AI Fundamentals: rule-based systems

Maria Simi
Logic programming and Prolog

LESSON 2: PROLOG- PROCEDURAL CONTROL OF REASONING- CONSTRAINT LOGIC PROGRAMMING-META INTERPRETERS
Summary

- Logic programming and Prolog
- Prolog as a full fledged programming language
  - Data structures, special constructs
- Procedural control of reasoning (ALGORITHM = LOGIC + CONTROL)
  - Specifying goal ordering
  - Controlling backtracking
  - Negation as failure
  - Implementing search strategies
- Constraint logic programming
- Meta-interpreters
SLD resolution and Prolog

Prolog is a rule-based/logic-programming language based on SLD resolution.

1. The declarative semantics is given by Horn clause knowledge bases.
2. The procedural semantics is given by a specific strategy for generating SLD trees:
   - Successors are generated in the order they appear in the logic program
   - The SLD tree is generated left-to-right depth-first.
   - In the unification, the occur-check is omitted for efficiency.

Since the depth-first visiting strategy is not complete, the Prolog is not complete.

For example the program:

\[
\begin{align*}
\text{ancestor}(X, Y) & : - \text{ancestor}(Z, Y), \ \text{parent}(X, Z). \\
\text{ancestor}(X, Y) & : - \text{parent}(X, Y).
\end{align*}
\]

diverges on the query ?- ancestor(lia, mark).
Introduction to Prolog

FROM THE BOOK BY IVAN BRATKO
Prolog: basic data types

- **Atoms**
  - Identifiers with initial lowercase: tom, x-1, ... (logical constants)
  - Strings of characters: ‘Tom’, ’Sarah Johnes’, ...
  - Strings of special characters: <=>, ::=, ...

- **Numbers**
  - Integers: 0, 1, -10, 1313, ...
  - Real: 3.4, -0.0035, ...

- **Variables**
  - Identifiers with initial **UPPERCASE**: Tom, X1, Result ...
  - Anonymous variables: ‘_’ as in hasChild(X) :- parent(X, _)
Prolog: structured objects

- Structured objects are terms including functions (functor applied to arguments)
  - date(1, may, 2001), date(Day, may, 2001) any day in May
  - point(1, 1), point(2, 2), ... segment(point(1,1), point(2, 2))
  - Arithmetic expressions: *(+(a, b), -(c, 5))

- Lists
  - [Head | Tail]: Head is the first element, Tail is the rest.
  - [1 | 2, 3, 4, 5] = [1, 2, 3, 4, 5] = [1, 2, 3 | 4, 5]

- Examples: membership in a list and concatenation of two lists:

  member(X, [X | Tail]).
  member(X, [Head | Tail]) :- member(X, Tail).
  conc([], L, L).
  conc([X | L1], L2, [X | L3]) :- conc(L1, L2, L3).
Basic arithmetic operations:
- +, −, *, / (division), // (integer division), ** (power), mod ...
- The ‘is’ infix operator forces the evaluation of the expressions: A is (5−2) + 1

Comparison operators:
- >, <, >=, <=, =:= (equal), \= (not equal) they force evaluation

While with 1+2 = 2+1 unification fails, 1+2 =:= 2+1 forces the evaluation and the answer is YES. If X =:= Y, the variables must be instantiated.

Example: length of a list. Compare the two programs.

\[
\begin{align*}
\text{len}([ ], 0). \\
\text{len}([\_|\text{Tail}], N) & : \text{len}(\text{Tail}, N1), \text{N is } 1+N1. \\
?\text{- }\text{len}([a, b, c], N).
\end{align*}
\]
Using structures: a family knowledge base

family(
    person(tom, fox, date(7, may, 1950), works(bbc, 15200)),
    person(ann, fox, date(9, may, 1951), unemployed),
    [person(pat, fox, date(5, may, 1973), unemployed),
     person(jim, fox, date(5, may, 1973), unemployed)])
Using structures: retrieving from the KB

