FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2021/2022 – 1st Semester

Introduction to Prolog

12/15/2021

Daniel Castro Silva
dcs@fe.up.pt

Agenda

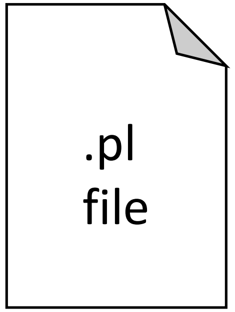
- Prolog
- Facts and Rules
- Queries
- How Prolog works
- Arithmetic
- Recursion
    - Recursion
        - Recursion
            - Recursion
- Lists

Prolog

- Prolog is the most widely used logic programming language
  - There are some language dialects, such as Edinburgh Prolog, and also a standardization – ISO Prolog
- There are several Prolog systems, both free and commercial
  - Some of the most popular are SICStus Prolog and SWI-Prolog
  - Another notable system is YAP, developed at DCC, FCUP

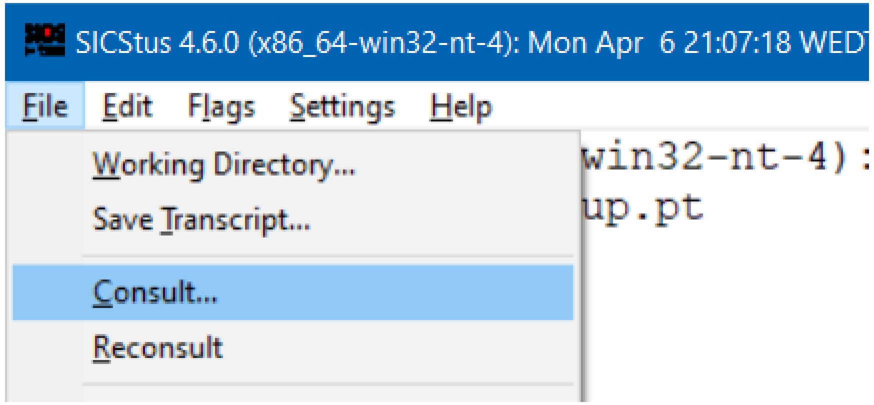
In this course, we'll be using SICStus Prolog v 4.7
(link to installer and keys are available in Moodle)

Prolog



- Write your code in a text file with a .pl extension
  - Use the text editor of your choice
- In SICStus, load it using the *File -> Consult…* Menu
  - Or call directly on SICStus:

```
| ?- consult('path/to/file.pl').
```



Alternatively, you can use SPIDER (SICStus Prolog IDE, based on Eclipse)

Prolog Programs

- A Prolog program is a finite set of predicates
  - Predicates use facts and rules to express knowledge as relations
  - Relations are generalizations of functions
    - Usually more versatile, usable in multiple directions

- A computation is a proof of a goal from a program
  - Using [a form of] SLD resolution with a unification algorithm

- A **correct** program does not allow the deduction of unwanted facts
- A **complete** program allows the deduction of everything intended

Terms

- Everything in Prolog is a *term*, which can be a *constant*, a *variable* or a *compound term*
- **Constants** represent elementary objects
  - **Numbers**
    - **Integers** (eg. 4, -8) (bases other than decimal can also be used, eg. 8’755)
    - **Floats** (eg. 1.5, -1.6) (also supports exponent, eg. 23.4E-2)
  - **Atoms**
    - Start with lower-case letter (eg. john_doe, johnSmith42)
    - String within single quotes (eg. ‘John Doe’, ‘John Smith 42’)

Terms

- **Variables** act as placeholders for arbitrary terms
    - Start with a capital letter (eg. Variable1)
    - Start with an underscore (eg. _Var2)
    - Single underscore (_) (anonymous variable)
- **Compound terms** are comprised of a *functor* and *arguments* (which are terms)
    - The functor is characterized by its *name* (an atom) and *arity* (the number of arguments), usually represented as *name/arity*
    - Eg. point/2 represents a functor named *point* with two arguments
        - point(4, 2) is a possible instance of point/2, and so is point(foo, point(3, bar))

Agenda

- Prolog
- Facts and Rules
- Queries
- How Prolog works
- Arithmetic
- Recursion
    - Recursion
        - Recursion
            - Recursion
- Lists

Facts

- Facts express a relation that is true
  - You can (kind of) interpret them as lines in a database table

