Introduction to AI: Useful Built-in predicates in Prolog (1/2)

François Jacquenet

Laboratoire Hubert Curien
Université de Saint-Etienne
Francois.Jacquenet@univ-st-etienne.fr

December, 2020

Outline

Input/Output on terms

- write/1 and writeq/1 : write(X) and writeq(X) display X on the output stream
- read/1: read(X) reads a term on the input stream and assigns it to X
- nl/0: displays a new line on the output stream

Outputting terms

write(T) causes the term T to be written to the current output stream, which by default is the user's screen

```
?- write(42).
42
true.

?- write(3.14).
3.14
true.

?- write('Artificial Intelligence').
Artificial Intelligence
true.
```

Outputting terms

write(T) causes the term T to be written to the current output stream, which by default is the user's screen

```
?- write(date(year(2020),month(december),day(1))).
date(year(2020),month(december),day(1))
true.

?- write([1,'a',date(year(2020),month(december),day(1)),[a,b]]).
[1,a,date(year(2020),month(december),day(1)),[a,b]]
true.

?- write([1,'a',date(year(Y),month(M),day(D)),[a,b,c]]).
[1,a,date(year(_18088),month(_18092),day(_18096)),[a,b,c]]
true.
```

Outputting terms

writeq(X) causes a quoted atom X to be written to the current output stream, as a quoted atom

```
?- writeq('Artificial Intelligence').
'Artificial Intelligence'
true.

?- writeq('student').
student
true.

?- writeq('Student').
'Student'
true.

?- writeq(42).
42
true.
```

Outputting terms

writeq(X) causes a quoted atom term X to be written to the current output stream, as a quoted atom

```
?- writeq(date(year(2020),month(december),day(1))).
date(year(2020),month(december),day(1))
true.

?- writeq(date(year(2020),month('M'),day(1))).
date(year(2020),month('M'),day(1))
true.

?- write(date(year(2020),month('M'),day(1))).
date(year(2020),month(M),day(1))
true.
```

Outputting newline

nl/0 causes a new line to be written to the current output stream

```
go :-
   X=42, Y=73,
   write('Here is the value of X : '), write(X),
   nl,
   write('Here is the value of Y : '),write(Y).

?- go.
Here is the value of X : 42
Here is the value of Y : 73
true.
```

Inputting terms

read(T) causes the term T to be read from the current input stream, which by default is the user's keyboard

```
?- read(T).
|: 42.

T = 42.

?- read(T).
|: 3.14.

T = 3.14.

?- read(T).
|: 'Artificial Intelligence'.

T = 'Artificial Intelligence'.
```

Note that you must enter a dot after the term entered.

Inputting terms

read(T) causes the term T to be read from the current input stream, which by default is the user's keyboard

```
?- read(T).
|: [1,'a',date(year(2020),month(december),day(1)),[a,b]].

T = [1, a, date(year(2020), month(december), day(1)), [a, b]].


?- read(T).
|: [1,'a',date(year(Y),month(M),day(D)),[a,b,c]].

T = [1, a, date(year(_21720), month(_21724), day(_21728)), [a, b, c]].
```

Inputting terms

read(T) causes the term T to be read from the current input stream, which by default is the user's keyboard

T can be instantiated

```
?- T = mary, read(T).
|: mary.

T = mary.


?- ?- T = mary, read(T).
|: john.

false.
```

Input/Output on characters

- put/1: put(A) displays the character whose ASCII code is A on the output stream
- get0/1: get0(X) reads a character from the input stream and assigns it to X
- get/1: get(X) reads the next non-white-space character (i.e. character with an ASCII value less than or equal to 32) from the input stream

Input/Output on characters

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Outputting characters

put(A) displays the character whose ASCII code is A on the current output stream, which by default is the user's screen

```
?- put(97).
a
true.

?- put(90).
Z
true.

?- put(35).
#
true.
```

Inputting characters

get0(C) reads a character on the input stream (by default, the user's keyboard) and assigns it to C

```
?- get0(C).
|: a

C = 97.

?- get0(C).
|: Z

C = 90.

?- get0(C).
|: #

C = 35.
```