?- family(person(_, fox, _, _), _).
?- family(X, [_, _]).
husband(X):=- family(X, _).
wife(X):=- family(_, X, _).
child(X) :- family(_, _, Children), member(X, Children).
dateofbirth(person(_, Date, _), Date).
?- child(X), dateofbirth(X, date(_, may, _)).
Using structures: computing

exists(P) :- husband(P); wife(P); child(P).  syntax for or
?- exists(P), dateofbirth(P, date(_, _, Y)), Y < 1970.
salary(person(_, _, works(_, S)), S).
salary(person(_, _, unemployed), 0).
total([], 0).
total([P|L], Sum) :- salary(P, S), total(L, R), Sum is S + R.
?- family(H, W, C), total([H, W | C], FamilyIncome).

family

person

person

person

person

person

person

person

person
“Algorithm = logic + control”

ROBERT KOWALSKI
Procedural control of reasoning

**ALGORITHM = LOGIC + CONTROL** [Robert Kowalski]

“An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used. The logic component determines the meaning of the algorithm whereas the control component only affects its efficiency.”

Declarative encoding of knowledge and general deduction are appealing but inefficient. For efficiency we must have some domain dependent control on the reasoning process. Given a KB made of facts and rules, how to make the most effective use of the rules?
Ordering subgoals

Suppose we are looking for an American cousin of Sally:

?- americanCousin(X, sally)

we could either:

1. find an American and then check to see if she is a cousin of Sally
   
   americanCousin(X, Y) :- american(X), cousin(X, Y).

2. find a cousin of Sally and then check to see if she is an American
   
   americanCousin(X, Y) :- cousin(X, Y), american(X).

Both programs are correct, but the choice makes a difference in performance.

PROLOG takes ordering of clauses and subgoals very seriously; the burden is on the programmer. In this case, it is better to generate all cousins and for each one test whether she is American ...
Consider three logically equivalent ways to express the relationship between the two predicates:

1. \textit{ancestor (X, Y) :- parent (X, Y).}
   \textit{ancestor (X, Y) :- parent (X, Z), ancestor(Z, Y).}

2. \textit{ancestor (X, Y) :- parent (X, Y).}
   \textit{ancestor (X, Y) :- parent (Z, Y), ancestor (X, Z).}

3. \textit{ancestor (X, Y) :- parent (X, Y).}
   \textit{ancestor (X, Y) :- ancestor (X, Z), ancestor (Z, Y).}

The three versions give the same results on all questions. However they could lead to substantially different amounts of computation.
Controlling backtracking

Prolog will automatically backtrack if this is necessary to satisfy a goal.

**Uncontrolled backtracking** however may cause inefficiency in a Prolog programs.

f1(X, 0) :- X < 3. % Rule 1
f1(X, 2) :- 3 =< X, X < 6. % Rule 2
f1(X, 4) :- 6 =< X. % Rule 3

?- trace, f1(1, Y), 2 < Y.
  ?- 1 < 3, 2 < 0 fail {X/1, Y/0}
  ?- 3 =< 1 fail {X/1, Y/2}
  ?- 6 =< 1 fail {X/1, Y/4}

Since the conditions in the body are **mutually exclusive**, we know that only one of them will succeed. After trying the first rule and failing on 2 < Y, we could give up.
Controlling backtracking with CUT

\[ f_2(X, 0) : - X < 3, !. \quad \text{CUT!} \quad % \text{Rule 1} \]
\[ f_2(X, 2) : - 3 =< X, X < 6. \quad % \text{Rule 2} \]
\[ f_2(X, 4) : - 6 =< X. \quad % \text{Rule 3} \]

?- f2(1, Y), 2 < Y.

No backtracking after first failure

?- trace, f2(7, Y).
  ?- 7 < 3 fail
  ?- 3 =< 7, 7 < 6 fail this test is redundant
  ?- 6 =< 7 ok this test is redundant

\[ f_3(X, 0) : - X < 3, !. \]
\[ f_3(X, 2) : - X < 6, !. \]
\[ f_3(X, 4). \]

if \( X < 3 \) then \( Y = 0 \),
else if \( X < 6 \) then \( Y = 2 \),
else \( Y = 4. \)
General behavior of CUT

The example was a case where, given a clause of the form “$G :- T, R.$” goal $T$ is needed only as a test for the applicability of subgoal $R$; if $R$ fails we do not want to backtrack to $T$ nor try any other alternative for $G$.

