La programmazione logica come metodologia per la codifica e la risoluzione di rompicapi

Agostino Dovier

Dipartimento di Matematica e Informatica
Università di Udine

13 Novembre 2015

Un esempio di commonsense reasoning

Attività lavorativa dottorandi

- D.M. 45/2013 art. 12: *L'ammissione al dottorato comporta un impegno esclusivo e a tempo pieno [. . .]*

Un esempio di commonsense reasoning

Attività lavorativa dottorandi

- D.M. 45/2013 art. 12: *L'ammissione al dottorato comporta un impegno esclusivo e a tempo pieno [. . .]*

- NOTA MIUR del 24/03/2014. *l'impegno esclusivo a tempo pieno del dottorando va disciplinato nell'ambito del regolamento di dottorato di Ateneo, atteso che compete al collegio dei docenti autorizzare il dottorando a svolgere attività [. . .] verificandone la compatibilità con il proficuo svolgimento delle attività formative (didattiche e di ricerca) relative al corso.*

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*.

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

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) $\mathtt{ar}(p) \geq 0$
  (e.g., $\mathtt{ar}(\mathtt{leq}) = 2, \mathtt{ar}(\mathtt{father}) = 2, \mathtt{ar}(\mathtt{integer}) = 1$).

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.

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.

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

$$H \leftarrow B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n \tag{1}$$

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

Rules

$$H \leftarrow B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n \tag{1}$$

where $H, B_1, \ldots, B_n$ are atoms, $n \geq 0$, $m \geq 0$ is said an (ASP) rule. The comma "," stands for $\wedge$ (and). The arrow "$\leftarrow$" is written ":-"
Terminology:

$$\underbrace{H}_{\textit{head}} \leftarrow \underbrace{B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n}_{\textit{body}}$$

Rules

$$H \leftarrow B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n \qquad (1)$$

where $H, B_1, \ldots, B_n$ are atoms, $n \geq 0$, $m \geq 0$ is said an (ASP) rule. The comma "," stands for $\wedge$ (and). The arrow "$\leftarrow$" is written ":-"
Terminology:

$$\underbrace{H}_{\textit{head}} \leftarrow \underbrace{B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n}_{\textit{body}}$$

If $m = n$ (i.e. `not` does not occur in (1)) the rule is said a *definite rule* (or *definite clause*)

Rules

$$H \leftarrow B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n \qquad (1)$$

where $H, B_1, \ldots, B_n$ are atoms, $n \geq 0$, $m \geq 0$ is said an (ASP) rule.
The comma "," stands for $\wedge$ (and). The arrow "$\leftarrow$" is written ":-"
Terminology:

$$\underbrace{H}_{\textit{head}} \leftarrow \underbrace{B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n}_{\textit{body}}$$

If $m = n$ (i.e. `not` does not occur in (1)) the rule is said a *definite rule* (or *definite clause*)
If $m = n = 0$ the rule is said a *fact*

Rules

$$H \leftarrow B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n \qquad (1)$$

From a logical point of view (1) is equivalent to:

$$H \vee \neg B_1 \vee \cdots \vee \neg B_m \vee B_{m+1} \vee \cdots \vee B_n$$

namely, a disjunction of literals (a.k.a. a clause)

If $m = n$ (i.e. `not` does not occur in (1)) there is exactly one positive literal in the clause.

Constraints

$$\leftarrow B_1, \ldots, B_m, \texttt{not}\ B_{m+1}, \ldots, \texttt{not}\ B_n \qquad (2)$$

where $B_1, \ldots, B_n$ are atoms, $n \geq 0$, $m \geq 0$ is said an (ASP) constraint.

From a logical point of view (2) is equivalent to:

$$\neg B_1 \vee \cdots \vee \neg B_m \vee B_{m+1} \vee \cdots \vee B_n$$

again, a disjunction of literals (a.k.a. a clause)
If $m = n$ (i.e. `not` does not occur in (2)) there are no positive literals.
Horn clauses are those that have at most one.

Programs

A program is simply a set of rules (1) and constraints (2) (order does not matter, in principle).

To start, let us focus on definite clause programs (a.k.a. pure Prolog programs), namely programs made exclusively by rules

$$H \leftarrow B_1, \ldots, B_m \tag{3}$$

where $H, B_1, \ldots, B_m$ are atoms, $m \geq 0$.

Programs

Databases

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.

Programs

Databases

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`"?

Programs

Databases

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.

Programs

Databases

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.

Programs

Databases

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:

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

view 2:

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

Programs

Databases

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:

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

view 2:

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

Luckily, they are equivalent (exercise!)

