

## **Lecture 3 - Deterministic and non-deterministic predicates**

### **Content**

- 1. Deterministic and non-deterministic predicates
  - 1.1 Predefined predicates
  - 1.2 Collect all solutions
  - 1.3 Negation - “not”, “\+”
  - 1.4 Lists and recursion
    - 1.4.1 Head & tail of a list
    - 1.4.2 List processing
    - 1.4.3 Using lists
- 2. Examples

### **Bibliography**

Czibula, G., Pop, H.F., Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligența Artificială, Ed. Albastră, Cluj-Napoca, 2012

## 1. Deterministic and non-deterministic predicates

Types of predicates, **ideal implementation**:

- **deterministic**
  - a deterministic predicate has only one solution
- **non-deterministic**
  - an non-deterministic predicate has several solutions

**SWI Prolog implementation**:

- A **deterministic predicate** always succeeds exactly once and does not leave a choicepoint.
- A **semi-deterministic predicate** succeeds at most once. If it succeeds it does not leave a choicepoint.
- A **non-deterministic predicate** is the most general case and no claims are made on the number of solutions and whether or not the predicate leaves a choicepoint on the last solution.

**Remark.** A predicate can be deterministic in one flow model, and non-deterministic in other flow models.

### 1.1 Predefined predicates

<code>var (X)</code>	= true if X is free, false if bound
<code>number (X)</code>	= true if X is bound to a number
<code>integer (X)</code>	= true if X is bound to an integer
<code>float (X)</code>	= true if X is bound to a real number
<code>atom (X)</code>	= true if X is bound to an atom
<code>atomic (X)</code>	= <code>atom (X)</code> or <code>number (X)</code>

## 1.2 Collect all solutions

Prolog provides ways to collect in a list all the solutions of a predicate.

**findall**(X, goal\_on\_X, L)  
**setof**(X, goal\_on\_X, L)  
**bagof**(X, goal\_on\_X, L)

They have the following parameters:

- the first parameter specifies the predicate parameter to be collected in the list;
- the second parameter specifies the predicate to be solved;
- the third parameter specifies the list where all the solutions will be collected.

The difference between these three predicates follows:

- **findall**
  - collects all the solutions of X in L
  - any variable in goal not in the first argument is not instantiated
- **setof**
  - removes from L all duplicate outputs and answers are sorted
  - any variable in goal not in the first argument results in separate solutions
- **bagof**
  - does not remove from L duplicates and the answers may not be sorted
  - any variable in goal not in the first argument results in separate solutions

### **EXAMPLE 1:**

p (a, b).  
p (b, c).  
p (a, c).  
p (a, d).  
all (X, L) :- findall (Y, p (X, Y), L).

?- all (a, L).  
L = [b, c, d]

### **EXAMPLE 2:**

age(peter, 7).  
age(ann, 5).  
age(pat, 8).  
age(tom, 5).  
age(ann, 5).

**?- setof(Child, age(Child, Age), Results).**

Age = 5  
Results = [ann, tom] ;

Age = 7  
Results = [peter] ;

Age = 8  
Results = [pat] .

**?- bagof(Child, age(Child, Age), Results).**

Age = 5

Results = [ann, tom, ann] ;

Age = 7

Results = [peter] ;

Age = 8

Results = [pat] .

**?- findall(Child, age(Child, Age), Results).**

Results = [peter, ann, pat, tom, ann] .

A way to group parameters in pairs uses the notation **A/B** or **A-B**:

**?- setof(Age/Child, age(Child, Age), Results).**

Results = [5/ann, 5/tom, 7/peter, 8/pat].

**?- setof(Age-Child, age(Child, Age), Results).**

Results = [5-ann, 5-tom, 7-peter, 8-pat].

**?- bagof(Age/Child, age(Child, Age), Results).**

Results = [7/peter, 5/ann, 8/pat, 5/tom, 5/ann].

**?- findall(Age/Child, age(Child, Age), Results).**

Results = [7/peter, 5/ann, 8/pat, 5/tom, 5/ann].

