Lecture 3 - Determinist and non-determinist predicates

Content

Bibliography

Czibula, G., Pop, H.F., Elemente avansate de programare în Lisp şi Prolog. Aplicaţii în Inteligenţa Artificială, Ed. Albastră, Cluj-Napoca, 2012

1. Determinist and non-determinist predicates

Types of predicates

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

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

1.1 Predefined predicates

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

1.2 The predicate "findall" (determination of all solutions)

Prolog provides a way to find all the solutions of a predicate at the same time: the findall predicate, which collects in a list all the solutions found.

findall(arg1, arg2, arg3)

It has the following parameters:

- the first parameter specifies the argument in the predicate considered to be collected in the list;
- the second parameter specifies the predicate to be solved;
- the third parameter specifies the list in which the solutions will be collected.

EXAMPLE

```
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]
```

1.3 Negation - "not", "\ +"

not(subgoal (Arg1, ..., ArgN)) true if subgoal fails (can't be proven to be true)
\ + subgoal (Arg1, ..., ArgN)

?- \ + (2 = 4).	?- not (2 = 4).
true.	**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 list (homogeneous) 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 of the list is the list.

Here are some examples:
The head of the list [a, b, c] is a
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

Prologue 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 an empty list, which can break down in the head and 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
$F \leftarrow P$

