Programming in Prolog

Recursion

Romain Barnoud

Thanks to:
Dr Fariba Sadri
Claudia Schulz

Recursion

Definition

A ***recursive*** predicate is a predicate that calls itself.


```
rec_pred(x_1, x_2, ..., x_n) :-
    goal_1,
    ...,
    goal_p,
    rec_pred(y_1, y_2, ..., y_n),
    goal_p_1,
    ...,
    goal_q.
```


A predicate is ***tail recrusive*** if the recursive call is the last goal in each recursive rule.

Example 1 – Factorial

Mathematical Definition

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

$$n! = (n-1)! \times n$$

Prolog Implementation


```
factorial(N, FN) :-
    M is N-1,
    factorial(M, FM),
    FN is N*FM.
```


Can you spot the problem?

Example 1 – Factorial

Prolog Implementation – Corrected

```
factorial(0, 1).

factorial(N, FN) :-
    N > 0,
    M is N-1,
    factorial(M, FM),
    FN is N*FM.
```


Do not forget the base case!

Usually, the base case(s) is/are placed *before* the recursive case(s).

Recursion vs. Loops

Loops do not exist in Prolog: you have to use recursion!

C++

```
while(i<n) {
    // do A
    ++i;
}
// do B
```

Prolog

```
while_loop(I, N, X1, X2, ...) :-
    I =:= N,
    % do B

while_loop(I, N, X1, X2, ...) :-
    I < N,
    % do A
    NewI is I+1,
    while_loop(NewI, N, Y1, Y2, ...).
```

Example 2 – Ackermann's Function

Mathematical Definition

$$A(m, n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Prolog Implementation – Revisited

```
ackermann(0, N, R) :-
    R is N+1.

ackermann(M, 0, R) :-
    M > 0,
    M1 is M-1,
    ackermann(M1, 1, R).
```

```
ackermann(M, N, R) :-
    M > 0, N > 0,
    N1 is N-1,
    ackermann(M, N1, R1),
    M1 is M-1,
    ackermann(M1, R1, R).
```

Different flavours of recursion

Program 1

```
natural_number(0).

natural_number(N) :-
    natural_number(M),
    N is M+1.
```

Program 2

```
natural_number(0).

natural_number(N) :-
    natural_number(M),
    M is N-1.
```

Program 3

```
natural_number(0).

natural_number(N) :-
    N is M+1,
    natural_number(M).
```

Program 4

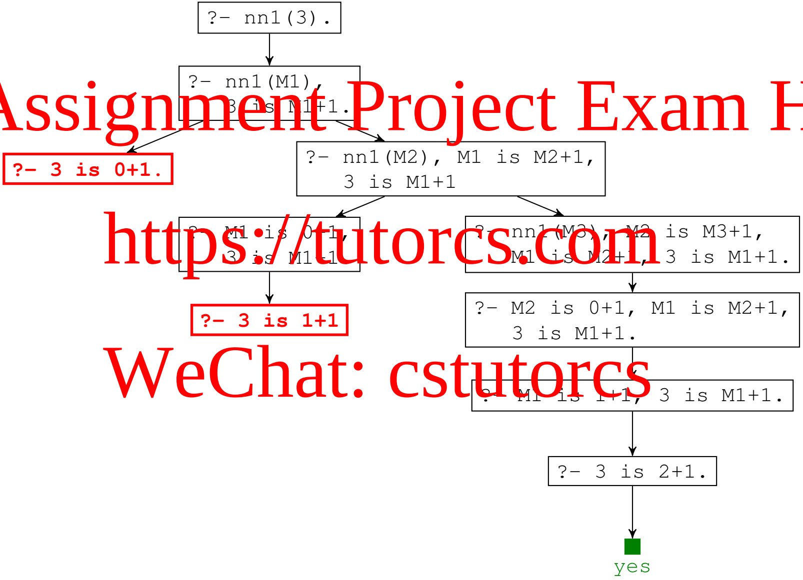
```
natural_number(0).

natural_number(N) :-
    M is N-1,
    natural_number(M).
```

