Artificial Intelligence

7. Logic Programming in Prolog

Prof. Bojana Dalbelo Bašić
Assoc. Prof. Jan Šnajder

University of Zagreb
Faculty of Electrical Engineering and Computing

Academic Year 2019/2020

 v1.10

Outline

Outline

1. Logic programming and Prolog
2. Inference on Horn clauses
3. Programming in Prolog
4. Non-declarative aspects of Prolog

Logic programming

- **Logic programming**: use of logic inference as a way of programming
- Main idea: define the problem in terms of logic formulae, then let the computer do the problem solving (program execution = inference)
- This is a typical **declarative programming** approach: express the logic of computation, don't bother with the control flow
- We focus on the description of the problem (**declarative**), rather than on how the program is executed (**procedural**)
- However, we still need some flow control mechanism, thus:

Algorithm = Logic + Control

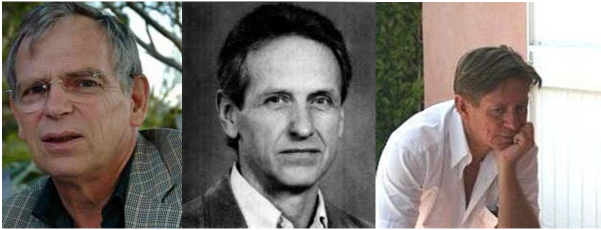
- Different from automated theorem proving because:
  1. explicit control flow is hard-wired into the program
  2. not the full expressivity of FOL is supported

Refresher: Declarative programming

- Describe **what** is being computed instead of **how** (control flow)
- The programmer specifies a set of constraints that define the solution space, while finding the actual solution is left to the interpreter
- Hallmarks of declarative PL:
  - **explicit** rather than implicit state
  - **no side effects** or limited side effects
  - programming with **expressions**
- Two main flavors:
  - **functional PL**: expression is a function
  - **logic PL**: expression is a relation (represented by a predicate)
- **Upsides**: formally concise, high level of abstraction, lend themselves to formal analysis, less error prone
- **Downsides**: inefficiency, steep learning curve, no wide adoption

Prolog

- **Prolog** – "Programming in Logic"
- A declarative programming language
- "The offspring of a successful marriage between natural language processing and automated theorem-proving"
- Alan Colmerauer, Robert Kowalski, and Philippe Roussel in 1972



Colmerauer, A., & Roussel, P. (1996). The Birth of Prolog. In History of programming languages (pp. 331–367)

SWI Prolog

www.swi-prolog.org

SWI Prolog

SWI-Prolog's features

HOME | DOWNLOAD | DOCUMENTATION | TUTORIALS | COMMUNITY | USERS | WIKI

Overview

SWI-Prolog is a versatile implementation of the Prolog language. Although SWI-Prolog gained its popularity primarily in education, its development is mostly driven by the needs for **application development**. This is facilitated by a rich interface to other IT components by supporting many document types and (network) protocols as well as a comprehensive low-level interface to C that is the basis for high-level interfaces to C++, Java (bundled), C#, Python, etc (externally available). Data type extensions such as dicts and strings as well as full support for Unicode and unbounded integers simplify smooth exchange of data with other components.

SWI-Prolog aims at **scalability**. Its robust support for multi-threading exploits multi-core hardware efficiently and simplifies embedding in concurrent applications. Its *Just In Time Indexing* (JITI) provides transparent and efficient support for predicates with millions of clauses.