Inputting characters

get0(C) reads a character on the input stream (by default, the user's keyboard) and assigns it to C

T can be instantiated

```
?- C = 90, get0(C).
|: Z

C = 90.

?- C = 90, get0(C).
|: a

false.
```

Inputting characters

We can use get0 several times to input several characters

```
readline :-
   get0(X), continue(X).
continue(35).
continue(C):-
   C \== 35, put(C), readline.

?- readline.
|: this is a test#
this is a test
true .
```

Inputting characters

We can extend this program, for example, to count the number of characters

```
readline(NumberChars):-
   readline(0,NumberChars).

readline(Count,NumberChars):-
   get0(C),continue(C,Count,NumberChars).

continue(35,Count,Count).
continue(C,Count,NumberChars):-
   C\==35,Count2 is Count+1,readline(Count2,NumberChars).
```

Inputting characters

Let's run it

```
?- readline(N).
|: This is a test#

N = 14 .

?- readline(N).
|: this is a test
|: and another one#

N = 30 .

?- readline(N).
|: #

N = 0 .
```

Inputting characters

Note we can write a simpler program (without using an accumulator):

```
readline(NumberChars):-
   get0(C),continue(C,NumberChars).

continue(35,0).
continue(C,NumberChars):-
   C\==35,
   readline(NumberChars2),
   NumberChars is NumberChars2 + 1.
```

Inputting characters

We can extend this program to count the number of vowels

```
readline(NumberChars,NumberVowels):-
   readline(0,NumberChars, 0, NumberVowels).

readline(CountChars,NumberChars,CountVowels,NumberVowels):-
   get0(C),
   continue(C,CountChars,NumberChars,CountVowels,NumberVowels).

continue(35,CountChars,CountChars,CountVowels,CountVowels).
continue(C,CountChars,NumberChars,CountVowels,NumberVowels):-
   C\==35,CountChars2 is CountChars+1,
   isVowels(C,CountVowels,NumberVowels2),
   readline(CountChars2,NumberChars,NumberVowels2,NumberVowels).

isVowels(C,CountVowels,NumberVowels) :-
   vowel(C), NumberVowels is CountVowels + 1.
isVowels(_,CountVowels,CountVowels).

vowel(97). vowel(101). vowel(105). vowel(111). vowel(117). vowel(121).
```

Inputting characters

Let's run it!

```
?- readline(N,V).
|: this is a test#

N = 14,
V = 4 .

?- readline(N,V).
|: zrtpml#

N = 6,
V = 0 .

?- readline(N,V).
|: #

N = V, V = 0 .
```

Inputting characters

Note we can write a simpler program (without using accumulators):

```
readline(NumberChars,NumberVowels):-
   get0(C),continue(C,NumberChars,NumberVowels).

continue(35,0,0).
continue(C,NumberChars,NumberVowels):-
   C\==35,
   vowel(C),
   readline(NumberChars2,NumberVowels2),
   NumberVowels is NumberVowels2 + 1,
   NumberChars is NumberChars2 + 1.
continue(C,NumberChars,NumberVowels):-
   C\==35,
   readline(NumberChars2,NumberVowels),
   NumberChars is NumberChars2 + 1.

vowel(97). vowel(101). vowel(105). vowel(111). vowel(117). vowel(121).
```

Defining the input stream

- The default current input stream is the user's keyboard
- The current input stream can be changed using the see/1 predicate
- We can restore the current input stream to be the user's keyboard by using the seen/0 predicate
- We can determine the name of the current input stream by solving seeing(F). F is unified to this name

Defining the input stream

For example we can run some previous predicates on a particular file `foo.txt` instead of the keyboard

```
readFile(NumberChars,NumberVowels) :-
   see('foo.txt'),
   readline(NumberChars,NumberVowels),
   seen.
```

