DEPARTMENT OF INFORMATICS ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2025/2026 – 1st Semester

Recursion and Arithmetic

Jácome Cunha
jacome@fe.up.pt

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Recursion
  - Recursion
    - Recursion
      - Recursion
        - Recursion

- Arithmetic

Agenda

- Recursion
  - Recursion
    - Recursion
      - Recursion
        - Recursion
- Arithmetic

Recursion

- Some relations are recursive

```
ancestor(X, Y):-                    % X is an ancestor of Y
      parent(X, Y).                 % if X is a parent of Y

ancestor(X, Y):-                    % X is an ancestor of Y
      parent(X, Z),                 % if X is a parent of Z
      ancestor(Z, Y).               % and Z is an ancestor of Y
```

- Recursion is based on the inductive proof
  - One or more base clauses
  - One or more recursion clauses

The order of clauses and goals may influence performance, or even cause infinite computations

Recursion

- Example: sum all numbers between 1 and N

```
sumN(0, 0).                                % Base clause

sumN(N, Sum):- N > 0,                      % Guard - make sure we don't
                                           %    have infinite recursion
               N1 is N-1,
               sumN(N1, Sum1),             % Recursive call
               Sum is Sum1 + N.
```

Recursion

- Example: sum all numbers between 1 and N

```
sumN(0, 0).

sumN(N, Sum):- N > 0,
               N1 is N-1,
               sumN(N1, Sum1),
               Sum is Sum1 + N.
```

```
| ?- sumN(2, Sum).
        1      1 Call: sumN(2,_925) ?
        2      2 Call: 2>0 ?
        2      2 Exit: 2>0 ?
        3      2 Call: _1935 is 2-1 ?
        3      2 Exit: 1 is 2-1 ?
        4      2 Call: sumN(1,_1955) ?
        5      3 Call: 1>0 ?
        5      3 Exit: 1>0 ?
        6      3 Call: _6589 is 1-1 ?
        6      3 Exit: 0 is 1-1 ?
        7      3 Call: sumN(0,_6609) ?
?       7      3 Exit: sumN(0,0) ?
        8      3 Call: _1955 is 0+1 ?
        8      3 Exit: 1 is 0+1 ?
?       4      2 Exit: sumN(1,1) ?
        9      2 Call: _925 is 1+2 ?
        9      2 Exit: 3 is 1+2 ?
?       1      1 Exit: sumN(2,3) ?
Sum = 3 ?
```

Tail Recursion

- Tail Recursion can increase efficiency
  - Add a new argument to the predicate: the accumulator
  - Make the recursive call the last call

```
sumN(N, Sum):- sumN(N, Sum, 0).            % Encapsulate
sumN(0, Sum, Sum).                         % Base case – the result is
                                           %     in the accumulator
sumN(N, Sum, Acc):- N > 0,
                    N1 is N-1,
                    Acc1 is Acc + N,
                    sumN(N1, Sum, Acc1).      % Recursive call is now
                                              %   the last sub-goal
```

To increase efficiency, we actually need to add a *cut* in the base clause - we'll see this operator next week

Tail Recursion

```
| ?- trace, sumN(2, S), notrace.
% The debugger will first creep -- showing everything
        1      1 Call: sumN(2,_941) ?
        2      2 Call: 2>0 ?
        2      2 Exit: 2>0 ?
        3      2 Call: _2067 is 2-1 ?
        3      2 Exit: 1 is 2-1 ?
        4      2 Call: sumN(1,_2087) ?
        5      3 Call: 1>0 ?
        5      3 Exit: 1>0 ?
        6      3 Call: _6721 is 1-1 ?
        6      3 Exit: 0 is 1-1 ?
        7      3 Call: sumN(0,_6741) ?
?       7      3 Exit: sumN(0,0) ?
        8      3 Call: _2087 is 0+1 ?
        8      3 Exit: 1 is 0+1 ?
?       4      2 Exit: sumN(1,1) ?
        9      2 Call: _941 is 1+2 ?
        9      2 Exit: 3 is 1+2 ?
?       1      1 Exit: sumN(2,3) ?
       10      1 Call: notrace ?
% The debugger is switched off
S = 3 ?
yes
```

```
| ?- trace, sumN(2, S, 0), notrace.
% The debugger will first creep -- showing
        1      1 Call: sumN(2,_941,0) ?
        2      2 Call: 2>0 ?
        2      2 Exit: 2>0 ?
        3      2 Call: _2111 is 2-1 ?
        3      2 Exit: 1 is 2-1 ?
        4      2 Call: _2129 is 0+2 ?
        4      2 Exit: 2 is 0+2 ?
        5      2 Call: sumN(1,_941,2) ?
        6      3 Call: 1>0 ?
        6      3 Exit: 1>0 ?
        7      3 Call: _8679 is 1-1 ?
        7      3 Exit: 0 is 1-1 ?
        8      3 Call: _8697 is 2+1 ?
        8      3 Exit: 3 is 2+1 ?
        9      3 Call: sumN(0,_941,3) ?
        9      3 Exit: sumN(0,3,3) ?
        5      2 Exit: sumN(1,3,2) ?
        1      1 Exit: sumN(2,3,0) ?
       10      1 Call: notrace ?
% The debugger is switched off
S = 3 ?
yes
```

