

# Advanced Topics in Data Mining and Logic Programming

**Ricardo Rocha**

**DCC-FCUP, University of Porto**

*ricroc@dcc.fc.up.pt*

## Logic Programming

- Logic programming languages, together with functional programming languages, form a major class of languages called **declarative languages**. A common characteristic of both groups of languages is that they have a strong mathematical basis:
  - Logic programming languages are based on the predicate calculus.
  - Functional programming languages are based on the lambda calculus.
- Declarative languages are considered to be very **high-level languages** when compared with conventional imperative languages because, generally, they allow the programmer to concentrate more on **what the problem is**, leaving much of the details of how to solve the problem to the computer. The programmer can specify the problem at a more application-oriented level and thus simplify the formal reasoning about it.

## Logic Programming

- Logic programming is a programming paradigm based on Horn Clause Logic, a subset of First Order Logic. Logic programming is a simple theorem prover that given a theory (or program) and a query, uses the theory to search for alternative ways to satisfy the query:
  - Variables are **logical** variables which can be instantiated **only once**.
  - Variables are **untyped** until instantiated.
  - Variables are instantiated via **unification**, a pattern matching operation finding the most general common instance of two data objects.
  - At unification failure the execution **backtracks** and tries to find another way to satisfy the original query.

## Logic Programming

- Logic programming is often mentioned to include the following major advantages:
  - **Simple declarative semantics**: a logic program is simply a collection of predicate logic clauses.
  - **Simple procedural semantics**: a logic program can be read as a collection of recursive procedures.
  - **High expressive power**: logic programs can be seen as executable specifications that despite their simple procedural semantics allow for designing complex and efficient algorithms.
  - **Inherent non-determinism**: since in general several clauses can match a goal, problems involving search are easily programmed in these kind of languages.
- These advantages lead to compact code that is easy to understand, program and transform. Furthermore, they make logic programming languages very attractive for the exploitation of implicit parallelism.

## Logic Programs

- A logic program consists of a collection of Horn clauses. Using Prolog's notation, each clause may be a **rule** of the form

$$A \text{ :- } B_1, \dots, B_n.$$

where  $A$  is the **head of the rule** and the  $B_1, \dots, B_n$  are the **body subgoals**, or it may be a **fact** and simply written as

$$A.$$

Rules represent the logical implication

$$B_1 \wedge \dots \wedge B_n \rightarrow A$$

while facts assert  $A$  as true.

- A separate type of clauses is that where the head goal is false. These type of clauses are called **queries** and, in Prolog, they are written as

$$\text{:- } B_1, \dots, B_n.$$

## Logic Programs

- A subgoal is a **predicate** applied to a number of **terms**

$$p(t_1, \dots, t_n)$$

where  $p$  is the predicate name, and the  $t_1, \dots, t_n$  are the terms used as arguments.

- A term can be either a:

- **Variable**
- **Atom**
- **Compound term**

Compound terms have the form  $f(u_1, \dots, u_m)$  where  $f$  is a **functor** and the  $u_1, \dots, u_m$  are themselves terms.

- Terms in a program represent world objects while predicates represent relationships among those objects. Variables represent unspecified terms while atoms represent symbolic constants.

## Logic Programs

- Information from a logic program is retrieved through query execution. The execution of a query  $Q$  against a logic program  $P$ , leads to consecutive **assignments** of terms to the variables of  $Q$  till a **substitution**  $\theta$  satisfied by  $P$  is found.
- Answers (or solutions) for  $Q$  are retrieved by reporting for each variable  $X$  in  $Q$  the corresponding assignment  $\theta(X)$ . When a variable  $X$  is assigned a term  $T$ , then  $X$  is said to be **bound** and  $T$  is called the **binding** of  $X$ . A variable can be bound to another different variable or to a non-variable term.
- Execution of a query  $Q$  with respect to a program  $P$  proceeds by reducing the initial conjunction of subgoals of  $Q$  to subsequent conjunctions of subgoals according to a refutation procedure. The refutation procedure of interest here is called **Selective Linear Definite resolution** or simply **SLD resolution**.

## SLD Resolution

- Let us assume that

$$:- G_1, \dots, G_n.$$

is the current conjunction of subgoals.

- Initially and according to a predefined *select<sub>literal</sub>* rule, a subgoal (or literal)  $G_i$  is selected.
- Assuming that  $G_i$  is the selected subgoal, then the program is searched for a clause whose head goal unifies with  $G_i$ . If there are such clauses then, according to a predefined *select<sub>clause</sub>* rule, one is selected.
- In a computer implementation, the *select<sub>literal</sub>* and *select<sub>clause</sub>* rules must be specified. Different specifications lead to different algorithms and different languages (or semantics) can thus be obtained.



