Prolog (part 1)

Andrea Sterbini – sterbini@di.uniroma1.it

Prolog: logic programming

Created in France by Alain Colmerauer & co. at Marseille, France in the ‘70 for AI and computational linguistics

Declarative style of:

- representing data/relations (facts)
- representing how to solve a problem (rules/clauses)
- representing data structures (built through unification)

Used for:

- AI: natural language parsing and generation, planning, theorem proving, math, symbolic manipulation, …
- meta-programming (programs that create programs)
- …

SWI-Prolog

Implementation for Windows/OsX/Linux at https://swi-prolog.org

- OOP, GUI programming, Web programming, Semantic web …

IDE, editor and Web-based:

- Browser-based interface at https://swish.swi-prolog.org
  - available also as a Docker image (swipl/swish)
- SwiPrologEditor/IDE at Hessen University
- Eclipse plugin (ProDT) at Bonn University

Interactive books to learn Prolog:

- Learn Prolog Now! at ~~http://lpn.swi-prolog.org~~
- Simply Logical at https://book.simply-logical.space

Data types and program elements

Integers 42 Floats 3.14 Strings ”Hello world”

Atoms andrea Lists [one, 2, 3.14, ”four”]

Terms height(andrea, 186) Dicts movie{ director: “Martin… }

Variables are NOT typed, and start with Capital or _underscore

the assignment is UNDONE on backtrack!!!

Facts describe relations that are always true

```
parent( maurizio, andrea ).    % Maurizio is parent of Andrea
```

Predicates/rules/clauses describe conditional relations

```
ancestor( Kid, Grandpa ) :-           % Grandpa is ancestor of Kid IF
    parent( Somebody, Kid ),          % there exists Somebody, parent of Kid
    ancestor( Somebody, Grandpa ).    % that has Grandpa as an ancestor

ancestor(Kid, Parent) :-              % (base case of the recursive ancestor relation)
    parent(Parent, Kid).              % all parents are ancestors of Kid
```

Program execution = query for a proof

A program execution is the response to a query asking the system to find a proof that something (a fact) is true

The system looks for a way to prove your query by searching:

- a fact that directly satisfies your query
- or else for a predicate/rule that would be able to satisfy your query:
    - if the head matches then recursively prove all its preconditions

If more than one ways exists to satisfy a query, all are tried in order (by backtracking/undoing last choice if some of the sub-queries fails)

Facts/clauses are searched in their textual order in the program

Values assigned to the variables to satisfy the query are returned

Declarative style

Facts can be considered as a database of known data

Could be used to teach data normalization

- 1NF: values are atomic/there is a unique key/reduced form
- 2NF: + no partial dependencies (create other tables)
- 3NF: + no transitive dependencies (create other tables)

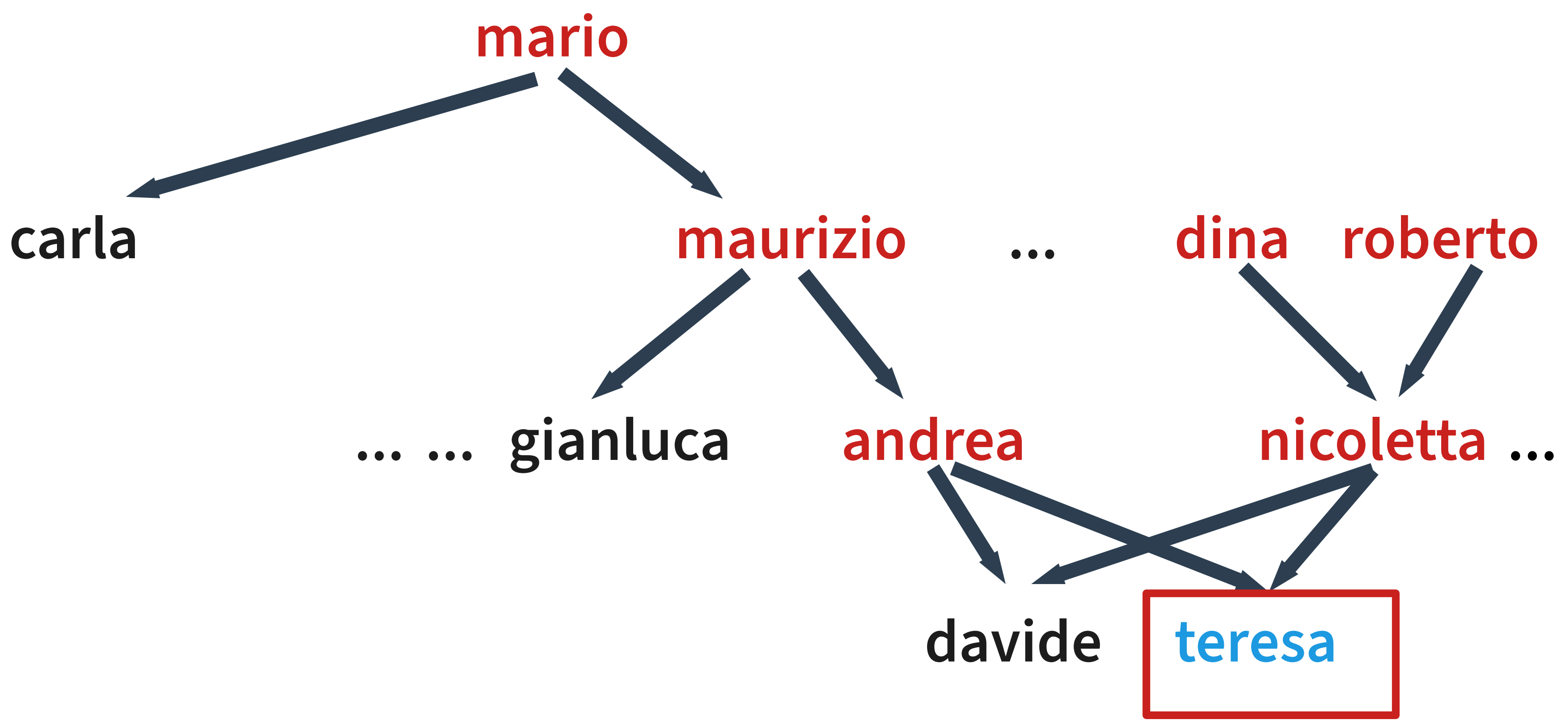
To retrieve a record with simple WITH constraints just QUERY with partial arguments and get variable values filled with found data