### 1.3 Negation - “not”, “\+”

Negation is true if the subgoal fails (can't be proven to be true).

```
not(subgoal (Arg1, ..., ArgN))  
\+ subgoal (Arg1, ..., ArgN)
```

```
?- \+ (2 = 4).  
true.
```

```
?- not (2 = 4).  
true.
```

### 1.4 Lists and recursion

In Prolog, a list is an object that contains an arbitrary number of other objects. The lists in SWI-Prolog are heterogeneous (the component elements can have different types). Lists are built using square brackets. Their elements are separated by a comma.

Here are some examples:

```
[1, 2, 3]
```

```
[dog, cat, canary]
```

```
[“valerie ann”, “jennifer caitlin”, “benjamin thomas”]
```

If we were to declare the type of a (homogeneous) list with integer elements, we would use a domain declaration of the following type

```
element = integer  
list = element *
```

### 1.4.1 Head & tail of a list

A list is a recursive object. It consists of two parts: the head, which is the first element of the list, and the tail, which is the rest of the list. The head of the list is an element, and the tail is a list:

The head of the list [a, b, c] is	The tail of [a, b, c] is [b, c]
The head of the list [c] is c	The tail of [c] is []

The empty list [] cannot be split into head and tail.

### 1.4.2 List processing

Prolog provides a way to make the head and tail of a list explicit. Instead of separating the elements of a comma list, we will separate the head of the queue with the character '|'.

For example, the following lists are equivalent:

[a b c]      [a | [b, c]]      [a | [b | [c]]]      [a | [b | [c | []]]]

Also, before the '|' sign several elements may be written, not just the first. For example, the list [a, b, c] above is equivalent to

[a | [b, c]]      [a, b | [c]]      [a, b, c | []]

- Following the unification of list [a, b, c] with list [H | T] (H, T being free variables)
  - H binds to a; T binds to [b, c]
- Following the unification of list [a, b, c] with list [H | [H1 | T]] (H, H1, T being free variables)
  - H binds to a; H1 binds to b; T binds to [c]

### 1.4.3 Using lists

Because a list is a recursive data structure, recursive algorithms are needed to process it. The basic way to process the list is to work with it, performing certain operations with each element of it, until the end is reached.

Such an algorithm generally needs two clauses. One of them says what to do with an empty list. The other says what to do with a non-empty list, which can be broken down in its head and its tail.



## 2. Examples

**EXAMPLE 2.1** Given a non null natural number  $N$ , compute  $F = N!$ . Simulate the iterative computation process.

```
 $I \leftarrow 1$   
 $P \leftarrow 1$   
While  $I < N$  do  
     $I \leftarrow I + 1$   
     $P \leftarrow P * I$   
End While  
 $F \leftarrow P$ 
```

$$fact(n) = fact\_aux(n, 1, 1)$$

$$fact\_aux(n, i, p) = \begin{cases} p & \text{if } i = n \\ fact\_aux(n, i + 1, p * (i + 1)) & \text{otherwise} \end{cases}$$

The description is not directly recursive, collector variables ( $i, P$ ) are used

```
% fact (N: integer, F: integer)  
% (i, i), (i, o) - deterministic  
fact (N, F) :- fact_aux (N, F, 1, 1).
```

```
% fact_aux (N: integer, F: integer, I: integer, P: integer)  
% (i, i, i, i), (i, o, i, i) - deterministic  
fact_aux (N, F, N, F) :- !. % fact_aux (N, F, I, P) :- I is N, F is P,!.  
fact_aux (N, F, I, P) :- NewI is I + 1,  
    NewP is P * NewI,  
    fact_aux (N, F, NewI, NewP).
```

The result is a tail recursion. All cycling variables were introduced as arguments to the `fact_aux` predicate.

**HOMEWORK:**

Write a factorial predicate `(N, F)` that works in all three flow models `(i, i)`, `(i, o)` and `(o, i)`.

**EXAMPLE 2.2** Verify the membership of an item in a list.

% (i, i)

To describe the membership in a list we will construct the member predicate (element, list) which will investigate whether a certain element is a member of the list. The algorithm to be implemented would (declaratively) be the following:

1. E is a member of the list L if it is its head.
2. Otherwise, E is a member of the list L if it is a member of the tail of L.

