Introduction to Logic Programming

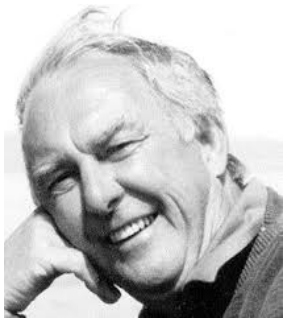
Agostino Dovier

Università di Udine
CLPLAB

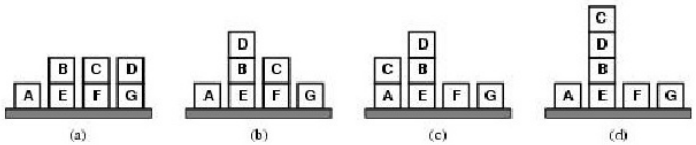
Udine, November 20, 2018

Where did Logic Programming come from?

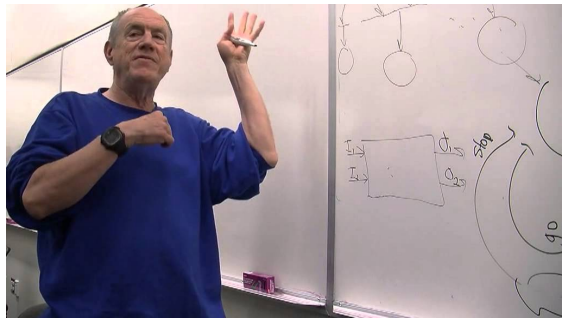
- Logic Programming was born 1972 (*circa*), presaged by related work by Ted Elcock (left), Cordell Green, Pat Hayes and Carl Hewitt (right) on applying theorem proving to problem solving (planning) and to question-answering systems.



Block world



Initial state (a)
Action(*Move*(*x*,*y*),
PRECOND: *Clear*(*x*) ∧ *Clear*(*y*) ∧ *On*(*x*,*z*)
EFFECT: *On*(*x*,*y*) ∧ *Clear*(*z*) ∧ ¬*On*(*x*,*z*) ∧ ¬*Clear*(*y*)
The agent first need to formulate a goal: *On*(*C*,*D*) ∧ *On*(*D*,*B*)



Where did Logic Programming come from?

- It blossomed from Alan Robinson's (left) seminal contribution, including the Resolution Principle, all the way into a practical, declarative, programming language with automated deduction at its core, through the vision and efforts of Alain Colmerauer and Bob Kowalski (right) .



Where did Logic Programming come from?

Deductive Reasoning and Formal Logic

Two basic ingredients: Modus Ponens (if p implies q is true and p is true, then q is true) and Generalization (if something is true of a class of things in general, this truth applies to all legitimate members of that class) allows to state the syllogism by Aristotele:

All human beings are mortal. Socrates is human.
Therefore, Socrates is mortal.

Formal Logic is a formal version of human deductive logic. It provides a formal language with an unambiguous syntax and a precise meaning, and it provides rules for manipulating expressions in a way that respects this meaning.

Where did Logic Programming come from?

Computational Logic

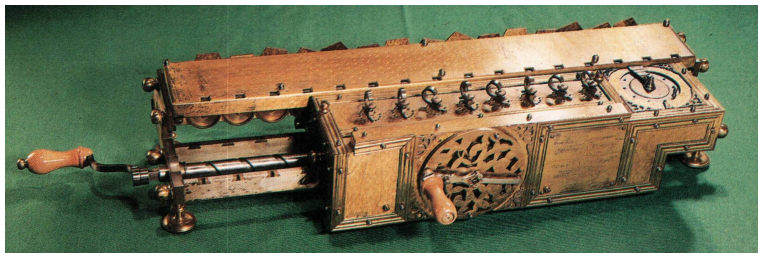
The existence of a formal language for representing information and the existence of a corresponding set of mechanical manipulation rules together make automated reasoning using computers possible. Computational logic is a branch of mathematics that is concerned with the theoretical underpinnings of automated reasoning. Like Formal Logic, Computational Logic is concerned with precise syntax and semantics and correctness and completeness of reasoning. Decidability and complexity issues arise.

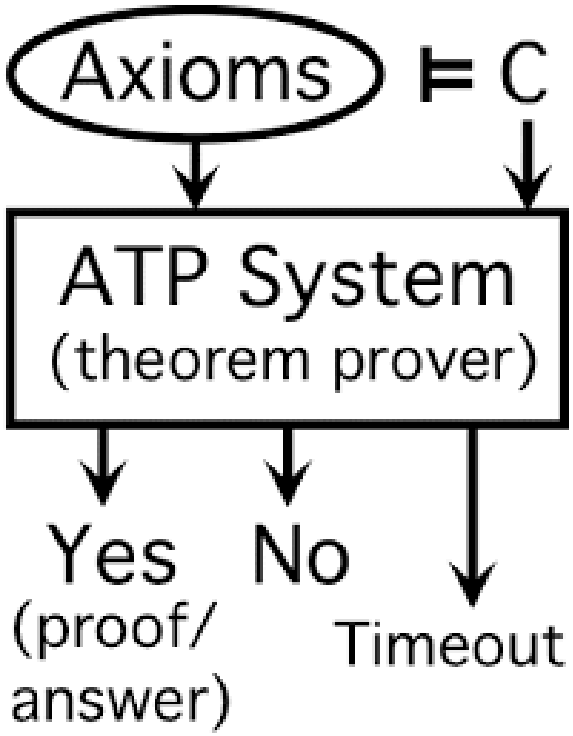
```
mortal(X) :- human(X).
human(socrates).
?- mortal(socrates).
?- yes
?- mortal(Y).
?- Y = socrates ? ;
no
```

Where did Logic Programming come from?

Theorem Proving

From (Gottfried Wilhelm von) Leibniz dream of mechanizing human reasoning using machines to modern computer-based automatic theorem proving

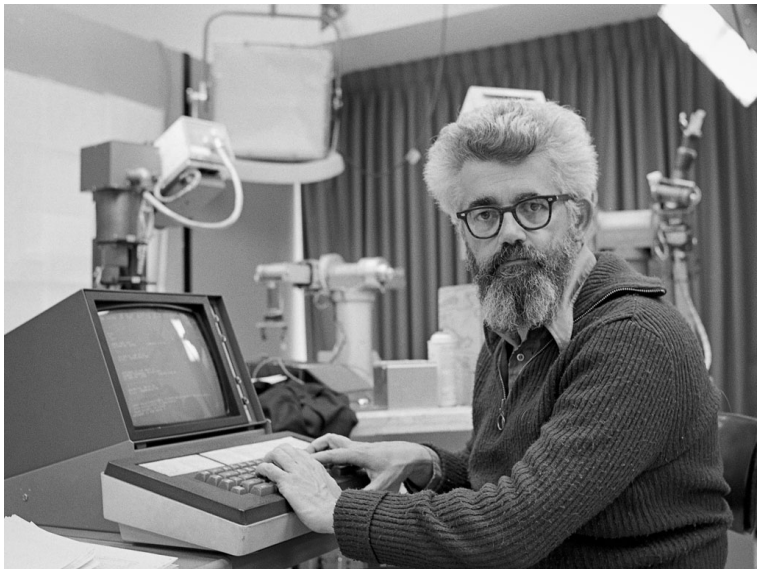




Where did Logic Programming come from?

AI

Expressing information in declarative sentences is far more modular than expressing it in segments of computer programs or in tables. Sentences can be true in a much wider context than specific programs can be used. The supplier of a fact does not have to understand much about how the receiver functions or how or whether the receiver will use it. The same fact can be used for many purposes, because the logical consequences of collections of facts can be available. [McC59]



John McCarthy (1927–2011)

Where did Logic Programming come from?

Programming Languages (late sixities, early seventies)

- Predicate/Function definition in imperative languages
  $\langle \text{HEAD} \rangle \langle \text{BODY} \rangle$
- Niklaus Wirth: Program = Algorithm + Data Structure
  [Prentice-Hall, 1976]

- Predicate Logic as a Programming Language
  $\langle \text{HEAD} \rangle \leftarrow \langle \text{BODY} \rangle$
- Bob Kowalski: Algorithm = Logic+Control

What is Logic Programming?

