Prolog

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)

Used for:
- AI: natural language parsing, planning, natural language generation, theorem proving, ...
- meta-programming (programs that create programs)
- ...

Data types and program elements

Integers 42 **Float** 3.14 **Strings** “Hello world”

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

Terms height(andrea, 186)

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

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:
- if a fact is directly available to satisfy your query
- or else if there exists a predicate that would satisfy your query:
    - then to prove it all its preconditions must be proved

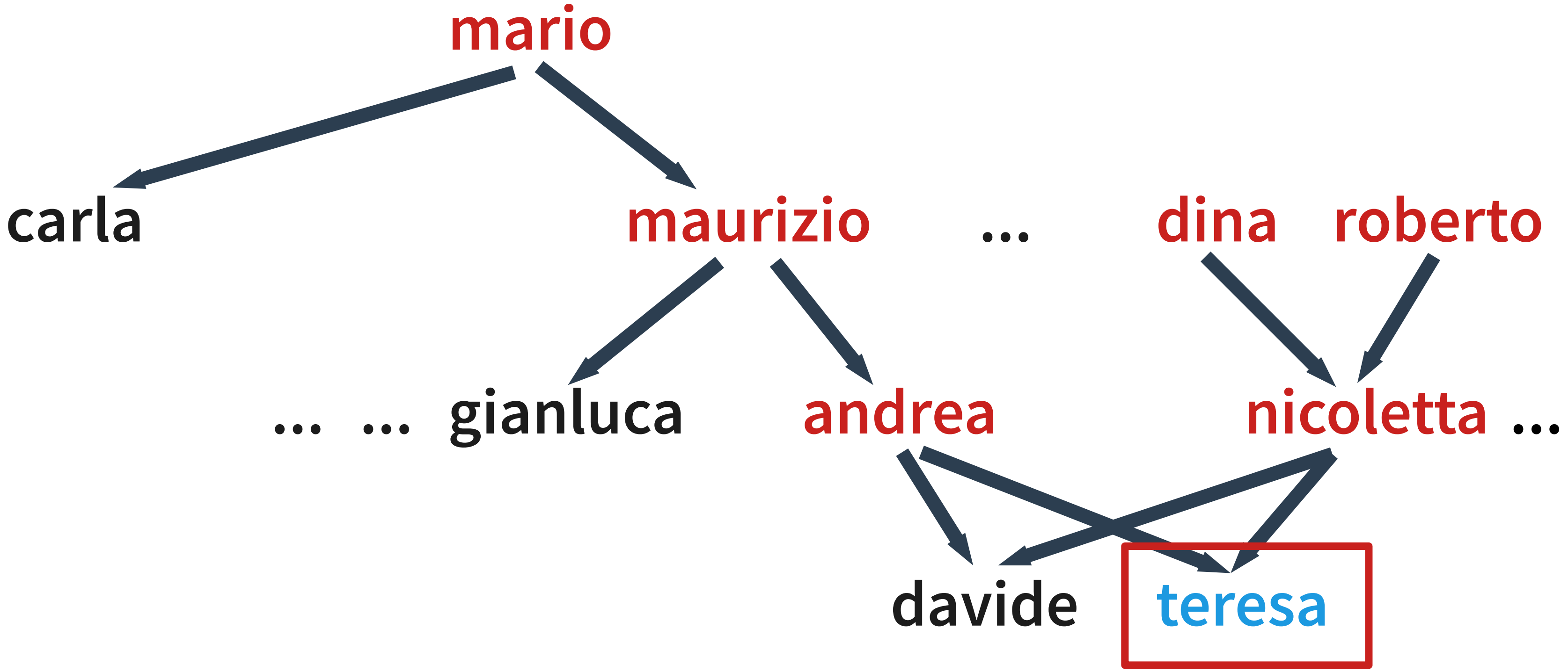
If more than one ways exists to satisfy a query, all are tried in order (by backtracking/undoing last choice if some of the subqueries fails)

