Logic Programming

The descriptive power of Prolog
How Prolog works

Alípio Jorge
DCC-FCUP
vsc@dcc.fc.up.pt (room: 1.45)

These slides are largely based on Prof. Inês Dutra's and Prof. Alípio Jorge

17 de março de 2022

The West/East trains

How to recognize a train going East?

Trains going East

Trains going West

- We need a rule that is true for all Eastbound trains
- and false for the others.
- Let's define predicate **eastbound/1**.

The West/East trains

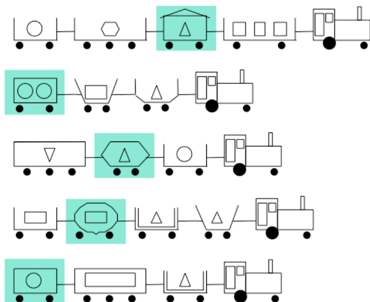
Let's describe the first train

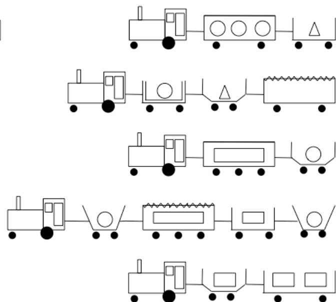
```
train(t1).
car(t1,c1).  car(t1,c2).  car(t1,c3).  car(t1,c4).
open(c1).   open(c3).   open(c4).
closed(c2).   small(c2).
load(c1,o1). load(c1,o2). load(c1,o3).
square(o1).   square(o2).   square(o3).
% to complete first train
```

How to complete?

The West/East trains

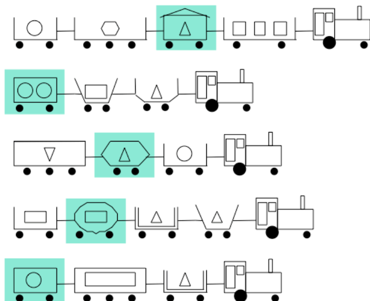
- How to describe some of the other trains?
- Are predictes missing?
- When to stop describing?

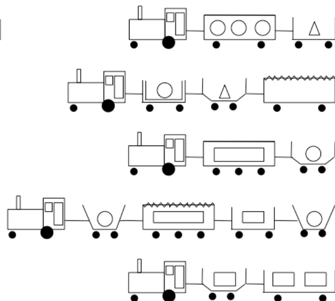




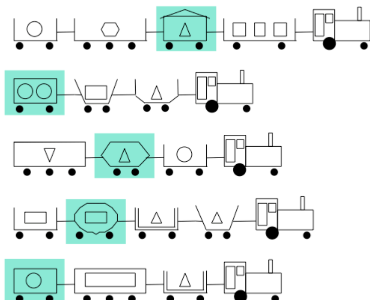
The West/East trains

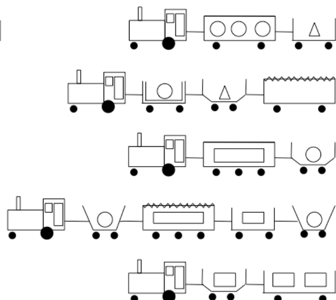
- Define predicate **eastbound/1**.





The West/East trains






```
eastbound(T):-car(T,C), closed(C), small(C).
```

The West/East trains

What if the rule is:

- “trains with a square load in an open car” ?
- “trains with a square load and an open car” ?
- “trains with a square load or a round load” ?
- “trains with an empty car” ?
- “trains with no car” ?

