AUTOMATED REASONING

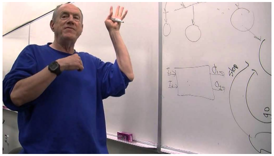
Agostino Dovier

Università di Udine
CLPLAB

Udine, November 2016

Where Logic Programming came from?

- Logic Programming was born circa 1972, 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.



WHERE LOGIC PROGRAMMING CAME FROM?

- It blossomed from Alan Robinson's (left) seminal contribution, 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 LOGIC PROGRAMMING CAME FROM?

DEDUCTIVE REASONING AND FORMAL LOGIC

Deductive reasoning argues from the general to a specific instance. The basic idea is that if something is true of a class of things in general, this truth applies to all legitimate members of that class.

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

(syllogism by Aristotele)

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.

$$\frac{\forall X\,(\mathrm{human}(X) \rightarrow \mathrm{mortal}(X)) \qquad \mathrm{human}(\mathrm{Socrates})}{\mathrm{mortal}(\mathrm{Socrates})}$$

Where Logic Programming came from?

Deductive Reasoning and Formal Logic

Deductive reasoning argues from the general to a specific instance. The basic idea is that if something is true of a class of things in general, this truth applies to all legitimate members of that class.

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

(syllogism by Aristotele)

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.

$$\frac{\forall X\,(\mathrm{human}(X) \rightarrow \mathrm{mortal}(X)) \qquad \mathrm{human}(\mathrm{Socrates})}{\mathrm{mortal}(\mathrm{Socrates})}$$

Where Logic Programming came 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.

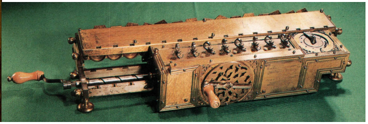
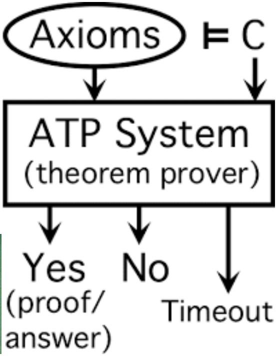
However, it is also concerned with decidability and complexity issues.

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

WHERE LOGIC PROGRAMMING CAME FROM?

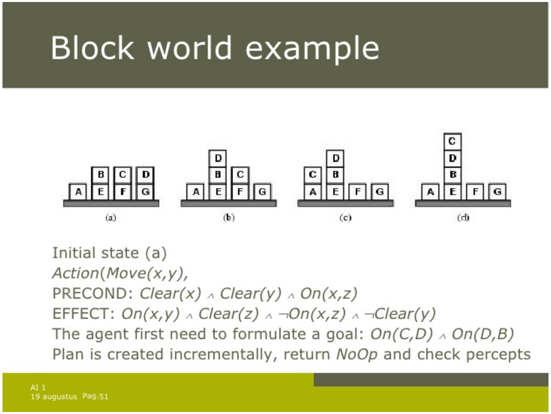
THEOREM PROVING

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

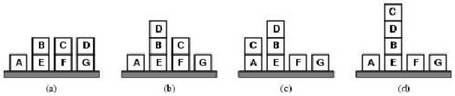



What is Logic Programming?

Planning



Block world example



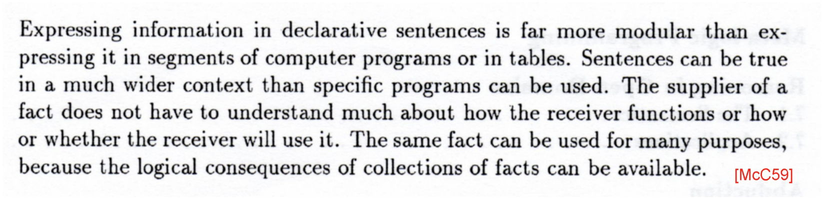
Initial state (a)
Action(Move(x,y),
PRECOND: *Clear(x)* ∧ *Clear(y)* ∧ *On(x,z)*
EFFECT: *On(x,y)* ∧ *Clear(z)* ∧ ¬*On(x,z)* ∧ ¬*Clear(y)*
The agent first need to formulate a goal: *On(C,D)* ∧ *On(D,B)*
Plan is created incrementally, return *NoOp* and check percepts

AI 1
19 augustus Pag.51

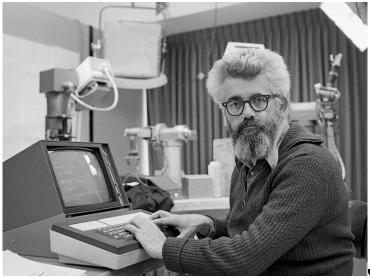
Planner system by Hewitt (1969)

Where Logic Programming came 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 Logic Programming came from?