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

Values assigned to the variables to satisfy the query are returned

Example: a small genealogy problem

From the ‘parent’ relation



If we want teresa’s ancestors we could 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, Ancestor ) :-
    parent(Ancestor, Kid).
ancestor( Kid, Ancestor ) :-
    parent( P, Kid ),
    ancestor( P, Ancestor).
```

QUERY

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

Many queries from the same facts/rules

```
% find known dina’s nephews
?- ancestor( N, dina).
   N = nicoletta ;   N = teresa ;   N = davide ;   false (no more solutions)

% find known siblings
?- parent( Parent, Kid1), parent( Parent, Kid2 ), Kid1 \= Kid2.
   Parent = mario,      Kid1 = maurizio,   Kid2 = carla ;
   Parent = mario,      Kid1 = carla,      Kid2 = maurizio ;
   Parent = maurizio,   Kid1 = andrea,     Kid2 = gianluca ;
   Parent = maurizio,   Kid1 = gianluca,   Kid2 = andrea ;
   Parent = andrea,     Kid1 = teresa,     Kid2 = davide ;
   Parent = andrea,     Kid1 = davide,     Kid2 = teresa ;
   Parent = nicoletta,  Kid1 = teresa,     Kid2 = davide ;
   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 <u>possibly with multiple alternative implementations</u>

When you query for a given term, you CALL the corresponding <u>set of clauses</u>, which are tried one at a time

When a clause is called, its inner prerequisites are CALLED <u>sequentially</u>

When it 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 <u>with its variable assignments</u>

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 the order of appearance in the file (by backtrack)

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

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 the condition is met

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

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

Unification = Matching between data-structures

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

E.g.

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

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

(unification is way more powerful than Python multiple assignment used to pack/unpack, as unification goes both ways)

Assignment …. normally through unification except for 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)

When some computation is required we use the 'is' predicate

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, …

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]
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

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

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]
```

Repeating a query N times

Repeating N times is normally 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 <u>you CAN collect results</u> through the predicate variables

Else you can repeat some operation by failing, backtracking and retrying

```
repeat_something(N) :-
    between(1, N, X),           % 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 <u>you CANNOT collect results</u> (unless you use side-effects)

Or else you could collect all solutions by:

Collect all solutions of a Predicate with bagof(Term, Predicate, ListOfTerms)

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

Or all <u>unique</u> solutions with predicate setof(Term, Predicate, Set)

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

Or just repeat DoSomething for each of the solutions of a Predicate with forall(Predicate, DoSomething)

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

Meta-programming and alternative syntax (DCG)

You can build terms from lists and viceversa

```
term( 1, two, three ) =.. [ term, 1, two, three ]
```

You can call/prove predicates built from data

```
call( Term )
```

You can add/remove new facts or clauses to the program

```
asserta( Head :- Body )          assertz( Head :- Body )
asserta( Fact )                  assertz( Fact )
retract( FactOrClause )
```

You can use an alternative syntax (e.g. Definite Clause Grammars)

```
sentence --> subject, verb, complement.   % automatically transformed to
sentence( Words, R3 ) :-
    subject( Words, R1 ), verb( R1, R2 ), complement(R2, R3).
verb --> [ run ].                         % is transformed to
verb( [ run | Rest ], Rest ).
```

Common extensions

Grammars

grammar rules map easily to Prolog predicates, both for parsing and for text generation

Constraints

the domain of the possible values of a variable can be constrained in many ways (e.g. the sudoku game)

OOP

terms could represent objects and their properties

rules could represent methods

GUI

widgets, events, callbacks and so on

Grammar example

```
Sentence          --> subject, verb, object.
subject           --> article(Gender), actor(Gender).
object            --> article(Gender), object(Gender).
article(female)   --> [ la ].            article(male)    --> [ il ].
actor(_)          --> [ chirurgo ].
actor(female)     --> [ elefantessa ].   actor(male) --> [ elefante ].
verb              --> [ mangiava ].
verb              --> [ guardava ].
object( female ) --> [ insalata ].
object( male )    --> [ cavolfiore ].
```

Constraint example (Sudoku)

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

- Natural Language processing and generation
- Symbolic manipulation
  (Math, Algebra, Physics, …)
- Recursion everywhere!

CONS

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

Demo

DEMO