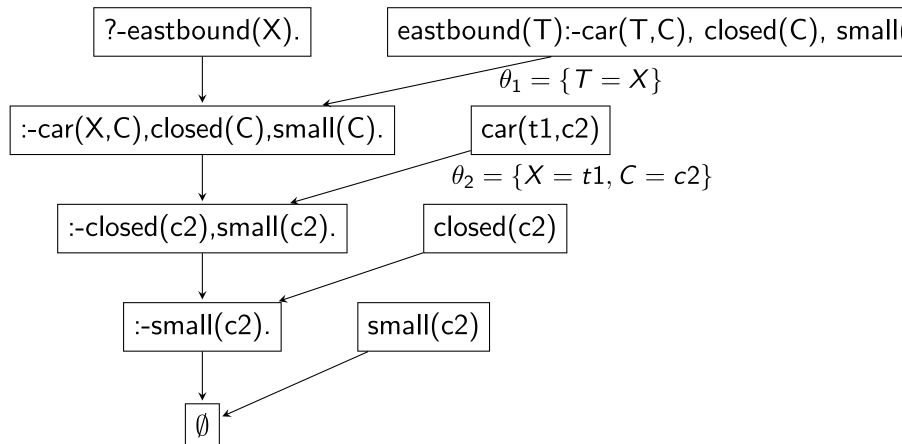
How Prolog works

Given the query `?-eastbound(X).`

- Look for a clause with a head that **unifies** with `eastbound(X)`.
- Prove that the **body** of the clause is true.
- Provide the resulting substitution.

How Prolog works

A **proof** that `eastbound(X)` is true for some `X`.



Unification

Two terms unify if they:

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

Unifyable pairs

- `adam` and `adam`
- `adam` and `X`
- `father(adam,abel)` and `father(X,abel)`

Unification

Two terms unify if they:

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

Non unifyable pairs

- `adam` and `eve`
- `adam` and `male(X)`
- `father(adam,abel)` and `father(X,X)`
- `X` and `s(X)`

Unification

- At each step, Prolog performs the strictly necessary substitutions to unify (if possible).
- This is also called mgu = *most general unifier*.
- We can obtain the mgu of two terms with **=/2**.

```
?- s(X,f(a)) = s(b,Y).
x=b,
Y=f(a).
```

Unification: Occur Check

- For efficiency reasons Prolog implementations do not check the occurrence of a variable inside a term.
- In other words no **occur check** is done.
- So sometimes Prolog can give a wrong answer.
- In practice this is (typically) not a problem.

Example of a wrong unification in Prolog:

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

SLD-resolution

SLD stands for *Selective, Linear for Definite Clauses*

- SLD-resolution is the *inference* method used in Logic Programming.
- It is a particular, more restricted, form of resolution of first order logic.
- It is used in logic programming due to its efficiency.
- It is **correct** and **complete** for **Horn clauses**.

A Horn Clause is a FOL disjunction with at most one positive literal.

$$\neg L_1 \vee \neg L_2 \vee \neg L_3 \vee \ldots \vee \neg L_n \vee P$$

Examples of Horn Clauses:

```
p(X):-q(x),r(X,Y).
:-man(X),mortal(X).
man(socrates).
:-true. %The null clause
```

SLD-resolution

Definition of SLD-derivation:

- Given a set of Horn clauses S and a set of goals $G = G_1, ..., G_q$
- An SLD-derivation is a sequence of negative clauses $< N_0, N_1, ..., N_p >$ such that:
  - $N_0 = G_j$, where G_j is one of the goals
  - For every N_i with the form `:- A1, ..., Ai, ... An` ($:-\ A_1, \ldots, A_i, \ \ldots \ \ A_n$)
  - ...there is some clause $A \ :-\ B_1, \ldots, B_m$
  - ...such that A and A_i unify
  - N_{i+1} is :-$\sigma_i(A_1, ..., A_{k-1}, B_1, ..., B_m, A_{k+1}, ..., A_n)$
- If $N_p = \square$ (the empty clause) we have a **refutation**, i.e., a proof.
- In that case the combination of all σ_i is the **answer substitution**

SLD-resolution

In Prolog typically:

- Goals are processed from left to right
- Clause are searched from top to bottom

SLD-resolution

Example program:

