INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Computer Science Department

Ch 4 – Arithmetics in Prolog

Mireille Ducassé

Last revision April 2024

Remember: Lists

- Lists are the most important structured terms of Prolog
- The empty list ([]) is an important special list
- List treatment is in general recursive, following most of the time the pattern:

```
do_something([], Res0):-
        do_empty_list(Res0).
do_something ([Head | Tail], [ResHead | ResTail):–
        do_one(Head, ResHead),
        do_something (Tail, ResTail).
```

A “theoretical” way to logically handle numbers : successor

Suppose we use the following way to write numerals:

1. **0** is a numeral.
2. If **X** is a numeral,
   then so is **succ(X)**.

```
numeral(0).
numeral(succ(X)):-
    numeral(X).
```

See Chapter 2

```
/*

?- numeral(succ(succ(succ(0)))).
yes

?- numeral(succ(1)).
No

?- numeral(2).
No

?- numeral(X).
X=0;
X=succ(0);
X=succ(succ(0));
X=succ(succ(succ(0)));
X=succ(succ(succ(succ(0))))

*/
```

Addition with succ/1

A program that adds two numbers represented with functor succ/1

?- add(succ(succ(0)),succ(succ(succ(0))), Result).
Result = succ(succ(succ(succ(succ(0)))))
yes

Code

```
add(0,X,X).
add(succ(X),Y,succ(Z)):-
    add(X,Y,Z).
```

In real programs
we cannot afford to use succ !

- The strength of Prolog is that everything is a term
  - **Symbolic treatment is made easy**
- However, at some point we need to calculate on numbers
  - But foo(3, 6) or foo(bar, crush) or 3+6 are handled in the same way (**trees**)
  - You can tell Prolog to go beyond logic programming and evaluate/calculate numbers
    - With predicate **is/2**

Extra-logical predicate is/2

- Ex: X is 3+6
  - Asks Prolog to take 3+6 as a number that needs to be evaluated
  - The result of the evaluation is unified with X
- Beware: "is" is extra-logical
  - Errors thus can occur if left hand side or right hand side are ill-typed

Examples

- X is 3+4
  X=7
- 7 is 3+4
  Yes
- 8 is 3+4
  No
- 3+4 is 3+4
  No

- 3 is Y+1
  Error
- X is Y+1
  Error
- X=3+4
  X=3+4

⚠ When an error occurs the whole execution aborts !

Comparison operators

- 5> 3

  yes
- 5*2 >= 3+4

  Yes
- X < 3

  Error

⚠ The arithmetic evaluation is done on both sides of comparison operators

length/2

Let's write a predicate that calculate the length of a list (my_length/2)

Test cases

```
?- my_length([], X).
X=0
?- my_length([a, 1, [c, d]], X).
X=3
?- my_length([a, 1, [c, d]], 5).
No
```

Code

```
my_length([], 0).
my_length ([_ | L], Len0):-
    my_length (L, Len),
    Len0 is Len + 1.
```

Try to execute

```
my_length([], 0).
my_length ([_ | L], Len0):-
    Len0 is Len + 1 ,
    my_length (L, Len).
```

What happens ? Why ?

Beware of the order of predicates, especially for extra-logical predicates

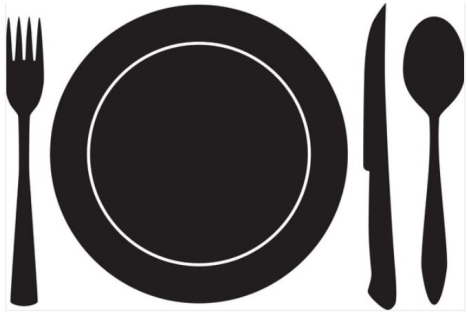
factorial/2

- The factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n.
  For example, 5!=5×4×3×2×1=120.
- The value of 0! is 1, according to the convention for an empty product.

```
factorial(N, F) :-
        N>0,
        N1 is N-1,
        factorial(N1, F1),
        F is N*F1.
factorial(0, 1).
```

Warning: this predicate can only be used to calculate F from N (not N from F)
:- mode factorial(++, ?).