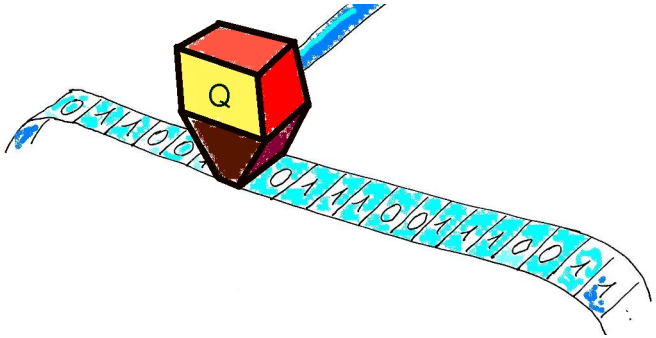
- The language Prolog, from the beginning, is a programming paradigm useful for Knowledge Representation and Reasoning, Deductive Databases, Computational linguistic, ...
- Prolog is often identified with Logic Programming (correct in the seventies, wrong nowadays)
- The first efficient implementation of Prolog is due to

  D.H.D. Warren (WAM–1983) 
- Now we have many: BProlog, SICStus Prolog, SWI Prolog, Yap Prolog, CIAO Prolog, ... all of them based on the WAM
- SWI Prolog (Jan Wielemaker et al) will be used in this course

  `http://www.swi-prolog.org/`

What is Logic Programming?

A small subset of Prolog (definite clause programming) is already *Turing complete*.




```
delta(q0,0,qi,1,left).
...
delta(qn,1,qj,0,right).
```

```
turing(Left,halt,S,Right,Left,halt,S,Right).
turing([L|L_i],Q,S,R_i,L_o,Q_o,S_o,R_o) :-
     delta(Q,S,Q1,S1,left),
     turing(L_i,Q1,L,[S1|R_i],L_o,Q_o,S_o,R_o).
turing(L_i,Q,S,[R|R_i],L_o,Q_o,S_o,R_o) :-
     delta(Q,S,Q1,S1,right),
     turing([S1|L_i],Q1,R,R_i,L_o,Q_o,S_o,R_o).
turing([],Q,S,R_i,L_o,Q,S_o,R_o) :-
      turing([0],Q,S,R_i,L_o,Q,S_o,R_o).
turing(L_i,Q,S,[],L_o,Q,S_o,R_o) :-
      turing(L_i,Q,S,[0],L_o,Q,S_o,R_o).
```

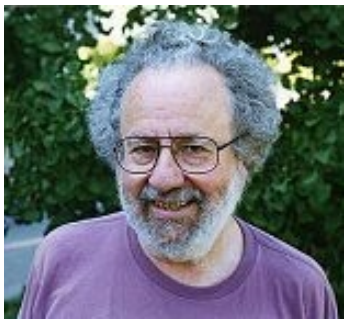
What is Logic Programming?

Moreover, the same subclass (definite clause programming) has lovely semantical properties.

$$P \text{ has a model} \Leftrightarrow P \text{ has a Herbrand model}$$

$$M_P = \bigcap_{M \text{ is a Herbrand model of } P} = T_P \uparrow \omega(\emptyset)$$

$$M_P = \{A \ : \ \text{there is a SLD resolution for } A \text{ from } P\}$$

What is Logic Programming?

- The declarative nature allows extensions/variations such as *constraint logic programming*, *functional logic programming*, *probabilistic logic programming*, *inductive logic programming*, ...
- All current Prolog systems have a large set of built-ins and complete interfaces with other languages and/or OS primitives, graphics, DB, etc.
- Search in logic programming is naturally parallelized.
- Inference techniques are inherited by part of big systems (e.g., IBM Watson)
- And, since 1999 we have Answer Set Programming (Knowledge Representation and Reasoning tool with a fast solver)

Logic Programming in the World

ALP (1986)

- website `www.logicprogramming.org`
- International meeting (ICLP) since 1982
- International Journal Theory and Practice of Logic Programming
- Other meetings (PADL, LOPSTR, ILP, LPNMR, . . .)
- International Schools
- Newsletter (every 3 months – ask for being included in the mailing list)

Logic Programming in Italy

GULP (1985)

- website: `www.programmazionelogica.it`
- GULP is the Italian Association for Logic Programming (Gruppo Utenti e ricercatori di Logic Programming)
- GULP is affiliated to ALP (but older!)
- AIM: to keep the interest in LP and related themes alive by promoting various initiatives both in research and education; an opportunity for young researchers to be introduced into an active and challenging research area in a *very informal* and *friendly* way
- Annual meeting (last one in Sept 2017 in Bolzano), summer/winter schools, workshops, student's grants, PhD theses prizes, ...

Syntax of Logic Programming

Terms

- Let $\mathcal{C}$ be a set of constant symbols
  (e.g., `a`, `b`, `c`, `socrate`, `uomo`, ...)
- Let $\mathcal{V}$ be a set of variable symbols
  (e.g., `X`, `Y`, `Z`, `X1`, `X2`, ...)
- Let $\mathcal{F}$ be a set of function symbols
  (e.g., `f`, `g`, `h`, `sqrt`, `piu`, `per`, ...)
- Each symbol $f \in \mathcal{F}$ has its own arity (number of arguments)
  $\texttt{ar}(f) > 0$ (e.g., $\texttt{ar}(\texttt{sqrt}) = 1, \texttt{ar}(\texttt{piu}) = 2$).
- We assume that $\texttt{ar}(c) = 0$ for $c \in \mathcal{C}$ and $\texttt{ar}(X) = 0$ for $X \in \mathcal{V}$

TERMS

- If $\mathtt{c} \in \mathcal{C}$ then $\mathtt{c}$ is a *term*
- If $\mathtt{X} \in \mathcal{V}$ then $\mathtt{X}$ is a *term*
- If $f \in \mathcal{F}$ and $\mathtt{ar}(f) = n$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a *term*.

- A term without variables is said a ground term
- E.g., $0, s(s(0)), s(s(X)), \mathtt{sqrt}(\mathtt{piu}(s(s(Y)), s(0)))$ are terms
- E.g., $0, s(s(0)), \mathtt{sqrt}(\mathtt{piu}(s(s(0)), s(0)))$ are ground terms $(\mathtt{ar}(s) = \mathtt{ar}(\mathtt{sqrt}) = 1, \mathtt{ar}(\mathtt{piu}) = 2)$

TERMS

- If $\mathtt{c} \in \mathcal{C}$ then $\mathtt{c}$ is a *term*
- If $\mathtt{X} \in \mathcal{V}$ then $\mathtt{X}$ is a *term*
- If $f \in \mathcal{F}$ and $\mathtt{ar}(f) = n$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a *term*.

- A term without variables is said a ground term
- E.g., $0, s(s(0)), s(s(X)), \mathtt{sqrt}(\mathtt{piu}(s(s(Y)), s(0)))$ are terms
- E.g., $0, s(s(0)), \mathtt{sqrt}(\mathtt{piu}(s(s(0)), s(0)))$ are ground terms $(\mathtt{ar}(s) = \mathtt{ar}(\mathtt{sqrt}) = 1, \mathtt{ar}(\mathtt{piu}) = 2)$

Terms

- If $\mathtt{c} \in \mathcal{C}$ then $\mathtt{c}$ is a *term*
- If $\mathtt{X} \in \mathcal{V}$ then $\mathtt{X}$ is a *term*
- If $f \in \mathcal{F}$ and $\mathtt{ar}(f) = n$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a *term*.

