PROLOG. Advanced Issues.
Knowledge Representation, Reasoning and Inference Control.
Predicates: not, cut, fail.

Antoni Ligęza

Katedra Automatyki, AGH w Krakowie

2021

References

1. Ulf Nilsson, Jan Maluszyński: Logic, Programming and Prolog, John Wiley & Sons Ltd., pdf, http://www.ida.liu.se/ ulfni/lpp
2. Dennis Merritt: Adventure in Prolog, Amzi, 2004 http://www.amzi.com/AdventureInProlog
3. Quick Prolog: http://www.dai.ed.ac.uk/groups/ssp/bookpages/quickprolog/quickprolog.html
4. W. F. Clocksin, C. S. Mellish: Prolog. Programowanie. Helion, 2003
5. SWI-Prolog's home: http://www.swi-prolog.org
6. Learn Prolog Now!: http://www.learnprolognow.org
7. http://home.agh.edu.pl/ ligeza/wiki/prolog
8. http://www.im.pwr.wroc.pl/ przemko/prolog

Prolog and Relational Databases

Similarities: analogies between RDB and PROLOG

- records are PROLOG facts,
- tables are PROLOG collections of facts,
- links can be representad as binary facts,
- associative tables can be representad as collections binary facts,
- SELECT – easily implemented by backtracking search and pattern matching,
- all relational/set operations can be easily mimic with PROLOG constructs.

Dissimilarities: advantages of PROLOG

- terms and unification – structural objects and matching them accessible,
- lists – no problem with multiple values,
- clauses – inference instead of pure search,
- recursion – intrinsic in PROLOG,
- iteration – simple in PROLOG,
- repeat-fail – built-in backtracking search,
- operators – custom knowledge representation language creation,
- meta-programming – a unique feature;

Problems with SQL

SQL is not a complete language!
SQL has no recursion.

Example problems with SQL

- variables — lack of variables,
- transitive cosure,
- recurency,
- lists,
- structures,
- trees,
- graphs,
- loops,
- combinatorial probelms,
- constraint satisfaction,
- search,
- backtracking search.

Two Example Problems

Shopping: a Knapsack Problem

```
item(item1,123).
item(item2,234).
item(item3,345).
item(itme4,456).
item(item5,567).
```

Task: find all sets of items that can be purchased for a given total cost.

Travelling: Path Generation

```
link(a,b).
link(a,c).
link(b,c).
link(b,d).
link(c,d)
link(c,e).
link(d,e).
```

Task: find all paths from a to d.

Library: an example database structure:

```
book(signature_1,title_1,author_1,publisher_1,year_1).
book(signature_2,title_2,author_2,publisher_2,year_2).
book(signature_3,title_3,author_1,publisher_3,year_3).
book(signature_4,title_4,author_4,publisher_4,year_4).
book(signature_5,title_5,author_1,publisher_5,year_5).

journal(journal_id_1,title_1,volume_1,number_1,year_1).
journal(journal_id_2,title_2,volume_2,number_2,year_2).
journal(journal_id_3,title_3,volume_3,number_3,year_3).
journal(journal_id_4,title_4,volume_4,number_4,year_4).
journal(signature_5,title_5,volume_5,number_5,year_5).

user(id_1,surname_1,forename_1,born_1,address_1).
user(id_2,surname_2,forename_2,born_2,address_2).
user(id_3,surname_3,forename_3,born_3,address_3).
user(id_4,surname_4,forename_4,born_4,address_4).
user(id_5,surname_5,forename_5,born_5,address_5).

register(id_1,signature_1,borrow_date_1,return_date_1).
register(id_1,signature_2,borrow_date_2,return_date_2).
register(id_2,signature_1,borrow_date_3,return_date_3).
```

Library: example database operations

```
sum(Id,Title,Year):-
   book(Id,Title,_,_,Year);
   journal(Id,Title,_,_,Year), not(book(Id,Title,_,_,Year)).

intersect(Id,Title,Year):-
   book(Id,Title,_,_,Year),
   journal(Id,Title,_,_,Year).

except(Id,Title,Year):-
   book(Id,Title,_,_,Year),
   not(journal(Id,Title,_,_,Year)).

projection(Title,Author,Year):-
   book(_,Title,Author,_,Year).

selection(Signature,Title,Author,Publisher,Year):-
   book(Signature,Title,Author,Publisher,Year),
   Author = author_1.

cartesian_product(Signature,Uid):-
   book(Signature,_,_,_,_),
   user(Uid,_,_,_,_).
```

Prolog Database Representation. Datalog Convention.