Agenda

- Recursion
  - Recursion
    - Recursion
      - Recursion
        - Recursion

- Arithmetic

Arithmetic

- Arithmetic expressions are not evaluated immediately
  - Example: A = 4+2 unifies A with the term +(4, 2), not the value 6

```
| ?- A = 4+2.
A = 4+2 ?
yes
| ?- B is 4+2.
B = 6 ?
yes
| ?- 6 is 4+2.
yes
| ?- 4+2 is 4+2.
no
```

- The *is* predicate can be used to evaluate an arithmetic expression
  - The right-side of *is* needs to be instantiated

```
| ?- C is 4+B.
! Instantiation error in argument 2 of (is)/2
! goal:  _419 is 4+_427
```

See section 4.7 of the SICStus Manual for more information on Arithmetic

Arithmetic

- Arithmetic expressions can be compared for (in)equality
  - Expr1 =:= Expr2 evaluates both expressions and if they are equal
  - Expr1 =\= Expr2 evaluates both expressions and if they are different
  - Comparison

    `E1 < E2`   `E1 > E2`   `E1 =< E2`   `E1 >= E2`

- Prolog can also compare and order terms

    `T1 @< T2`   `T1 @> T2`   `T1 @=< T2`   `T1 @>= T2`

  - Term1 == Term2 verifies whether the two terms are literally identical
  - Term1 \== Term2 checks if the two terms are not literally identical

Arithmetic

- There are several functions available
  - X + Y, X - Y, X * Y, X / Y (float quotient)
  - X // Y is the integer quotient, truncated towards 0
  - X div Y is the integer quotient (rounded down)
  - X rem Y is integer remainder: X – Y * (X // Y)
  - X mod Y is integer remainder: X – Y * (X div Y)
  - Many other functions
    - round(X), truncate(X), floor(X), ceiling(X)
    - abs(X), sign(X), min(X, Y), max(X, Y)
    - sqrt(X), log(X), exp(X), X ** Y, X ^ Y
    - sin(X), cos(X), tan(X), …

```
| ?- A is 5 // 2.
A = 2 ?
yes
| ?- A is -5 // 2.
A = -2 ?
yes
| ?- A is 5 div 2.
A = 2 ?
yes
| ?- A is -5 div 2.
A = -3 ?
yes
| ?- A is 5 rem 2.
A = 1 ?
yes
| ?- A is -5 rem 2.
A = -1 ?
yes
| ?- A is 5 mod 2.
A = 1 ?
yes
| ?- A is -5 mod 2.
A = 1 ?
yes
```

Natural Numbers

- Arithmetic in Prolog deviates from pure Logic Programming
  - It is, however, necessary for efficiency

- A more '*logical*' representation of (natural) numbers
  - 0 is natural
  - The successor of X - *s(X)* - is natural if X is natural
    - 0, s(0), s(s(0)), s(s(s(0))), …

```
natural_number(0).
natural_number(s(X)):- natural_number(X).
```