Statements end with a period

```
male(homer).              % homer is a male
female(marge).            % marge is a female
father(homer, bart).      % homer is the father of bart
mother(marge, bart).      % marge is the mother of bart
```

Arguments between parentheses and separated by commas

Predicate (relation) names start with lowercase letter

Semantics

- The semantics (interpretation) needs to be defined and shared

```
father(homer, bart).        % homer is the father of bart
father(homer, bart).        % the father of homer is bart
```

- This inherent ambiguity only highlights the importance of using appropriate and descriptive names as well as code comments

```
% single-line comment

/* multi-line
   comment */
```

Naming conventions and code comments represent a part of the evaluation of the practical assignment

Rules

- Rules allow for the deduction of new knowledge from existing knowledge (facts and other rules)
  - Rules are expressed in the form of Horn Clauses:
    - Head :- Body

```
grandfather(X, Y):- father(X, Z), parent(Z, Y).     % X is the grandfather of Y
                                                      % if X is the father of Z
                                                      % and Z is a parent of Y
```

neck

and

```
%multiple definitions of a rule with the same head: rule one or rule two or...

parent(X, Y):- father(X, Y).      % X is a parent of Y if X is the father of Y
parent(X, Y):- mother(X, Y).      % X is a parent of Y if X is the mother of Y
```

Disjunction

- Disjunction can also be expressed with the ; operator

```
parent(X, Y):- father(X, Y).      % X is a parent of Y if X is the father of Y
parent(X, Y):- mother(X, Y).      % X is a parent of Y if X is the mother of Y

% is equivalent to

parent(X, Y):- father(X, Y) ; mother(X, Y)
```

- The disjunction operator (;) should be used sparingly
  - Always use parentheses to clarify

Rules

- Rules have both a declarative and a procedural interpretation
  - Declarative interpretation

```
grandfather(X, Y):-          % X is the grandfather of Y
        father(X, Z),        % if X is the father of Z
        parent(Z, Y).        % and Z is a parent of Y
```

  - Procedural interpretation

```
grandfather(X, Y):-          % to solve grandfather(X,Y)
        father(X, Z),        % first solve father(X, Z)
        parent(Z, Y).        % and then parent(Z, Y)
                             % (solve = execute)
```

Rules

- The head of a rule can have 0 or more arguments

```
parent(X, Y):- father(X, Y).      % X is a parent of Y if X is the father of Y

fathers:- father(X, Y).           % fathers is true if there is a(t least one)
                                  % father/child relation
```

A rule with no arguments is a good entry point to a program

Variables in Programs

- Variables are universally instantiated in logic programs

```
plus(0, S, S).                 % 0 is the neutral element of addition
mult(1, V, V).                 % 1 is the neutral element of multiplication


human(Homer).                  % everything is human
father(homer, Bart)            % homer is the father of everything


grandfather(X, Y):- father(X, Z), parent(Z, Y).
```

Variables occurring only in the body of a rule can be seen as existentially quantified

We need to be careful when using variables with facts

Agenda

- Prolog
- Facts and Rules
- Queries
- How Prolog works
- Arithmetic
- Recursion
  - Recursion
    - Recursion
      - Recursion
- Lists

Queries

- Computations in Prolog start with a question, which has two possible answers:
  - Yes (possible with answer substitution – variable binding)
  - No
- The attempt to prove the question right/wrong (is it a consequence of the program?) produces the computations

```
| ?- male(homer).
yes

| ?- father(homer, bart).
yes
```

```
| ?- female(marge).
yes

| ?- father(marge, bart).
no
```

Variables in Queries

- Queries can include variables
  - Variables are existentially quantified in queries
- A variable starting with an underscore is a ‘*don’t care*’

```
| ?- father(X, bart).
X = homer ?
yes

| ?- father(_X, bart).
yes

| ?- male(_).
yes
```

```
| ?- male(X).
X = homer ?
yes

| ?- male(X).
X = homer ? ;
X = bart ? n
no
```

If satisfied with the answer, just hit enter
If you want another answer, type ‘n’, ‘no’ or ‘;’

Variables and Compound Queries

- Queries can be more complex, combining goals
- Variables are used to glue together the different goals
  - Underscore alone (_) is the exception

```
| ?- male(X), parent(X, bart).
X = homer ? ;
no

| ?- male(_X), parent(_X, bart).
yes
```

```
| ?- male(_X), parent(Y, bart).
Y = homer ? ;
Y = marge ? ;
Y = homer ? ;
Y = marge ? ;
no
```

Why the duplicates?
Just wait a few slides!