Library: example database operations

```
inner_join(Uid,Surname,Name,Signature):-
   user(Uid,Surname,Name,_,_),
   register(Uid,Signature,_,_).

left_outer_join(Uid,Surname,Name,Signature):-
   user(Uid,Surname,Name,_,_),
   register(Uid,Signature,_,_).
left_outer_join(Uid,Surname,Name,Signature):-
   user(Uid,Surname,Name,_,_),
   not(register(Uid,Signature,_,_)).

who_what_book(Uid,Surname,Name,Signature,
              Title,BorrowDate,ReturnDate):-
   register(Uid,Signature,BorrowDate,ReturnDate),
   user(Uid,Surname,Name,_,_),
   book(Signature,Title,_,_,_).

subquery_has_book(Uid,Surname,Name):-
   user(Uid,Surname,Name,_,_),
   register(Uid,_,_,_).
```

```
rodzina(osoba(jan, kowalski,data(5,kwiecien,1946),pracuje(tpsa,3000)),
        osoba(anna,kowalski,data(8,luty,1949),    pracuje(szkola,1500)
        [osoba(maria,kowalski,data(20,maj,1973),     pracuje(argo_turi
         osoba(pawel,kowalski,data(15,listopad,1979),zasilek)]).

rodzina(osoba(krzysztof, malinowski, data(24,lipiec,1950), bezrobocie)
        osoba(klara, malinowski, data(9,styczen,1951), pracuje(kghm,80
        [osoba(monika, malinowski, data(19,wrzesien,1980), bezrobocie)

maz(X) :- rodzina(X,_,_).
zona(X) :- rodzina(_,X,_).
dziecko(X) :- rodzina(_,_,Dzieci), member(X,Dzieci).

istnieje(Osoba) :- maz(Osoba); zona(Osoba); dziecko(Osoba).

data_urodzenia(osoba(_,_,Data,_),Data).

pensja(osoba(_,_,_,pracuje(_,P)),P).
pensja(osoba(_,_,_,zasilek),500).
pensja(osoba(_,_,_,bezrobocie),0).

zarobki([],0).
zarobki([Osoba|Lista],Suma) :-
        pensja(Osoba,S),
        zarobki(Lista,Reszta),
        Suma is S + Reszta.
```

Built-in Predicates

1. `fail` – always fails. Since it is impossible to pass through, it is used to force *backtracking*.
2. `true` – always succeeds.
3. `repeat` – always succeeds; provides an infinite number of choice points.
4. `!` – cut; prohibits backtracking to the goals located left of its placement.
5. `+Goal1, +Goal2` – conjunction; prove `Goal1`, then `Goals2`.
6. `+Goal1; +Goal2` – disjunction; either `Goal1` or `Goal2` should be proved.It is translated into:

```
Goal12 :- Goal1.
Goal12 :- Goal2.
```

Disjunction in clauses:

```
h :- p;q.
```

can be translated into:

```
h :- p.
h :- q.
```

Definition of `repeat`

```
repeat.
repeat:- repeat.
```

The Role and Use of fail

Role of fail

The fail predicate does not unify with any other predicate.

1. fail — stops further inference, and
2. enforces backtracking.

Applications of fail

1. to ensure exploration of all the possible executions of a clause; fail is placed at the end of this clause.
2. definition of repeated actions (a loop),
3. ensuring program execution (robust programming).

An action-fail loop

```
loop :-
   action,
   fail.
loop.
```

A repeat-test loop

```
loop :-
   repeat,
   action,
   test.
test :- termination_ok,!,go_out.
test :- fail.
```

Example loop solutions

```
loop_infinite:-
   repeat,
   actions,
   fail.

loop_infinite_read_write:-
   repeat,
   read(X), process(X,Y), write(Y),nl,
   fail.

loop_find_fail:-
   d(X),
   process(X,Y), write(Y),nl,
   fail.
loop_find_fail.

loop_repeat_test:-
   repeat,
   d(X),
   process(X,Y), write(Y),nl,
   test_for_final(Y), write('***end: '),write(Y),nl.
d(0). d(1). d(2). d(3). d(4). d(5). d(6). d(7). d(8). d(9).
```

The cut predicate

Description of cut

- The *cut* predicate is a standard predicate which helps avoiding exploration of further inference possibilities, provided that the one currently explored satisfies certain predicates.
- The cut predicate allows for pruning branches in the search tree generated by the depth-first search algorithm.
- The *cut* predicate is symbolized with the exclamation mark '!'.