The pattern:
- $G :- T, !, R$
- $G :- S.$

More efficient than:
- $G :- T, R.$
- $G :- \overline{T}, S.$ $\overline{T}$ a goal mutually exclusive with $T$

In general:
- $G :- T1, T2, ..., Tm, !, G1, G2, ... , Gn.$

means that once $T1, T2, ..., Tm$ have been established we can commit to the rest of goals without looking for alternatives.
Other CUT examples

1. Anybody, except Adam and Eve, has two parents.

   numberOfParents (adam, V) :- !, V=0.
   numberOfParents (eve, V) :- !, V=0.
   numberOfParents (P, 2).

2. The maximum of two numbers

   max( X, Y, X) :- X >= Y, !.
   max( X, Y, Y).

3. This version of member stops as soon as it finds an element equal X.

   member( X, [X | L] ) :- !.
   member( X, [Y | L] ) :- member( X, L).

4. Classify people in categories according to this schema:
   Winner: always wins
   Fighter: sometime wins, sometime not
   Sportsman: always beated

   beat(tom, jim).
   beat(ann, tom).
   beat(pat, jim).
   class( X, fighter) :- beat( X, _), beat(_, X), !.
   class( X, winner) :- beat(X, _), !.
   class( X, sportsman) :- beat(_, X).
Negation as failure

Suppose we want to represent “Mary likes all animals but snakes”.

Let’s try with “If \( X \) is a snake then ‘Mary likes \( X \)’ is not true, otherwise if \( X \) is an animal then Mary likes \( X \)”. This can be done introducing a special goal \texttt{fail} that always fails:

\[
\begin{align*}
\text{likes}(\text{mary}, \text{X}) :& - \text{snake}(	ext{X}), !, \text{fail}.
\text{likes}(\text{mary}, \text{X}) :& - \text{animal}(\text{X}).
\end{align*}
\]

This example, and many others, indicate that it would be useful to have a unary predicate '\texttt{not}' such that \texttt{not}(P) is true if \( P \) fails. It could be defined as follows:

\[
\begin{align*}
\text{not}(P) :& - P, !, \text{fail}. \\
\text{not}(P) :& \text{fail if } P \text{ succeeds} \\
\text{not}(P) :& \text{else succeed}
\end{align*}
\]

\texttt{not} is a \textbf{built-in} Prolog procedure that behaves as defined above.
Negation as failure

This new type of goal, not(G), is understood to succeed when the goal G fails and to fail when the goal G succeeds.

Failure must occur in a finite number of steps.

Other examples:

1. noChildren(X) :- not(parent(X, Y)). we assume a closed world.
   :- noChildren(john) succeeds if :- parent(john, Y) fails
   Different from proving KB |= ∀y ¬parent(john, y) = ¬∃y parent(john, y). It is rather KB |= ¬∃y parent(john, y). This makes the behavior nonmonotonic.

2. composite(N) :- N > 1, not (primeNumber(N)).

3. Easier to read solution to the classification problem.
   class(X, fighter) :- beat(X, _), beat(_, X).
   class(X, winner) :- beat(X, _), not(beat(_, X)).
   class(X, sportsman) :- beat(_, X), not(beat(X, _)).
Problems with CUT and negation

Using CUT has advantages and drawbacks:

1. With cut we can often improve the efficiency of the program. The idea is to explicitly tell Prolog: do not try other alternatives because they are bound to fail.