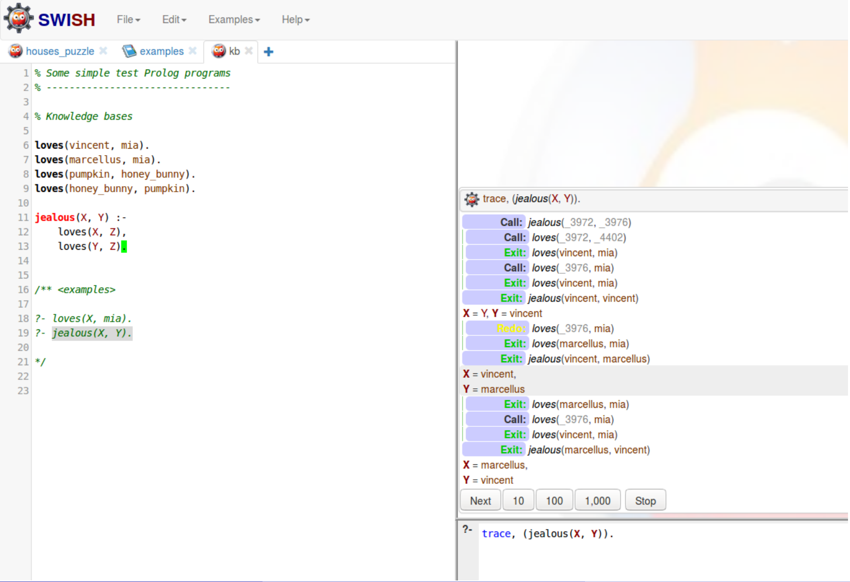
SWI-Prolog **unifies many extensions** of the core language that have been developed in the Prolog community such as *tabling*, *constraints*, *global variables*, *destructive assignment*, *delimited continations* and *interactors*.

SWI-Prolog offers a variety of **development tools**, most of which may be combined at will. The native system provides an editor written in Prolog that is a close clone of Emacs. It provides *semantic* highlighting based on real time analysis of the code by the Prolog system itself. Complementary tools include a graphical debugger, profiler and cross-referencer. Alternatively, there is a mode for GNU-Emacs and, Eclipse plugin called PDT and a VSC plugin, each of which may be combined with the native graphical tools. Finally, a *computational notebook* and web based IDE is provided by SWISH. SWISH is a versatile tool that can be configured and extended to suit many different scenarios.

SWI-Prolog provides an add-on distribution and installation mechanism called **packs**. A *pack* is a directory with minimal organizational conventions and a *control* file that describes the origin, version, dependencies and automatic upgrade support. Packs

SWI Prolog

https://swish.swi-prolog.org/



Outline

Horn clauses (1)

- Prolog uses a subset of FOL called **Horn clause logic**

Horn clause

A **Horn clause** is a clause (a disjunction of literals) with at most one positive literal:

$$\neg P_1 \vee \neg P_2 \vee \cdots \vee \neg P_n \vee Q$$

or, equivalently:

$$(P_1 \wedge P_2 \wedge \cdots \wedge P_n) \rightarrow Q$$

The negative literals constitute the **body** of the clause, while the positive literal constitutes its **head**.

Definite clause

A **definite clause** is a Horn clause with exactly one positive literal.

- A Horn clause can be propositional or first-order.

Horn clauses (2)

- A Prolog program is made up of a sequence of definite clauses
- Each definite clause defines a **rule of inference** or a **fact**
- A fact is simply a definite clause of the form $True \rightarrow Q \equiv Q$
- Inference rules and facts are an intuitive way of formalizing human knowledge

- A Horn clause with only negative literals is called a **goal clause**
- Given the logic program as input, the aim is to prove the goal clause using **refutation resolution**

Horn clauses (3)

- Horn clauses are of **restricted expressivity**: one can not transform every FOL formula into an equivalent Horn clause
  - E.g., $\neg P \rightarrow Q$ or $P \rightarrow (Q \vee R)$
- However, in practice, this turns out not be a severe limitation
- The upside of limiting ourselves to Horn clauses is that we can implement efficient reasoning procedures using **resolution** with either **forward chaining** or **backward chaining**
- Forward/backward chaining over Horn clauses is **complete**
- Propositional Horn clauses: time complexity of inference is **linear** in the number of clauses
- First-order Horn clauses: inference is still undecidable, but generally more efficient than in unrestricted FOL

