Practical Work of Languages, Technologies, and Paradigms of Programming Part III Logic Programming 2018-2019

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Practice 7: Introduction to Prolog

Contents

1 Objetives of this practical session

The objectives are twofold:

1. To introduce the use of a logic language (Prolog) to represent a knowledge base and to query that database.
2. To deepen in two basic concepts of this logical paradigm that allow solving queries: the concepts of *unification* and *backtracking* as well as to introduce a simple mechanism for performing simple arithmetic computations.

2 SWI-Prolog

SWI-Prolog is an open source implementation of the Prolog programming language. The name derives from Sociaal-Wetenschappelijke Informatica, the ancient name of a research group at the University of Amsterdam where its development began.

2.1 Loading and executing programs

SWI-Prolog is already installed in the practice labs and is available from any folder by typing the `swipl` command from a Linux shell.

Open a terminal and create a folder for working in this practice (for example, `cd DiscoW/LTP/pr7`). From Linux, open a shell in this folder and invoke SWI-Prolog by typing the `swipl` command:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.6.1)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

The system executes an interpreter which expects queries to the knowledge base (Prolog program) loaded into memory. In SWI-Prolog, these queries can be performed using an interactive shell whose prompt is `?-`. From this interactive shell, we can load programs saved in a file. To load a program we can use the command `consult`(filename), `compile`(filename) or [filename] followed by a period, where *filename* is the name of the file **excluding the extension**. It is possible to exit from the interpreter by means of the `halt.` command or by pressing `CTRL + D`.

Exercise 1 *Load the program saved in the file* `family.pl` *(available at* PoliformaT*) using any of the previous load instructions. Do not forget that all queries must end with a period.*

You have to obtain the following output:

```
consult(family).
% family compiled 0.00 sec, 13 clauses
true.
```

We can consult the loaded predicates by means of the `listing` command. We can also use it for some predicate in particular, for instance, `listing(isparent)`.

Exercise 2 *Execute* `listing` *from the* `swipl` *interpreter and list the knowledge base for several of the predicates loaded into memory.*
You should obtain something like this:

```
?- listing.

:- thread_local thread_message_hook/3.
:- dynamic thread_message_hook/3.
:- volatile thread_message_hook/3.

isrelative(A, B) :-
        isparent(A, B).
isrelative(A, B) :-
        ischild(A, B).
isrelative(A, B) :-
        issibling(A, B).
isrelative(A, B) :-
        isgrandparent(A, B).

isgrandparent(A, C) :-
        isparent(A, B),
        isparent(B, C).

issibling(A, C) :-
        isparent(B, A),
        isparent(B, C),
        A\==C.

isparent(juan, maria).
isparent(pablo, juan).
isparent(pablo, marcela).
isparent(carlos, debora).
isparent(luisa, debora).

ischild(B, A) :-
        isparent(A, B).
true.
```

It is important to remark that the use of lowercase and uppercase letters is relevant. Variables start by uppercase or by the underline symbol "_". For instance, `A` is a variable whereas `juan` is not. Moreover, we can remark that a program is made from:

- facts, for instance `isparent(pablo,juan).`
- rules, for instance `ischild(B, A) :- isparent(A, B).`

The predicate `isparent(pablo,juan)` expresses the fact that `pablo` is *the father of* `juan`. The definition of this predicate is *extensional*, meaning that those facts appear explicitly in the program.

The predicate `ischild(B,A)` is defined in an *intensional* way by means of a rule:

```
ischild(B,A) :- isparent(A,B).
```

which states that *"is son or daughter"* `B` of `A` *if "is mother or father"* `A` *of* `B`. Otherwise stated, we can solve the problem of knowing who is the child of anyone by means of the `isparent` relation. Observe that the conditional *if* from the former sentence is expressed in the program by using the symbol `:-`.
The predicate `isrelative(A, B)` is defined in an intensional way by using 4 rule clauses stating that `A` is a relative of `B` *if* `A` is parent, child, sibling or grandparent of `B`. Observe that the use of several rules with the same head (`isrelative(A, B)` in this example) is a simple way of expressing disjunction in Prolog.
Finally, the definition of the `issibling` predicate contains three predicates separated by commas in the body of the rule (after `:-`). The commas are used to represent the conjunction of predicates, which means that all of them must return a successful result to fulfill the rule.