From a procedural point of view:

1. To find a member of the list L, find his head;
2. Otherwise, find a member of the tail of L.

$$member(E, l_1 l_2 \dots l_n) = \begin{cases} false & \text{if } \text{is empty} \\ true & \text{if } l_1 = E \\ member(E, l_2 \dots l_n) & \text{otherwise} \end{cases}$$

**Version 1**

% member1(e: element, L: list)

% (i, i) - (non) deterministic

% (o, i) - non-deterministic

member1(E, [E | \_]).

member1(E, [\_ | L]) :-  
    member1(E, L).

## Version 2

% member2 (e: element, L: list)

% (i, i) - deterministic

%(o, i) - deterministic

member2(E, [E | \_]) :- !.

member2(E, [\_ | L]) :-  
    member2(E, L).

## Version 3

% member3 (e: element, L: list)

% (i, i) - (non) deterministic

% (o, i) - non-deterministic

member3(E, [\_ | L]) :-  
    member3(E, L).

member3(E, [E | \_]).

?- member3(E, [1,2,3]).

E = 3;

E = 2;

E = 1.

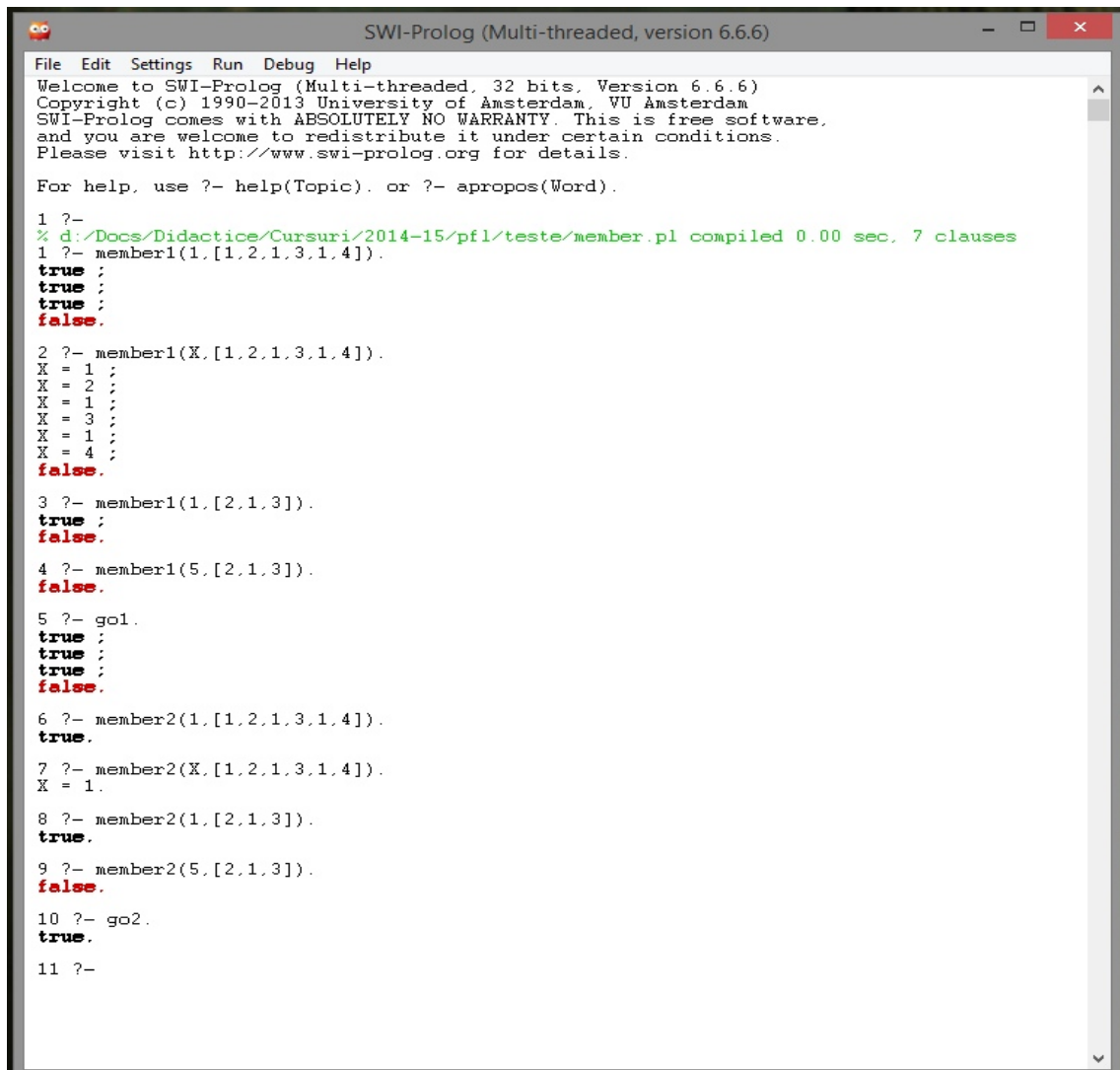
?-member3 (4, [1,2,3]).