Definition and operation of cut

Consider a clause:

$$h \ :- \ p_1, p_2, \ldots, p_i, !, p_{i+1}, \ldots, p_m.$$

1. ! divides a clause in the left and right part,
2. the left part atoms are removed form stack,
3. no backtracking for h, $p_1, p_2, \ldots, p_i$,
4. backtracking is still possible for $p_{i+1}, \ldots,$ up to p_m.

Examples with cut

```
cyfra(0).
cyfra(1).
cyfra(2).
cyfra(3).
cyfra(4).
cyfra(5).
cyfra(6).
cyfra(7).
cyfra(8).
cyfra(9).

liczba(X):-        %%% 0-99
   cyfra(S),!,
   cyfra(D),
   cyfra(J),
   X is 100*S+10*D+J.

liczba(X):-        %%% 0-9
   cyfra(S),
   cyfra(D),!
   cyfra(J),
   X is 100*S+10*D+J.
```

Example: definition of a function

```
%%% Function definition: mutually exclusive conditions

f(X,2):- X>=0,X<3.
f(X,4):- X>=3,X<6.
f(X,6):- X>=6,X<9.
f(X,8):- X>=9.

%%% Function definition: preventing backtrackin with cut

fc(X,2):- X>=0,X<3,!.
fc(X,4):- X>=3,X<6,!.
fc(X,6):- X>=6,X<9,!.
fc(X,8):- X>=9.

%%% Function definition: preventing backtracking + optimization

fco(X,2):- X>=0,X<3,!.
fco(X,4):- X<6,!.
fco(X,6):- X<9,!.
fco(X,8):- X>=9.
```

Closed World Assumption

- positive knowledge about the world of interest is stated explicitly; it takes the form of facts and rules (clauses in PROLOG),
- it is assumed that all the positive knowledge is available or can be deduced; in other words, that the world is *closed*,
- if so, if a certain fact does not follow from the current knowledge base, it is assumed to be false,
- it does not mean it is *really false*; the definition is *operational* — it provides a way to decide whether something is false in the case the negation of it is not stated in an explicit way.

Basic concept of negation in PROLOG

1. Negation in PROLOG is based on the *Closed World Assumption*,
2. it is implemented as negation as failure to prove a goal,
3. predicate `not` is in fact a meta-predicate — its argument is a predicate p, and $not(p)$ succeeds if an attempt to prove p fails,
4. `not(p)` — attempts to prove `p`; if the interpreter fails, `not(p)` succeeds.

Syntax

Example use of `not`

```
%%% Large base of positive facts
  pracownik(adam).
  pracownik(bogdan).
  pracownik(czesiek).
  pracownik(damian).
  pracownik(eustachy).
  pracownik(walery).
  pracownik(xawery).

%%% Few exception examples
  bumelka(bogdan).
  bumelka(walery).
  bumelka(damian).

  premia(X):-
     pracownik(X),
     not(bumelka(X)).

%%% Variable instantiation first!!!
   premia_wrong_example(X):-
     not(bumelka(X)),
     pracownik(X).
```

Implementation of not

```
not(P):- call(P),!,fail.
not(_).
```

Use of not

not(goal) should be always invoked with all variables of the `goal` instantiated!

Example use of not

```
%%% Variable instantiation first!!!
   premia_wrong_example(X):-
     not(bumelka(X)),
     pracownik(X).
%%% Explanation of the work of not
   pracowity(X):-
     bumelka(X),!,
     fail.
  pracowity(_).

w:- pracownik(X),write(X),nl, pracowity(X), write(X),nl,fail.
```

Asserting and Retracting knowledge

1. Programs in PROLOG can easily access global memory.
2. There are two basic operations on it — one can *assert* new facts and the other can *retract* them.
3. The facts can be read from any place of the program.
4. The standard predicates *assert/1* and *retract/1* are in fact meta-predicates; their arguments are facts.
5. practical use:

```
    assert(p).

    retract(p).
```

6. The *retract* and *assert* operations produce results which are not removed during backtracking.
7. Predicates `assert` and `retract` can be used in any place of the program
8. At any stage of program execution all the clauses have access to the knowledge contained in the global memory; a useful mechanism for communication among clauses without parameter passing is available in this way.