Backward chaining

- Starting with a knowledge base of facts and rules (the logic program) Γ and the negated goal $\neg P$, we aim to derive a NIL clause

$$\neg P \text{ resolves with } C_1 \in \Gamma \text{ and generates new negated goal } \neg P_2$$
$$\neg P_2 \text{ resolves with } C_2 \in \Gamma \text{ and generates new negated goal } \neg P_3$$
$$\vdots$$
$$\neg P_k \text{ resolves with } C_k \in \Gamma \text{ and generates NIL}$$

- This process can be viewed as a **state space search** where:
  - each state is the negated current goal
  - initial state: $\neg P$
  - goal state: the NIL clause
  - operator: resolving $\neg P_i$ with a clause C_j from Γ
- **NB:** Horn clauses are closed under resolution: the resolvent of two Horn clauses is itself a Horn clause

Backward chaining – Example 1

- Logic program:

$$
\begin{array}{llcl}
(1) & A & \equiv & A \\
(2) & B & \equiv & B \\
(3) & (A \wedge B) \rightarrow C & \equiv & \neg A \vee \neg B \vee C \\
(4) & (C \vee D) \rightarrow E & \equiv & \neg C \vee E \\
(5) & & & \neg D \vee E
\end{array}
$$

- Goal: clause E
- Initial state: $\neg E$
- Step 1: Resolving goal $\neg E$ and clause (4), new goal is $\neg C$
- Step 2: Resolving goal $\neg C$ and clause (3), new goal is $\neg A \vee \neg B$
- Step 3: Resolving goal $\neg A \vee \neg B$ and clause (1), new goal is$\neg B$
- Step 4: Resolving goal $\neg B$ and clause (2), deriving NIL

Backward chaining – Example 2

- Logic program:

$$
\begin{array}{llcl}
(1) & A & \equiv & A \\
(2) & B & \equiv & B \\
(3) & (A \wedge B) \rightarrow D & \equiv & \neg A \vee \neg B \vee D \\
(4) & (C \vee D) \rightarrow E & \equiv & \neg C \vee E \\
(5) & & & \neg D \vee E
\end{array}
$$

- Goal: clause E
- Initial state: $\neg E$
- Step 1: Resolving goal $\neg E$ and clause (4), new goal is $\neg C$
- Step 2: Backtracking to the most recent choice point
- Step 3: Resolving goal $\neg E$ and clause (5), new goal is $\neg D$
- Step 4: Resolving goal $\neg D$ and clause (3), new goal is $\neg A \vee \neg B$
- Step 5: Resolving goal $\neg A \vee \neg B$ and clause (1), new goal is $\neg B$
- Step 6: Resolving goal $\neg B$ and clause (2), deriving NIL

Backward chaining – algorithm

- Non-deterministic algorithm for resolution over Horn clauses:

Backward chaining

function $\mathrm{BackwardChaining}(P, \Gamma)$
  if $P = \mathrm{NIL}$ **then return** true
  $L \leftarrow \mathrm{SelectLiteral}(P)$
  $C \leftarrow \mathrm{SelectResolvingClause}(L, \Gamma)$
  if $C = \mathit{fail}$ **then return** false
  $P' \leftarrow \mathrm{resolve}(P, C)$
  $\mathrm{BackwardChaining}(P', \Gamma)$

- $\mathrm{SelectLiteral}$ selects one literal from the clause P (a negated literal from the negated goal)
- $\mathrm{SelectResolvingClause}$ selects the clause from Γ whose head (which is the only positive literal of a Horn clause) resolves with L
- There can be many such clauses, thus the search branches here

Backward chaining in Prolog

- The clauses in Γ are ordered by the programmer (typically: more specific clauses come first, followed by more general clauses)
- The negative literals in each clause (i.e., the order of atoms in the antecedent) are also ordered by the programmer
- The state space search is carried out in **depth-first order**: when the negated goal P is resolved with clause C, the negative literals in C are placed **in the original order at the beginning** of P
- The $\mathrm{SelectLiteral}$ selects the **first** literal in P. Thus, P is implemented as a **stack** (LIFO)
- This proof strategy is known as **SLD (Selective Linear Definite clause) resolution**