2.2 Queries to facts

Now, let us perform several queries to the knowledge database loaded into memory. These queries can be directly performed by means of the interpreter. We will start by making simple queries relating facts described in the program. For example, we can ask if `pablo` is the father of `juan` as follows:

```
?- isparent(pablo,juan).
true.
```

Exercise 3 *What happens when you ask if* `juan` *is father of* `pablo`*? Enter that query in the interpreter and explain the obtained response.*

The information obtained in response in the former queries was always a statement (`true`) or a negation (`false`) about the existence of a proof for the query. Queries can have variables in order to extract more information from the knowledge base. For example, suppose we want to know if `maria` has a parent and, if so, find out who he or she is. We can launch the same query as before, but replacing the first argument with a variable `X`. The variable will be assigned a value making true the query.

```
?- isparent(X,maria).
X = juan.
```

That is, the interpreter can demonstrate that `isparent(X,maria)` is true from the facts stored in the program if `X=juan`.

When there is more than one affirmative answer to a query, the system returns the first that is computed. If we want to receive more answers, we must press *r* (*redo*) in order to know other responses. Test it by performing the following exercise.

Exercise 4 *Ask who are the children of* `pablo` *by replacing, in the query of the former example, the variable* `X` *by the constant* `pablo` *and the constant* `maria` *by a variable (for example,* `X`*). What happens when pressing the* `r` *key after the first response?*

The following execution illustrates the use of two variables to extract, from the database, all the pairs `(X,Y)` where `X` *is mother or father of* `Y` (the semicolon symbol appear in the terminal after pressing the keys `;` or `r`):

```
?- isparent(X,Y).
X = juan,
Y = maria ;
X = pablo,
Y = juan ;
X = pablo,
Y = marcela ;
X = carlos,
Y = debora.
X = luisa,
Y = debora.
```

Exercise 5 *Try the query* `isparent(X,X).` *What answer is obtained? Explain it.*

In this section we have addressed queries which only required to *search* if a pair of terms (`juan`, `maria`, `pablo`...) were related by means of a fact (`iffather`). When using variables, they were instantiated to the terms which made the fact true.

2.3 Queries to rules

Now, let's perform queries requiring to *infer* information that was not expressed as facts contained in the program, but which can be obtained from the combination of facts and rules. Rules are of the form "If the antecedent (body) is true, then it is also true the consequent (head)". For instance, the rule `sonof(A,B) :- isparent(B,A).` expresses that if `B` is parent of `A` then `A` is a child of `B`. Using this rule, the interpreter can deduce that `maria` is a daughter of `juan` after the following query:

```
?- ischild(maria,juan).
true.
```

it is also possible to ask who are the parents of `maria`:

```
?- ischild(maria,X).
X = juan.
```

or who are the parents of Débora:

```
?- ischild(debora,X).
X = carlos ;
X = luisa.
```

None of the facts `sonof(maria,juan)`, `sonof(debora,carlos)` or `sonof(debora,luisa)` are explicitly stored in the program. However, the interpreter is able to conclude that they are deductible from the facts and from the program rules.

Exercise 6 *Making use of* `family.pl`*:*

- *Query who are the grandparents of* `maria`*.*
- *Query all the persons constituting the known family of* `pablo`*.*
- *Who are the persons who are family of Débora?*

- *Using a text editor, add a rule to the* `family.pl` *file defining the* `isgrandchild` *relationship similar to the* `ischild` *but using the* `isgrandparent` *predicate instead of* `isparent`*. Store the changes and reload the file using* `reconsult(family)`*. Now, try to determine who is the grandfather of* `maria` *by using the predicate* `isgrandchild`*.*
- *Launch the query* `isrelative(A,B)`*. Include the* `isrelative(A,B) :- grandsonof(A,B)` *rule into the program and reconsult. How do the two responses differ?*

3 Two basic concepts in Prolog

Each paradigm makes use of different concepts in its operational semantics. Thus, the operational semantics of imperative languages is based on the change of the state. In the functional programming paradigm, pattern matching and term reduction are two basic concepts. In logic languages, unification and backtracking constitute the two main concepts. This section is devoted to the study of these concepts.