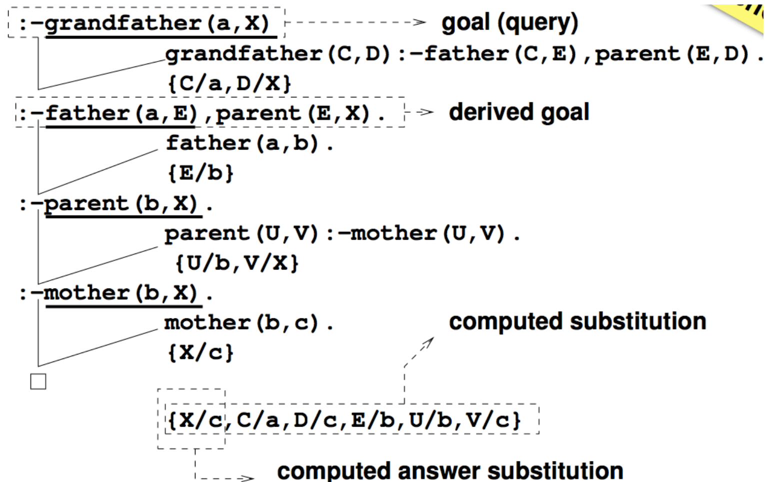
```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(a,b).
mother(b,c).
```

Goal:

```
:-grandfather(a,X).
```

SLD-resolution

Refutation proof tree:

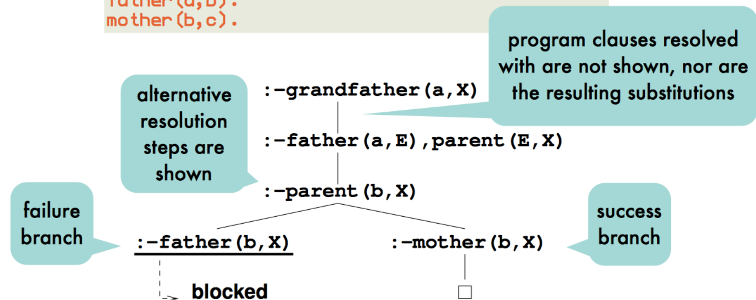


[http://soft.vub.ac.be/~cderoove/declarative_programming/decprog3_sld_cut_naf.pdf]

SLD-resolution

SLD-tree:

- a proof tree shows a possible resolution path
- an **SLD-tree** represents the search for that path

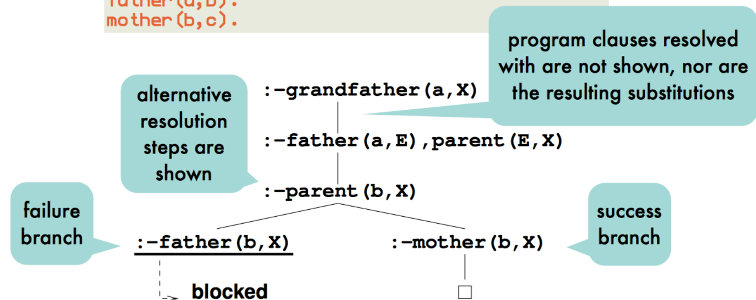


[http://soft.vub.ac.be/~cderoove/declarative_programming/decprog3_sld_cut_naf.pdf]

SLD-resolution

SLD-tree:

- a proof tree shows a possible resolution path
- when one branch fails, Prolog **backtracks**



[http://soft.vub.ac.be/~cderoove/declarative_programming/decprog3_sld_cut_naf.pdf]

SLD-resolution

Backtracking example:

```
daughter(X,Y) :- parent(Y,X), female(X).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
father(homer,bart).
father(homer,lisa).
mother(marge,bart).
mother(marge,lisa).
male(homer).  male(bart).
female(marge).   female(lisa).
```

Query 1: `?-daughter(lisa,homer).`

Query 2: `?-daughter(lisa,marge).`