Outline

Prolog facts and rules

$$\forall x\big(\mathrm{HUMAN}(x) \rightarrow \mathrm{MORTAL}(x)\big) \wedge \mathrm{HUMAN}(Socrates)$$

```
mortal(X) :- human(X).   % rule
human(socrates).         % fact
```

- Variables are uppercased, predicate symbols are lowercased
- Implications are in the form `consequent :- antecedent`
- Variables are implicitly universally quantified
- Every line ends with a full stop

Prolog queries

```
?- human(socrates).    % Is Socrates human?
true
?- human(doughnut).    % Is a doughnut human?
false
?- mortal(socrates).   % Inference: is Socrates mortal?
true
?- mortal(X).          % Who is mortal?
X = socrates
true
```

Adding conditions to rules

$$\forall x \big((\mathrm{MAMMAL}(x) \land \mathrm{SPEAKS}(x)) \rightarrow \mathrm{HUMAN}(x) \big)$$

```
human(X) :- mammal(X), speaks(X).  % comma denotes "and"
```

$$\forall x \big((\mathrm{MAMMAL}(x) \land \mathrm{SPEAKS}(x) \land \mathrm{PAYS\_TAXES}(x)) \rightarrow \mathrm{HUMAN}(x) \big)$$

```
human(X) :-
  mammal(X),
  speaks(X),
  pays_taxes(X).
```

Adding disjunctions to rules

$$\forall x\big((\mathrm{HUMAN}(x) \vee \mathrm{ALIVE}(x)) \rightarrow \mathrm{MORTAL}(x)\big)$$

- The disjunction in the rule condition can be written in two ways
- Either the rule is factored into separate clauses:

```
mortal(X) :- human(X).   % first clause
mortal(X) :- alive(X).   % second clause
```

- Or a disjunction is introduced in the rule body:

```
mortal(X) :-
  human(X); alive(X).   % semicolon denotes "or"
```

n-ary predicates

- Binary predicates model binary relations:

```
teacher(socrates, plato).    % Socrates is Plato's teacher
teacher(cratylus, plato).
teacher(plato, aristotle).
```

- A rule for disciple relation as the inverse of teacher relation:

```
disciple(X, Y) :- teacher(Y, X).
```

- A rule for defining that someone is taught by a teacher:

```
taught(X) :- disciple(X, Y).
```

- As we don't care about the value of Y, we can also write:

```
taught(X) :- disciple(X, _).
```

Query examples

```
?- teacher(X, plato).   % Who is Plato's teacher?
X = socrates
X = cratylus
true
?- teacher(socrates, Y), teacher(cratylus, Y).
   % Whom do they both teach?
Y = plato
true
?- taught(X).  % Who is being taught?
X = plato
X = plato
X = aristotle
true
?- disciple(aristotle, socrates).
false
```

Recursively defined predicates

- $\mathrm{DISCIPLE}(x, y)$ captures only the direct relation
- How can we capture transitive relations?
- E.g., $\mathrm{FOLLOWER}(x, y)$, iff x is an either direct or indirect follower of philosopher y:
  - Base case: x is an disciple of y
  - Recursive cause: x is an disciple of z, who in turn is the follower of y

```
follower(X, Y) :-   % base clause
  disciple(X, Y).
follower(X, Y) :-   % recursive clause
  disciple(X, Z),
  follower(Z, Y).
```

```
?- follower(aristotle, socrates).
true
```

Prolog search tree

- Applying the SLD resolution on the logic program and the negated goal generates a search tree which corresponds to a **proof tree**
- Each node is a **stack of negative literals** to be resolved
- The goal is to derive NIL, i.e., empty the stack
- Prolog attempts to resolve the top literal from the stack against the **head literal** of every clause from the program (remember: a literal from the stack is negative, while head literals from clauses are positive)
- Such literals are potentially **complementary unifiable**
- The clauses are searched from **top to bottom** of the program (hence the order of clauses is important)