Programs

Databases

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).
```

Programs

Databases

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).

Programs

Databases

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?

Programs

Databases

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?

Programs

Databases

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.
```

Programs

(Infinite) Arithmetics

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).
```

Programs

(Infinite) Arithmetics

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

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

What are we expecting?

Programs

(Infinite) Arithmetics

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 $\overline{n}$.

Programs

(Infinite) Arithmetics

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?

Programs

(Infinite) Arithmetics

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., $\texttt{sum}(\overline{5}, 0, \overline{5}), \texttt{sum}(\overline{2}, \overline{4}, \overline{6})$

An infinite set of answers

Let's stop for a while

- Infinite sets of answers can be managed (one item per time) by top-down methods (Prolog).

Let's stop for a while

- Infinite sets of answers can be managed (one item per time) by top-down methods (Prolog).
- These methods work very badly when negation is required

Let's stop for a while

- Infinite sets of answers can be managed (one item per time) by top-down methods (Prolog).
- These methods work very badly when negation is required
- For definite clause programs we can use Prolog (a programming language with top-down *solver*) and deal with infiniteness — sometimes 

Let's stop for a while

- Infinite sets of answers can be managed (one item per time) by top-down methods (Prolog).
- These methods work very badly when negation is required
- For definite clause programs we can use Prolog (a programming language with top-down *solver*) and deal with infiniteness — sometimes 
- For KR/programs with negation we add a finiteness restriction and use ASP (a modeling language with bottom-up *solver*)

Universes and Interpretations

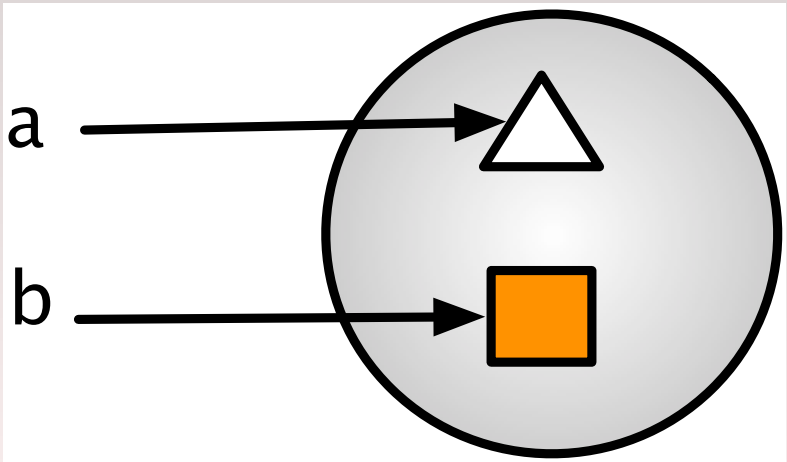
```
p(a).
p(b).
```

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

Universes and Interpretations

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

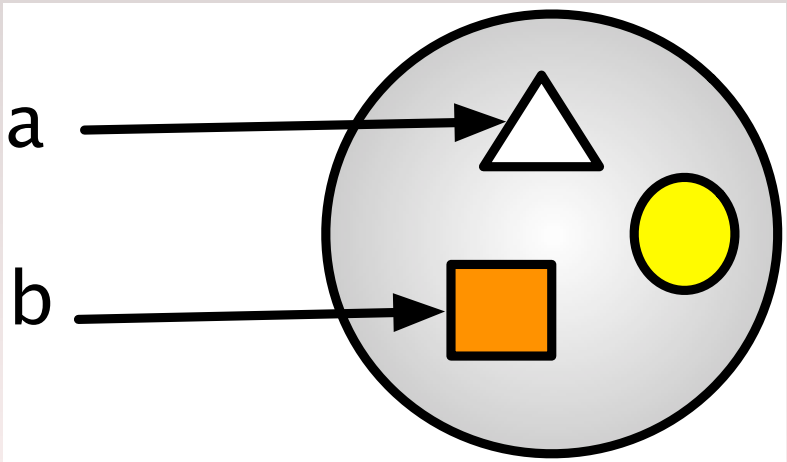
A program (or in general, a first-order theory) *P* is a built from a list of symbols.
In this case constants *a* and *b*, and one unary predicate symbol *p*
We would like to assign a meaning (interpretation) to these symbols on a universe of objects.



Universes and Interpretations

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

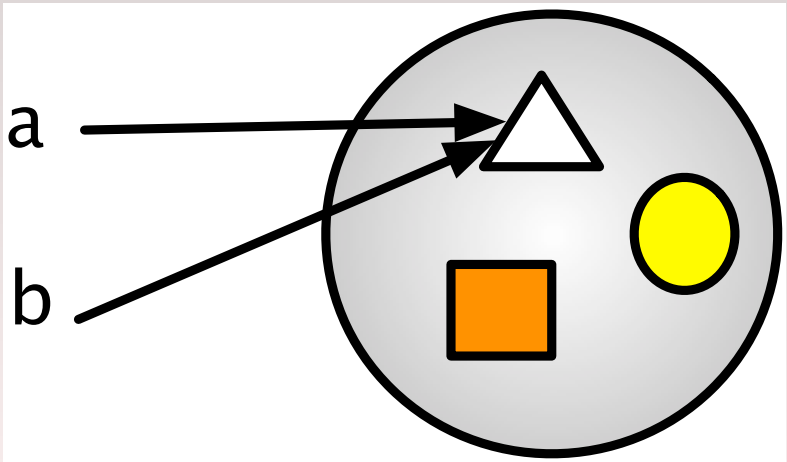
A program (or in general, a first-order theory) *P* is a built from a list of symbols.
In this case constants *a* and *b*, and one unary predicate symbol *p*
We would like to assign a meaning (interpretation) to these symbols on a universe of objects.



Universes and Interpretations

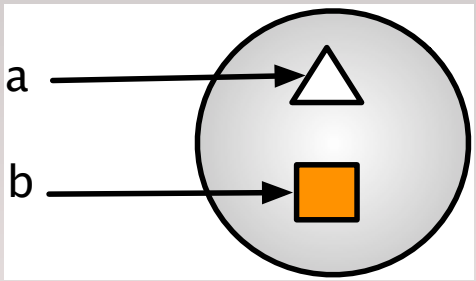
```
p(a).
p(b).
```

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.

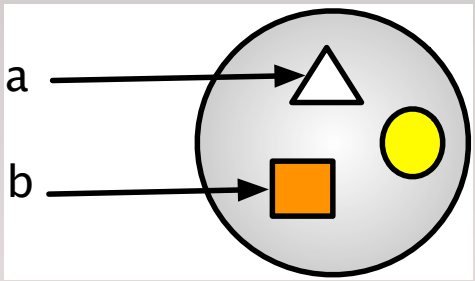


Universes and Interpretations

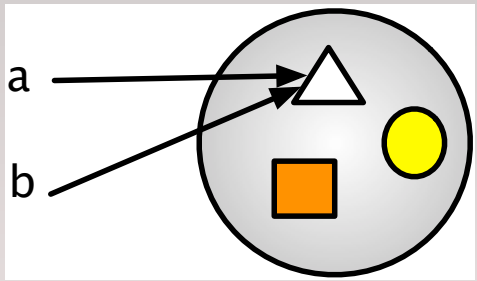
Different interpretations for constant symbols induce different interpretations for first-order formulas (with equality)



$a \neq b$



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



$a = b$

Universes and Interpretations

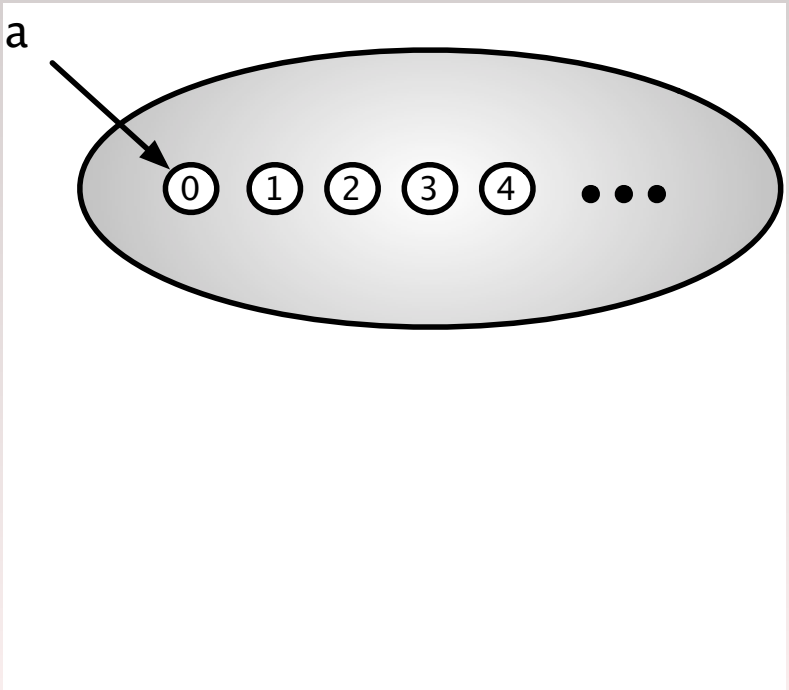
```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

