Functional and
Logic programming

Tutorial 3: Tail-recursion, Cut and effective programming

Task 1: Factorial

- $5! = 5 \times 4 \times 3 \times 2 \times 1$
- Define **factorial(N,F)** such that $F = N!$
- **X = 1 + 1** means **unification**, not **computation**.
- Instead you want to use **X is 1 + 1**.
- Variables must be instantiated, hence **X is Y + Z** fails.

Task 2: Let's make factorial faster

- You can evaluate performance using **time(factorial(10000,_)).**
- Not impressive? Use *tail-recursion*.
- Let's make **factorial2(N,·)**, which calls **factorial2 (N-1,·)** as the last subgoal of its definition.

CUT !

GOTO of logic programming

What does ! do?

1) Cuts off clauses below

q(b).
q(c).

p(a).
p(X) :- q(X), !.
p(d).

Give me the answers for:

?- p(X).
?- p(a).
?- p(b).
?- p(c).
?- p(d).

What does ! do?

2) Cut of search tree in front of "!"

- Study the code to see this effect.

Cut the search space

- Take your assignment 1 and modify the definition of **father(X,Y)**.
- If the father of *X* is found to be *Y*, it is no longer needed to search for other possibilities (no one has 2 fathers).
- Call **trace.** and see the length of the derivation.

Cut the search space

- Make two definitions of **max(X,Y,Z)**
  Z is the maximum of $\{X, Y\}$.

1. With "!"
2. Without "!"

- Which is simpler? *More* effective?

Declaring your own X \= Y

- In the assignment you have already encountered **X \= Y** which fails if X and Y can be unified.
- Now try defining your own **diff(X,Y).**
- You may need **fail/0** which always fails.

Declaring your own "not"

- In the assignment you have already encountered **not(·)**.
- Now try defining your own **my_not(Goal)** which succeeds only if the Goal fails.
- You may need two predicates:
  - **call(Goal)** executes Goal
  - **fail** always fails