Examples

```
assert(p(a)).
assert(p(a,f(b),g(a,c))).
assert(p(X,X)).
assert(list,[a,b,c,d,e],5).
assert(register(id_1,signature_1,borrow_date_1,return_date_1)).
```

Cleaning the Database

The predicate which performs clearing of the memory is called *retractall/1*; its definition is as follows:

```
retractall(P):-
   retract(P),
   fail.
retractall(_).
```

`retractall/1` is built-in predicate in SWI-PROLOG.

Dynamic facts declaration

Dynamic facts should be declared as such

```
:- dynamic p/n.
```

Database predictaes

1. `dynamic p/n` – declared a dynamic term p with arity n.
2. `consult(+File)` – reads the file 'File' into the program (A Prolog source code; usually a dynamic database).
3. `assert(+Term)` – asserts `Term` to the database (at the last position).
4. `asserta(+Term)` – asserts `Term` to the database (at the first position).
5. `assertz(+Term)` – asserts `Term` to the database (at the last position).
6. `retract(+Term)` – retracts the first term unifiable with `Term`.
7. `retractall(+Term)` – retracts all the occurrences of terms unifiable with `Term`.
8. `abolish(Functor/Arity)` – Removes all clauses of a predicate with functor Functor and arity Arity from the database.
9. `abolish(+Name, +Arity)` – the same as abolish(Name/Arity).

Syntax

```
:-  op( 50, xfy, :).
route(P1, P2, Day, [P1 / P2 / Fnum / Deptime]):- % Direct flight
  flight( P1, P2, Day, Fnum, Deptime, _).
route(P1, P2, Day, [(P1 / P3 / Fnum1 / Dep1)|RestRoute] ):- %Indirect
  route( P3, P2, Day, RestRoute),
  flight( P1, P3, Day, Fnum1, Dep1, Arr1),
  deptime( RestRoute, Dep2),      % Departure time of Route
  transfer( Arr1, Dep2).          % Enough time for transfer
flight( Place1, Place2, Day, Fnum, Deptime, Arrtime)  :-
   timetable( Place1, Place2, Flightlist),
   member( Deptime / Arrtime / Fnum / Daylist , Flightlist),
   flyday( Day, Daylist).
flyday( Day, Daylist)  :-
   member( Day, Daylist).
flyday( Day, alldays)  :-
   member( Day, [mo,tu,we,th,fr,sa,su] ).
deptime( [ _ / _ / _ / Dep | _], Dep).
transfer( Hours1:Mins1, Hours2:Mins2)  :-
   60 * (Hours2 - Hours1) + Mins2 - Mins1 >= 40.
% A FLIGHT DATABASE
timetable( edinburgh, london,
           [  9:40 / 10:50 / ba4733 / alldays,
             13:40 / 14:50 / ba4773 / alldays,
             19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).
timetable( london, edinburgh,
           [  9:40 / 10:50 / ba4732 / alldays,
             11:40 / 12:50 / ba4752 / alldays,
```

The Zebra Puzzle — Einstein Problem

```
left(L,P,[L,P,_,_,_]).
left(L,P,[_,L,P,_,_]).
left(L,P,[_,_,L,P,_]).
left(L,P,[_,_,_,L,P]).

near(X,Y,L):- left(X,Y,L).
near(X,Y,L):- left(Y,X,L).

einstein(S):-
   S = [[norweg,_,_,_,_],_,[_,_,_,mleko,_],_,_],
   member([anglik,czerwony,_,_,_],S),
   member([szwed,_,psy,_,_],S),
   member([dunczyk,_,_,herbata,_],S),
   left([_,zielony,_,_,_],[_,bialy,_,_,_],S),
   member([_,zielony,_,kawa,_],S),
   member([_,_,ptaki,_,pallmall],S),
   member([_,zolty,_,_,dunhill],S),
   near([_,_,_,_,blends],[_,_,koty,_,_],S),
   near([_,_,konie,_,_],[_,_,_,_,dunhill],S),
   member([_,_,_,piwo,blumaster],S),
   member([niemiec,_,_,_,prince],S),
   near([norweg,_,_,_,_],[_,niebieski,_,_,_],S),
   near([_,_,_,_,blends],[_,_,_,voda,_],S),
   member([_,_,rybki,_,_],S).
```