Programming languages – Prolog

Introductory instruction (2021/22)

T. Goluch

1. Programming environments

The ECL^{i}PS^{e} environment will be used in the project. It is available at: http://eclipseclp.org/. After installing, we will have available two versions:

- **console** (for Windows it is `<installation-directory>\lib\i386_nt\eclipse.exe`). We can add an access path using the command: `PATH = %PATH%;<installation-directory> lib\x86_64_nt` (applies to 64 bit version). Compiling command: `eclipse -f <nazwa-pliku>`. The file should have an extension: .ecl or .pl and exist in current directory. However, the file name can be given without the extension. Nevertheless, after the correct installation of the ECL^{i}PS^{e} environment, it should be enough to double-click .ecl files to run them. After launching, information about the license and the version of the compiler should be displayed, as well as an suffix: `[eclipse 1]: _` after which you can enter subsequent queries. To exit the program and return to the command line, use `halt` instruction.
- **graphical** (TkEclipse). After creating the program file (in your favorite editor or using: *File→Edit new...*) we can compile it using command *File→Compile...* or by typing in *Query Entry* field: `[<filename-without-extension>]`. The file should have an extension: .ecl or .pl and exist in current directory. You can enter the new path of the current directory by selecting the menu option *File→Change Directory...* . enter your queries in *Query Entry* Field. If you want to clean all previously compiled modules, you can use *File→Clear toplevel module* menu option.

The alternative environment is SWI-Prolog. It is available at: https://www.swi-prolog.org/download/stable. After installation a console version is available. The program is compiled in a file with the `.pl` extension using the command:

```
<file_name_without_extension>.
```

In case of failure, check if the file is in the default folder:

```
ls.
```

You can also load the file when running SWI with the command:

```
swipl -s <file_name>
```

Available is also SWISH WEB environment, which is a subset SWI-Prolog. The application is available at: https://swish.swi-prolog.org/. Its advantage is easy accessibility and the ability to save and share code.

2. Logic programming

The PROLOG program consists of a set of clauses definitions. The simplest clause `p/1` definition (`1` means number of clause parameters) has a form:

```
    p(1).
```

It means that this clause is satisfiable (if and only if) for the parameter equal to `1` (and only for it). Please create a new file with the program containing this clause and then compile it. If we now execute an query (text on a black background means queries preceded by `?-` and answers):

```
?- p(X).
```

we will get the answer:

```
X = 1
Yes
```

After executing the query, PROLOG looks through all defined clauses and tries to match the query to them. In this and subsequent examples, please add/update clauses in frames to the program file. For example, when `p` is satisfiable for `1` and `2`:

PROVA 1

```
    p(1).
    p(2).
```

```
?- p(X).
X = 1
Yes (solution 1, maybe more)
X = 2
Yes (solution 2)
```

To obtain further matches, depending on the version, press the semicolon (console version – eclipse.exe) or press *more* button (graphical version – TkEclipse). It is also worth noting that the order of the clauses is significant, replacing it we obtain following answer:

```
    p(2).
    p(1).
```

```
?- p(X).
X = 2
Yes (solution 1, maybe more)
X = 1
Yes (solution 2)
```

General clause definition has form:

```
    predicate(Variables) :- goal.
```

The clause is satisfiable for variables Variables if the goal is satisfiable. Wherein the goal is a list of sub-goals connected by conjunction (as: goal-1, goal-2, ..., goal-n) or an alternative (as: goal-1; goal-2; ...; goal-n). Eg clause:

PROVA 2

```
      q( 0,0 ).
      q( X,Y ) :- X > 0, T is X - 1, q( T,Y ).
```

is satisfiable for parameters 0,0 and for all X,Y such that q(X-1, Y) is satisfiable. A clause is/2 is needed to calculate the value of the expression. Otherwise, to logical variable T would be assigned X – 1 symbolic expression. Clauses form p(X). (i.e. with an empty goals list) we call facts, other clauses we call rules. Now let's ask: q(7,0). It was not possible to match the first clause, but the match to the second will succeed: the variable X will be associated with 7 value, and Y with zero. and then the truth of the sub-goals will be checked. The first two are satisfiable. To check the truth of the third sub-goal, check if q(6,0) is true. There will be another attempt to match the query (this time q(6,0)). It will be true if q(5,0) is true etc., up to the query q(0,0), which is true of "definitions" (first fact). Variable names always begin with a uppercase letter, and predicates with a lowercase one.

3. Debugger

For a better understanding of the program or for an error analysis, you can use the debugger. Choose the option *Tools→Tracer* from the menu and run the query again:

```
q(7,0).
```

In the *Call Stack* window the current goal (blue) and child goals will be displayed. Satisfiable goals are marked in green and not satisfiable in red. To continue tracking, use *Creep* button. All called goals in chronological order can be viewed by browsing the *Trace Log* tab.