Defining the output stream

- The default current output stream is the user's terminal
- The current output stream can be changed using the tell/1 predicate
- We can restore the current output stream to be the user's terminal by using the told/0 predicate
- We can determine the name of the current output stream by solving telling(F). F is unified to this name

Defining the output stream

For example we can copy a text from the keyboard to a file

```
copyText(Outfile):-
   tell(Outfile),
   copy_characters,
   told.

copy_characters:-
   get0(C),continue(C).

continue(35). /* ASCII value of # */
continue(N):-
   N\==35,
   put(N),copy_characters.
```

Defining the output stream

And then run:

```
?- copyText('fic1.txt').
|: this is a test
|: let's look at the content of
|: the file now#

true .
```

If we look at `fic1.txt`, it contains

```
this is a test
let's look at the content of
the file now
```

repeat

Recursivity may be good, but sometimes it's better to backtrack

- repeat/0
- it succeeds whenever it is called
- it also succeeds (as many times as necessary) on backtracking

This can be used to create a looping effect

repeat

An example

```
get_answer(A):-
   write('Enter an answer to the question'),nl,
   repeat,
   write('Your answer (yes or no): '),read(A),
   valid(A),
   write('Your answer is '),write(A),nl.

valid(yes).
valid(no).
```

repeat

Let's run it!

```
?- get_answer(A).
Enter an answer to the question
Your answer (yes or no): azer.
Your answer (yes or no): |: abc.
Your answer (yes or no): |: yes.
Your answer is yes
A = yes .
```

fail

fail/0 is a predicate that always fail
Why it may be useful?

```
student(mary).
student(john).
student(suzy).

allStudents :-
   student(S), write(S), write(' is a student'),nl,
   fail.
allStudents.
```

fail

Let's run it!

```
?- allStudents.
mary is a student
john is a student
suzy is a student
true.
```

fail

Consider another example

```
person(john,smith,45,lyon,teacher).
person(martin,williams,33,paris,teacher).
person(henry,smith,26,dijon,student).
person(jane,wilson,62,lyon,teacher).
person(mary,smith,29,paris,student).

allJobs(J) :-
   person(FirstName,Name,_,_,J),
   write(FirstName),write(' '),write(Name),nl,
   fail.
allJobs(_).
```

fail

Let's run it!

```
?- allJobs(teacher).
john smith
martin williams
jane wilson
true.

?- allJobs(student).
henry smith
mary smith
true.

?- allJobs(policeman).
true.
```

Exercise: Write a predicate findUnderAge(A) that list all the persons that are less than A years old.

fail

A last example

```
find_all_routes(Town1,Town2):-
   findroute(Town1,Town2,Route),
   write('Possible route: '),write(Route),nl,
   fail.
find_all_routes(_,_).

findroute(Town1,Town2,Route) :-
   ...     % definition of the findroute predicate
```

The cut

What we call cut is the symbol !

It changes the way Prolog naturally backtracks

It prunes the derivation tree

The cut

Consider the following program and goals

```
max(A,B,B) :- B > A.
max(A,B,A).

?- max(3,4,M).
M = 4 ;
M = 3.

?- max(4,3,M).
M = 4.
```

This is not a good way to write max because the first goal gives one good solution but also one bad, due to the second clause

The cut

Obviously, we could write

```
max(A,B,B) :- B > A.
max(A,B,A) :- B =< A.

?- max(3,4,M).
M = 4 ;
false

?- max(4,3,M).
M = 4.
```

For this very basic example, finding the good condition to avoid unwanted backtracking is easy

BUT, this is not always the case

The cut

Consider a second example with some goals

```
sum(1,1).
sum(N,S):-
   N1 is N-1,
   sum(N1,S1),
   S is S1+N.

?- sum(6,S).
S = 21 ;
ERROR: Out of local stack
```

We get a solution, but if we ask for another one, the program loops

The cut

Obviously, we could write

```
sum(1,1).
sum(N,S):-
   N > 1,
   N1 is N-1,
   sum(N1,S1),
   S is S1+N.

?- sum(6,S).
S = 21 ;
false.
```