- A term without variables is said a ground term
- E.g., $0, s(s(0)), s(s(X)), \mathtt{sqrt}(\mathtt{piu}(s(s(Y)), s(0)))$ are terms
- E.g., $0, s(s(0)), \mathtt{sqrt}(\mathtt{piu}(s(s(0)), s(0)))$ are ground terms ($\mathtt{ar}(s) = \mathtt{ar}(\mathtt{sqrt}) = 1, \mathtt{ar}(\mathtt{piu}) = 2$)

Atomic Formulas (atoms)

- Let $\mathcal{P}$ be a set of predicate symbols (e.g., `p`, `q`, `r`, `genitore`, `allievo`, `coetaneo`, `eq`, `leq`, `integer`,...)
- Each symbol $p \in \mathcal{P}$ has its own arity (number of arguments) $\texttt{ar}(p) \geq 0$
  (e.g., $\texttt{ar}(\texttt{leq}) = 2, \texttt{ar}(\texttt{father}) = 2, \texttt{ar}(\texttt{integer}) = 1$).
- If $p \in \mathcal{P}$, $\texttt{ar}(p) = n$, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula* (or, simply, an atom)
- E.g., $\texttt{integer}(s(s(s(0)))), \texttt{leq}(0, s(s(0)))$, $\texttt{father}(\texttt{abramo}, \texttt{isacco}), \texttt{p}(X, Y, \texttt{a})$ are atoms.
- A *literal* is either an atom or `not` A where A is an atom.
- We'll make use of 0-ary atoms. E.g. $p, q, r, \ldots$ This way, we can encode propositional logics (vs first-order logic)

Atomic Formulas (atoms)

- Let $\mathcal{P}$ be a set of predicate symbols (e.g., `p`, `q`, `r`, `genitore`, `allievo`, `coetaneo`, `eq`, `leq`, `integer`,...)
- Each symbol $p \in \mathcal{P}$ has its own arity (number of arguments) $\texttt{ar}(p) \geq 0$
  (e.g., $\texttt{ar}(\texttt{leq}) = 2, \texttt{ar}(\texttt{father}) = 2, \texttt{ar}(\texttt{integer}) = 1$).
- If $p \in \mathcal{P}$, $\texttt{ar}(p) = n$, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula* (or, simply, an atom)
- E.g., $\texttt{integer}(s(s(s(0)))), \texttt{leq}(0, s(s(0))),$ $\texttt{father}(\texttt{abramo}, \texttt{isacco}), \texttt{p}(X, Y, \texttt{a})$ are atoms.
- A *literal* is either an atom or `not` A where A is an atom.
- We'll make use of 0-ary atoms. E.g. $p, q, r, \ldots$ This way, we can encode propositional logics (vs first-order logic)

Atomic Formulas (atoms)

- Let $\mathcal{P}$ be a set of predicate symbols (e.g., `p`, `q`, `r`, `genitore`, `allievo`, `coetaneo`, `eq`, `leq`, `integer`,...)
- Each symbol $p \in \mathcal{P}$ has its own arity (number of arguments) $\texttt{ar}(p) \geq 0$
  (e.g., $\texttt{ar}(\texttt{leq}) = 2, \texttt{ar}(\texttt{father}) = 2, \texttt{ar}(\texttt{integer}) = 1$).
- If $p \in \mathcal{P}$, $\texttt{ar}(p) = n$, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula* (or, simply, an atom)
- E.g., $\texttt{integer}(s(s(s(0)))), \texttt{leq}(0, s(s(0))),$ $\texttt{father}(\texttt{abramo}, \texttt{isacco}), \texttt{p}(X, Y, \texttt{a})$ are atoms.
- A *literal* is either an atom or `not` A where A is an atom.
- We'll make use of 0-ary atoms. E.g. $p, q, r, \ldots$ This way, we can encode propositional logics (vs first-order logic)

Atomic Formulas (atoms)

- Let $\mathcal{P}$ be a set of predicate symbols (e.g., `p`, `q`, `r`, `genitore`, `allievo`, `coetaneo`, `eq`, `leq`, `integer`,...)
- Each symbol $p \in \mathcal{P}$ has its own arity (number of arguments) $\texttt{ar}(p) \geq 0$
  (e.g., $\texttt{ar}(\texttt{leq}) = 2, \texttt{ar}(\texttt{father}) = 2, \texttt{ar}(\texttt{integer}) = 1$).
- If $p \in \mathcal{P}$, $\texttt{ar}(p) = n$, and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula* (or, simply, an atom)
- E.g., $\texttt{integer}(s(s(s(0)))), \texttt{leq}(0, s(s(0))),$ $\texttt{father}(\texttt{abramo}, \texttt{isacco}), \texttt{p}(X, Y, \texttt{a})$ are atoms.
- A *literal* is either an atom or `not` A where A is an atom.
- We'll make use of 0-ary atoms. E.g. $p, q, r, \ldots$ This way, we can encode propositional logics (vs first-order logic)

RULES (DEFINITE)

$$\underbrace{H}_{head} \leftarrow \underbrace{B_1, \ldots, B_m}_{body} \tag{1}$$

where $H, B_1, \ldots, B_m$ are atoms, $m \geq 0$ is said a definite rule.
The comma "," stands for $\wedge$ (and). The arrow "$\leftarrow$" is written ":-"

If $m = 0$ the rule is said a *fact*

$$H \leftarrow B_1, \ldots, B_m$$

From a logical point of view it is equivalent to:

$$H \vee \neg B_1 \vee \cdots \vee \neg B_m$$

namely, a disjunction of literals (a.k.a. a clause) with exactly one positive literal (Horn clauses have at most one positive literal).

A program is simply a set of rules (order does not matter, in principle).

Here is a simple program giving information about the British Royal family:

```
parent(elizabeth,charles).
parent(philip,charles).
parent(diana,william).
parent(diana,harry).
parent(charles,william).
parent(charles,harry).
```

Here `parent` is a predicate. All names are (terms consisting of) constant symbols.
These above rules have empty bodies and thus they are *facts*. They allow to populate extensionally a Database.

Let us write some rules, trying to *define* intensionally a predicate:

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

This is a definite clause. How can we interpret such a "Z"?

VIEW 1 : (Given X and Y) if *there exists* a Z such that X is parent of Z, and Z is parent of Y, then X is a grandparent of Y.

VIEW 2 : (Given X and Y and Z) if X is parent of Z, and Z is parent of Y, then X is a grandparent of Y.

Let us write some rules, trying to *define* intensionally a predicate:

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

This is a definite clause. How can we interpret such a "Z"?

VIEW 1 : (Given X and Y) if *there exists* a Z such that X is parent of Z, and Z is parent of Y, then X is a grandparent of Y.

VIEW 2 : (Given X and Y and Z) if X is parent of Z, and Z is parent of Y, then X is a grandparent of Y.

Let us write some rules, trying to *define* intensionally a predicate:

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

This is a definite clause. How can we interpret such a "Z"?

VIEW 1 : (Given X and Y) if *there exists* a Z such that X is parent of Z, and Z is parent of Y, then X is a grandparent of Y.

VIEW 2 : (Given X and Y and Z) if X is parent of Z, and Z is parent of Y, then X is a grandparent of Y.

Let us write some rules, trying to *define* intensionally a predicate:

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

This is a definite clause. How can we interpret such a "Z"?

view 1:

$$(\forall \mathtt{X})(\forall \mathtt{Y})\big((\exists \mathtt{Z})(\mathtt{parent}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{parent}(\mathtt{Z},\mathtt{Y})) \rightarrow \mathtt{granparent}(\mathtt{X},\mathtt{Y})\big)$$

view 2:

$$(\forall \mathtt{X})(\forall \mathtt{Y})(\forall \mathtt{Z})\big(\mathtt{parent}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{parent}(\mathtt{Z},\mathtt{Y}) \rightarrow \mathtt{granparent}(\mathtt{X},\mathtt{Y})\big)$$

Luckily, they are equivalent (exercise!)

Let us write some rules, trying to *define* intensionally a predicate:

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

This is a definite clause. How can we interpret such a "Z"?

view 1:

$$(\forall \mathtt{X})(\forall \mathtt{Y})\big((\exists \mathtt{Z})(\mathtt{parent}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{parent}(\mathtt{Z},\mathtt{Y})) \rightarrow \mathtt{granparent}(\mathtt{X},\mathtt{Y})\big)$$

view 2:

$$(\forall \mathtt{X})(\forall \mathtt{Y})(\forall \mathtt{Z})\big(\mathtt{parent}(\mathtt{X},\mathtt{Z}) \wedge \mathtt{parent}(\mathtt{Z},\mathtt{Y}) \rightarrow \mathtt{granparent}(\mathtt{X},\mathtt{Y})\big)$$

Luckily, they are equivalent (exercise!)

Let us write some rules, trying to *define* intensionally a predicate:

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

A solver should be able to deduce:

```
grandparent(elizabeth,william).
grandparent(elizabeth,harry).
grandparent(philip,william).
grandparent(philip,harry).
```

Let us enlarge the database (order does not matter)

```
parent(william,george).
parent(william,charlotte).
parent(kate, george).
parent(kate, charlotte).
```

We can define now the "`ancestor`" predicate.

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

This is the first use of "*recursion*". Recursion is fundamental in declarative programming (either functional or logic).