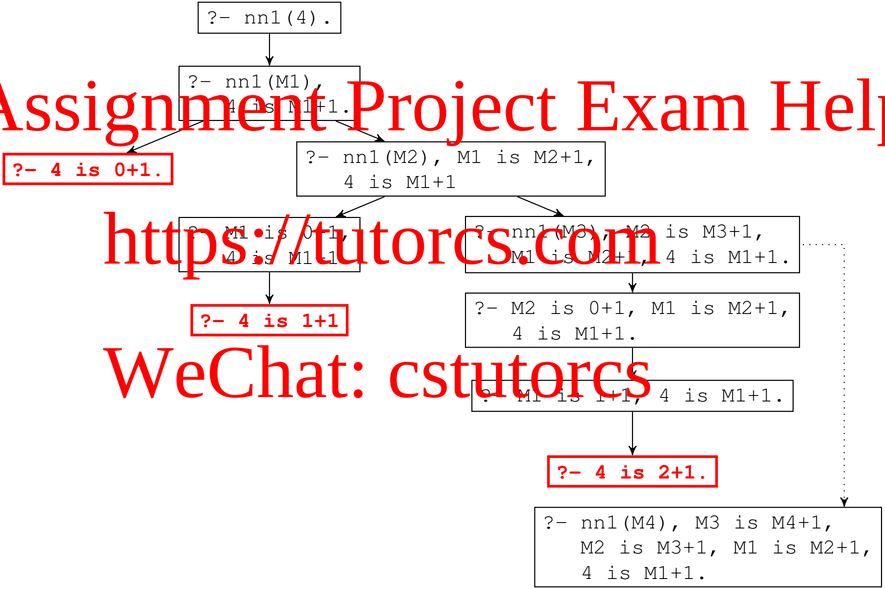
Different flavours of recursion

	Test	Generate	Remark
Program 1	✓	✓	Slower than Program 4
Program 2	✗	✗	
Program 3	✗	✗	Tail recursive, but does not work
Program 4	✓	✗	Tail recursive, most efficient (for testing)

Why is the tail recursive predicate much faster?



Why is the tail recursive predicate much faster?



Why is the tail recursive predicate much faster?

```
?- nn1(3).
?- nn1(M1), 3 is M1+1.
?- 3 is 0+1.
?- nn1(M2), M1 is M2+1, 3 is M1+1
?- M1 is 0+1, 3 is M1+1.
?- 3 is 1+1
?- nn1(M3), M2 is M3+1, M1 is M2+1, 3 is M1+1.
?- M2 is 0+1, M1 is M2+1, 3 is M1+1.
?- M1 is 1+1, 3 is M1+1.
?- 3 is 2+1.
yes
```

```
?- nn4(3).
?- M is 3-1, nn4(M).
?- nn4(2).
?- M is 2-1, nn4(M).
?- nn4(1).
?- M is 1-1, nn4(M).
?- nn4(0).
yes
```

Computing 'nn1(N)' will take $N(N-1)/2$ additional steps compared to 'nn4(N)'

Factorial revisited

Non tail-recursive version

```
factorial(0, 1).

factorial(N, FN) :-
    N > 0,
    M is N-1,
    factorial(M, FM),
    FN is N*FM.
```

How to make `factorial` tail-recursive?

Hint:

How would you code `factorial` in an imperative language?

C/C++ version

```
int factorial(int n) {
    int x = 1;
    while(n > 0) {
        x *= n;
        --n;
    }
    return x;
}
```

Solution: use an ***accumulator***!

Tail-recursive version

```
factorial(N, FN) :-
    trf(N, 1, FN).  ---------->  int acc = 1;

trf(0, Acc, Res) :-
    Res is Acc.  ------------->  return acc;
trf(N, Acc, Res) :-
    N > 0,  ------------------>  while(n > 0) {
    NewAcc is Acc * N,  ------>      acc *= n;
    M is N-1,  --------------->      --n;
    trf(M, NewAcc, Res).  ---->  }
```

Factorial revisited

Tail-recursive version

```
factorial(N, FN) :-
    trf(N, 1, FN).

trf(0, Acc, Res) :-
    Res is Acc.
trf(N, Acc, Res) :-
    N > 0,
    M is N-1,
    NewAcc is Acc * N,
    trf(M, NewAcc, Res).
```

```
?- factorial(4, F).
```

```
?- trf(4, 1, F).
```

```
?- 4 > 0,
   M is 4-1,
   NewAcc is 1*4,
   trf(M, NewAcc, F).
```

```
?- trf(3, 4, F).
```

```
?- trf(2, 12, F).
```

```
?- trf(1, 24, F).
```

```
?- trf(0, 24, F).
```

F = 24

Summary

- Do not forget the base case.

- Think about how you will use your predicate
(and in particular, which arguments will be ground).

- The order of the rules, the calls in the rules and the calls in the query are extremely important (both for recursive and non-recursive procedures): try to design tail-recursive programs.

- Use `trace` to see how your predicate is working.

Declarative vs. Procedural Meaning

Consider the rule '`A :- B, C.`'

- Declarative meaning (logical interpretation): $A \leftarrow B \wedge C$, *i.e.* A is true if B is true and C is true.
- Procedural Meaning (how Prolog interprets the rule): to prove A, prove B and ***then*** prove C (order matters!).

Consider the rules '`A :- B. A :- C.`'

- Declarative meaning: $A \leftarrow B \vee C$
- Procedural meaning: to prove A, prove B. If B does not hold, then to prove A, prove C (again, order matters).

Example: '`p :- p.`'

- Declarative meaning: $p \leftarrow p$ (tautology)
- Procedural meaning: to prove p, prove p... Infinite loop!

Declarative vs. Procedural Meaning

Consider the program `P` and the query `Q`

	Prolog	Logic
`Q` ground	`yes.`	$P \models Q$
	`no.`	$P \not\models Q$
`Q` contains variables	θ (Prolog outputs a valid substitution)	$P \models \forall X_1, \ldots, X_k (Q\theta)$
	`no.`	$\forall \theta, P \not\models \forall X_1, \ldots, X_k (Q\theta)$