Closed World Assumption

- Assumption that everything that is true is known to be true (ie, is represented as a clause in the program)
- Therefore, everything that cannot be deduced from the clauses in the program is assumed to be false

```
| ?- male(donald).
no
```

  - Requires attention to make sure everything we want to deduce can be deduced from the program clauses

Coding Efficiency Considerations

- Use implicit unification instead of additional variables

```
change_player(X, Y):- X = 1, Y = 2.
change_player(X, Y):- X = 2, Y = 1.
```

Should instead be written as

```
change_player(1, 2).
change_player(2, 1).
```

- Always place input arguments before output arguments
  - SICStus indexes predicates by their first argument

= is the unification operator (kind of ‘[possibly] equal’);
\= (not unifiable) can be interpreted as ‘can’t be equal’

Coding Style Considerations

- Although white space and code indentation are meaningless, there are some coding style guidelines you should consider following, to increase code readability:
  - Indent the code consistently
  - Put each sub-goal on a separate, indented line
  - Use human-readable names for predicates and variables
  - Try to limit the length of code lines and number of lines per clause
  - …

See Covington et al. (2012). Coding Guidelines for Prolog. Theory and Practice of Logic Programming, 12(6): 889-927

Horn Clauses

- Everything in Prolog is expressed as a Horn Clause
  - Rules are complete horn clauses (head :- body)

```
parent(X, Y):- father(X, Y).
```

  - Fact are horn clauses where the body is always true (just the head)

```
male(homer):- true          ⇔          male(homer).
```

  - Queries are horn clauses without a head (just the body)

```
| ?- father(X, bart).
```

Predicates

- A **predicate** is a set of clauses for the same functor
  - **Clauses** are either facts or rules
- Functors with the same name but different arity are different predicates

```
father(X):- father(X, Y).     % X is a father
                              % if X is the father of some Y
```

Documentation

- Documentation should include a **mode declaration** for each argument:
  - + (input): the argument is instantiated when the predicate is called
  - - (output): the argument is not instantiated in the predicate call
  - ? (in/out): the argument can be instantiated or not

```
% square(+number, -square)          % calculates the square
                                    % of a given number

% parent(?parent, ?child)           % parent/child relation
```

- One of the most powerful properties of Prolog is its versatility

Prolog Versatility

- The versatility of Prolog can be seen in most predicates:
  - For instance, parent/2 allows:
    - Confirming that two given people are parent/child
    - Obtaining the children of a given person
    - Obtaining the parents of a given person
    - Obtaining all parent/child pairs
  - In most other languages, we would need to implement four different functions to achieve this, or include extra logic to test instantiation