Constant a and function symbols $s/1$ and $g/2$. Functions symbols must be interpreted as functions ($f(x, y) = z$ means $(x, y, z) \in F$)

Universes and Interpretations

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

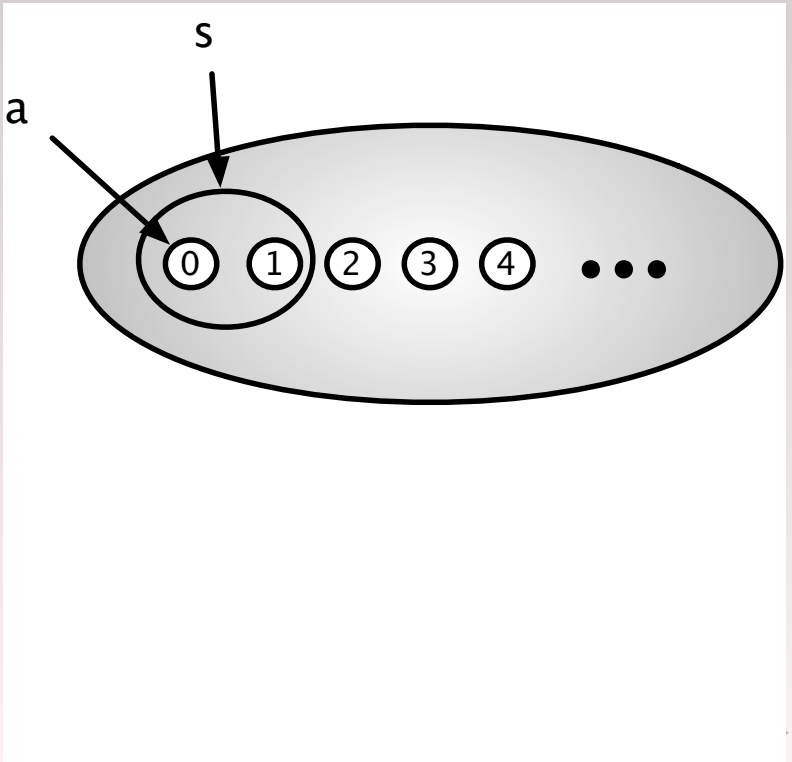
Constant a and function symbols $s/1$ and $g/2$. Functions symbols must be interpreted as functions ($f(x, y) = z$ means $(x, y, z) \in F$)



Universes and Interpretations

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

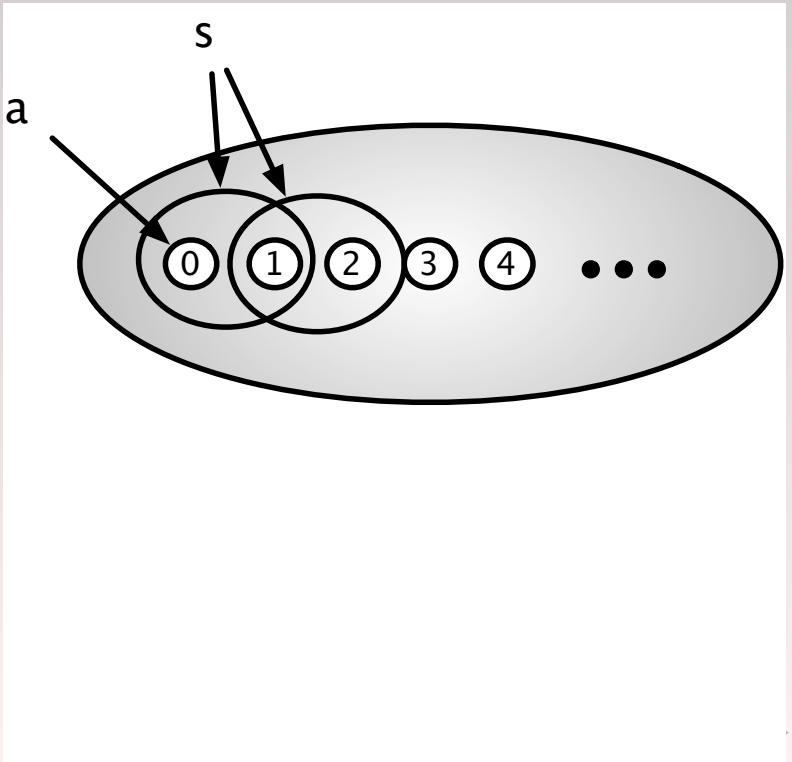
Constant a and function symbols $s/1$ and $g/2$. Functions symbols must be interpreted as functions ($f(x, y) = z$ means $(x, y, z) \in F$)