4. Data types

In the Prolog, we can distinguish the following data types:

a) **numbers,** we can distinguish four types:

- **integers**, in case of use GMP library, their size is limited only by the device's memory:

```
123 -27 34923748927492749495867593039484746374859589174
```

```
?- X is 123 + 34923748927492749495867593039484746374859589174.
X = 34923748927492749495867593039484746374859589297
```

- **float**, the double type from the C language is used for representation:

```
3.141592653589793 6.02e23 -35e-12 -1.0Inf
```

```
?- X is 3.141592653589793 + 6.02e23.
X = 6.02e+23
```

- **rational**, the value is represented as quotient of two unlimited integers:

```
2_6, rational(2.5), 4_3 + rational(1.5)
2 rdiv 5 or 2r5  in the case  SWI-Prolog
```

```
?- X is 6_1 + 5_8.
X = 53_8
```

or in the case of SWI:

```
X is 6r1 + 5r8.
X = 53r8.
```

An integer and float number can be converted to a rational form using a predicate `rational/1` or `rationalize/1`.

```
?- X is rational(1).
X = 1_1
```

```
?- X is rationalize(1.5).
X = 3_2
```

In the case of the `rational/1` the result of such a conversion may be surprising, due to the fact that the number saved in binary form is converted. The example below shows that the number `1.1` does not have a finite representation in the form of a binary fraction.

```
?- X is rational(1.1).
X = 2476979795053773_2251799813685248
```

```
?- X is rationalize(1.1).
X = 11_10
```

- **bounded reals**, real number that is contained between two floating point values:

```
1.5__2.0, 1.0__1.0, 3.1415926535897927__3.1415926535897936
```

```
?- X is 1.5__2.0 + 1.0__1.0.
X = 2.5__3.0
```

b) **strings,** are a representation of any sequence of bytes and are written in quotes:

```
"hello"
"I am a string!"
"string with a newline \n and a null \000 character"
```

```
?- string_concat("butter", "fly", L).
L = "butterfly"
```

c) **atoms**, it's simple symbolic constants (similar to enumeration types from other languages). Syntactically, all words starting with a lowercase letter mean atoms, atoms are also symbols sequences and anything in apostrophes:

```
atom  quark  i486  -*-  ??? [] 'Atom' 'an atom'
```

```
?- atom(atom).
Yes
```

```
?- atom(123).
No
```

d) **lists**, it is an ordered sequence of elements contained between square brackets and separated by commas, eg: [1,2,3,5A special case is an empty list, marked with: [].Each non-empty list is constructed from the head (first element) and tail (list of other items, potentially empty). The construction of the list from H head and tail T has the form: [H|T]. Thus, the list [1,2,3] can be saved in one of the equivalent ways:

```
[1,2,3]
[1|[ 2,3 ]]
[1|[2|[3]]]
[1|[2|[3|[]]]]
.(1, .(2,[]))
```

```
?- is_list([1|[2|[3|[]]]])
Yes
```

An example predicate, designating the length of the list has the form:

```
length( [],0 ).
length( [_|T],L ) :- length( T,P ), L is P + 1.
```

```
?- length([1, 2, 3, 4], L).
L = 4
```

The length of the empty list is zero, and the non-empty list is one more than the length of its tail. _ means that we do not associate the heads of the list with any logical variable. Another example – attach one list at the end of the other:

```
attach( [],L,L ).
attach( [H|T],L,[H|X] ) :- attach( T,L,X ).
```

List L attached to an empty list gives list L. The list L attached at the end of the list consisting of the head H and the tail T forms a list built from the head H and the tail X resulting from attaching to the end T the list L. Another example – check if X is equal to one of the list items:

```
?- attach([1, 2], [3, 4], N).
```

```
N = [1, 2, 3, 4]
```

Prova5.pl

```
      belongs( X,[X|_] ).
      belongs( X,[H|T] ) :- belongs( X,T ).
```

```
?- belongs(1, [1, 2, 3, 4]).
Yes (solution 1, maybe more)
```

e) **structures**, is a named set with a fixed number of arguments with the following syntax:

```
<name>(<arg>1,...<arg>n)
```

examples of structure:

Prova6.pl

```
      date(december, 25, "holiday").
      element(hydrogen, composition(1,0)).
      flight(london, new_York, 12.05, 17.55).
```

```
?- date(december, 25, X).
X = "holiday "
```

```
?- element(hydrogen, X).
X = composition(1, 0)
```

```
?- element(hydrogen, composition(P, N)).
P = 1
N = 0
```

```
?- flight(From, Dest, Beg, End).
From = londyn
Dest = nowy_york
Beg = 12.05
End = 17.55
```