Back to French menu: Update 5



- We want to build/verify balanced menus, in terms of Calories
- How to proceed
  - **add one argument to rule predicates in order to collect the calories in a list**
    - see next slide for facts
  - **define predicate add_cal(CalList, Cal)**
    - that adds all the calories of CalList into cal (It is recursive!)
  - **define predicate check_french_menu(M, Cal, MaxCal)**
    - that calls french_menu/3,
    - adds all the calories and
    - check that the result Cal =< MaxCal

Test each step before moving to the next one

Test case for first step

```
?- french_menu(M, CalList).
M = [salad, steak_with_vegetables]
CalList = [5, 150, 0]
…
```

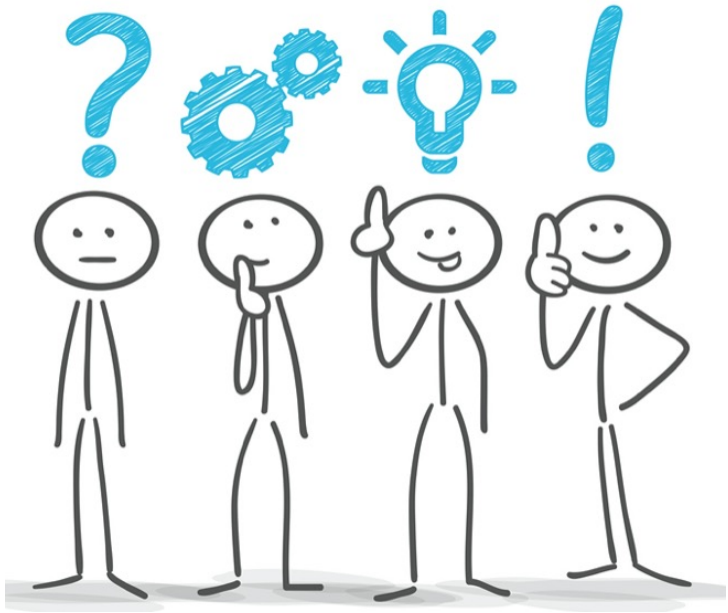
Final test case

```
?- check_french_menu(Menu, Cal, 600).
Menu = [salad, steak_with_vegetables]
Cal = 155
…
```

French Menu Update 5: facts

```
appetizer(salad, 5).
appetizer(poached_egg, 5).
appetizer(artichoke , 6).

meat_course(steak_with_vegetables , 150).
meat_course(chicken_with_fries , 250).

fish_course(trout_with_rice , 200).
fish_course(salmon_with_eggplant , 140).

veggy_course(falafel_with_rice , 220).
veggy_course(vegetable_lasagna , 350).

dessert(fruit_salad , 150).
dessert(fresh_fruit , 80).
dessert(cake , 200).

cheese(roquefort , 80).
cheese(camembert , 100).
```

The actual numbers in the facts have little to do with reality. They are given for the sake of testing.



Take your time to search, code and test your own program

Then take your time to understand the following solution

French Menu Update 5 (bis)

```
appetizer(salad, 5).
appetizer(poached_egg, 5).
appetizer(artichoke , 6).

meat_course(steak_with_vegetables , 150).
meat_course(chicken_with_fries , 250).

fish_course(trout_with_rice , 200).
fish_course(salmon_with_eggplant , 140).

veggy_course(falafel_with_rice , 220).
veggy_course(vegetable_lasagna , 350).

dessert(fruit_salad , 150).
dessert(fresh_fruit , 80).
dessert(cake , 200).

cheese(roquefort , 80).
cheese(camembert , 100).

%add_cal/2
add_cal([], 0).
add_cal([Head | Tail], Cal) :-
        add_cal(Tail, Cal1),
        Cal is Head + Cal1.

% check_french_menu/3
check_french_menu(M, Cal, MaxCal) :-
        french_menu(M, CalList),
        add_cal(CalList, Cal),
        Cal =< MaxCal.
%french_menu/1
french_menu([A | Tail], [ApCal | TailCal]) :-
        appetizer(A , ApCal),
        menu_main(Tail, TailCal).
french_menu(M, Cal) :-
        menu_main(M, Cal).
% menu_main/1
menu_main([M | Tail], Cal) :-
        main_course(M, MaCal),
        menu_dessert(Tail, TailCal),
        append(MaCal, TailCal, Cal).
% menu_dessert/1  (and/or cheese)
menu_dessert([], [0]).
menu_dessert([A], [Cal]) :-
        dessert_or_cheese(A, Cal).
menu_dessert([C, D], [CalC, CalD]) :-
        cheese(C, CalC),
        dessert(D, CalD).
```