## SLD Resolution

- Consider that

$$A :- B_1, \dots, B_m.$$

is the selected clause that unifies with  $G_i$ . The unification process has determined a substitution  $\theta$  to the variables of  $A$  and  $G_i$  such that  $A\theta = G_i\theta$ .

- Execution proceeds by replacing  $G_i$  with the body subgoals of the selected clause and by applying  $\theta$  to the variables of the resulting conjunction of subgoals:

$$:- (G_1, \dots, G_{i-1}, B_1, \dots, B_m, G_{i+1}, \dots, G_n)\theta.$$

- If the selected clause is a fact,  $G_i$  is simply removed from the conjunction of subgoals:

$$:- (G_1, \dots, G_{i-1}, G_{i+1}, \dots, G_n)\theta.$$

## SLD Resolution

- A sequence of the previous reductions is called an SLG derivation. Finite SLD derivations may be successful or failed.
- A **successful SLD derivation** is found whenever the conjunction of subgoals is reduced to the true subgoal, which therefore corresponds to the determination of a query substitution (answer) satisfied by the program.
- When there are no clauses unifying with a selected subgoal, then a **failed SLD derivation** is found. In Prolog, failed SLD derivations are resolved through applying a **backtracking** mechanism. Backtracking exploits alternative execution paths by:
  - Undoing all the bindings made since the preceding selected subgoal  $G_p$ .
  - Reducing  $G_p$  with the next available clause unifying with it.

The computation stops either when all alternatives have been exploited or when an answer is found.

## The Prolog Language

- Prolog is the most popular logic programming language. The name Prolog was invented in 1973 by Colmerauer and colleagues as an abbreviation for **PRO**gramation en **LOG**ic to refer to a software tool designed to implement a man machine communication system in natural language.
- In 1977, David H. D. Warren made Prolog a viable language by developing the first compiler for Prolog. This helped to attract a wider following to Prolog and made the syntax used in this implementation the *de facto* Prolog standard.
- In 1983, Warren proposed a new abstract machine for executing compiled Prolog code that has come to be known as the **Warren Abstract Machine**, or simply **WAM**. The WAM became the most popular way of implementing Prolog and almost all current Prolog systems are based on WAM's technology.

## The Prolog Language

- The operational semantics of Prolog is based on SLD resolution. Prolog specifies that the *select<sub>literal</sub>* rule chooses the leftmost subgoal in a query and that the *select<sub>clause</sub>* rule follows the textual order of the clauses in the program.

member(Elem, [ Elem | \_ ]).

member(Elem, [ \_ | Tail ]) :- member(Elem, Tail).

0. member(b, [a, b]).



1. member(b, [b]).



2. **yes**

0. member(E, [a, b]).



1. **E = a**

2. member(E, [b]).



3. **E = b**

4. member(E, [ ]).

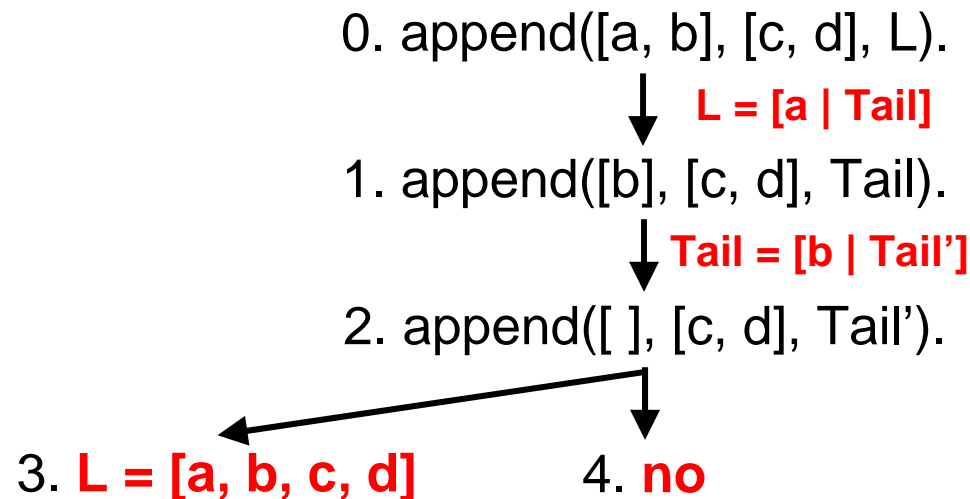


5. **no**

## The Prolog Language

append([ ], List, List).