5. Operators

NON HAI CAPITO

In the Prolog, we can easily get a list of all 87 operators along with information on their precedence and commutativity. Just do the following query:

```
current_op(Precedence, Type, Name).
```

where:

- `Precedence` – an integer from the range of 0..1200, a higher number means a higher precedence,
- `Type` – determines how many arguments the operator is and whether it is cumulative. In addition, in the case of binary operators, if it is cumulative, then either right or left side. Eg. yfx – is a two-argument operator (infix) left-sided where:

  - f – operator's place,
  - y – the side from which the expression executing will be started,
  - x – no connectivity from this side.
- `Name` – operator's symbol.

```
?- current_op(Precedence, Type, Name).
Precedence = 1100
Type = xfy
Name = ;
Yes (solution 1, maybe more)
```

If you want for example to ask for all (unary) integer prefix operators, run the query:

```
?- current_op(Precedence, fy, Nazwa).
Precedence = 1000
Nazwa = local
Yes (solution 1, maybe more)
```

In particular, knowing the operator's symbol, we can get additional information, e.g.:

```
?- current_op(Precedence, Type, :-).
Precedence = 1200
Type = xfx
Yes (solution 1, maybe more)
Precedence = 1200
Type = fx
Yes (solution 2)
```

6. Comments

Comments in Prolog are contained between /* i */ (like in C) or between % and the end of the line, e.g.:

```
?- p(1). /* this is a comment */
?- p(2). % this is a comment too
```

7. Backtracking predicate `fail/0`and cut operator `!/0`

NON HAI CAPITO IL FUNZIONAMENTO

Executing `fail` predicate always fails and forces backtracking. Cut operator `!` divides the clause into two parts and does not allow re-execution (backtracking) to the left part.

```
      different(X, Y) :- X=Y, !, fail.
      different(_, _).
```

```
?- different(a, a).
No
?- different(a, b).
Yes
```

Consider the example above. If the variables X and Y are different then the first sub-goal will not be satisfiable and therefore the match of the next predicate will be executed, which for any two variables is always satisfiable. The answer will be correct – Yes. In the case when the variables will be different, the first sub-target will be satisfiable but the second one will always be unsatisfiable. However, the cut operator will not allow you to return to the left part of the clause and execute the predicate again.

8. Metapredicates

CAPITO POCHISSIMO

Metapredicates family `maplist/N` applies a passed-in goal (usually a predicate) to each item in N passed-in lists.

```
?- maplist(=(E), Ls).
E = E
Ls = []
Ls = [E]
Ls = [E, E]
...
```

```
?- maplist(=(E), [2, 2, 2]).
E = 2
Yes
?- maplist(=(E), [2, 3]).
No
```

```
?- maplist(plus(2), [1, 2, 3, 4, 5], [3, 4, 5, 6, 6]).
No (0.00s cpu)
```

```
?- maplist(plus(2), [1, 2, 3, 4, 5], L).
L = [3, 4, 5, 6, 7]
```

```
int_succ(A, B) :- A is B + 1.
```

```
?- maplist(int_succ, I, [1, 2, 3]).
I = [2, 3, 4]
```

```
mul(N,R) :- R is N*N.
```

```
?- maplist(maplist(mul), [[1, 2, 3], [3, 4, 5]], Rss).
Rss = [[1, 4, 9], [9, 16, 25]]
```

More complex metapredicates defined only in SWISH.

```
int_sum(A, B, C) :- A is B + C.
```

```
?- maplist(int_sum, I, [1, 2, 3], [2, 3, 4]).
I = [3, 5, 7]
```

Metapredicates family `foldl/N`.

```
     maxL( [H|T], M ) :- foldl( max_, T, H, M ).
     max_( A, B, C ) :- C is max( A, B ).
```

```
?- maxL([3,7,5], M).
M = 7
```

9. Constraint Logic Programming CLP

To use CLP, you must import the ic library using the command (at the beginning of the program):

```
:- lib(ic). % The ECLiPSe environment only
```

And we can solve simple equations:

```
?- 4 #= 2*Y
Y = 2
```

or

```
?- X #= 2*16.
X = 32.
```

A typical solution to a problem using CLP has form:

```
solve(Variables):- load_data(Data),
set_ constraints(Data, Variables),
labeling(Variables).
```

where `set_ constraints/2` defines the problem model. Predicate `labeling/1` tries to find solutions by checking all substitutions for variables. Consider the following problem in which we have 8 variables: S, E, N, D, M, O, R, Y, each of them means a different number, and the following equality is fulfilled:

```
  S E N D
+ M O R E
-----------
= M O N E Y
```