Adding Natural Numbers

- Addition can then be seen as a ternary relation

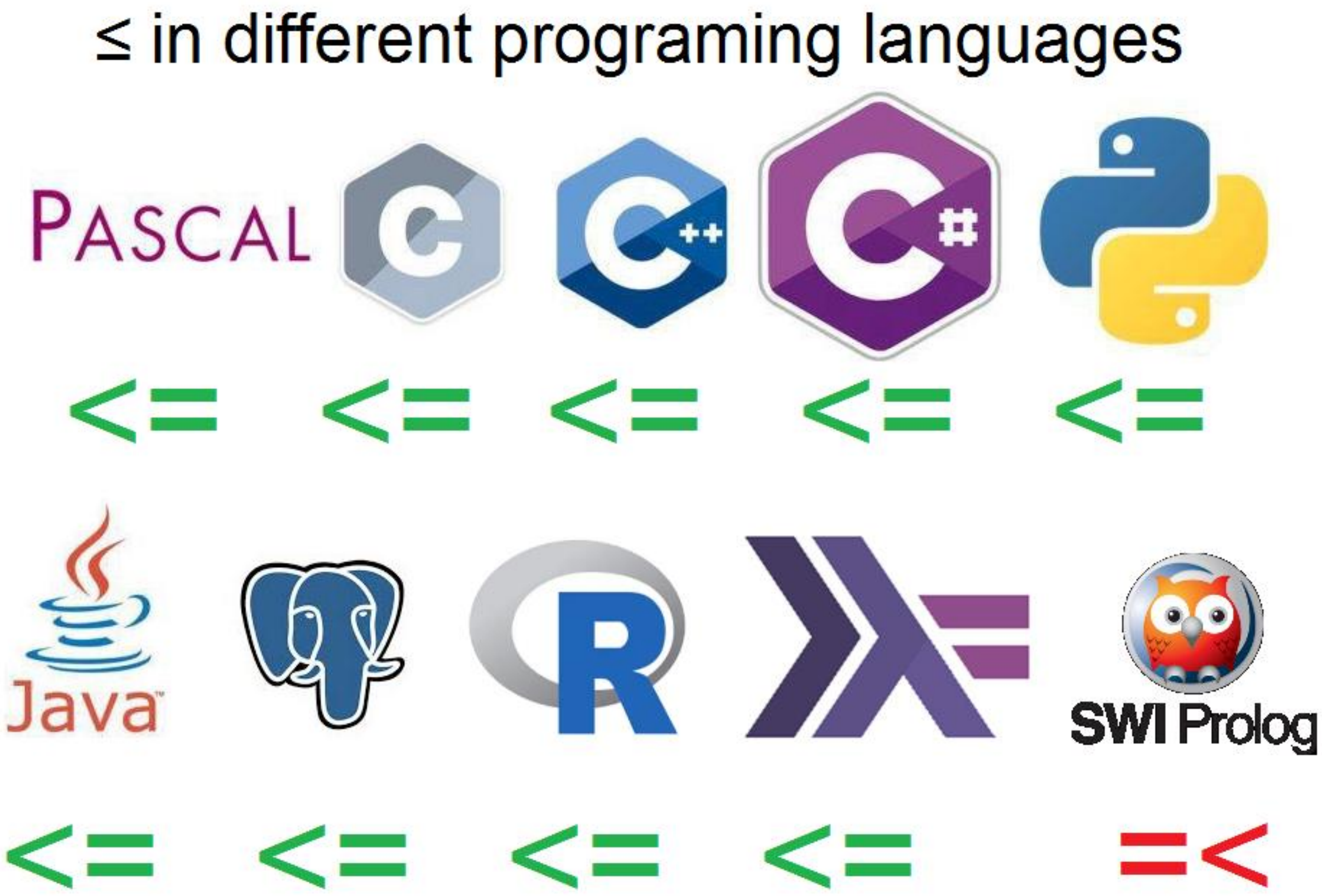
```
% plus(X, Y, Z): X + Y = Z

plus(0, X, X):-
        natural_number(X).

plus(s(X), Y, s(Z)):-
        plus(X,Y,Z).
```

```
| ?- plus( s(s(0)), s(0), Z).
Z = s(s(s(0))) ?
yes
| ?- plus( s(s(0)), Y, s(s(s(0)))).
Y = s(0) ?
yes
| ?- plus( X, s(0), s(s(s(0)))).
X = s(s(0)) ?
yes
| ?- plus( X, Y, s(s(0))).
X = 0,
Y = s(s(0)) ? ;
X = s(0),
Y = s(0) ? ;
X = s(s(0)),
Y = 0 ? ;
no
```

Q & A


≤ in different programing languages
PASCAL
<= <= <= <= <=
Java
R
SWI Prolog
<= <= <= <= =<