**false.**

?- member3 (2, [1,2,3]).

true;

**false.**



```
SWI-Prolog (Multi-threaded, version 6.6.6)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.6)
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).

1 ?-
% d:/Docs/Didactice/Cursuri/2014-15/pfl/teste/member.pl compiled 0.00 sec, 7 clauses
1 ?- member1(1,[1,2,1,3,1,4]).
true ;
true ;
true ;
false.

2 ?- member1(X,[1,2,1,3,1,4]).
X = 1 ;
X = 2 ;
X = 1 ;
X = 3 ;
X = 1 ;
X = 4 ;
false.

3 ?- member1(1,[2,1,3]).
true ;
false.

4 ?- member1(5,[2,1,3]).
false.

5 ?- go1.
true ;
true ;
true ;
false.

6 ?- member2(1,[1,2,1,3,1,4]).
true.

7 ?- member2(X,[1,2,1,3,1,4]).
X = 1.

8 ?- member2(1,[2,1,3]).
true.

9 ?- member2(5,[2,1,3]).
false.

10 ?- go2.
true.

11 ?-
```

As it can be seen, the predicate **member** it also works in the flow model (o, i), in which it is **non-deterministic**.

### **EXAMPLE 2.3 Add an item at the end of a list.**

? add (3, [1, 2], L).

L = [1, 2, 3]

Recursive formula:

$$add(e, l_1 l_2 \dots l_n) = \begin{cases} (e) & \text{daca } l \text{ e vida} \\ l_1 \oplus add(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

#### **Version 1**

% add (e: element, L: list, LRez: list)

% (i, i, o) – deterministic

add (E, [], [E]).

add (E, [H | T], [H | Rez]) :-

add (E, T, Rez).

% add (E, [H | T], Rez) :-

% add (E, T, L), Rez = [H | L].

#### **Version 2**

% add (L: list, e: element, LRez: list)

% (i, i, o) – deterministic

add ([], E, [E]).

add ([H | T], E, [H | Rez]) :-

add (T, E, Rez).

% add ([H | T], E, Rez) :-

% add (T, E, L), Rez = [H | L].

The time complexity of the operation of adding an item to the end of the list of a list of n items is  $\theta(n)$ .

### **EXAMPLE 2.4 Determine the inverse of a list.**

**Version A** (direct recursion)

$$invers(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ invers(l_2 \dots l_n) \oplus l_1 & \text{altfel} \end{cases}$$

% inverse (L: list, LRez: list)

% (i, o) – deterministic

inverse ([], []).

inverse ([H | T], Rez) :-

    inverse (T, L),

    add (H, L, Rez).

The time complexity of the operation of reversing a list of n elements (using the addition at the end) is  $\theta(n^2)$ .

**Version B** (with collector variable)

$$\begin{aligned} & invers\_aux(l_1 l_2 \dots l_n, Col) \\ & = \begin{cases} Col & \text{daca } l \text{ e vida} \\ invers\_aux(l_2 \dots l_n, l_1 \oplus Col) & \text{altfel} \end{cases} \\ & invers(l_1 l_2 \dots l_n) = invers\_aux(l_1 l_2 \dots l_n, \emptyset) \end{aligned}$$

% inverse (L: list, LRez: list)

% (i, o) – deterministic

inverse (L, Rez) :-

    invers\\_aux ([], L, Rez).

```
% invers_aux (Col: list, L: list, LRez: list)
% Col is the collector
% (i, i, o) - deterministic
```

```
invers_aux (Col, [], Col).
invers_aux (Col, [H | T], Rez) :-
    invers_aux ([H | Col], T, Rez).
```

The time complexity of the operation of reversing a list of  $n$  elements (using a collector variable) is  $\theta(n)$ .