Let us define the "`married`" and "`sibling`" predicates:

```
married(X,Y) :- parent(X,Z), parent(Y,Z).
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

Is this definition completely correct?

Are you sibling of yourself?

Patch:

```
married(X,Y) :- parent(X,Z), parent(Y,Z), X \= Y.
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y.
```

Let us define the "`married`" and "`sibling`" predicates:

```
married(X,Y) :- parent(X,Z), parent(Y,Z).
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

Is this definition completely correct?

Are you sibling of yourself?

Patch:

```
married(X,Y) :- parent(X,Z), parent(Y,Z), X \= Y.
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y.
```

Let us define the "`married`" and "`sibling`" predicates:

```
married(X,Y) :- parent(X,Z), parent(Y,Z).
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

Is this definition completely correct?

Are you sibling of yourself?

Patch:

```
married(X,Y) :- parent(X,Z), parent(Y,Z), X \= Y.
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y.
```

Let us add some extra information:

```
female(elizabeth).      female(diana).
female(kate).           female(charlotte).
male(philip).           male(charles).
male(william).          male(harry).
male(george).
```

Then we can define other predicates, e.g.

```
isfather(X)   :- parent(X,Y), male(X).
ismother(X)   :- parent(X,Y), female(X).
brother(X,Y)  :- sibling(X,Y), male(X).
has_a_sister(X) :- sibling(X,Y), female(Y).
```

Let us define the notion of being a natural number "`nat`":

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

What are we expecting?

`nat(0), nat(s(0)), nat(s(s(0))), ...`

An infinite set of answers (is it viable?)

In the following, let us denote $\underbrace{s(s(\cdots(s(}_{n}0))\cdots))$ by $\overline{n}$.

Let us define the notion of being a natural number "`nat`":

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

What are we expecting?

```
nat(0), nat(s(0)), nat(s(s(0))), ...
```

An infinite set of answers (is it viable?)

In the following, let us denote $\underbrace{s(s(\cdots(s(}_{n}0))\cdots))$ by $\overline{n}$.

Let us define now the "`sum`" predicate:

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

What are we expecting?

E.g., $\texttt{sum}(\overline{5}, 0, \overline{5})$
$\texttt{sum}(\overline{2}, \overline{4}, \overline{6})$
$\texttt{sum}(\overline{2}, \overline{4}, Z) \mapsto Z = \overline{6}$
$\texttt{sum}(X, \overline{4}, \overline{6}) \mapsto X = \overline{2}$

Last line is interesting ...

Let us define now the "`sum`" predicate:

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

What are we expecting?

E.g., $\texttt{sum}(\overline{5}, 0, \overline{5})$
$\texttt{sum}(\overline{2}, \overline{4}, \overline{6})$
$\texttt{sum}(\overline{2}, \overline{4}, Z) \mapsto Z = \overline{6}$
$\texttt{sum}(X, \overline{4}, \overline{6}) \mapsto X = \overline{2}$

Last line is interesting . . .

Semantics of Logic Programs

Universes and Interpretations

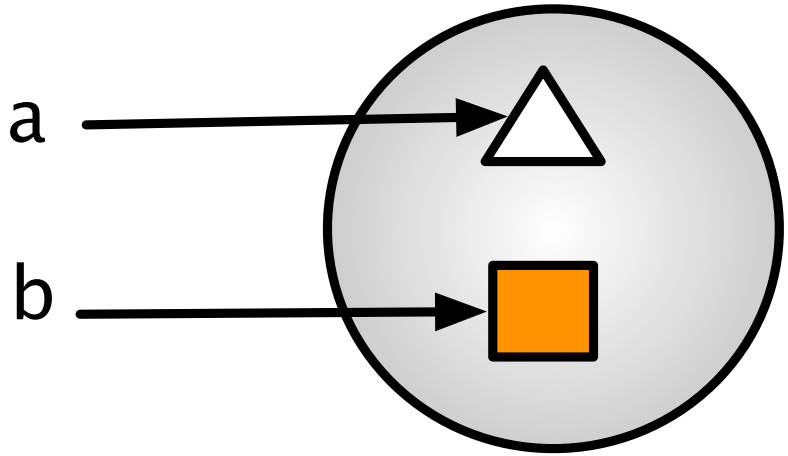
```
p(a).
p(b).
```

A program (or in general, a first-order theory) P is a built from a list of symbols.
In this case constants a and b, and one unary predicate symbol p
We would like to assign a meaning (interpretation) to these symbols on a universe of objects.

Universes and Interpretations

```
p(a).
p(b).
```

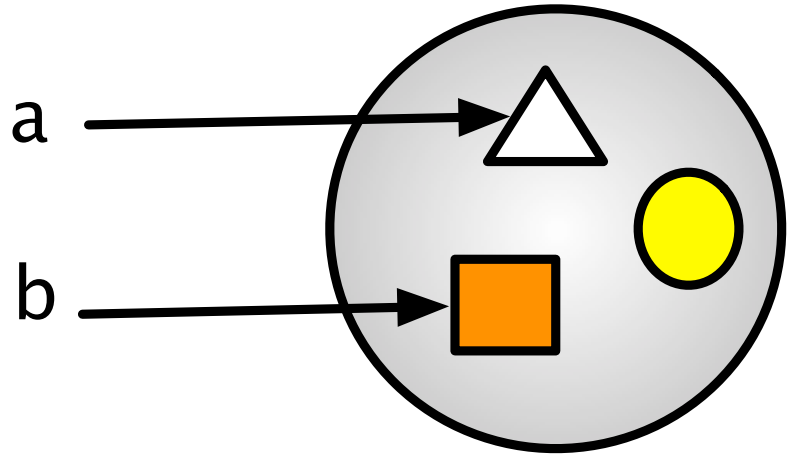
A program (or in general, a first-order theory) *P* is a built from a list of symbols.
In this case constants *a* and *b*, and one unary predicate symbol *p*
We would like to assign a meaning (interpretation) to these symbols on a universe of objects.



Universes and Interpretations

```
p(a).
p(b).
```

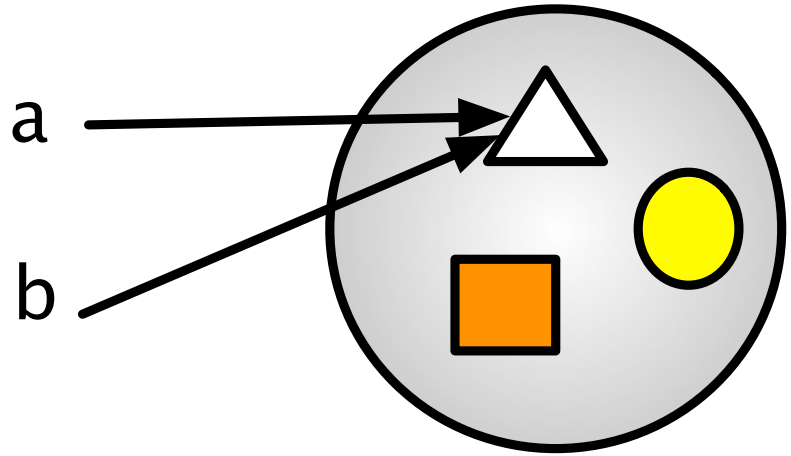
A program (or in general, a first-order theory) *P* is a built from a list of symbols.
In this case constants *a* and *b*, and one unary predicate symbol *p*
We would like to assign a meaning (interpretation) to these symbols on a universe of objects.



Universes and Interpretations

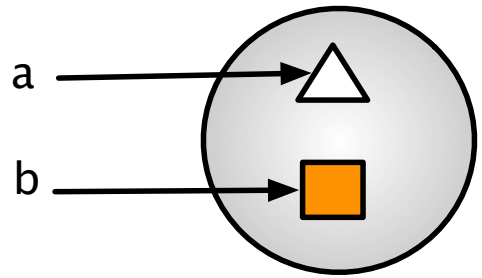
```
p(a).
p(b).
```

A program (or in general, a first-order theory) *P* is a built from a list of symbols.
In this case constants *a* and *b*, and one unary predicate symbol *p*
We would like to assign a meaning (interpretation) to these symbols on a universe of objects.

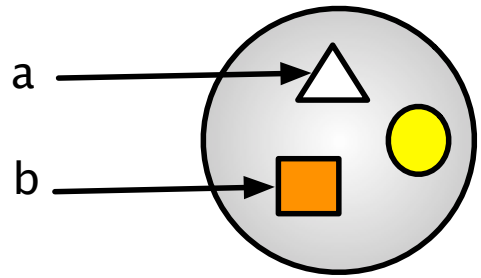


Universes and Interpretations

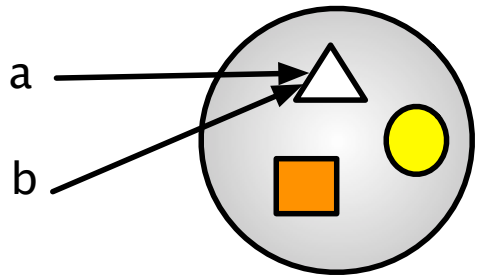
Different interpretations for constant symbols induce different interpretations for first-order formulas (with equality)



$a \neq b$



$\exists X\,(X \neq a \wedge X \neq b)$



$a = b$

Universes and Interpretations

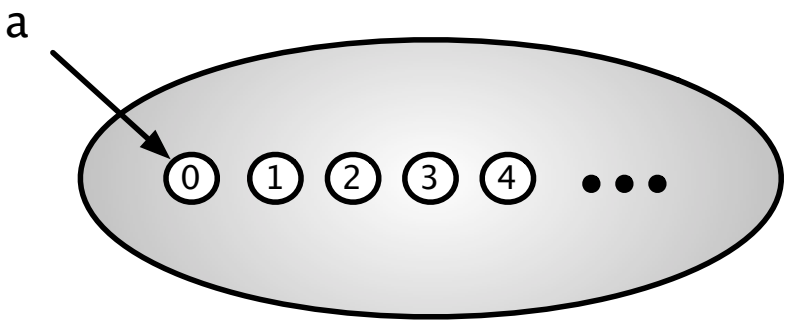
```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