Universes and Interpretations

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

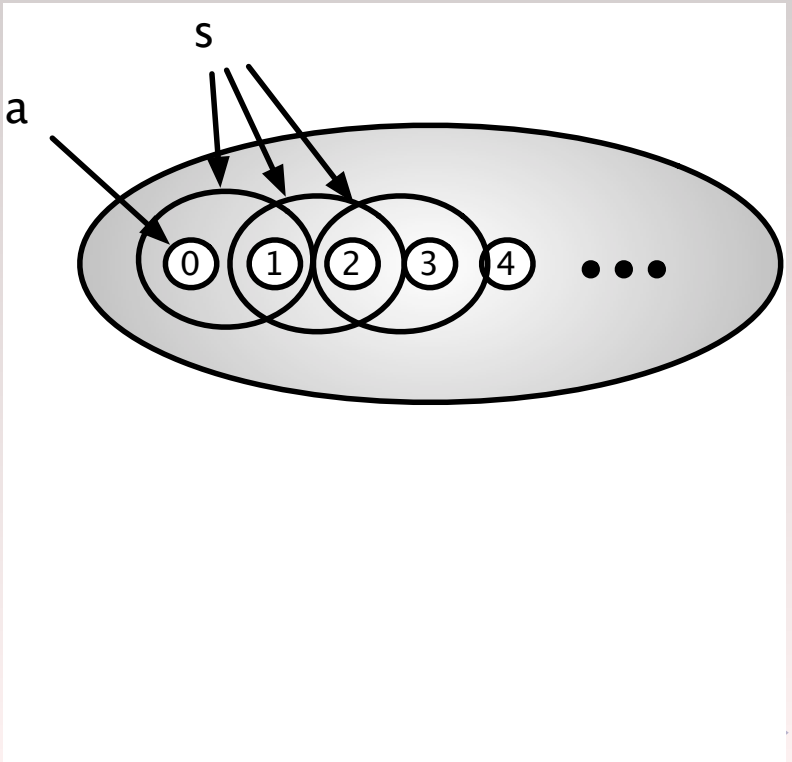
Constant a and function symbols $s/1$ and $g/2$. Functions symbols must be interpreted as functions ($f(x, y) = z$ means $(x, y, z) \in F$)



Universes and Interpretations

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

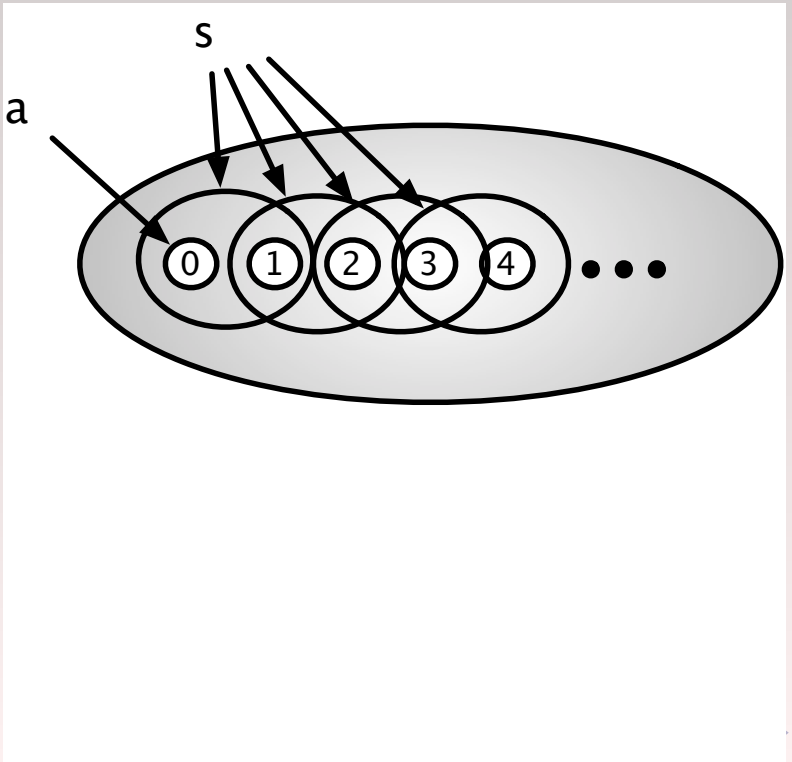
Constant a and function symbols $s/1$ and $g/2$. Functions symbols must be interpreted as functions ($f(x, y) = z$ means $(x, y, z) \in F$)