2. Using cut we can specify mutually exclusive rules; so we can add expressivity to the language.

The main disadvantage is that we can lose the correspondence between the declarative and procedural meaning of programs. Compare:

\[
\begin{align*}
p & : - a, b. \\
p & : - c. \\
p & \iff (a \land b) \lor c
\end{align*}
\]

\[
\begin{align*}
p & : - a, !, b. \\
p & : - c. \\
p & \iff (a \land b) \lor (\neg a \land c)
\end{align*}
\]

\[
\begin{align*}
p & : - c. \\
p & : - a, !, b. \\
p & \iff c \lor (a \land b)
\end{align*}
\]

We can distinguish

- **Green cuts**: that do not change the meaning (safer)
- **Red cuts**: that change the meaning, we have to be careful to the actual meaning.
Consider the Fibonacci series: 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

Solution 1:

\[ \text{fib}(0, 1) \]
\[ \text{fib}(1, 1) \]
\[ \text{fib}(N, V) \quad \text{:-} \quad X_2 \text{ is } N-2, \text{fib}(X_2, Y), X_1 \text{ is } N-1, \text{fib}(X_1, Z), \text{plus}(Y, Z, V). \]

\[ \text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1) \]

Solution 2:

\[ \text{fib}(N, V) \quad \text{:-} \quad f(N, 1, 0, V). \]
\[ f(0, Y, Z, Y). \]
\[ f(N, Y, Z, V) \quad \text{:-} \quad X_1 \text{ is } (N-1), \text{plus}(Y, Z, S), f(X_1, S, Y, V). \]

\[ f(n, y, z, v) \text{ iff } v = y \cdot \text{fib}(n) + z \cdot \text{fib}(n-1) \]

This equivalent characterization avoids the redundancy of the previous version and requires only a linear number of Plus subgoals. Fib of 100 is computable in solution 2 but not in solution 1.
AI programming in Prolog

Basic search algorithms for problem solving are easy to implement (see Part II of Bratko book). The following is a depth-first search with cycle breaking.

You need to define:
- States
- Initial state
- Goal-test (s)
- Successors (s)

for your problem, then ask:

?- solve(initial-state-node, Solution).

*Note:* the test

not (member(Node1, Path))

is to break cycles.

```
solve(Node, Solution) :-
    depthfirst([], Node, Solution).

depthfirst(Path, Node, [Node | Path]) :-
    goal(Node).

depthfirst(Path, Node, Sol) :-
    s(Node, Node1),
    not (member(Node1, Path)),
    depthfirst([Node | Path], Node1, Sol).
```
A well known example: blocks world

Initial state: 

```
[[c, a, b], [], []]  
```

*a list of stacks*

Goal state: 

```
...[a, b, c]...
```

Goal test:

```
goal(State) :- member([a, b, c], State).
```

goal(State) :- member([a, b, c], State).

Transition function:

• moves the top of one stack to another stack
  (empty stack means table).

• uses the del function to delete an item from
  a stack

```
s(Stacks, [Stack1, [Top1|Stack2]|Other]) :-
del([Top1 | Stack1], Stacks, Stacks1),
del(Stack2, Stacks1, Other).
```

```
del(X, [X|L], L). % utility
del(X, [Y|L], [Y|L1]) :- del(X, L, L1).
```

? solve([[c, a, b], [], []], Solution).

The solution found is very long, not optimal.
Adding a depth-limit, iterative deepening

Iterative deepening can be obtained by starting with depth 0 and incrementing depth until a solution is found.

solve(Node, Solution, Max) :-
   depthfirstL(Node, Solution, Max).

depthfirstL(Node, [Node], _ ) :-
   goal(Node).

depthfirstL(Node, [Node|Sol], Max) :-
   Max > 0,
   s(Node, Node1),
   Max1 is Max-1,
   depthfirstL(Node1, Sol, Max1).

?- solve([[c, a, b], [ ], [ ]], Solution, 4).
This is an alternative version of iterative deepening.

The function \textit{path} generates, for the given initial node, all the possible paths of increasing length.

Each one is then goal tested by \textit{solve}.

\begin{verbatim}
path(Node, Node, [Node]).

path(FirstN, LastN, [LastN | Path]) :-
    path(FirstN, OneButLast, Path),
    s(OneButLast, LastN),
    not(member(LastN, Path)).

solve(Node, Solution) :-
    path(Node, GoalNode, Solution),
    goal(GoalNode).