3.1 Term unification

As we saw in the previous section, Prolog programs consist of definitions of two types of predicates: facts and rules. As with functional languages, logic languages also contain terms. In logic languages, terms are not reduced using a set of equations. However, in logic languages, variables can appear in query terms.[1] This allows the retrieval of information from the parameters of predicates. Term unification makes it possible the reversibility of the flow information of parameters. That is, parameters can be used to input or to output information in a query, even in the same query.

We have studied an algorithm, in the theory lectures, to determine whether or not two terms with variables can be unified and, in case they can, to return the (most general) unifier. This algorithm can be seen as an extension of the pattern matching used in functional languages which also allows instantiating the variables of the query term.

Two terms containing variables can be unified if they have identical structure when the appropriate values to their variables is given. That is, they have the same names in the same places after instantiating the variables. Variables which appear in the same positions in both terms do not need to be instantiated and are considered identical or equivalent. Let us see some examples:

- `date(10,nov,2030)` and `date(10,nov,2030)` unify because they are identical.
- `date(10,nov,2030)` and `date(X,nov,2030)` unify if `X` = 10.
- `date(10,nov,2030)` and `date(X,nov,X)` do not unify since `X` cannot be 10 and 2030 at the same time.
- `date(10,nov,2030)` and `time(13,05)` do not unify because `date` and `time` are different functors.[2]
- `X` and `time(13,05)` unify if `X` = `time(13,05)`.
- `X` y `date(10,nov,Y)` unifican si `X` = `date(10,nov,Y)`.

[1] A classification of these terms appears in Appendix A.
[2] A functor is the name of a predicate or a function.

- `X` and `date(10,nov,Y)` unify if `X` = `date(10,nov,Y)`.
- `date(10,oct,2030)` and `date(X,nov,2030)` do not unify since months are different.
- terms `moment(date(10,nov,2030),Y)` and `moment(X,time(13,05))` unify if `X` = `date(10,nov,2030)` and `Y = time(13,05)`.
- terms `moment(time(Time,Minutes))` and `moment(time(13,05))` unify if `Time` = 13 and `Minutes` = 05.

In order to unify two predicates, they must have the same name, the same arity and should unify each of the terms of its parameters. For example:

- facts `isparent(X,debora)` and `isparent(carlos,debora)` unify if `X` = `carlos`.
- facts `isparent(X,debora)` and `isparent(luisa,debora)` unify if `X` = `luisa`.
- facts `isparent(X,debora)` and `ischild(X,debora)` do not unify because the names of the predicates are different.
- facts `isparent(X,debora)` and `isparent(luisa,Y)` unify if `X` = `luisa` and `Y` = `debora`.

Exercise 7 *Check at least one of the statements made in the above example running the unification algorithm of the SWI-Prolog interpreter. In order to do that, select pairs of terms and place them to the left and in the right part of the equal sign as in the following example:*

```
date(10,nov,2030) = date(X,nov,2030).
isgrandchild(time(Time,moment)) = moment(time(13,05)).
```

Exercise 8 *Make the same as in previous exercise but using predicates. For instance:*
`isparent(X,debora) = isparent(luisa,Y).`

Unification with composed terms

Now we will try the unification with predicates containing composed terms in their arguments. To this end, the file `flights.pl`, available at *PoliformaT*, will be used. This file contains information about flights. For each direct flight there is a fact represented by the predicate `flight` with eight parameters: origin, destiny, departure time, departure hour, arrival date, arrival time, duration and price. All the dates are represented by means of the functor `date` which groups three terms as arguments: the day, the month and the year (for instance: `date(10,nov,2030)`). All the parameters representing a time are described with the functor `time` with two arguments (hour and minutes). An example of a complete fact representing a file is as follows:

```
flight(barcelona,madrid,
       date(10,nov,2030),time(13,05),
       date(10,nov,2030),time(15,05),
       120,80).
```

where you can observe that there is a direct flight from Barcelona to Madrid with departure on November 10, 2030 at 13:05 and arrival on the same day at 15:05 with a duration of 120 minutes at a price of 80 euros. The program contains the following five facts:

```
flight(barcelona,madrid,
       date(10,nov,2030),time(13,05),
```

```
        date(10,nov,2030),time(15,05),
        120,80).
flight(barcelona,valencia,
        date(10,nov,2030),time(13,05),
        date(10,nov,2030),time(15,05),
        120,20).
flight(madrid,london,
        date(10,nov,2030),time(16,05),
        date(10,nov,2030),time(17,35),
        90,140).
flight(valencia,london,
        date(10,nov,2030),time(16,05),
        date(10,nov,2030),time(17,35),
        90,50).
flight(madrid,london,
        date(10,nov,2030),time(23,05),
        date(11,nov,2030),time(00,25),
        80,50).
```