Universes and Interpretations

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

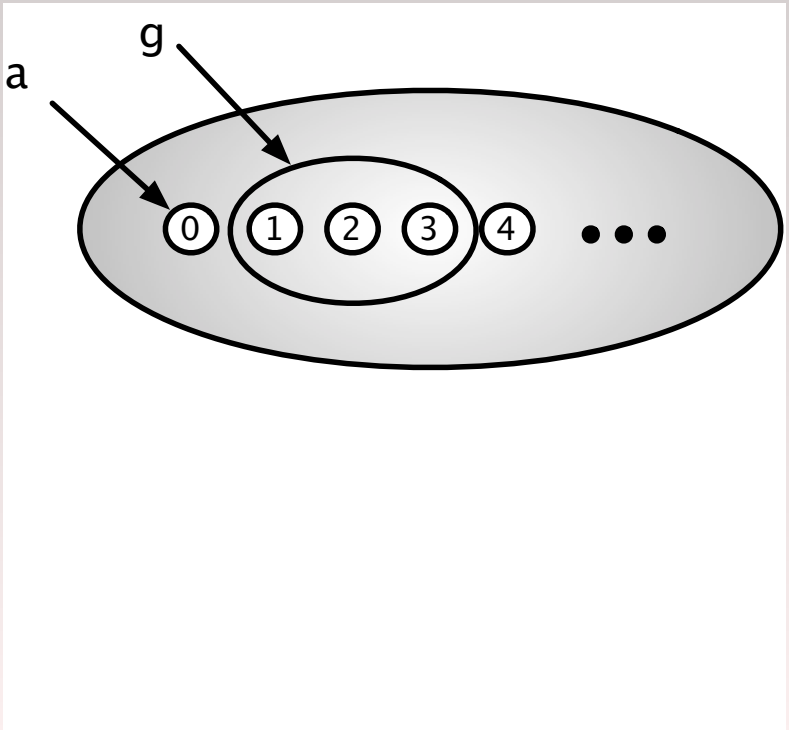
Constant a and function symbols $s/1$ and $g/2$. Functions symbols must be interpreted as functions ($f(x, y) = z$ means $(x, y, z) \in F$)



Universes and Interpretations

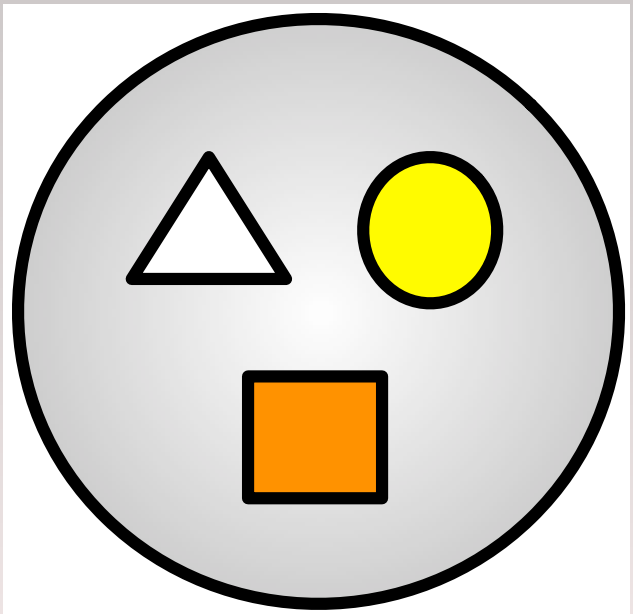
```
p(a).
p(s(X)) ← p(X).
q(g(X,Y)) ← p(X),p(Y).
```

Constant a and function symbols $s/1$ and $g/2$. Functions symbols must be interpreted as functions ($f(x, y) = z$ means $(x, y, z) \in F$)