Prolog search tree

- If the top literal L is complementary unifiable using MGU θ with the head of some clause C, then L gets popped from the stack, the body of C is pushed onto the stack, substitution θ is applied to all literals on the stack, and the search continues
- If no clause from the program is complementary unifiable with the L, the search **backtracks** to the last choice point $\Rightarrow$ **depth-first search**
- If the stack is emptied, this means NIL is derived, hence return `true`
- If the search is complete and the stack is not emptied, return `false`

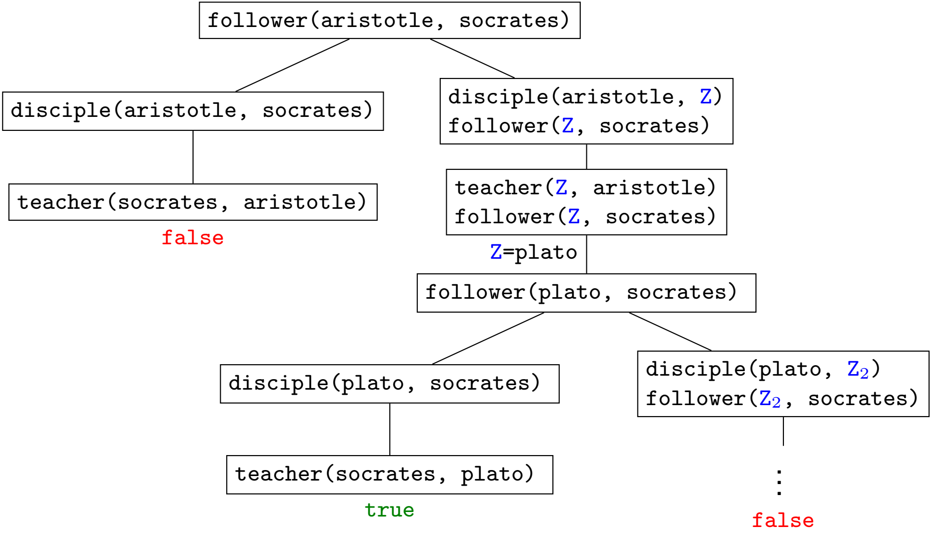
Prolog program – example

```prolog
% Rules
follower(X, Y) :-
  disciple(X, Y).
follower(X, Y) :-
  disciple(X, Z),
  follower(Z, Y).

disciple(X, Y) :-
  teacher(Y, X).

% Facts
teacher(socrates, plato).
teacher(cratylus, plato).
teacher(plato, aristotle).
```

Prolog search tree – example



Prolog search tree – example – remarks

- Each node = downward-growing stack of negative literals
- The search branches on resolving the `follower` literal, because there are two rules for the `follower` predicate
- On request, after proving the goal, Prolog can be made to continue to search for alternative resolutions (which fails in this case)
- Do not confuse the stack of negative literals (constituting the state in the search space) with the DFS stack (used to implement the DFS; not shown on previous slide)

Prolog execution trace

```
[trace] ?- follower(aristotle, socrates).
  Call:  (7) follower(aristotle, socrates) ?  creep
  Call:  (8) disciple(aristotle, socrates) ?  creep
  Call:  (9) teacher(socrates, aristotle) ?  creep
  Fail:  (9) teacher(socrates, aristotle) ?  creep
  Fail:  (8) disciple(aristotle, socrates) ?  creep
  Redo:  (7) follower(aristotle, socrates) ?  creep
  Call:  (8) disciple(aristotle, _G5025) ?  creep
  Call:  (9) teacher(_G5024, aristotle) ?  creep
  Exit:  (9) teacher(plato, aristotle) ?  creep
  Exit:  (8) disciple(aristotle, plato) ?  creep
  Call:  (8) disciple(plato, socrates) ?  creep
  Call:  (9) teacher(socrates, plato) ?  creep
  Exit:  (9) teacher(socrates, plato) ?  creep
  Exit:  (8) disciple(plato, socrates) ?  creep
  Exit:  (7) follower(aristotle, socrates) ?  creep
true.
```

Outline

Order of atoms/clauses

```
follower(X, Y) :-    % base clause
  disciple(X, Y).
follower(X, Y) :-    % recursive clause
  disciple(X, Z),
  follower(Z, Y).
```