Constant a and function symbols $s/1$ and $g/2$. Function symbols must be interpreted as functions ($g(x, y) = z$ means $(x, y, z) \in G$, where G is the interpretation viewed as set of triples)

Universes and Interpretations

```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

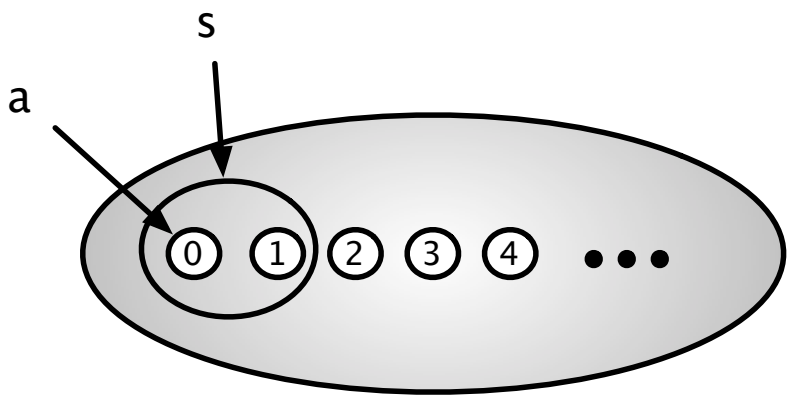
Constant a and function symbols $s/1$ and $g/2$. Function symbols must be interpreted as functions ($g(x, y) = z$ means $(x, y, z) \in G$, where G is the interpretation viewed as set of triples)



Universes and Interpretations

```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

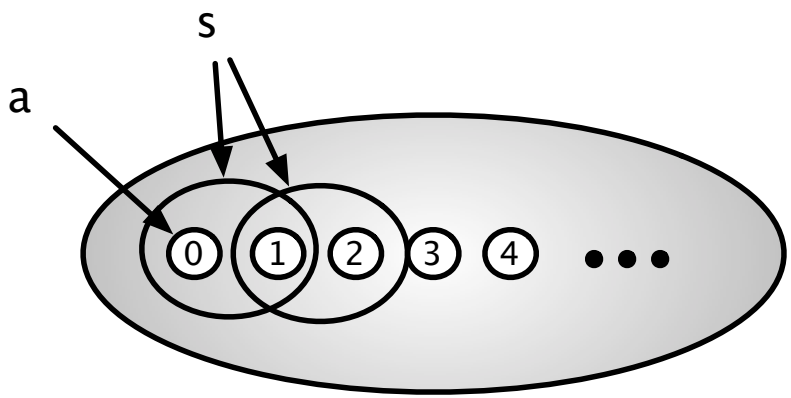
Constant a and function symbols $s/1$ and $g/2$. Function symbols must be interpreted as functions ($g(x, y) = z$ means $(x, y, z) \in G$, where G is the interpretation viewed as set of triples)



Universes and Interpretations

```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

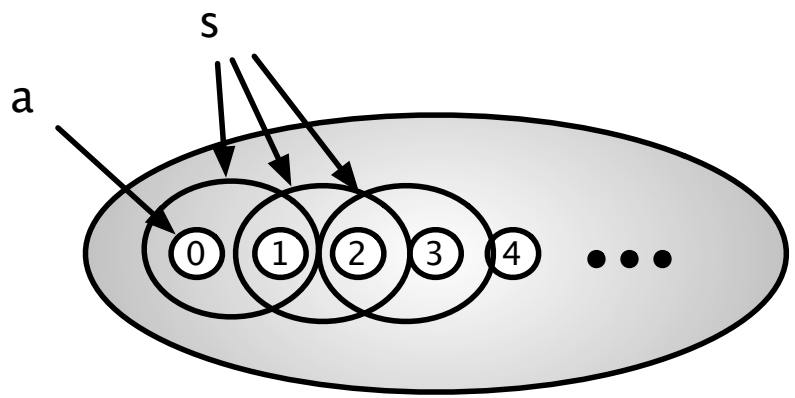
Constant a and function symbols $s/1$ and $g/2$. Function symbols must be interpreted as functions ($g(x, y) = z$ means $(x, y, z) \in G$, where G is the interpretation viewed as set of triples)



Universes and Interpretations

```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

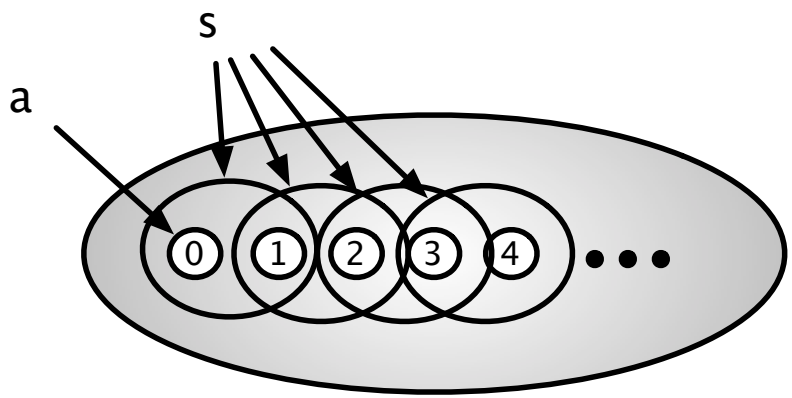
Constant a and function symbols $s/1$ and $g/2$. Function symbols must be interpreted as functions ($g(x, y) = z$ means $(x, y, z) \in G$, where G is the interpretation viewed as set of triples)