For this very basic example, finding the good condition to avoid unwanted backtracking is easy

BUT, this is not always the case

The cut

The cut may be used in such situations

```
max(A,B,B) :- B > A, !.
max(A,B,A).

?- max(3,4,M).
M = 4.

?- max(4,3,M).
M = 4.
```

The cut

The cut may be used in such situations

```
sum(1,1) :- !.
sum(N,S):-
   N1 is N-1,
   sum(N1,S1),
   S is S1+N.

?- sum(6,S).
S = 21.
```

The cut

More generally

```
p(X) :-
   condition(X),r(X,Y),s(Y).
p(X) :-
   \+ condition(X), t(X), u(X,Y).
```

BUT solving condition(X) may be expensive (time and memory)

Moreover, if condition(X) is false, we need to prove it again to prove the negation is true

It can be a big waste of time

The cut

Using the cut we can be more efficient

In fact it's an if then else...

```
p(X) :-
   condition(X),!, r(X,Y),s(Y).
p(X) :-
   t(X), u(X,Y).
```

Cut with failure

Consider this program

```
bird(eagle).
bird(duck).
bird(crow).
bird(ostrich).
bird(swan).

can_fly(ostrich) :-
   !,
   fail.
can_fly(X) :-
   bird(X).
```

Here the cut is useful to specify exceptions to general rules

Cut (recap)

Consider this program

```
p(a).
p(b) :- !.
p(c).
```

What will produce the following goals:

```
?- p(X).
?- p(X), !.
?- p(X), p(Y).
?- !, p(X), p(Y).
?- p(X), !, p(Y).
?- p(X), p(Y), !.
```

Cut and derivation tree

Let's look at the impact of the cut on the derivation tree
Consider the following program

```
p3(X,Y,Z) :- p2(X,Z), t(Y).

p2(X,Y) :- s(X), q(Y).
p2(X,Y) :- q(Z), r(X,Z,Y).

s(a).  s(b). s(c).
q(a).  q(e).
t(42). t(21).

r(f,a,g).
r(g,e,d).
```

Draw the derivation tree of the resolution of the goal:

```
?- p3(A,B,C).
```

Cut and derivation tree

Now consider the following program that contains a cut

```
p3(X,Y,Z) :- p2(X,Z), t(Y).

p2(X,Y) :- s(X), !, q(Y).
p2(X,Y) :- q(Z), r(X,Z,Y).

s(a).  s(b). s(c).
q(a).  q(e).
t(42). t(21).

r(f,a,g).
r(g,e,d).
```

Show on the previous derivation tree which branches are pruned

What happens if we run the goal: `?- p3(e,B,C).`

assert/1 and retract/1

Allow dynamic change of the Prolog program

- asserta/1: asserta(C) adds clause C in the program (at the top of other clauses with the same predicate)
- assertz/1: assertz(C) adds clause C in the program (at the bottom of other clauses with the same predicate)
- retract/1: retract(C) deletes from the program the first clause that unifies with C (first from the top)
- retractall/1: retractall(H) deletes from the program all the clauses C whose head unifies with H

Those predicates are generally used to add new knowledge that has been learnt or delete knowledge that becomes outdated

asserta/assertz

To allow predicate `p1` of arity `n` to be asserted or retracted it must be defined as dynamic: `:- dynamic(p1/n).`
If you do not do that, you get an error while running assert or retract on `p1`

```
?- retract(p1(X)).
ERROR: No permission to modify static procedure `p/1'
ERROR: In:
ERROR:    [8] retract(p(_8920))
ERROR:    [7] <user>
```

With SWI Prolog `assert(p1(a))` works even if you did not define `p1/1` as dynamic (but `retract` fails)

Adding clauses

Suppose we have no clause defining the predicate p/1

```
?- assertz(p(1)).
?- assertz(p(2)).

?- listing(p).
p(1).
p(2).
```

Adding clauses

Let's continue to add facts

```
?- asserta(p(3)).
?- assertz(p(4)).
?- asserta(p(5)).

?- listing(p).
p(5).
p(3).
p(1).
p(2).
p(4).
```

Adding clauses

We can also add clauses with head and body

```
?- asserta((grand_parent(X,Y) :- parent(X,Z), parent(Z,Y))).

?- listing(grand_parent).
:- dynamic grand_parent/2.

grand_parent(A, C) :-
   parent(A, B),
   parent(B, C).
```

Note the parentheses around the clause! and there is no dot at the end of the clause!