?- solve([[c, a, b], [ ], [ ]], Solution).
\end{verbatim}
AI programming in Prolog

- We can easily implement other search strategies: breadth-first, A* ...
- The book shows other nice examples of AI Programming ...
  - Constraint logic programming
  - Expert systems
  - Planning
  - Machine learning
  - Language processing
  - Game planning
  - Meta-programming.

You are free to explore in the Your turn session.
Constraint logic programming
Constraint Logic Programming

Constraint logic programming (CLP) combines the constraint satisfaction approach with logic programming, creating a **new language** where a logic program works along a specialized constraint solver.

The basic Prolog can be seen as a very specific constraint satisfaction language where the constraints are of a limited form, that is **unifications constraints** or **bindings**. Prolog is extended introducing other types of constraints.

**CLP(X)** differ in the domain and type of constraints they can handle.

1. **CLP(R)**: constraints on real numbers
2. **CLP(Z)**: integers
3. **CLP(Q)**: rational numbers
4. **CLP(B)**: boolean values
5. **CLP(FD)**: finite domains
Trying the SWISH Prolog CLP’s libraries

Constraint logic programming (CLP) allows variables to be constrained rather than bound.

A CLP solution is the most specific set of constraints on the variables that can be derived from the knowledge base. A specific solution if the constraints are tight enough.

Compare the behavior a classical Prolog program with a CLP program.

convert(Euro, USD) :-
    USD is Euro * 0.842.
?- convert(150, USD).
USD = 126.3
?- convert(Euro, 200).
Arguments are not sufficiently instantiated

convert(Euro, USD) :-
    {USD = Euro * 0.842}.
?- convert(150, USD).
USD = 126.3
?- convert(Euro, 200).
Euro = 237.52969121140143
?- convert(Euro, USD).
{USD=0.842*Euro}
Syntax and built-in functions for CLP

**CLP(R) and CLP(Q), real and rational**

Syntax for constraints:

\{
A < 2, \ B = 5, \ C > A\}
\{
1 + \ X = 5\}

Built-in functions for constraints:

?- \{X =< 5\}, maximize(X).
X=5.0

?- \{X =< 5, \ 2 =< X\}, minimize(2*X + 3).
X=2.0

**CLP(FD), finite domains**

Syntax for constraints:

X in Set to declare the domain of X

Set can be:

\{1, 2, 3, 4, 5\} a list of integers
1 .. 10 a range
Set1 \(\lor\) Set2 union
Set1 \(\land\) Set2 intersection
\Set complement

Comparison operators:

#= equal
\#\= not equal
#< less than
#=< less or equal

06/12/17 AI FUNDAMENTALS - M. SIMI 32
Trying the SWISH Prolog CLP’s libraries

In Swish Prolog you can load the following libraries:
1. clpfd (for finite domains)
2. clpq (for rational domains)
3. clpr (for real domains)

```
triangle(X, Y, Z) :-
X > 0, Y > 0, Z > 0,
X+Y >= Z, Y+Z >= X, X+Z >= Y.
?- triangle(3, 4, 5).
YES.
?- triangle(3, 4, Z)
cannot be solved

:- use_module(library(clpfd)).
triangle(X, Y, Z) :-
{X #> 0, Y #> 0, Z #> 0,
X+Y #>= Z, Y+Z #>= X,
X+Z #>= Y}.
?- triangle(3, 4, Z).
Z in 1..7
```

% {Z #> 1.0, Z #< 7.0}
Crypto-arithmetic CLP

DONALD +
GERALD =
ROBERT

Built-in functions:
all_different(L):
all the variables have different values
labeling([], L): assigns values from left to right.

:- use_module(library(clpfd)).

Vars = [D,O,N,A,L,G,E,R,B,T],
D in 0..9, O in 0..9, N in 0..9, A in 0..9,
L in 0..9, G in 0..9, E in 0..9, R in 0..9,
B in 0..9, T in 0..9,
all_different(Vars),
100000*D + 10000*O + 1000*N + 100*A + 10*L + D +
100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=
100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
labeling([], Vars).

