IN112 Mathematical Logic
Lab session on Prolog

Christophe Garion

DMIA – ISAE

License CC BY-NC-SA 3.0

Outline

1. Prolog: getting started!
2. Interpreting and querying
3. Unification and assignement
4. Lists
5. Negation by failure
6. Misc.

Outline

1. **Prolog: getting started!**
2. Interpreting and querying
3. Unification and assignement
4. Lists
5. Negation by failure
6. Misc.

Prolog interpreter

Prolog is an **interpreted** language:

- no executable is created (even if it is possible)
- a Prolog program is loaded into the interpreter
- queries are executed on this program through the interpreter

We will use the GNU Prolog interpreter.

Diaz, D. (2013a).
GNU Prolog.
http://www.gprolog.org.

— (2013b).
GNU Prolog Manual.
http://www.gprolog.org/manual/html_node/index.html.

Starting GNU Prolog and queries

Starting the interpreter:

shell

```
    c.garion@chabichou# gprolog
    GNU Prolog 1.3.1
    By Daniel Diaz
    Copyright (C) 1999-2009 Daniel Diaz
    | ?-
```

The `| ?-` prompt waits for a query.

To escape from the interpretor, the `halt` predicate is used:

```
    |?- halt.
```

Prolog syntax: a (concise) overview

Syntax (case)

Case is important in Prolog: identifiers beginning by an **uppercase letter** are **variables** names.

Examples: in `p(X,f(Y),a)`

- `p` is a predicate name
- `X` and `Y` are variable names
- `f` is a function name
- a is a constant name

...but `'P'('My_Constant')` is also valid!

N.B.

When speaking of a predicate, its arity is given (`ancestor/2` for instance).

Prolog syntax: a (concise) overview

Syntax (clauses)

A clause is written like this:

```
A :- B1,...,Bn.
```

where `A` is the head of the clause and `B1,...,Bn` its body.
When a clause is written `A :- `**`true`**`.`, it simply noted `A.` Such a clause is called a **fact**.
Do not forget the "." at the end!

Examples:

- `jack` is a parent of `mary`
  ➥ `parent(jack, mary).`
- for all `X` and `Y`, if there exists a `Z` such that `X` is an ancestor of `Z` and `Z` is a parent of `Y`, then `X` is an ancestor of `Z`
  ➥ `ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).`

Prolog syntax: a (concise) overview

Syntax (comments)

Comments are written like this:

```
/* Comments */
```

N.B.

Types cannot be declared explicitely.

A Prolog program for ancestors. . .

We will consider the following program:

ancestors.pl

```
/**************************/
/* Definition of parent/2 */
/**************************/
parent(jack, mary).
parent(louise, jack).
parent(franck, john).

/***************************/
/* Definition of ancestor/2 */
/***************************/
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).
```

Emacs mode for Prolog

There is a major mode for Prolog in Emacs.

Astrom, A., M. Zamazal, and S. Bruda.
EMACS major mode for Prolog.
http://turing.ubishops.ca/home/bruda/emacs-prolog/.

Add the following lines into your `.emacs`:

.emacs

```
(add-to-list ’load-path "/chemin/vers/repertoire/contenant/le/fichier/")
(autoload ’run-prolog "prolog" "Start a Prolog sub-process." t)
(autoload ’prolog-mode "prolog" "Major mode for Prolog programs." t)
(autoload ’mercury-mode "prolog" "Major mode for Mercury programs." t)
(setq prolog-system ’gnu)
(setq auto-mode-alist (append ’(("\\.pl$" . prolog-mode)
                                ("\\.m$" . mercury-mode))
                                auto-mode-alist))
```

This mode provides syntax highlighting + interpreter access.

Easy to use

The command line interpreter allows to:

- navigate into queries history with ↑ and ↓
- dynamically complete a predicate name with ⟶

From Emacs:

- history access with `M-p` and `M-p`
- a predicate, a region or a buffer can be evaluated
- predicate definitions can be retrieved
- predicate template insertion with `C-c C-n`
- ...

Outline

1. Prolog: getting started!
2. **Interpreting and querying**
3. Unification and assignement
4. Lists
5. Negation by failure
6. Misc.

Interpret a program

From interpreter:

```
| ?- ['/home/tof/Cours/IN112/exempleProlog/ancestors.pl'].
compiling /home/tof/Cours/IN112/exempleProlog/ancetres.pl for byte code...
/home/tof/Cours/IN112/exempleProlog/ancetres.pl compiled,
  12 lines read - 898 bytes written, 62 ms

(2 ms) yes
```

- do not forget the final “`.`”
- filename protection with “`''`”
- Prolog answers `yes`: the predicate **`consult`**/1 is evaluated!
- beware when reloading programs, verify with **`listing`** the predicates definitions

From Emacs, open the file and `C-c C-b`.