Universes and Interpretations

```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

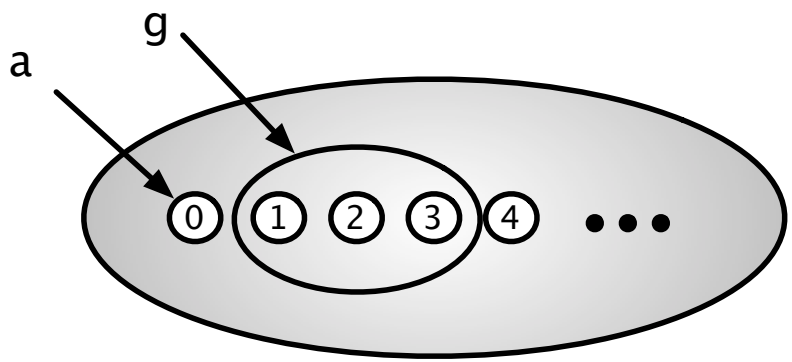
Constant a and function symbols $s/1$ and $g/2$. Function symbols must be interpreted as functions ($g(x, y) = z$ means $(x, y, z) \in G$, where G is the interpretation viewed as set of triples)



Universes and Interpretations

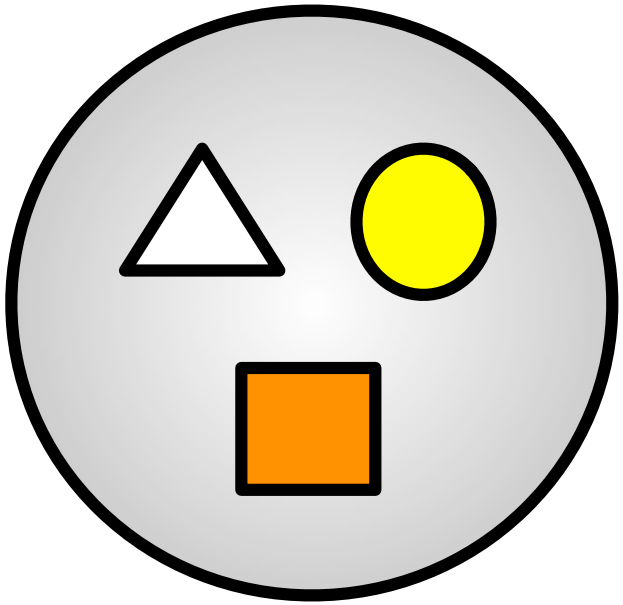
```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

Constant a and function symbols $s/1$ and $g/2$. Function symbols must be interpreted as functions ($g(x, y) = z$ means $(x, y, z) \in G$, where G is the interpretation viewed as set of triples)



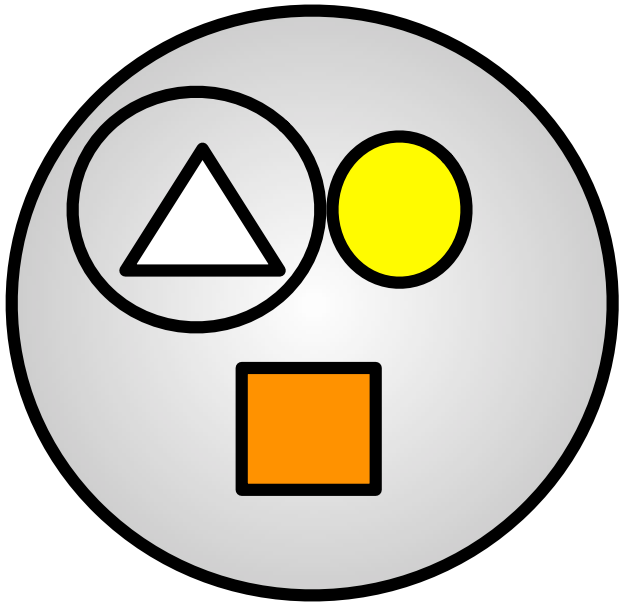
Universes and Interpretations

Predicate symbols (e.g. $p/1, q/n$) should be interpreted (as 1-ary, n-ary relations).



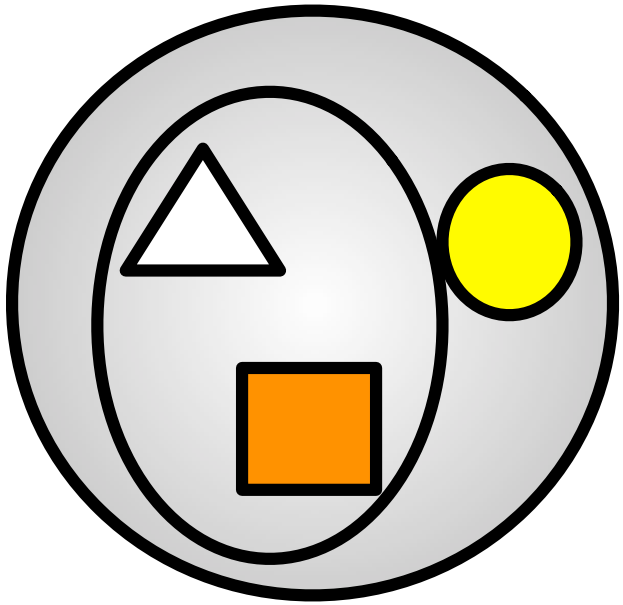
Universes and Interpretations

Predicate symbols (e.g. $p/1, q/n$) should be interpreted (as 1-ary, n-ary relations).



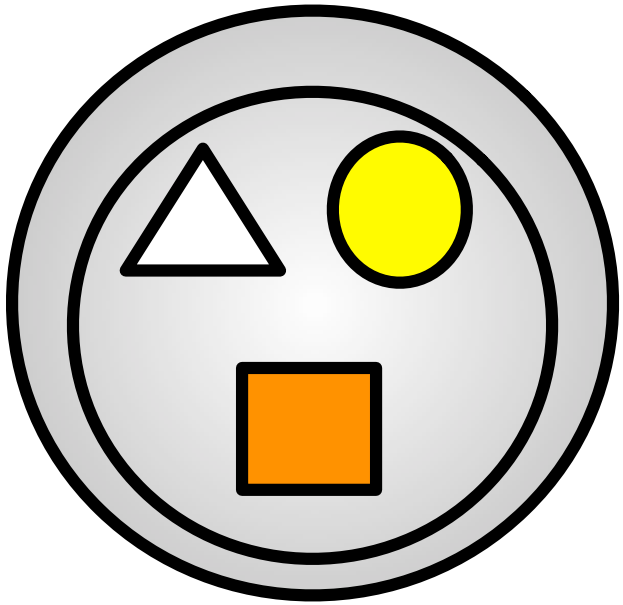
Universes and Interpretations

Predicate symbols (e.g. $p/1, q/n$) should be interpreted (as 1-ary, n-ary relations).



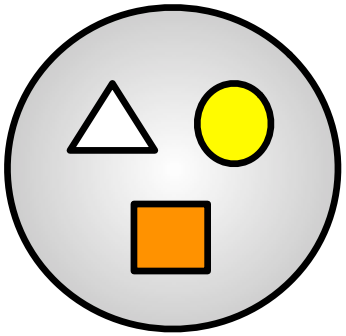
Universes and Interpretations

Predicate symbols (e.g. $p/1, q/n$) should be interpreted (as 1-ary, n-ary relations).

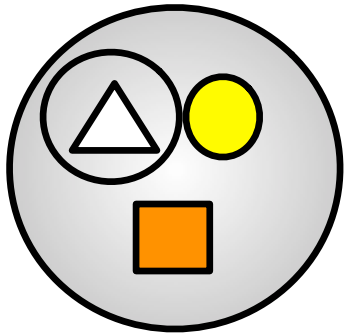


Universes and Interpretations

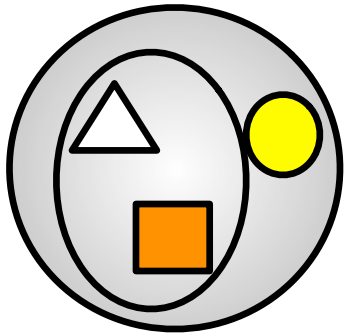
Different interpretations for predicate symbols induce different interpretations for first-order formulas (with equality)



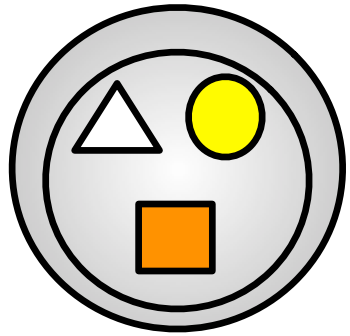
$\forall X\ \neg p(X)$



$\exists X\ p(X) \wedge$
$\exists X \exists Y (X \neq Y \wedge \neg p(X) \wedge \neg p(Y))$



$\exists X\ (\neg p(X)) \wedge$
$\exists X \exists Y (X \neq Y \wedge p(X) \wedge p(Y))$

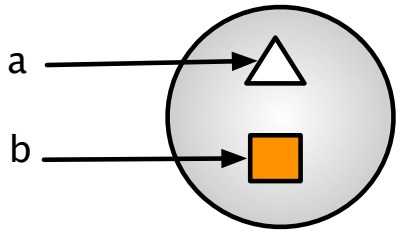


$\forall X\ p(X)$

MODELS

Some of the various interpretations of constant, function, and predicate symbols can be models of the program (or of the theory) P.

```
p(a).
q(b).
r(X) ← p(X).
```



Let us denote $\overline{a}$ =triangle and $\overline{b}$ =square. Let use denote with P, Q, R the interpretations of the predicate symbols p, q, r.