Programming Languages (late sixities, early seventies)

- Predicate/Function definition in imperative languages
  $\langle$ HEAD $\rangle$ $\langle$ BODY $\rangle$
- Niklaus Wirth: Program = Algorithm + Data Structure

- Predicate Logic as a Programming Language
  $\langle$ HEAD $\rangle \leftarrow \langle$ 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

What is Logic Programming?

Prolog toy example: Database (from Mathematics Genealogy Project)

```
father(alonzo,martin).        father(alonzo,alan).
father(martin,alberto).       father(martin,eugenio).
father(alberto,agostino).     father(alberto,carla).
father(agostino,alessandro).father(agostino,luca).

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

This simple example shows the power of the bi-directionality of predicate definitions.

- `ancestor(alonzo,alessandro).`
- `ancestor(X,alessandro).`
- `ancestor(alonzo,Y).`
- `ancestor(X,Y).`

WHAT IS LOGIC PROGRAMMING?

PROLOG TOY EXAMPLE: DATABASE (FROM MATHEMATICS GENEALOGY PROJECT)

```
father(alonzo,martin).          father(alonzo,alan).
father(martin,alberto).         father(martin,eugenio).
father(alberto,agostino).       father(alberto,carla).
father(agostino,alessandro).father(agostino,luca).

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

This simple example shows the power of the bi-directionality of predicate definitions.

- `ancestor(alonzo,alessandro).`
- `ancestor(X,alessandro).`
- `ancestor(alonzo,Y).`
- `ancestor(X,Y).`

What is Logic Programming?

Prolog Toy example: Database (from Mathematics Genealogy Project)

```
father(alonzo,martin).       father(alonzo,alan).
father(martin,alberto).      father(martin,eugenio).
father(alberto,agostino).    father(alberto,carla).
father(agostino,alessandro).father(agostino,luca).

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

This simple example shows the power of the bi-directionality of predicate definitions.

- `ancestor(alonzo,alessandro).`
- `ancestor(X,alessandro).`
- `ancestor(alonzo,Y).`
- `ancestor(X,Y).`

What is Logic Programming?

Prolog Toy example: Database (from Mathematics Genealogy Project)

```
father(alonzo,martin).        father(alonzo,alan).
father(martin,alberto).       father(martin,eugenio).
father(alberto,agostino).     father(alberto,carla).
father(agostino,alessandro).father(agostino,luca).

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

This simple example shows the power of the bi-directionality of predicate definitions.

- `ancestor(alonzo,alessandro).`
- `ancestor(X,alessandro).`
- `ancestor(alonzo,Y).`
- `ancestor(X,Y).`

WHAT IS LOGIC PROGRAMMING?

PROLOG TOY EXAMPLE: DATABASE (FROM MATHEMATICS GENEALOGY PROJECT)

```
father(alonzo,martin).        father(alonzo,alan).
father(martin,alberto).       father(martin,eugenio).
father(alberto,agostino).     father(alberto,carla).
father(agostino,alessandro).father(agostino,luca).

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

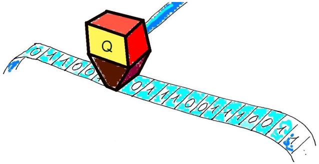
This simple example shows the power of the bi-directionality of predicate definitions.

- `ancestor(alonzo,alessandro).`
- `ancestor(X,alessandro).`
- `ancestor(alonzo,Y).`
- `ancestor(X,Y).`

What is Logic Programming?

Toy examples but not toy language

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?

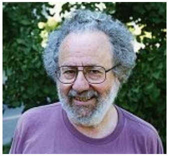
Toy examples but not toy language

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?

TOY EXAMPLES BUT NOT TOY LANGUAGE

Adding negation LP can be used for *Knowledge representation* and *non monotonic reasoning*.

```
flies(X) :- bird(X), not abnormal_bird(X).
abnormal_bird(Y) :- penguin(Y).
abnormal_bird(Y) :- roadrunner(Y).
bird(Z) :- penguin(Z).
bird(tweety).                    penguin(pingu).
```

Semantics become complex: Stable model semantics (Gelfond-Lifschitz) is the answer and it is NP-computable in programs that do not use arbitrary function symbols.

WHAT IS LOGIC PROGRAMMING?

TOY EXAMPLES BUT NOT TOY LANGUAGE

Adding negation LP can be used for *Knowledge representation* and *non monotonic reasoning*.