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

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

- the description is not directly recursive, collector variables (i, P) are used

```
% fact (N: integer, F: integer)
% (i, i), (i, o) - determinist
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) - determinist
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:

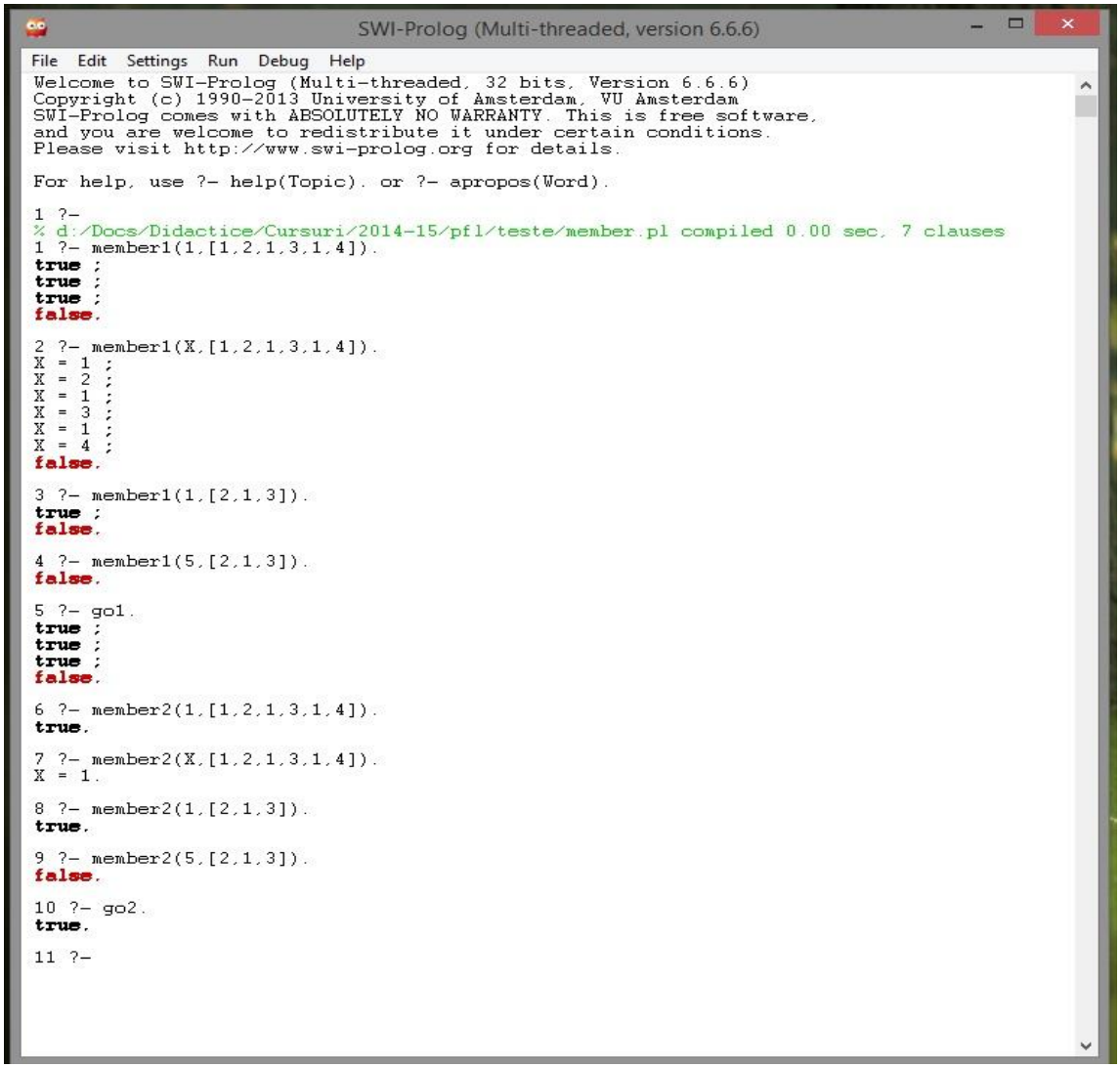
1. E is a member of the list L if it is its head.
2. Otherwise, E is a member of the list L if it is a member of the tail of L.

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.

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

1. Version 1	**2. Version 2**
% member (e: element, L: list) % (i, i) - (non) determinist % (o, i) – non-determinist member1 (E, [E | _]). member1 (E, [_ | L]) :- member1 (E, L). go1 :- member1 (1, [1,2,1,3,1,4]).	% member (e: element, L: list) % (i, i) - determinist, %(o, i) – determinist member2 (E, [E | _]) :-!. member2 (E, [_ | L]) :- member2 (E, L). go2 :- member2 (1, [1,2,1,3,1,4]).



3. Version 3

% member (e: element, L: list)
% (i, i) - (ne) determinist, (o, i) - nedeterminist

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

?- member3 (E, [1,2,3]).	?-member3 (4, [1,2,3]).	?- member3 (2, [1,2,3]).
E = 3;	**false.**	true;
E = 2;		**false.**
E = 1.		

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

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

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

L = [1, 2, 3]

Recursive formula:

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

Version 1

```
% add (e: element, L: list, LRez: list)
% (i, i, o) – determinist

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) – determinist

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 θ(n).

EXAMPLE 2.4 Determine the inverse of a list.

Version A (direct recursion)

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

```
% inverse (L: list, LRez: list)
% (i, o) – determinist

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)

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

$$invers(l_1 l_2 \dots l_n) = invers\_aux(l_1 l_2 \dots l_n, \varnothing)$$

```
% inverse (L: list, LRez: list)
% (i, o) – determinist

inverse (L, Rez) :- invers_aux ([], L, Rez).

% invers_aux (Col: list, L: list, LRez: list) - the first argument is the collector
% (i, i, o) - determinis

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)

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

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

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 θ(n), n being the number of items in the list.

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

Version B (with collector variable)

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

$$even(l_1 l_2 \dots l_n) = even\_aux(l_1 l_2 \dots l_n, \varnothing)$$

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

even(L, Rez) :-
        even_aux (L, Rez, []).
% even_aux(L: list, LRez: list, Col: list)
% (i, o, i) – determinist

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 θ(n2), 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 nondeterminist 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) $\quad e < l_1$
2. $pair(e, l_2 \dots l_n)$

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

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-determinist

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 nondeterminist predicate pairs.

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

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