An interpretation that satisfies the logical meaning of all the formulas of P is a model.

$P = \{\overline{a}\}, Q = \{\overline{b}\}, R = \{\overline{a}, \overline{b}\}$ is a model.

$P = \{\overline{a}, \overline{b}\}, Q = \{\overline{b}\}, R = \{\overline{a}\}$ is NOT a model.

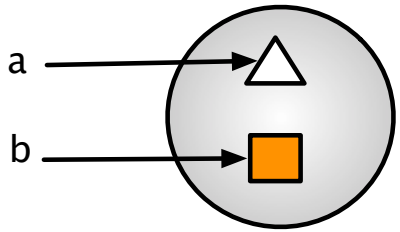
An atom $q(t_1, \ldots, t_n)$ is a logical consequence of a program/theory P if $(\overline{t_1}, \ldots, \overline{t_n}) \in Q$ in all (interpretations that are) models of P.

We say that $P \models q(t_1, \ldots, t_n)$.

MODELS

Some of the various interpretations of constant, function, and predicate symbols can be models of the program (or of the theory) P.

```
p(a).
q(b).
r(X) ← p(X).
```



Let us denote $\overline{a}$ =triangle and $\overline{b}$ =square. Let use denote with P, Q, R the interpretations of the predicate symbols p, q, r.

An interpretation that satisfies the logical meaning of all the formulas of P is a model.

$P = \{\overline{a}\}, Q = \{\overline{b}\}, R = \{\overline{a}, \overline{b}\}$ is a model.

$P = \{\overline{a}, \overline{b}\}, Q = \{\overline{b}\}, R = \{\overline{a}\}$ is NOT a model.

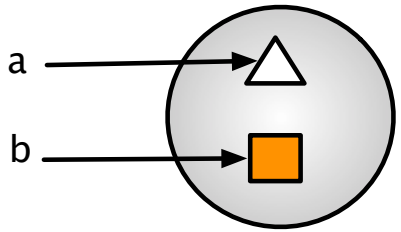
An atom $q(t_1, \ldots, t_n)$ is a logical consequence of a program/theory P if $(\overline{t_1}, \ldots, \overline{t_n}) \in Q$ in all (interpretations that are) models of P.

We say that $P \models q(t_1, \ldots, t_n)$.

Models

Some of the various interpretations of constant, function, and predicate symbols can be models of the program (or of the theory) P.

```
p(a).
q(b).
r(X) ← p(X).
```



Let us denote $\overline{a}$ =triangle and $\overline{b}$ =square. Let use denote with P, Q, R the interpretations of the predicate symbols p, q, r.
An interpretation that satisfies the logical meaning of all the formulas of P is a model.
$P = \{\overline{a}\}, Q = \{\overline{b}\}, R = \{\overline{a}, \overline{b}\}$ is a model.
$P = \{\overline{a}, \overline{b}\}, Q = \{\overline{b}\}, R = \{\overline{a}\}$ is NOT a model.

An atom $q(t_1, \ldots, t_n)$ is a logical consequence of a program/theory P if $(\overline{t_1}, \ldots, \overline{t_n}) \in Q$ in all (interpretations that are) models of P.
We say that $P \models q(t_1, \ldots, t_n)$.

Models

The set of logical consequences seems to be what we expect from the program.

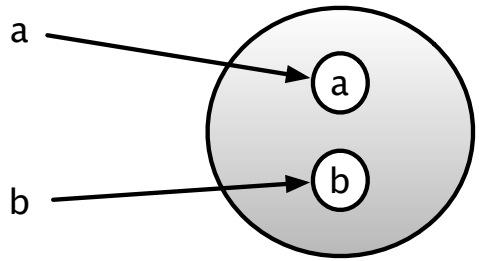
$P \models q(t_1, \ldots, t_n)$

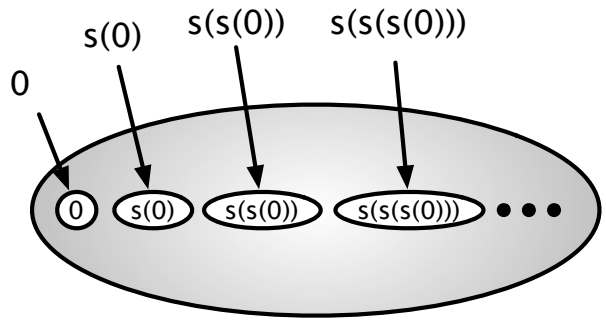
An atom $q(t_1, \ldots, t_n)$ is a logical consequence of a program/theory P if $(\overline{t_1}, \ldots, \overline{t_n}) \in Q$ in all (interpretations that are) models of P.

The questions are: Does it exist always? If yes, how to compute it?

HERBRAND INTERPRETATIONS

Let us consider the set of all ground terms that can be built with constant and function symbols in a program P.
This set can be used as a Universe for interpretations (the Herbrand Universe or H_P).
Ground terms are interpreted as themselves

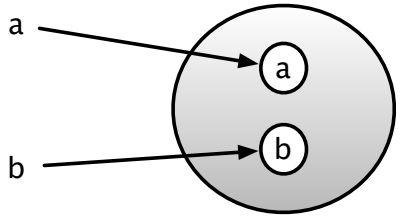




HERBRAND MODELS

Interpretations on the Herbrand Universe can be (or not) models (Herbrand models)

```
p(a).
q(b).
r(X) ← p(X).
```



Now, $\overline{a} = a$ and $\overline{b} = b$. Let us denote with P, Q, R the interpretations of the predicate symbols p, q, r.

1. $P = \{\overline{a}\}, Q = \{\overline{b}\}, R = \{\overline{a}, \overline{b}\}$ is a model.
2. $P = \{\overline{a}, \overline{b}\}, Q = \{\overline{b}\}, R = \{\overline{a}\}$ is NOT a model.

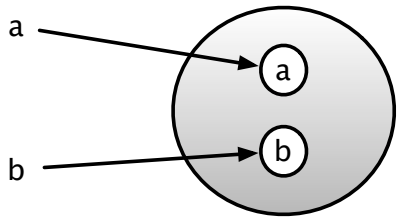
Herbrand interpretations and models can be represented uniquely by set of atoms:

1. $\{p(a), q(b), r(a), r(b)\}$ (model)
2. $\{p(a), p(b), q(b), r(a)\}$ (not a model)

HERBRAND MODELS

Interpretations on the Herbrand Universe can be (or not) models (Herbrand models)

```
p(a).
q(b).
r(X) ← p(X).
```



Now, $\overline{a} = a$ and $\overline{b} = b$. Let us denote with P, Q, R the interpretations of the predicate symbols p, q, r.

1. $P = \{\overline{a}\}, Q = \{\overline{b}\}, R = \{\overline{a}, \overline{b}\}$ is a model.
2. $P = \{\overline{a}, \overline{b}\}, Q = \{\overline{b}\}, R = \{\overline{a}\}$ is NOT a model.

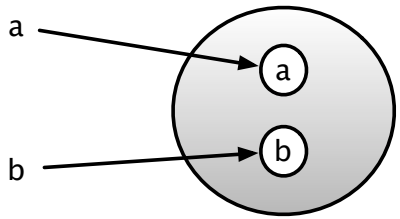
Herbrand interpretations and models can be represented uniquely by set of atoms:

1. $\{p(a), q(b), r(a), r(b)\}$ (model)
2. $\{p(a), p(b), q(b), r(a)\}$ (not a model)

HERBRAND MODELS

Interpretations on the Herbrand Universe can be (or not) models (Herbrand models)

```
p(a).
q(b).
r(X) ← p(X).
```



Now, $\overline{a} = a$ and $\overline{b} = b$. Let us denote with P, Q, R the interpretations of the predicate symbols p, q, r.

1. $P = \{\overline{a}\}, Q = \{\overline{b}\}, R = \{\overline{a}, \overline{b}\}$ is a model.
2. $P = \{\overline{a}, \overline{b}\}, Q = \{\overline{b}\}, R = \{\overline{a}\}$ is NOT a model.