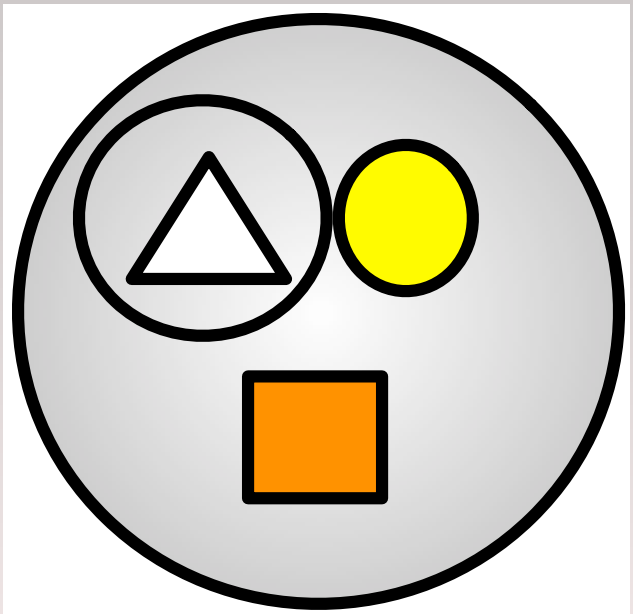
Universes and Interpretations

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



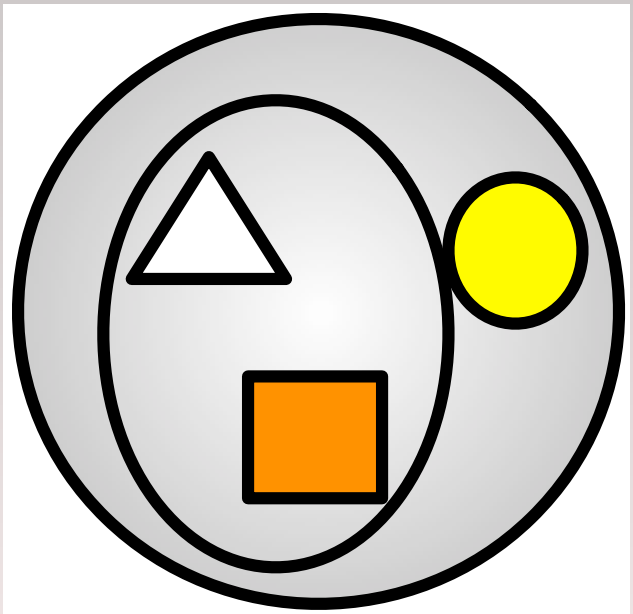
Universes and Interpretations

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



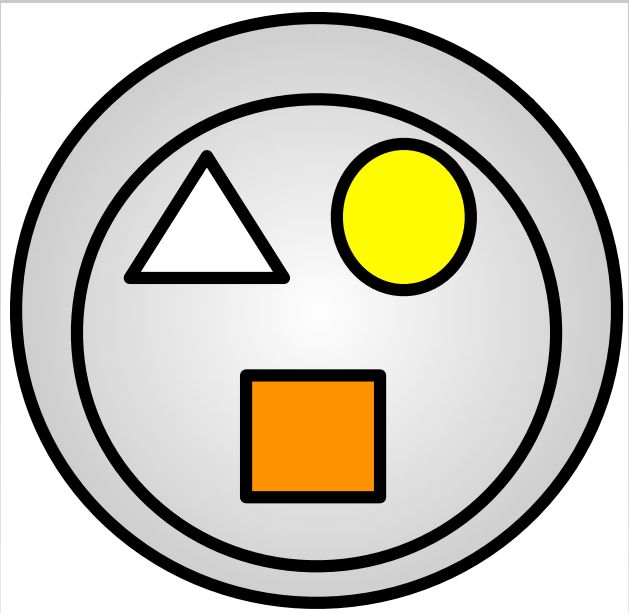
Universes and Interpretations

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



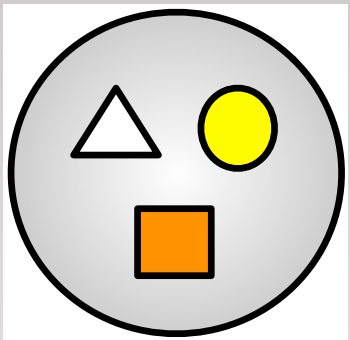
Universes and Interpretations

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

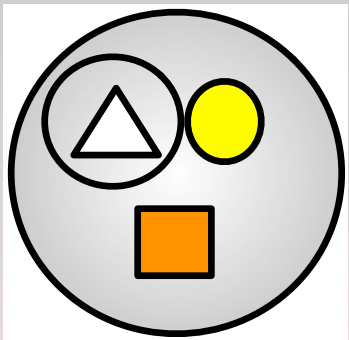


Universes and Interpretations

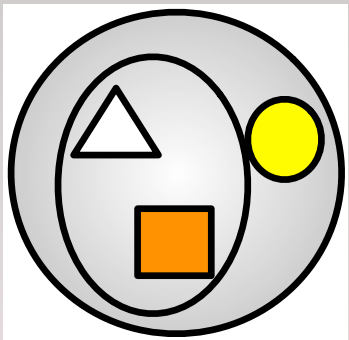
Different interpretations for predicate symbols induce different interpretations for first-order formulas (with equality)