Exercise 9 *Load the file* `flights.pl` *and search for all flights from Valencia to London. Make use of the proper variable names in order to obtain the following output:*

```
DepartureDay = ArrivalDay, ArrivalDay = date(10, nov, 2030),
DepartureTime = time(16, 5),
ArrivalTime = time(17, 35),
Duration = 90,
Price = 50.
```

Exercise 10 *Perform the following queries:*

- *Ask for all flights departing from Madrid on November 10, 2030. To do this, you have to perform a query in which the third parameter of the predicate* `flight` *uses the functor* `date` *with the parameters of the requested date.*
- *Look for all flights whose departure time is 13:05. You have to proceed as in the previous point but using the functor* `time`*.*
- *Perform a query to check the flights departing after 16:00. You must perform a query on the term of the fourth parameter of the predicate* `flight` *consisting of the functor* `time` *with two variables (*`H` *and* `M`*) as parameters. Then, and separated by a comma, you must require that the time is greater than or equal to 16 (*`H>=16`*). Remember that the comma between predicates represents the logical conjunction.*

The program also contains a rule defining the predicate `connection_same_day` with arity 3 to describe flights with just one scale in the same day. This predicate has three parameters to describe the origin, destination and date. Notice that the body of the clause has a variable `Connection` representing the city of the connection. Parameters using the underscore symbol are not relevant to the resolution of the problem. The body of this predicate uses the predicate `is` to evaluate arithmetic expressions. It works by assigning to the variable on the left the result of evaluating the expression on the right. In this expression, the minutes elapsed from the start of the day are added, leaving 60 minutes to perform the flight link. The last condition in ethe body of the clause makes it possible for the traveler to have enough time to catch the flight.

```
connection_same_day(Origin,Destination,Date):-
    flight(Origin,Connection,Date,_,Date,time(Hl1,Ms1),_,_),
    flight(Connection,Destination,Date,time(Hs2,Ms2),Date,_,_,_),
    Hl1_in_minutes is Hl1 * 60 + Ms1 + 60,
    Hs2_in_minutes is Hs2 * 60 + Ms2,
    Hl1_in_minutes =< Hs2_in_minutes.
```

Exercise 11 *Consult all flights with a single connection link that can be done on November 10, 2030.*

Exercise 12 *What would happen if the fourth and the last appearance of the variable* `Date` *in the body of the clause* `connection_same_day` *were changed to another variable* `Another_date`*?*

3.2 Backtracking

We have seen that the interpreter provides all the solutions that are deductible from the facts and rules defining predicates in a Prolog program. These solutions appear (are deducted) one after another as the user requests them. They are obtained by exploring all possibilities making use of both facts and rules. This part of the practice is devoted to observe, by means of traces, the strategy used by the interpreter to obtain those solutions.

Let us first show a brief introduction to this strategy. Prolog clauses comprise both facts and rules.[3] Thus, we can generally say that a Prolog program is composed of clauses. To unify with a clause consists of unifying with a fact or with the head of a rule. Variables of the clause as well as variables of the query can be instantiated in this unification process, leading to both input and output passing of information in the argument passing. Given a query, the interpreter has to locate a clause which unifies with it. Each time a clause is unified, the interpreter provides a new copy (or variant) of it. All variables in the new copy are new (they are renamed) and maintain the same relationship as those of the original clause.

Depending on the type of the clause, the interpreter has a different behavior. If the clause unifies with a fact, the solution if based on the result of this unification. If the unification is performed on the head of a rule, the result of the unification is applied to the body of the rule. In this way, the result of unifying the head (variables instantiated with terms) is propagated to the body predicates. Next, each of the predicates of this body are used as queries. If all these predicates succeed, their results are returned through the unifier of the head of the original query.

Let us remark that the reason to perform a search process is the possibility that several clauses unify with the query. Each time a unification succeeds, we are taking a choice between different possibilities that can lead to the right deduction. Backtracking is required when a choice cannot deduce the veracity of a given query. In this case, other alternatives are tried by undoing the last choice and trying to unify with another clause. In Prolog, this search process is performed in the following way: clauses are taken from top to bottom in the order they appear in the program.