To use more complex WITH constraints use rules.

To get table JOINS just AND queries (that must be all true)

Example: a small genealogy problem

From the ‘parent’ relation



To find all teresa’s ancestors we recursively climb the parent relation

Representing facts AND relations (deduction rules)

FACTS

```
parent(mario, maurizio).
parent(mario, carla).
parent(maurizio, andrea).
parent(maurizio, gianluca).
parent(andrea, teresa).
parent(andrea, davide).
parent(dina, nicoletta).
parent(roberto, nicoletta).
parent(nicoletta, teresa).
parent(nicoletta, davide).
```

RULES

```
ancestor( Kid, Parent ) :- % base case
    parent(Parent, Kid).
ancestor( Kid, Grandpa ) :-
    parent( Somebody, Kid ),
    ancestor( Somebody, Grandpa).
```

QUERY (find all ancestors)

```
?- ancestor( teresa, A ).
A = andrea ;      A = nicoletta ;
A = maurizio ;    A = mario ;
A = dina ;        A = roberto ;
false (no more solutions)
```

So many different queries from the same facts/rules!

```
% find known dina's nephews
?- ancestor( N, dina), not(parent(dina, N)).
    N = teresa ;   N = davide ;  false (no more solutions)
```

```
% find known sibling pairs
?- parent( Parent, Kid1), parent( Parent, Kid2 ), Kid1 @< Kid2.
    Parent = mario,      Kid1 = carla,       Kid2 = maurizio ;
    Parent = maurizio,   Kid1 = andrea,      Kid2 = gianluca ;
    Parent = andrea,     Kid1 = davide,      Kid2 = teresa ;
    Parent = nicoletta,  Kid1 = davide,      Kid2 = teresa ; false
```

Procedural interpretation of a Prolog program

You can see the rules/facts of your program as if they were a set of "subroutines", each with multiple alternative implementations

When you query for a given term proof, you CALL the corresponding set of clauses, which are tried one at a time (in textual order)

When a clause is called, its inner prerequisites are CALLED sequentially

When one FAILS, another clause is tried for the same term (by backtracking to the most recent choice, undoing it and trying the next)

This implies a DFS search of a solution in the execution tree

The first solution found is returned with its variable assignments

If you ask for another solution (tab or ;) Prolog bactracks and continues

Unification = Matching between data-structures

Unification: its powerful term-matching mechanism can be used to automatically pack/unpack terms and data structures

When they contains variables, Prolog looks for a suitable assignment of the variables (on both sides!)
Notice that the term functor and arity (# of args) should match

E.g.

parent(Dad, andrea, male) = parent(maurizio, andrea, Gender)
is true when Dad = maurizio AND Gender = male

Unification is way more powerful than Python multiple assignment used to pack/unpack, as unification goes both ways and inside terms

Two different types of “assignment” term unification vs. math computation

Unification is used to pack/unpack data structures (terms, lists, …)

term(X, two, three(X)) = term(four, B, C)

=> X=four B=two C=three(four)

NOTICE how the X value appears now in the term assigned to C

Unification CANNOT compute math expressions (but CAN do symbolic manipulation)

To do computation, instead, we use ‘is’ to evaluate expressions

A is max(3, 5) => A=5

B is A * 10 => B=50

C is 12 mod 7 => C=5

Functions available: min, max, arithmetic, random, trigonometric, logarithms logical (bits), ascii, ...