Adding clauses

We can also add clauses with head and body

```
?- asserta((grand_parent(X,Y) :- parent(X,Y))).

?- listing(grand_parent).
:- dynamic grand_parent/2.

grand_parent(A, B) :-
   parent(A, B).
grand_parent(A, C) :-
   parent(A, B),
   parent(B, C).
```

Deleting clauses

`retract(C)` deletes from the program the first clause that unifies with `C` (first from the top)
Suppose we have those facts

```
:- dynamic(student/1).

student(mary).
student(john).
student(suzy).
```

Deleting clauses

Then look at what happens when we run various goals using `retract/1`

```
?- retract(student(john)).
true.

?- listing(student)
:- dynamic(student/1).

student(mary).
student(suzy).

true.
```

Deleting clauses

Take care:

```
?- retract(student(X)).
X = mary ;
X = suzy

?- listing(student).
:- dynamic student/1.

true.
```

Deleting clauses

But if we do not hit ";"

```
?- retract(student(X)).
X = mary

?- listing(student).
:- dynamic student/1.

student(suzy).

true.
```

Deleting clauses

Suppose we have the program:

```
p(X,Y) :- q(X), r(X).
p(X,Y) :- s(X), t(Y).
p(a,b).
p(c,d).
```

Deleting clauses

Then run `retract/1` (and hit ";" each time we get a solution):

```
p(X,Y) :- q(X), r(Y).
p(X,Y) :- s(X), t(Y).
p(a,b).
p(c,d).

?- retract(p(A,B)).
A = a, B = b

A = c, B = d.

?- listing(p/2).
:- dynamic p/2.

p(A, B) :- q(A),r(B).
p(A, B) :- s(A),t(B).

true.
```

Deleting clauses

Continue to run `retract/1` on the program:

```
p(X,Y) :- q(X), r(Y).
p(X,Y) :- s(X), t(Y).
```

and get:

```
?- retract((p(A,B) :- s(A),t(B))).
true.

?- listing(p/2).
:- dynamic p/2.

p(A, B) :- q(A),r(B).

true.
```

Deleting clauses

Finally, on the program:

```
p(X,Y) :- q(X), r(Y).
```

we can run:

```
?- retract((p(A,B) :- B)).
B =  (q(A), r(B)).

?- listing(p/2).
:- dynamic p/2.

true.
```

Another example

Consider a file containing some information about persons

```
mary. smith. 22. 'Saint-Etienne'. student.
john. smith. 25. 'Lyon'. 'Computer Scientist'.
mike. portnoy. 51. 'New York'. musician.
stephen. hough. 58. 'London'. musician.
suzy. gallagher. 21. 'Paris'. student.
end.
```

Another example

We want to write a program that reads this file and add this information into the Prolog database

```
readfile :-
   see('people.txt'),
   read_data,
   write('Finished converting data.'),nl,
   seen.

read_data:-
   read(FirstName), continue(FirstName).

continue(end).
continue(FirstName):-
   read(Name), read(Age), read(Town), read(Job),
   assertz(person(FirstName,Name,Age,Town,Job)),
   read_data.
```

Another example

Running `?- readfile.` we get the knowledge based defined by the `person/5` predicate

```
?- listing(person).
:- dynamic person/5.

person(mary, smith, 22, 'Saint-Etienne', student).
person(john, smith, 25, 'Lyon', 'Computer Scientist').
person(mike, portnoy, 51, 'New York', musician).
person(stephen, hough, 58, 'London', musician).
person(suzy, gallagher, 21, 'Paris', student).

true.
```