Herbrand interpretations and models can be represented uniquely by set of atoms:

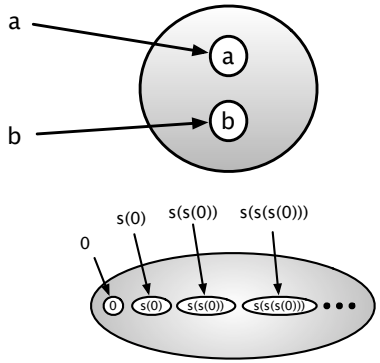
1. $\{p(a), q(b), r(a), r(b)\}$ (model)
2. $\{p(a), p(b), q(b), r(a)\}$ (not a model)

A LATTICE OF INTERPRETATIONS

- Given a program P, the corresponding Herbrand Universe H_P is determined uniquely

```
p(a).        q(b).
r(X) ← p(X).

nat(0).
nat(s(X)) ← nat(X).
```



- Let $B_P = \{p(t_1, \ldots, t_n) : p$ is a predicate symbol in P, t_is are ground terms made with constant and function symbols in $P\}$ B_P is called the Herbrand base.
- Any subset of B_P uniquely determines an Herbrand Interpretation (some of them can be models)
- $(\wp(B_P), \subseteq)$ forms a complete lattice

The fundamental Theorem(s)

Theorem

Let P be a set of clauses. Then P admits a model if and only if it admits a Herbrand model.

Theorem

Let P be a (definite clause) program. Then P admits a (unique) minimum Herbrand model M_P (M_P is the semantics of P, it is also the intersection of all Herbrand models).
(i.e., if I is a Herbrand model of P, then $M_P \subseteq I$).

Theorem

Let P be a (definite clause) program and $A \in B_P$ be a ground atom. Then $P \models A$ if and only if $A \in M_P$.

Computing M_P

1. Top-Down: using SLD resolution (Prolog).
   Query the SLD interpreter with the goal $:- A$.
   Use the following (meta) rule untile the goal becomes empty.

$$\frac{:- A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m \qquad\qquad H :- B_1, \ldots, B_n}{:- (A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_m)\theta}$$

   where $H :- B_1, \ldots, B_n$ is the renaming of a clause in P and θ is the most general unifier of A_i and H. Non-determinism and failures are handled via backtracking. $i = 1$ in actual implementations. "Derivations" leading to an empty goal are searched.
2. Bottom-Up: using the T_P (immediate consequence) operator (Datalog/ASP).

$$T_P(I) = \{a : a \leftarrow b_1, \ldots, b_n \in \text{ground}(P), \{b_1, \ldots, b_n\} \subseteq I\}$$

EXAMPLE

Let P be the program:

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

We can produce the successful SLD derivation

$$\dfrac{\dfrac{\dfrac{:-q(a). \qquad q(X_1) :- r(X_1), p(X_1). \qquad \theta = [X_1/a]}{:-r(a),p(a). \qquad\qquad r(a).}}{:-p(a). \qquad\qquad p(a).}}{\square}$$

Example

Let P be the program:

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

Then:

$$
\begin{aligned}
T_P(\emptyset) &= \{r(a), r(b), p(a)\} \\
T_P(\{r(a), r(b), p(a)\}) &= \{q(a), r(a), r(b), p(a)\} \\
T_P(\{q(a), r(a), r(b), p(a)\}) &= \{q(a), r(a), r(b), p(a)\} \quad \Leftarrow \text{ Fixpoint!}
\end{aligned}
$$

EXAMPLE

Let P be the program:

```
p(a).
q(X) :- q(X).
```

SLD might loop

$$\frac{:- q(a). \qquad q(X_1) :- q(X_1). \qquad \theta = [X_1/a]}{:- q(a). \qquad q(X_2) :- q(X_2). \qquad \theta = [X_2/a]}$$
$$\frac{:- q(a). \qquad q(X_3) :- q(X_3). \qquad \theta = [X_3/a]}{\ldots}$$

Example

Let P be the program:

```
p(a).
q(X) :- q(X).
```

Then:

$$\begin{aligned} T_P(\emptyset) &= \{p(a)\} \\ T_P(\{p(a)\}) &= \{p(a)\} \Leftarrow \text{Fixpoint!} \end{aligned}$$

EXAMPLE

Let P be the program:

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

Then:

$$\frac{:- nat(s(s(s(0)))). \qquad nat(s(X_1)) :- nat(X_1). \qquad \theta = [X_1/s(s(0))]}{:- nat(s(s(0))). \qquad nat(s(X_2)) :- nat(X_2). \qquad \theta = [X_1/s(0)]}$$

$$\frac{:- nat(s(0)). \qquad nat(s(X_3)) :- nat(X_3). \qquad \theta = [X_3/0]}{:- nat(0). \qquad nat(0).}$$

$$\frac{}{\square}$$

Example

Let P be the program:

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

Then:

$$
\begin{array}{rcl}
T_P(\emptyset) & = & \{nat(0)\} \\
T_P(\{nat(0)\}) & = & \{nat(0), nat(s(0))\} \\
T_P(\{nat(0), nat(s(0))\}) & = & \{nat(0), nat(s(0)), nat(s(s(0)))\} \\
\vdots & \vdots & \vdots \\
T_P(\{nat(0), nat(s(0)), nat(s(s(0))), \ldots\}) & = & \{nat(0), nat(s(0)), nat(s(s(0))), \ldots\} \\
 & \Uparrow & \text{Fixpoint!}
\end{array}
$$

Basically, it might loop.

T_P is monotone: $I \subseteq J \rightarrow T_P(I) \subseteq T_P(J)$ and
upward continuous: if $I_0 \subseteq I_1 \subseteq I_2 \cdots$ then $T_P(\bigcup_{i \geq 0} I_i) = \bigcup_{i \geq 0} T_P(I_i)$

Let us define

$$\begin{aligned} T_P \uparrow 0 &= \emptyset \\ T_P \uparrow n+1 &= T_P(T_P \uparrow n) \\ T_P \uparrow \omega &= \bigcup_{n \geq 0} T_P \uparrow n \end{aligned}$$

THEOREM

If P is a definite clause program, then $T_P \uparrow \omega = M_P = T_P(T_P \uparrow \omega)$.

THEOREM

If P is a definite clause program, then M_P is the set of ground atoms that admit a successful SLD derivation.

Computing M_P

Equivalence of the three semantics

T_P is monotone: $I \subseteq J \rightarrow T_P(I) \subseteq T_P(J)$ and
upward continuous: if $I_0 \subseteq I_1 \subseteq I_2 \cdots$ then $T_P(\bigcup_{i \geq 0} I_i) = \bigcup_{i \geq 0} T_P(I_i)$

Let us define

$$\begin{aligned} T_P \uparrow 0 &= \emptyset \\ T_P \uparrow n+1 &= T_P(T_P \uparrow n) \\ T_P \uparrow \omega &= \bigcup_{n \geq 0} T_P \uparrow n \end{aligned}$$

Theorem

If P is a definite clause program, then $T_P \uparrow \omega = M_P = T_P(T_P \uparrow \omega)$.

Theorem

If P is a definite clause program, then M_P is the set of ground atoms that admit a successful SLD derivation.

SUMMARY

- The semantics of definite clause logic programming is based on the minimum Herbrand model M_P. It is the set of logical consequences. It is computable, i.e. it is a recursively enumerable set. You can compute it top down by SLD resolution (as in Prolog) or bottom up by $T_P \uparrow \omega$ (the least fixpoint). It is recursive (PTIME) if there are not function symbols in P. [J.W. Lloyd: Foundations of Logic Programming; K.R. Apt: From Logic Programming to Prolog]
- Focusing on definite clause logic programming one can be interested in the greatest fixpoint of T_P for coinductive reasoning (Coinductive Logic Programming). This set is not computable (it is a productive set) but can be under approximated. [D. Ancona, A. Dovier: A theoretical perspective of Coinductive Logic Programming. 2015]

Negation

- Allowing negation in Body introduces a number of issues

$$H \leftarrow B_1, \ldots, B_m, \texttt{not } C_1, \ldots, \texttt{not } C_n$$

- T_P is no longer monononic: a minimum model is not ensured to exist
- The notion of Stable Model (Gelfond-Lifschitz) is introduced.
- Semantics is bottom-up (finiteness restrictions are imposed)
- A new dialect of logic programming called Answer Set Programming (ASP), emerged for Knowledge Representation (where non monotonicity is needed)

See you next week