(a third type of assignment as constraint over the variable domain is available in Constraint Logic Programming predicates)

Demo

Genealogy demo

SWISH examples: kb, movies

Lists (dynamic, heterogeneous)

```
List = [ one, two, three, four ]        % list syntax

[ Head | Tail ] = List                  % how to extract the first element
    Head   = one                        % fails if the list is empty
    Tail   = [ two, three, four ]

[ First, Second | Rest ] = List         % extracting first and second element
    First  = one                        % fails if the list has less than 2 elements
    Second = two
    Rest   = [ three, four ]

EmptyList = []                          % the empty list

is_empty([]).                           % test for empty list through unification

length([], 0).                          % recursively compute the list length
length([H|T], N1) :- length(T,N), N1 is N + 1.
```

Predicates are relations and works in many ways/directions

```
append( [a], [b, c], L)        => L = [a, b, c]                % concatenation
append( A,   [b, c], [a, b, c]) => A = [a]                     % split
append( A,   B,      [a, b, c]) => A = [],      B = [a, b, c] ;  % all possible splits
                                   A = [a],     B = [b, c] ;
                                   A = [a, b], B = [c] ;
                                   A = [a, b, c],  B = [] ; fail

member( a, [a, b, c] )  =>  true                        % check membership
member( A, [a, b, c] )  =>  A=a   or  A=b   or  A=c     % find members
member( a, B)   =>      B = [a|_] ;         % generate list starting with a
                        B = [_,a|_] ;       % generate list with a in 2° place
                        B = [_,_,a|_] ;     % generate list with a in 3° place
                        … (infinite solutions)
```

Functional programming

Predicates can be used as if they were functions or to test values

You just add an argument to collect the result

```
square( X, Result ) :- Result is X * X.      % function
is_odd(X) :- 1 is X mod 2.                   % test = compute+unify
```

You can map functions over lists (with the apply library)

```
List = [ 1, 2, 3, 4 ], maplist( square, List, List1 ).
    => List1 = [ 1, 4, 9, 16 ]
```

Or get all elements satisfying some property

```
List = [1, 2, 3, 4], include(is_odd, List, Odd).
    => Odd = [1, 3]
List = [1, 2, 3, 4], partition(is_odd, List, Odd, Even).
    => Odd = [1, 3]   Even = [2, 4]
```

There is no need for looping constructs
Recursion, recursion everywhere!

NORMAL WAY: Repeating N times is done through recursion

```
repeat_something(0).          % base case
repeat_something(N) :-
    N > 0,                    % we are in the recursive case
    do_something,
    N1 is N-1,
    repeat_something(N1).
```

NOTICE: in this case you CAN collect results through the predicate variables

FAILURE-DRIVEN-WAY: repeat by failing, backtracking and retrying

```
repeat_something(N) :-
    between(1, N, X),         % generate X=1, 2, 3, 4, 5 … N by backtracking
    do_something,
    fail.                     % to avoid failure of the predicate
repeat_something(_).          % add a default “always true” clause
```

NOTICE: in this case you CANNOT collect results (unless you use side-effects)

More general ways to collect all solutions or to repeat

<u>All solutions</u> (with repetitions): bagof(Term, Predicate, ListOfTerms)

```
?- bagof( X, (member(X, [3, 2, 3, 4]), 1 is X mod 2), Odd)
        => Odd = [ 3, 3 ]
```

All <u>unique solutions</u>: setof(Term, Predicate, SetOfTerms)

```
?- setof( X, (member(X, [3, 2, 3, 4]), 1 is X mod 2), Odd)
        => Odd = [ 3 ]
```

Repeat a call <u>for each solution</u> of a Predicate: forall(Predicate, DoSomething)

```
?- forall( member(El, [1, 2, 3]), writeln(El) ).
     1
     2
     3
```

Programming styles

Single threaded

Declarative: data AND rules

- declarative data => relational data representation (SQL-like)

Functional: rules as functions transforming data

Meta-programming: programs that BUILD programs

Predicate/Relations can be used in many directions

Recursion, recursion everywhere!

Parallelism in some particular Prolog (Sicstus, Parlog, GHC)

Simple multiprocessing with the ‘spawn’ library

Prolog Pro/Cons for teaching

PRO

- Focus on data abstraction
- Focus on relations instead than procedures
- easy Natural Language processing and generation
- easy Symbolic manipulation
  (Math, Algebra, Physics, …)
- AI
- Recursion everywhere!

CONS

- Not typed (but you can use terms for dynamic typing)
- There is no really nice IDE (or you can use Eclipse PDT)
- Recursion everywhere!

Demo

DEMO

(to be continued)