The solution to this problem is:

```
:- lib(ic).
model(Variables) :-
Variables = [S,E,N,D,M,O,R,Y],
Variables :: 0..9,
alldifferent(Variables),
S #\= 0,
    M #\= 0,
    1000 * S + 100 * E + 10 * N + D +
    1000 * M + 100 * O + 10 * R + E #=
    10000 * M + 1000 * O + 100 * N + 10 * E + Y.
    solve(Variables) :- model(Variables), labeling(Variables).
```

```
?- solve (Variables).
Variables = [9, 5, 6, 7, 1, 0, 8, 2]
Yes
```

Predicate `model` first creates a list of variables in our problem. In the second step, we assign to all variables (using `::/2` predicate) as a domain a set of natural numbers in the range <0, 9>. To create a domain that is a subset of real numbers, the range 0..9 should be changed to 0.0..9.0 or use `$::/2` predicate. In the next line, we specify that all variables must be different. Finally, we conclude that neither S nor M can be equal to zero (restrictions expressed by comparison operators are created by adding # sign in front of the operator) and describe our equation.

```
    :- use_module(library(clpfd)).  % Finite domain constraints

    sudoku(Rows) :-
            length(Rows, 9), maplist(same_length(Rows), Rows),
            append(Rows, Vs), Vs ins 1..9,
            maplist(all_distinct, Rows),
            transpose(Rows, Columns),
            maplist(all_distinct, Columns),
            Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
            blocks(As, Bs, Cs),
            blocks(Ds, Es, Fs),
            blocks(Gs, Hs, Is).

    blocks([], [], []).
    blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
            all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
            blocks(Ns1, Ns2, Ns3).

    problem(1, [[_,_,_,_,_,_,_,_,_],
                [_,_,_,_,_,3,_,8,5],
                [_,_,1,_,2,_,_,_,_],
                [_,_,_,5,_,7,_,_,_],
                [_,_,4,_,_,_,1,_,_],
                [_,9,_,_,_,_,_,_,_],
                [5,_,_,_,_,_,_,7,3],
                [_,_,2,_,1,_,_,_,_],
                [_,_,_,_,4,_,_,_,9]]).
```

```
problem(1, Rows), sudoku(Rows), maplist(portray_clause, Rows).
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1].
[2, 4, 6, 1, 7, 3, 9, 8, 5].
```

```
[3, 5, 1, 9, 2, 8, 7, 4, 6].
[1, 2, 8, 5, 3, 7, 6, 9, 4].
[6, 3, 4, 8, 9, 2, 1, 5, 7].
[7, 9, 5, 4, 6, 1, 8, 3, 2].
[5, 1, 9, 2, 8, 6, 4, 7, 3].
[4, 7, 2, 3, 1, 9, 5, 6, 8].
[8, 6, 3, 7, 4, 5, 2, 1, 9].
```

```
Rows = [[9, 8, 7, 6, 5, 4, 3, 2, 1], [2, 4, 6, 1, 7, 3, 9, 8, 5], [3, 5,
1, 9, 2, 8, 7, 4, 6], [1, 2, 8, 5, 3, 7, 6, 9, 4], [6, 3, 4, 8, 9, 2, 1,
5, 7], [7, 9, 5, 4, 6, 1, 8, 3, 2], [5, 1, 9, 2, 8, 6, 4, 7, 3], [4, 7, 2
, 3, 1, 9, 5, 6, 8], [8, 6, 3, 7, 4, 5, 2, 1, 9]]
```

10. Simplified NIM game

In the prologue you can also get answers to questions about the winning strategy of simple logic games. Take, for example, the simplified version of the NIM game. This is a deterministic game for 2 players with no hidden information and zero sum. The simplified variant assumes that we start with 3 piles of stones (described by three variables of the predicate is_winner/ 3). The player wins if his opponent cannot move. The move removes one and only one stone from any of the 3 piles. Below is the program that answers this question:

```
    is_winner(0,0,0) :- fail.
    is_winner(A,B,C) :- X is A - 1, X >= 0, not(is_winner(X,B,C)).
    is_winner(A,B,C) :- X is B - 1, X >= 0, not(is_winner(A,X,C)).
    is_winner(A,B,C) :- X is C - 1, X >= 0, not(is_winner(A,B,X)).
```

11. Literature

http://eclipseclp.org/

ECL^{i}PS^e – A Tutorial Introduction (http://eclipseclp.org/doc/tutorial.pdf)

ECL^{i}PS^e – User Manual (http://eclipseclp.org/doc/userman.pdf)