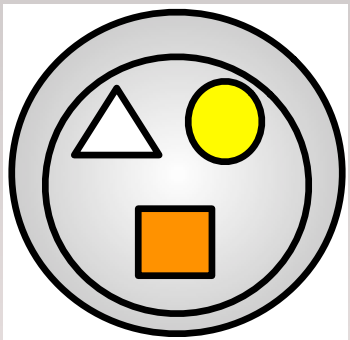
$\forall X\, \neg p(X)$



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



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

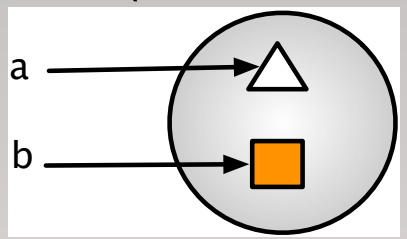


$\forall X\, p(X)$

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).
```

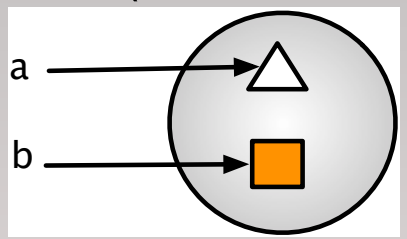


Let us denote $\overline{a}$ =triangle and $\overline{b}$ =square. Let use denote with P, Q, R the interpretations of the predicate symbols p, q, r.

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.

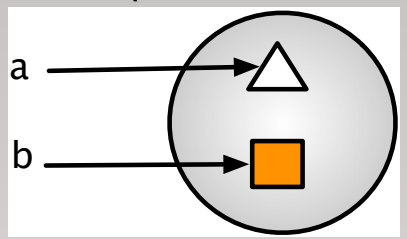
$P = \{\overline{a}\}, Q = \{\overline{b}\}, R = \{\overline{a}, \overline{b}\}$ is a model.

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

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).
```

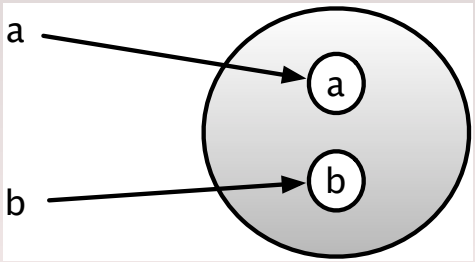


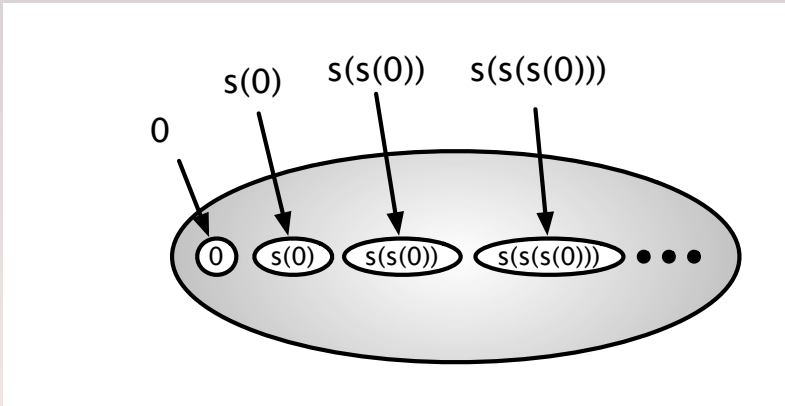
Let us denote $\overline{a}$ =triangle and $\overline{b}$ =square. Let use denote with P, Q, R the interpretations of the predicate symbols p, q, r.
An interpretation that satisfies the logical meaning of all the formulas of P is a model.
$P = \{\overline{a}\}, Q = \{\overline{b}\}, R = \{\overline{a}, \overline{b}\}$ 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 $P \models p(t_1, \ldots, t_n)$.

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

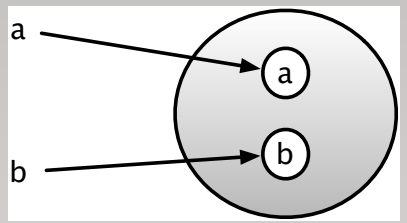




Herbrand Models

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

```
p(a).
q(b).
r(X) ← p(X).
```

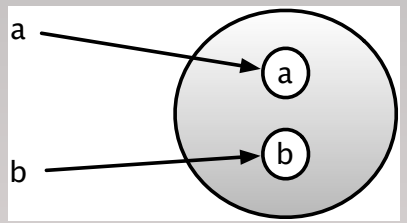


Now, $\overline{a} = a$ and $\overline{b} = b$. Let use denote with P, Q, R the interpretations of the predicate symbols p, q, r.

Herbrand Models

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

```
p(a).
q(b).
r(X) ← p(X).
```



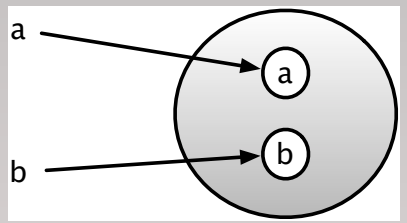
Now, $\overline{a} = a$ and $\overline{b} = b$. Let use denote with P, Q, R the interpretations of the predicate symbols p, q, r.

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 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 use 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)