When the unification process is performed with the head of a rule, the predicates of the body are solved from left to right. Each predicate solved in this way has previously applied the unifiers resulting from the unification with the head of the rule as well as those applied on the body

[3]Indeed, facts can be considered a special case of rules whose body is empty and whose head is the fact itself.

predicates of its left. Each time the interpreter fails when trying to solve one of these predicates (after trying all the unification possibilities from top to bottom), it performs one step back to the former predicate (the nearest at the left), undoes the last unification and tries to unify with the clause below.

Trace observation and debugging

In accordance with the mechanism explained above, the execution of an objective may entail the following phases or actions that are manifested in the execution trace:

- `Call`: the name of the predicate and the values of its arguments are displayed when a (sub)goal is invoked.
- `Exit`: if the (sub)goal is satisfied and ends successfully, the value of the arguments that make this (sub)goal satisfiable is accumulated to the partially computed response.
- `Fail`: in this case, the indication of a fail is shown.
- `Redo`: a new invocation of the same (sub)goal is reported. This invocation if produced by the backtracking mechanism in order to re-satisfy the (sub)goal.

The SWI-Prolog interpreter provides predicates allowing us to follow or to watch the backtracking search process: `debug`, `nodebug`, `trace` and `notrace`. These predicates change the status of the *shell* into debug mode and let us see how the interpreter performs the search in the knowledge base.

The `trace` predicate is interactive and allows us to interact in the process (you can see the options by pressing *h* during the trace). This is the predicate that will be used to observe and monitor the implementation of the interpreter. To enter into trace mode, simply perform the query `trace.` and the interpreter will enter into trace mode.

Let us check this process with the example below where the query `isrelative(pablo,X)` is performed on the `family.pl` program without the `isgrandparent` predicate and without the associated rule of `isrelative` using it. Queries are preceded by the word `Call [1]`.

Observe that variable `_G1658` comes from a variant of the clause due to the automatic renaming. If the interpreter is able to demonstrate that the query can be deduced from the clauses of the program, it shows the sample preceded by `Exit`, as well as its paremeters instantiated to the terms which make them true.

In `[2]` the variable `_G1658` appears instantiated to `juan`. If the user ask for a continuation of the search process (by pressing the `r` key after obtaining the first response in `[3]`), the interpreter will perform one step back, undoes the last unification and resumes the search of a new clause from the last one with which it unified. The continuation of the search is preceded by the word `Redo`. In `[4]` reappears the variable `_G1658` and a next clause is searched in order to unify with the query from the next the last one that unified. In `[5]` is unified with the following clause.

If we insist on asking for more solutions to the initial query, the interpreter can not find another fact `isparent` with which to unify and looks for another rule `isrelative [6]`. All the other clauses that define the predicate `isrelative` fail since the queries of their bodies do not unify with any clause. The errors in the search are preceded by the word `Fail`. Use the query

`notrace.` in order to exit the trace mode and make use of `nodebug.` in order to exit debugging mode.

```
[trace]  ?- isrelative(pablo,X).
   Call: (6) isrelative(pablo, _G1658) ? creep                [1]
   Call: (7) isparent(pablo, _G1658) ? creep
   Exit: (7) isparent(pablo, juan) ? creep                    [2]
   Exit: (6) isrelative(pablo, juan) ? creep
X = juan ;                                                    [3]
   Redo: (7) isparent(pablo, _G1658) ? creep                  [4]
   Exit: (7) isparent(pablo, marcela) ? creep                 [5]
   Exit: (6) isrelative(pablo, marcela) ? creep
X = marcela ;
   Redo: (6) isrelative(pablo, _G1658) ? creep                [6]
   Call: (7) ischild(pablo, _G1658) ? creep
   Call: (8) isparent(_G1658, pablo) ? creep
   Fail: (8) isparent(_G1658, pablo) ? creep
   Fail: (7) ischild(pablo, _G1658) ? creep
   Redo: (6) isrelative(pablo, _G1658) ? creep
   Call: (7) issibling(pablo, _G1658) ? creep
   Call: (8) isparent(_G1730, pablo) ? creep
   Fail: (8) isparent(_G1730, pablo) ? creep
   Fail: (7) issibling(pablo, _G1658) ? creep
   Fail: (6) isrelative(pablo, _G1658) ? creep
false.
```