?- solve(N1, N2, N3).
Meta interpreters
Meta-interpreters

- A meta-interpreter for a language is a program that is written in the language itself and treats other programs as data.
- Prolog has a powerful features for writing meta programs because Prolog treats programs and data both as terms.
- One can write meta interpreters for various applications, extending the implementation of Prolog in different directions.
- Applications:
  - exploring different execution strategies, i.e. on breadth first, limited depth search, combination of depth first and breadth first searches, etc.
  - generating proof trees, expert system shell, trace facilities ...
  - Implementing new languages
A vanilla meta-interpreter

To build the meta-interpreter we can rely on the built-in predicate:

\[
\text{clauses}(\text{Goal}, \text{Body})
\]

which retrieves a clause from the consulted program that matches \textit{Goal}.

The **vanilla meta-interpreter** does nothing, but it can be extended in several directions.

```prolog
member1(X, [X | _]). % example program
cache1(X, [X | Tail]) :-
    member1(X, Tail).
%?- member1(3, [1,2,3]).
%-------------------------------------------------------
% Vanilla meta-interpreter
prove(true).
prove(Goal) :-
    clause(Goal, Body),
    prove(Body).
prove(Goal1, Goal2) :-
    prove(Goal1), prove(Goal2).
%?- prove(member1(3, [1,2,3])).
```
A tracing meta-interpreter

The following code extends the vanilla meta-interpreter with a tracing facility.

```prolog
% a tracing meta-interpreter
prove(true) :- !.
prove(Goal) :-
    write('Call: '), write(Goal), nl,
    clause(Goal, Body),
    prove(Body),
    write('Exit: '), write(Goal), nl.
prove(Goal1, Goal2) :- !,
    prove(Goal1),
    prove(Goal2).
%?- prove(member1(3, [1,2,3])).
```
A breadth-first meta-interpreter

From *Artificial Intelligence Techniques in Prolog* by Yoav Shoham

```prolog
% A breadth-first meta-interpreter
% 
meta_bf( Goal ) :- b( [ conj( [ Goal ], Goal ) ], Goal ).

b( [ conj( [ ], InstanceOfGoal ) | _ ], InstanceOfGoal ).
b( [ conj( [ ], _ ) | S ], Goal ) :- b( S, Goal ).
b( [ conj( [ A | B ], OrigGoal ) | S ], Goal ) :-
  system( A ), !,
  append( S, [ conj( B, OrigGoal ) ], Newlist ),
  b( Newlist, Goal ).
b( [ conj( [ A | _ ], _ ) | S ], Goal ) :-
  system( A ), !,
  b( S, Goal ).
b( [ conj( [ A | B ], OrigGoal ) | S ], Goal ) :-
  findall( conj( BB, OrigGoal ),
    ( clause( A, Body ),
      mixed_append( Body, B, BB ) ),
    L ),
  append( S, L, Newlist ),
  b( Newlist, Goal ).

% mixed_append/3 appends an 'and' list and a regular one; see Chapter 1.
```
Conclusions

- Prolog is a very powerful and elegant rule-based programming language, very flexible and suitable for rapid prototyping of AI paradigms.
- Implementation is quite efficient (Warren abstract machine).
- Care must be taken in controlling the order of rules and subgoals, and of the CUT (!) operator.
- Meta-level interpreters can be used to extend the language and easily design new languages.
- The II part of the book by Bratko, implements with simple Prolog programs many AI paradigms, including machine learning.
- You are encouraged to experiment.
- Rules are used backwards. Next time we will discuss rule based running forward.
Your turn

✓ The II part of the book by Bratko, implements in Prolog programs AI paradigms, including machine learning.

✓ You are encouraged to experiment. For example:
  - Search algorithms
  - An expert system application
  - An application of constraint programming
  - An expert system shell
  - Planning algorithms
  - Learning algorithms
  - ...

AI FUNDAMENTALS - M. SIMI
References