Memorize previously computed values

Suppose we want to calculate the Fibonacci function of a number

The definition of this function is:

$$\begin{cases} fib(1) = 1 \\ fib(2) = 1 \\ fib(n) = fib(n-1) + fib(n-2) \text{ for } n > 2 \end{cases}$$

We can write the following Prolog program

```
fib(1, 1).
fib(2, 1).
fib(N, F) :-
   N > 2,
   N1 is N - 1, fib(N1, F1),
   N2 is N - 2, fib(N2, F2),
   F is F1 + F2.
```

Memorize previously computed values

Let's look at how it works on a small example

```
fib(5, F0)
   fib(4, F1)
      fib(3, F2)
         fib(2, F3) -> F3 = 1
         fib(1, F4) -> F4 = 1
         -> F2 = F3+F4 = 1+1 = 2
      fib(2, F5) -> F5 = 1
      -> F1 = F2+F5 = 2+1 = 3
   fib(3, F6)
      fib(2, F7) -> F7 = 1
      fib(1, F8) -> F8 = 1
      -> F6 = F7+F8 = 1+1 = 2
-> F0 = F1+F6 = 3+2 = 5
```

We can see that fib(3,X) is calculated twice

Memorize previously computed values

Using `asserta/1` we can memorize some intermediate calculations

```
lazyFib(1, 1).
lazyFib(2, 1).
lazyFib(N, F) :-
   N > 2,
   N1 is N - 1, lazyFib(N1, F1),
   N2 is N - 2, lazyFib(N2, F2),
   F is F1 + F2,
   asserta(lazyFib(N,F)).
```

Memorize previously computed values

Let's look at how it works on a small example

```
lazyFib(5, F0)
   lazyFib(4, F1)
      lazyFib(3, F2)
         lazyFib(2, F3) -> F3 = 1
         lazyFib(1, F4) -> F4 = 1
         -> F2 = F3+F4 = 1+1 = 2
         asserta(lazyFib(3,2))
      lazyFib(2, F5) -> F5 = 1
      -> F1 = F2+F5 = 2+1 = 3
      asserta(lazyFib(4,3))
   lazyFib(3, F6)
      -> use the fact lazyFib(3,2)
      -> F6 = 2
-> F0 = F1+F6 = 3+2 = 5
asserta(lazyFib(5,5))
```

A last example

We can define a predicate that can remove some information about a particular person

```
remove(FirstName,Name):-
   retract(person(FirstName,Name,_,_,_)).
```

A last example

```
person(mary, smith, 22, 'Saint-Etienne', student).
person(john, smith, 25, 'Lyon', 'Computer Scientist').
person(mike, portnoy, 51, 'New York', musician).
person(stephen, hough, 58, 'London', musician).
person(suzy, gallagher, 21, 'Paris', student).

?- remove(john,smith).
true.

?- listing(person).
:- dynamic person/5.

person(mary, smith, 22, 'Saint-Etienne', student).
person(mike, portnoy, 51, 'New York', musician).
person(stephen, hough, 58, 'London', musician).
person(suzy, gallagher, 21, 'Paris', student).

true.
```

A last example

Finally we can define a predicate that can update some information about a particular person

```
update(FirstName,Name,NewJob):-
   person(FirstName,Name,Age,City,Job),!,
   retract(person(FirstName,Name,Age,City,Job)),
   assertz(person(FirstName,Name,Age,City,NewJob)).
update(FirstName,Name,NewJob):-
   write('Sorry but '),
   write(FirstName), write(' '),
   write(Name), write(' is not in the database...').
```

A last example

And run it:

```
?- update(mary,smith,doctor).
true.

?- listing(person).
:- dynamic person/5.

person(john, smith, 25, 'Lyon', 'Computer Scientist').
person(mike, portnoy, 51, 'New York', musician).
person(stephen, hough, 58, 'London', musician).
person(suzy, gallagher, 21, 'Paris', student).
person(mary, smith, 22, 'Saint-Etienne', doctor).

true.
```