```
flies(X) :- bird(X), not abnormal_bird(X).
abnormal_bird(Y) :- penguin(Y).
abnormal_bird(Y) :- roadrunner(Y).
bird(Z) :- penguin(Z).
bird(tweety).                penguin(pingu).
```

Semantics become complex: Stable model semantics (Gelfond-Lifschitz) is the answer and it is NP-computable in programs that do not use arbitrary function symbols.

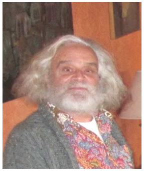
What is Logic Programming?

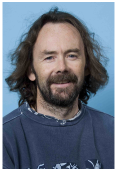
Toy examples but not toy language

Constraint Logic Programming:

		1						
		2		3				4
			5			6		7
5			1	4				
	7						2	
				7	8			9
8		7			9			
4				6		3		
						5		

$$
\begin{array}{l}
\texttt{sudoku}([X_{11},\dots,X_{99}]) :- \\
\quad \texttt{domain}([X_{11},\dots,X_{99}],1,9), \\
\quad X_{13}=1, X_{23}=2,\dots,X_{97}=5, \\
\quad \texttt{alldifferent}([X_{11},\dots,X_{19}]), \\
\qquad\qquad \vdots \\
\quad \texttt{alldifferent}([X_{11},\dots,X_{91}]), \\
\qquad\qquad \vdots \\
\quad \texttt{alldifferent}([X_{77},\dots,X_{99}]).
\end{array}
$$





What is Logic Programming?

Toy examples but not toy language

- The declarative nature allows extensions such as *constraint logic programming*, *functional logic programming*, ...
- All current Prolog systems have 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 ASP ...

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 june 2017 in Milano), summer/winter schools, workshops, student's grants, PhD theses prizes, . . .
- kind president

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 june 2017 in Milano), summer/winter schools, workshops, student's grants, PhD theses prizes, ...
- kind president

Syntax of Logic Programming

Terms

- Let $\mathcal{C}$ be a set of constant symbols (e.g., $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, $\mathtt{socrate}$, $\mathtt{uomo}$, ...)
- Let $\mathcal{V}$ be a set of variable symbols (e.g., $\mathtt{X}$, $\mathtt{Y}$, $\mathtt{Z}$, $\mathtt{X1}$, $\mathtt{X2}$, ...)
- Let $\mathcal{F}$ be a set of function symbols (e.g., $\mathtt{f}$, $\mathtt{g}$, $\mathtt{h}$, $\mathtt{sqrt}$, $\mathtt{piu}$, $\mathtt{per}$, ...)
- Each symbol $f \in \mathcal{F}$ has its own arity (number of arguments) $\mathtt{ar}(f) > 0$ (e.g., $\mathtt{ar}(\mathtt{sqrt}) = 1, \mathtt{ar}(\mathtt{piu}) = 2$).
- We assume that $\mathtt{ar}(c) = 0$ for $c \in \mathcal{C}$ and $\mathtt{ar}(X) = 0$ 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) $\mathtt{ar}(p) \geq 0$
  (e.g., $\mathtt{ar}(\mathtt{leq}) = 2$, $\mathtt{ar}(\mathtt{father}) = 2$, $\mathtt{ar}(\mathtt{integer}) = 1$).
- If $p \in \mathcal{P}$, $\mathtt{ar}(p) = n$, 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., $\mathtt{integer}(s(s(s(0))))$, $\mathtt{leq}(0, s(s(0)))$, $\mathtt{father}(\mathtt{abramo}, \mathtt{isacco})$, $\mathtt{p}(X, Y, \mathtt{a})$ 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) $\mathtt{ar}(p) \geq 0$
  (e.g., $\mathtt{ar}(\mathtt{leq}) = 2, \mathtt{ar}(\mathtt{father}) = 2, \mathtt{ar}(\mathtt{integer}) = 1$).
- If $p \in \mathcal{P}$, $\mathtt{ar}(p) = n$, 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., $\mathtt{integer}(s(s(s(0))))$, $\mathtt{leq}(0, s(s(0)))$, $\mathtt{father}(\mathtt{abramo}, \mathtt{isacco})$, $\mathtt{p}(X, Y, \mathtt{a})$ 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 $t_1, \dots, t_n$ are terms, then $p(t_1, \dots, t_n)$ 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. $p, q, r, \dots$ 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 ":-"
Terminology:

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

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

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

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:

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

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

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

where $H, B_1, \dots, B_n$ 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:

$$\underbrace{H}_{head} \leftarrow \underbrace{B_1, \dots, B_m, \mathtt{not}\ B_{m+1}, \dots, \mathtt{not}\ B_n}_{body}$$

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

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

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:

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

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

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

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

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

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.

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:

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

view 2:

$$(\forall \mathtt{X})(\forall \mathtt{Y})(\forall \mathtt{Z})\big(\mathtt{parent}(\mathtt{X},\mathtt{Z}) \land \mathtt{parent}(\mathtt{Z},\mathtt{Y}) \to \mathtt{granparent}(\mathtt{X},\mathtt{Y})\big)$$

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).
```

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).

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
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.
```

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?

`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 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?

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

An infinite set of answers

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).
- 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*)
- Let us see some examples of (simple) Prolog programs

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*)
- Let us see some examples of (simple) Prolog programs

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*)
- Let us see some examples of (simple) Prolog programs

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*)
- Let us see some examples of (simple) Prolog programs