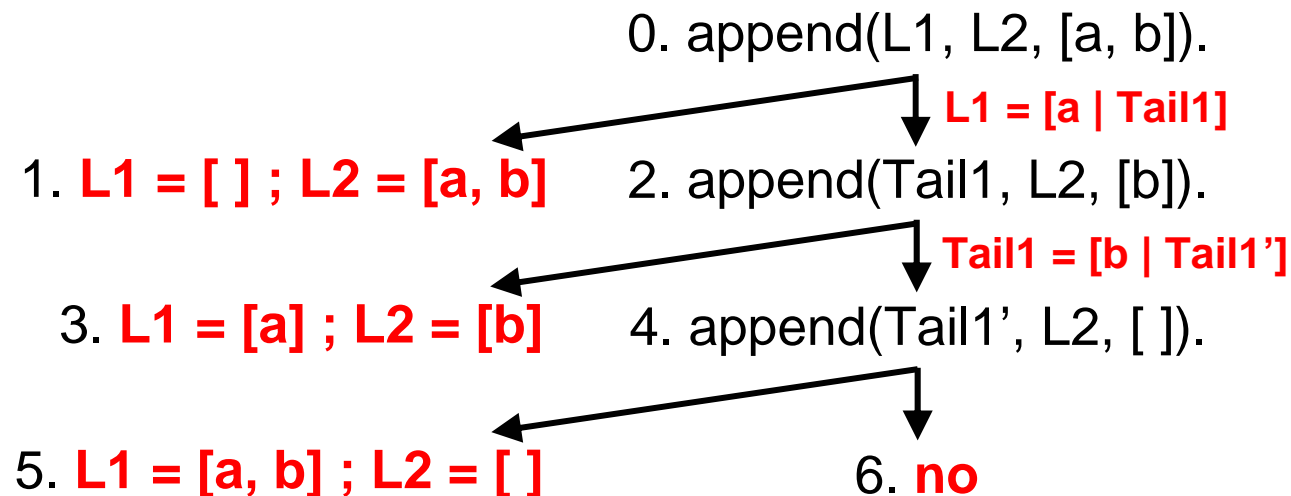
append([ Head | Tail1 ], List2, [ Head | Tail ]) :- append(Tail1, List2, Tail).



## The Prolog Language

append([ ], List, List).

append([ Head | Tail1 ], List2, [ Head | Tail ]) :- append(Tail1, List2, Tail).



## The Prolog Language

reverse([ ], [ ]).

reverse([ Head | Tail ], List) :- reverse(Tail, List1), append(List1, [Head], List).

0. reverse([a, b], L).



1. reverse([b], List1), append(List1, [a], L).



2. reverse([ ], List1'), append(List1', [b], List1), append(List1, [a], L).



3. append([ ], [b], List1), append(List1, [a], L).



...



6. **L = [b, a]**

## The Prolog Language

- To make Prolog a useful programming language for real world problems, some additional features not found within first order logic were introduced. These features include:
  - **Meta-logical predicates:** these predicates inquire the state of the computation and manipulate terms.
  - **Cut predicate:** this predicate adds a limited form of control to the execution. It prunes unexploited alternatives from the computation.
  - **Extra-logical predicates:** these are predicates which have no logical meaning at all. They perform input/output operations and manipulate the Prolog database, by adding or removing clauses from the program being executed.
  - **Other predicates:** these include arithmetic predicates to perform arithmetic operations, term comparison predicates to compare terms, extra control predicates to perform simple control operations, and set predicates that give the complete set of answers for a query.



## The WAM

- The WAM is a stack-based architecture with simple data structures and a low-level instruction set. At any time, the state of a computation is obtained from the contents of the WAM data areas, data structures and registers.
- The WAM defines the following execution stacks:
  - **Code area**: stores the WAM code corresponding to the loaded programs.
  - **Stack**: stores the environment and choice point frames. Environments track the flow of control in a program and choice points store open alternatives. Some WAM implementations use separate execution stacks to store environments and choice points.
  - **Heap**: sometimes also referred as global stack, it is an array of data cells used to store variables and compound terms that cannot be stored in the stack.
  - **Trail**: organized as an array of addresses, it stores the addresses of the (stack or heap) variables which must be reset upon backtracking.
  - **PDL**: a push down list used by the unification process.

## The WAM

- Four main groups of instructions can be enumerated in the WAM instruction set:
  - **Choice point instructions**: these instructions allow to allocate/remove choice points and to recover the state of a computation through the data stored in choice points.
  - **Control instructions**: these instructions allow to allocate/remove environments and to manage the call/return sequence of subgoals.
  - **Unification instructions**: these instructions implement specialized versions of the unification algorithm according to the position and type of the arguments.
  - **Indexing instructions**: these instructions accelerate the process of determining which clauses unify with a given subgoal call. Depending on the first argument of the call, they jump to specialized code that can directly index the unifying clauses.