Exercise 13 *Check that the knowledge base* `family.pl` *has been loaded (you can use* `listing.`*) and, in case it is not loaded, use* `consult(family).` *Execute the trace related to the* `isrelative(pablo,X).` *query taking into account the rules that were not in the example before (*`isgrandparent`*,* `isgrandchild` *and the corresponding* `isrelative`*).*

Exercise 14 *Make it sure that the knowledge base* `flights.pl` *has been loaded. Execute a trace of the query* `connection_same_day(Origin, Destination, Connection).`

For realistic examples, a detailed trace like the one we have shown in the previous example may lack practical value, due to the large amount of information that the programmer has to process. For this reason, Prolog systems provide predicates capable of producing a selective trace: `spy(P)` specifies the name of a predicate that you want to display a trace (i.e., when a target is launched only information is displayed of the predicate `P`); `nospy (P)` stops the "espionage" of the predicate named `P`. You can consult the manual to get more information about these and other commands for the debugging of Prolog programs.

4 Simple arithmetic in Prolog

Technically, equality is in Prolog a predicate like any other and can be written in prefix notation (for example, `= (X,2)`), although the infix notation is more stable and readable.

Exercise 15 *Try the following queries and think about what you get:*

```
?- 1 = 2.
?- X = 2.
?- Y = X.
?- X = 8, X = Y.
?- X = Y, X = 8.
```

Exercise 16 *Try now the following query:*

```
?- 1 + 2 = 3.
```

The ‘‘+’’ operator produces a symbolic result ‘‘1 + 2’’. In order to perform an arithmetic computation you have fo force it. This can be done with the `is` *operator. Try the following queries:*

```
?- 2 is 1 + 1.
?- X is 3 * 4.
?- Y is Z + 1.
```

Note the error message of the last example: Prolog requires the right side of the relation `is` *to be* ground*), meaning that it cannot contain variables. Otherwise, Prolog would have to solve some non-trivial equations!*

Exercise 17 *Write a predicate called factorial in order to be able to perform the following query* `?- factorial(6,Y).` *Is this predicate also able to solve the query* `?- factorial(X,24).`*?*

Exercise 18 *Consider the following Prolog program, where the call* `put_code(32)` *is used. The effect of this call is to write a whitespace on the screen:*

```
tab(0).
tab(N) :- put_code(32), N is N - 1, tab(N).
```

Try the query: `?- tab(8).` *Select the correct answer from the following choices:*

1. *an execution error is produced because the argument of* `put` *must be a character.*
2. *an execution error is produced because the first argument of* `is` *must be a non-instantiated variable.*
3. *the execution has the effect of a tabulation of 8 spaces.*
4. *the goal fails.*

Exercise 19 *How can you make the previous program work as expected? Replace the* `put_code(32)` *predicate by* `put_char('.')` *in order to better visualize the result.*

Appendix

A Classification of Prolog terms

- Simple
  - Variables: are represented by means of strings composed by letters, digits and the underscore symbol, but they have eo start by an uppercase letter or by the underscore symbol (for instance: `X`, `Result_1`, `_total3`). Variables starting with underscore symbol are anonymous variables and are used when we are not interested in their values.
  - Constants
    - Symbolic constants: strings composed by letters, digits and the underscore symbol which must start by a lowercase letter. For instance: `f`, `barcelona`, `libro33a`, `libro_blanco`. **Note**: These constants are also known as *atoms*, but this terminology is not used to avoid the confusion with atomic formula which are a predicate (for instance: `fatherof(...)`, `flight(...)`.
    - Numbers: are used to represent integral and floating point types.
      - Integral or integer types: are represented using the typical notation (0, 1, -320, 1520, etc).
      - Floating point: can be represented using both the decimal or the scientific notation. In both cases, you have to include at least one digit in both sides of the decimal point.
- Compound terms: are constructed by means of the symbol of a function, called *functor* (denoted by a symbolic constant) followed (between parenthesis) by a sequence of terms (arguments) separated by commas. For instance: `date(10,nov,2015)`, `time(13,05)`). **Note**: spaces between the functor and the opening parenthesis are not allowed when writing compound terms.