**Remark.** Using a collector variable does not lead to a decrease in complexity in all cases. There are situations in which the use of a collector variable increases the complexity (ex: the addition in collectors is done at the end of it, not at the beginning).



**EXAMPLE 2.5** Determine the list of even items in a list (keep the order of the items in the original list).

**Version A** (direct recursion)

$$even(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ l_1 \oplus even(l_2 \dots l_n) & \text{daca } l_1 \text{ par} \\ even(l_2 \dots l_n) & \text{altfel} \end{cases}$$

% even (L: list, LRez: list)

% (i, o) – deterministic

even([], []).

even([H | T], [H | Rez]) :-

    H mod 2 =:= 0,

    !,

    even(T, Rez).

even([\_ | T], Rez) :-

    even(T, Rez).

The time complexity of the operation is  $\theta(n)$ ,  $n$  being the number of items in the list.

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n - 1) + 1 & \text{otherwise} \end{cases}$$

**Version B** (with collector variable)

$$\begin{aligned} & even\_aux(l_1 l_2 \dots l_n, Col) \\ & = \begin{cases} Col & \text{daca } l \text{ e vida} \\ even\_aux(l_2 \dots l_n, Col \oplus l_1) & \text{daca } l_1 \text{ par} \\ even\_aux(l_2 \dots l_n, Col) & \text{altfel} \end{cases} \\ & even(l_1 l_2 \dots l_n) = even\_aux(l_1 l_2 \dots l_n, \emptyset) \end{aligned}$$

```

% even (L: list, LRez: list)
% (i, o) – deterministic

even(L, Rez) :-
    even_aux (L, Rez, []).

% even_aux(L: list, LRez: list, Col: list)
% (i, o, i) – deterministic

even_aux ([], Rez, Rez).
even_aux ([H | T], Rez, Col) :-
    H mod 2 =:= 0,
    !,
    add (H, Col, ColN), % add at the end
    even_aux (T, Rez, ColN).
even_aux ([_ | T], Rez, Col) :-
    even_aux (T, Rez, Col).

```

The time complexity of the operation is  $\theta(n^2)$ ,  $n$  being the number of items in the list.

**EXAMPLE 2.6** Given a numerical list, determine the list of strictly ascending pairs of elements in the list.

?- pairs ([2, 1, 3, 4], L)  
 L = [[2, 3], [2, 4], [1, 3], [1, 4], [3, 4]]

?- pairs ([5, 4, 2], L)  
**false**

We will use the following predicates:

- the nondeterministic predicate pair (element, list, list) (flow model (i, i, o)), which will produce pairs in ascending order between the given element and elements of the argument list

?- pair (2, [1, 3, 4], L)  
 L = [2, 3]  
 L = [2, 4]

$pair(e, l_1 l_2 \dots l_n) =$   
 1.  $(e, l_1)$              $e < l_1$   
 2.  $pair(e, l_2 \dots l_n)$

% pair (E: element, L: list, LRez: list)  
 % (i, i, o) – non-deterministic

pair (A, [B | \_], [A, B]) :-  
     A < B.  
 pair (A, [\_ | T], P) :-  
     pair (A, T, P).

- the nondeterministic predicate `pairs (list, list)` (flow model (i, o)), which will produce pairs in ascending order between the elements of the argument list

```
? pairs ([2, 1, 4], L)
L = [2, 4]
L = [1, 4]
```

```
pairs( $l_1 l_2 \dots l_n$ ) =
1. pair( $l_1, l_2 \dots l_n$ )
2. pairs( $l_2 \dots l_n$ )
```

```
% pairs (L: list, LRez: list)
% (i, o) – non-deterministic
```

```
pairs ([H | T], P) :-
    pair(H, T, P).
pairs ([_ | T], P) :-
    pairs(T, P).
```

- the main predicate `allPairs (list, listRez)` (flow model (i, o)), which will collect all solutions of the nondeterministic predicate `pairs`.

```
% allPairs (L: list, LRez: list)
% (i, o) –deterministic
```

```
allPairs (L, LRes) :-
    findall (X, pairs (L, X), LRez).
```