Query evaluation

Query evaluation:

```
| ?- ancestor(jack,mary).

true ?
```

The "?" symbol signifies that Prolog waits for a user command:

- ; to ask for the next solution (backtracking)
- a to ask for all solutions
- RETURN to stop

In our case, after "**;**":

```
Fatal Error: local stack overflow (size: 8192 Kb,
             environment variable used: LOCALSZ)
```

Query evaluation

After having corrected the program:

```
| ?- ancestor(W,mary).

W = jack ? a

W = louise

no
```

Evaluation with failure:

```
| ?- ancestor(john,jack).

no
```

Outline

1. Prolog: getting started!
2. Interpreting and querying
3. **Unification and assignement**
4. Lists
5. Negation by failure
6. Misc.

Arithmetics

Arithmetics is available in GNU Prolog.

- +, *, -, /
- `min`, `max`
- `=:=`, =/= (cf. next slides)
- <, >, =<, >=
- ...

All those operators are **infix**.

N.B. (important)

The operands of those operators must be **evaluable**.

Unification, assignment, equality: try it!

Unification operator: =

```
X = Y.
X = Y, f(Y) = Z.
f(X) = g(Y).
f(X) \= g(a).
```

Assignment operator: **is**

```
X is 2.
X is (2 + 3).
X is Y.
Y = 2, X is Y.
```

Terms equality operator: ==

```
X == X.
X == Y.
X \== Y.
2 == (1 + 1).
```

Arithmetic equality: =:=

```
2 =:= 2.
2 =:= (1 + 1).
2 =:= 3.
2 =\= 3.
```

Unification, assignement: example

Question

How to define a fact/2 predicate which represents the factorial function?

Using logic:

Factorial definition

$$\begin{array}{l} fact(0,1)\wedge \\ \forall x \forall y\ fact(x-1,y) \rightarrow fact(x, y*x) \end{array}$$

Unification, assignement: example

fact-1.pl

```
fact(0,1).
fact(N,Y * N) :- fact(N - 1,Y).
```

Trying with 0!:

```
| ?- fact(0,X).

X = 1 ? ;

Fatal Error: global stack overflow (size: 32768 Kb,
                          environment variable used: GLOBALSZ)
```

Unification, assignement: example

fact-1.pl

```
fact(0,1).
fact(N,Y * N) :- fact(N - 1,Y).
```

Explanation (with **trace.**, remove trace mode with `notrace.`):

```
    1    1  Redo: fact(0,1) ?
    2    2  Call: fact(0-1,_48) ?
    3    3  Call: fact(0-1-1,_78) ?
    4    4  Call: fact(0-1-1-1,_108) ?
    5    5  Call: fact(0-1-1-1-1,_138) ?
    ...
```

Unification, assignement: example

fact-2.pl

```
fact(0,1).
fact(N,Y * N) :- N > 0, fact(N - 1, Y).
```

Trying 0!:

```
| ?- fact(0,X).

X = 1 ? ;

no
```

Unification, assignement: example

fact-2.pl

```
fact(0,1).
fact(N,Y * N) :- N > 0, fact(N - 1, Y).
```

Trying 3!:

```
| ?- fact(3,X).

no
```

Unification, assignement: example

fact-2.pl

```
fact(0,1).
fact(N,Y * N) :- N > 0, fact(N - 1, Y).
```

Explanation (with **trace.**):

```
1    1  Call: fact(3,_16) ?
2    2  Call: 3>0 ?
2    2  Exit: 3>0 ?
3    2  Call: fact(3-1,_48) ?
4    3  Call: 3-1>0 ?
4    3  Exit: 3-1>0 ?
...
8    5  Call: 3-1-1-1>0 ?
8    5  Fail: 3-1-1-1>0 ?
7    4  Fail: fact(3-1-1-1,_158) ?
5    3  Fail: fact(3-1-1,_103) ?
3    2  Fail: fact(3-1,_48) ?
1    1  Fail: fact(3,_16) ?
```