- Formally, the order of clauses and their atoms should be arbitrary, due to commutativity of '$\wedge$' and '$\vee$':

$$
\begin{aligned}
& \big(\neg D(x,y) \vee F(x,y)\big) \wedge \big(\neg D(x,z) \vee \neg F(z,y) \vee F(x,y)\big) \\
\equiv\ & \big(\neg D(x,y) \vee F(x,y)\big) \wedge \big(\neg F(z,y) \vee \neg D(x,z) \vee F(x,y)\big) \\
\equiv\ & \big(\neg D(x,z) \vee \neg F(z,y) \vee F(x,y)\big) \wedge \big(\neg D(x,y) \vee F(x,y)\big) \\
\equiv\ & \big(\neg F(z,y) \vee \neg D(x,z) \vee F(x,y)\big) \wedge \big(\neg D(x,y) \vee F(x,y)\big)
\end{aligned}
$$

Order of atoms/clauses

- However, because Prolog uses SLD, commutativity doesn't hold and the order of clauses/atoms becomes important

1:
```
follower(X, Y) :-
  disciple(X, Y).
follower(X, Y) :-
  disciple(X, Z),
  follower(Z, Y).
```

2:
```
follower(X, Y) :-
  disciple(X, Z),
  follower(Z, Y).
follower(X, Y) :-
  disciple(X, Y).
```

3:
```
follower(X, Y) :-
  disciple(X, Y).
follower(X, Y) :-
  follower(Z, Y),
  disciple(X, Z)..
```

4:
```
follower(X, Y) :-
  follower(Z, Y),
  disciple(X, Z).
follower(X, Y) :-
  disciple(X, Y).
```

⇒ **declarative meaning** deviates from **procedural meaning**!

Negation

- A Horn close does not allow for negated atoms in the antecedent, e.g.:

$$(Q(x) \wedge \neg P(x)) \to R(x) \equiv \neg Q(x) \underbrace{\vee P(x) \vee R(x)}_{\text{two pos. literals!}}$$

- Logic programming without negation would be far too restrictive
- Prolog introduces the `not` operator, which can be used in the body of a rule:

```
R(X) :- Q(X), not(P(X)).
```

- The semantics of `not` is different from the one in logic:

Negation as failure – NAF

The literal `not(P(x))` is true if `P(X)` cannot be derived, otherwise it is false.

- In logic: `not(P(x))` is true iff `P(x)` is false

Negation – example

```
human(X) :-
    speaks(X),
    not(has_feathers(X)).

speaks(socrates).
speaks(polynesia).
has_feathers(polynesia).
```

```
?- human(polynesia).
false
?- human(socrates).
true
?- not(human(polynesia)).
true
```

Closed world assumption

- NAF: if `P(x)` can't be derived, then `not(P(x))` true
- Standard semantics: if `not(P(x))` is true, then `P(x)` is false
- Taken together: if `P(x)` can't be derived, then `P(x)` is false
- In other words, all things that can't be derived are false

Closed world assumption – CWA

Everything that cannot be derived (facts that are not in the knowledge base and that cannot be derived from the knowledge base) is false.

- We don't allow for facts to be unknown (neither true nor false)
- CWA causes Prolog to deviate from standard semantics. E.g., in logic:

$$P, (P \land \neg Q) \rightarrow R \not\models R$$

but in Prolog:

$$P, (P \land \neg Q) \rightarrow R \vdash R$$

Wrap-up

- **Logic programming** is a kind of declarative programming, and **Prolog** is a logic programming language
- **Horn clause logic** is a subset of FOL which allows for efficient inference using resolution
- **Horn clauses** are clauses with at most one positive literal, while **definite clauses** are Horn clauses with exactly one positive literal
- A Prolog program is a sequence of definite first-order clauses, which correspond to **facts** and **rules**
- The program is executed by applying **refutation resolution with backward chaining** (SLD resolution)
- Non-commutativity of disjunction/conjunction and **negation as failure** are the non-declarative aspects of Prolog

Next topic: Expert systems