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 (unification)

Used for:
- AI: natural language parsing, planning, natural language generation, theorem proving, ...
- 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
- 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 <u>NOT typed</u>, and start with Capital or _underscore
assignments are UNDONE on backtrack!!!

Facts describe relations that are <u>always true</u>

```
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
```

Program execution = searching 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

Prolog looks for a way to prove your query by searching:
- a fact to satisfy your query
- or else a predicate (rule) that could satisfy your query:
    - BUT: to prove it all its preconditions must be proved

If many ways exists to satisfy a query, all are tried in order (by backtracking/undoing last choice when subqueries fails)

The order of search is the order of the facts/clauses in the program

Values assigned (to 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)

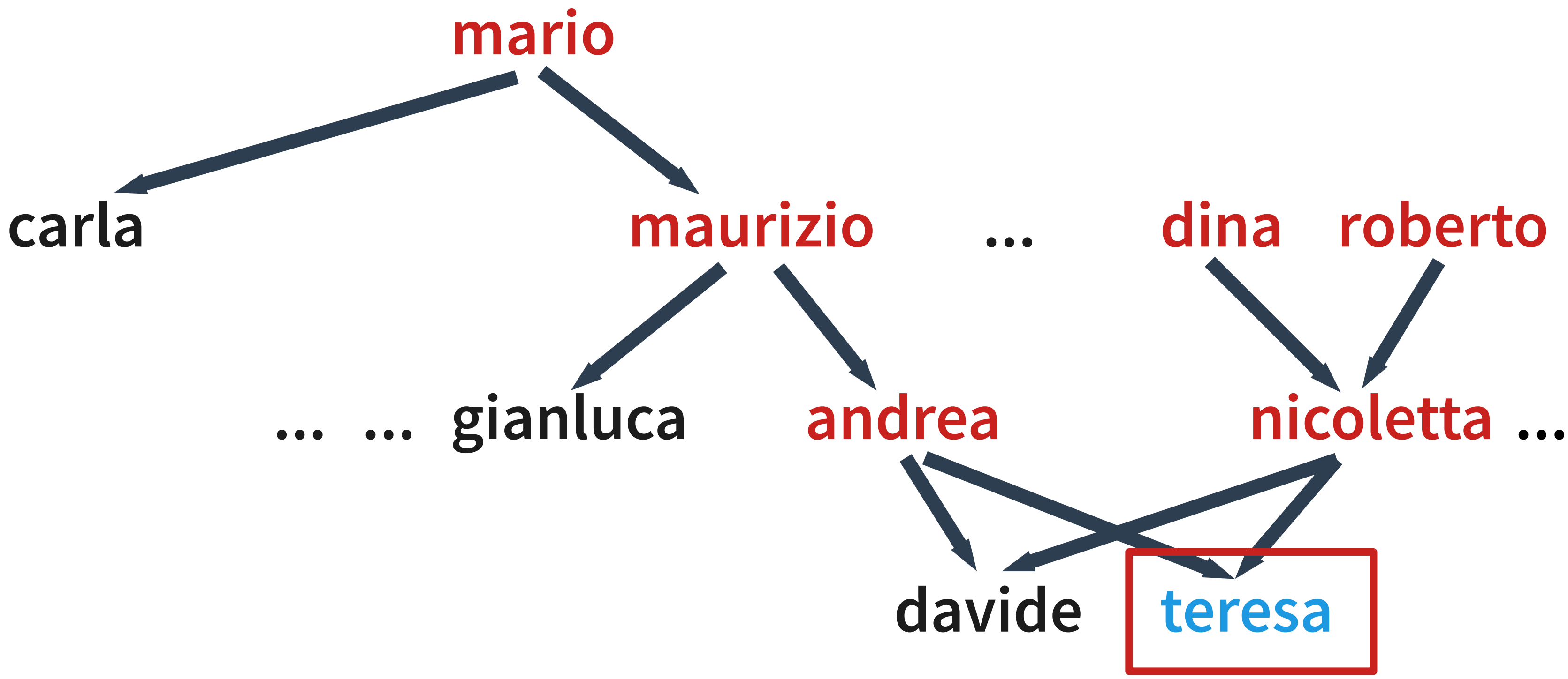
To retrieve a record with simple WITH constraints just QUERY with partial arguments and get variable values filled with found data

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 climb the parent relation

Representing facts AND relations (deduction rules)

FACTS (data)

```
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, Grandpa ) :-
    parent(Grandpa, Kid).
ancestor( Kid, Grandpa ) :-
    parent( P, Kid ),
    ancestor( P, 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

Multiple clauses as if-then-else? (not exactly)

When a predicate/rule has multiple clauses they are tried in order of appearance in the file (by backtrack)

(this IS NOT an if-then-else, as they are ALL tried on backtrack)

You could simulate if-then-else by using exclusive preconditions

```
clause(…) :-       condition, then.
clause(…) :- not(condition), else.
```

OR you can commit (!) to one clause as soon as its condition is met

```
clause(…) :- condition, !, then.   % no backtrack after '!'
clause(…) :- else.
```

OR you can use the -> operator   `clause(…) :- condition -> then ; else .`

The '!' (cut) predicate removes all current remaining choices and commits the execution to the only clause containing it (BUT BEWARE OF FAILURES AFTER THE CUT!)

Unification = Matching between data-structures

Unification is a powerful term-matching mechanism to automatically
pack/unpack terms and data structures used in clauses

E.g.

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

Variables are matched with the most general value (on both sides)
Notice that the term functor and arity (# of args) should match

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)

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.
```

Demo

Genealogy demo

SWISH examples: kb, movies

Predicates are relations and works in many ways/directions

```
append( [a], [b, c], L)          => L = [a, b, c]
append( A,   [b, c], [a, b, c])  => A = [a]
append( A,   B,      [a, b, c])  => A = [],          B = [a, b, c] ;
                                    A = [a],         B = [b, c] ;
                                    A = [a, b],      B = [c] ;
                                    A = [a, b, c],   B = [] ; fail

member( a, [a, b, c] )   => true
member( A, [a, b, c] )   => A=a     or  A=b     or  A=c
member( a, B)            => B = [a|_] ;         % list starting with a
                            B = [_,a|_] ;       % list with a in 2° place
                            B = [_,_,a|_] ;     % 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]
```

How to repeat without loops? 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,                % the index is reduced
    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 as a whole
repeat_something(_).          % add a default “always true” clause
```

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

Or else you could collect all solutions by:

All solutions of a Predicate: bagof(Term, Predicate, ListOfTerms)

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

=> Odd = [odd(3), odd(3)]

Unique solutions: setof(Term, Predicate, Set)

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

=> Odd = [odd(3)]

Just repeat DoSomething for each solution of a Predicate:

forall(Predicate, DoSomething)

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

1

2

3

Demo

DEMO