Unification, assignement: example

fact-2.pl

```
fact(0,1).
fact(N,Y * N) :- N > 0, fact(N - 1, Y).
```

Explanation (why `fact(0,1)` is not used):

```
| ?- 3-1-1-1 = 0.

no
```

Unification, assignement: example

fact-3.pl

```
fact(0,1).
fact(N, Y * N) :- N > 0, M = N - 1, fact(M, Y).
```

Trying 3!:

```
| ?- fact(3,X).
fact(3,X).

no
```

Unification, assignement: example

fact-3.pl

```
fact(0,1).
fact(N, Y * N) :- N > 0, M = N - 1, fact(M, Y).
```

Explanation (with **trace.**):

```
1    1  Call: fact(3,_16) ?
2    2  Call: 3>0 ?
2    2  Exit: 3>0 ?
3    2  Call: fact(3-1,_48) ?
4    3  Call: 3-1>0 ?
4    3  Exit: 3-1>0 ?
...
8    5  Fail: 3-1-1-1>0 ?
7    4  Fail: fact(3-1-1-1,_158) ?
5    3  Fail: fact(3-1-1,_103) ?
3    2  Fail: fact(3-1,_48) ?
1    1  Fail: fact(3,_16) ?
```

Unification, assignement: example

fact-3.pl

```
fact(0,1).
fact(N, Y * N) :- N > 0, M = N - 1, fact(M, Y).
```

Explanation (why `fact(0,1)` is not used):

```
| ?- M = 3 - 1.
M = 3-1

yes
```

Unification, assignement: example

fact-4.pl

```
fact(0,1).
fact(N, Y * N) :- N > 0, M is N - 1, fact(M, Y).
```

Trying 3!:

```
| ?- fact(3,X).

X = 1*1*2*3 ? ;

no
```

fact-5.pl (finally!)

```
fact(0,1).
fact(N, Y) :- N > 0, M is N - 1, fact(M, X), Y is X * N.
```

Outline

1. Prolog: getting started!
2. Interpreting and querying
3. Unification and assignement
4. **Lists**
5. Negation by failure
6. Misc.

Lists representation

Lists are basic data structures.

Building a new list **by induction**:

Syntax (list)

- empty list: `[]`
- or the list is composed of:
  - an **element**, the **head** of the list
  - a **list**, the **tail** of the list

  The list is then represented by `[head | tail]`.

A list containing **known elements** is represented with the ",": `[a, b, c]`.

Lists representation

Lists are basic data structures.

Building a new list **by induction**:

Syntax (list)

- empty list: `[]`
- or the list is composed of:
  - an **element**, the **head** of the list
  - a **list**, the **tail** of the list

  The list is then represented by `[head | tail]`.

Examples to evaluate:

```
[T | Q] = [a, b, c].
[T, Q] = [a, b, c].
[T, Q, R] = [a, b, c].
[T | Q] = [a].
[T] = [a].
```

Operations on lists

Avalaible operations on lists:

- `member/2` (e.g. `member(X, [a,b,c]).`)
- `append/3` (e.g. `append([d, e], [a,b,c], X).`)
- `reverse/2`
- `last/2`
- **`length`**`/2`
- `sort/2`
- ...

Outline

1. Prolog: getting started!
2. Interpreting and querying
3. Unification and assignement
4. Lists
5. **Negation by failure**
6. Misc.

Negation by failure

Negation does not exist in Prolog: only Horn clauses are used!

Syntax (negation by failure)

```
\+ (goal)
```

This operator represents the fact that **Prolog cannot prove `goal`**.

Try the following examples:

```
\+(X == Y).
\+(member(a, [b,c])).
```

NBF: the ancestors example

Add the following predicate: X is not a direct ancestor of Y if X is an ancestor of Y and X is not a parent of Y.

ancestor.pl

```
not_direct_ancestor(X, Y) :- ancestor(X, Y), \+(parent(X, Y)).
```

Try it!

NBF is not logical negation!

In classical logic:

$$\models A \land \neg B \leftrightarrow \neg B \land A$$

…but not with NBF!

Try the following definition for `not_direct_ancestor`:

ancestor.pl

```
not_direct_ancestor(X, Y) :- \+(parent(X, Y)), ancestor(X, Y).
```

Outline

1. Prolog: getting started!
2. Interpreting and querying
3. Unification and assignement
4. Lists
5. Negation by failure
6. **Misc.**

Misc.

`listing` is used to obtain a predicate definition or all program predicates definition.

ex: **`listing`**`(ancestor).`

Normally, a predicate should be defined contiguously. You can add the following line **at the beginning of the file**:

```
:- discontiguous(pred/n).
```

where `pred` is the predicate.

If a predicate is defined in more than one file, you should add the following in all files preambles:

```
:- multifile(pred/n).
```