothesis): construct new_hypothesis
%  covering all of pos_exs and none of the neg_exs
covering(Poss,Negs,Model,Hyp0,NewHyp) :-
  construct_hypothesis(Poss,Negs,Model,Hyp), !,
  remove_pos(Poss,Model,Hyp,NewPoss),
  % cover remaining posexs
  covering(NewPoss,Negs,Model,[Hyp|Hyp0],NewHyp).
covering(P,N,M,H0,H) :-
  append(H0,P,H). % add uncovered exs to hypothesis

% remove_pos(+old_pos_exs,+model,+clause,-new_pos_ex)
%   remove posexs that are covered by clause + model,
%   yielding new_pos_ex
remove_pos([],M,H,[]).
remove_pos([P|Ps],Model,Hyp,NewP) :-
  covers_ex(Hyp,P,Model), !,
  write('Covered example: '), write_ln(P),
  remove_pos(Ps,Model,Hyp,NewP).
remove_pos([P|Ps],Model,Hyp,[P|NewP]):-
  remove_pos(Ps,Model,Hyp,NewP).
```

```
% covers_ex(+clause,+example,+model):
%  example is covered by clause
covers_ex((Head:- Body),Example,Model):-
  verify(
    (Head=Example, forall(element(L,Body),element(L,Model))
    )).
% construct_hypothesis(+pos_exs,+neg_exs,+model,-clause)
construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
  write('RLGG of '), write(E1),
    write(' and '), write(E2), write(' is'),
  rlgg(E1,E2,Model,Cl),
  reduce(Cl,Negs,Model,Clause), !,
  nl,tab(5), write_ln(Clause).
construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
  write_ln(' too general'),
  construct_hypothesis([E2|Es],Negs,Model,Clause).
```

```
% reduce(+old_clause,+neg_exs,+model,-new_clause)
%   remove redundant literals from body and ensure
%   that no negexs are covered
reduce((H:- B0),Negs,M,(H:-B)):-
  % remove literals of M from B0, giving B1
  setof0(L, (element(L,B0), not(var_element(L,M))), B1),
  % body B consists of literals from B1 that are necessary
  % not to cover negative examples
  reduce_negs(H,B1,[],B,Negs,M).
% covers_neg(+clause,+negs,+model,-n)
%   n negative example from negs covered by clause + model
covers_neg(Clause,Negs,Model,N):-  element(N,Negs),
  covers_ex(Clause,N,Model).
```

```
% reduce_negs(+H,+In,+B0,-B,+Negs,+Model)
%   B is B0 + subsequence of In such that (H:- B) + Model
%   does not cover elements of Negs
reduce_negs(H,[L|Rest],B0,B,Negs,Model):-
  % try removing L
  append(B0,Rest,Body),
  not(covers_neg((H:- Body),Negs,Model,N)), !,
  reduce_negs(H,Rest,B0,B,Negs,Model).
reduce_negs(H,[L|Rest],B0,B,Negs,Model):-
  % L cannot be removed
  reduce_negs(H,Rest,[L|B0],B,Negs,Model).
reduce_negs(H,[],Body,Body,Negs,Model):-
  not(covers_neg((H:- Body),Negs,Model,N)).

var_element(X,[Y|Ys]):-
  X == Y.          % syntactic identity
var_element(X,[Y|Ys]):-
  var_element(X,Ys).
```

further developments

- Top-down (specializing) induction: cfr. book, section 9.3
- Application examples:
  - scientific discovery: e.g. predicting 3-dimensional shape of proteins from their amino acid sequence
  - data mining; this may use a probabilistic semantics