Agenda

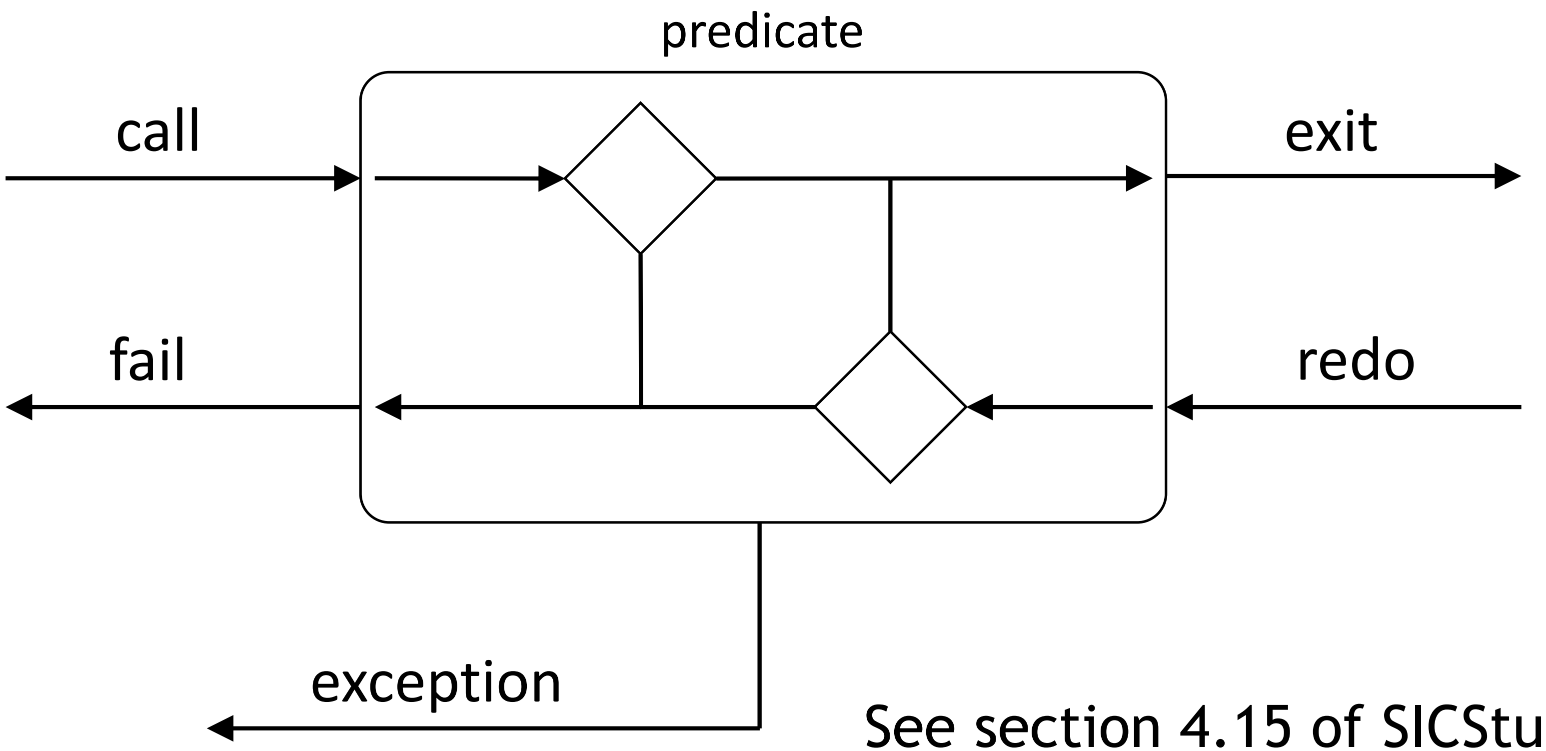
- Prolog
- Facts and Rules
- Queries
- How Prolog works
- Arithmetic
- Recursion
    - Recursion
        - Recursion
            - Recursion
- Lists

How Prolog Works

- Prolog's mechanics work
  - Top to bottom
    - The order of clauses is important
  - Left to right
    - In rules, prove sub-goals in left-to-right order
  - With backtracking
    - If a sub-goal fails, go back to previous decision point

The Prolog Box Model

- Each call to a goal can be modelled as a four-gate box model



See section 4.15 of SICStus's Manual for more information on exceptions

Tracing

- Trace mode allows us to follow the computations step by step
  - Can be activated from the menu Flags -> Debugging -> trace
  - Or in the code, by calling *trace*
    - Disable it by calling *notrace*

```
foo(bar):- fubar(bar, baz),
           trace,                  % activate trace mode
           qux(baz),               % the call to qux will be traced
           notrace,                % deactivate trace mode
           quux(bar).
```

See section 5 of SICStus's Manual for more information on Trace and Debugging

Tracing

- Trace message format:

```
N   S   InvID  Depth  Port: Goal ?
```

  - N (only visible at Exit ports) indicates that the goal call may backtrack to find alternative solutions
  - S indicates the existence of a spypoint
  - InvID (Invocation ID) is a unique identifier for each goal (can be used to match messages from the various ports)
  - Depth is an indication of the general call depth
  - Port is one of Call, Exit, Redo, Fail or Exception
  - Goal is the current goal of the computation

Agenda

- Prolog
- Facts and Rules
- Queries
- How Prolog works
- Arithmetic
- Recursion
    - Recursion
        - Recursion
            - Recursion
- Lists

Arithmetic

- Arithmetic expressions are not evaluated immediately
  - Example: A = 4+2 unifies A with the term +(4, 2), not the value 6
- The *is* predicate can be used to evaluate an arithmetic expression

```
| ?- 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 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
```

Agenda

- Prolog
- Facts and Rules
- Queries
- How Prolog works
- Arithmetic
- Recursion
  - Recursion
    - Recursion
      - Recursion
- Lists

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

- Prolog
- Facts and Rules
- Queries
- How Prolog works
- Arithmetic
- Recursion
    - Recursion
        - Recursion
            - Recursion
- Lists

Lists

- Lists are the quintessential data structure in Prolog
- Empty list represented as [ ]
- Elements separated by commas within square brackets
  - [a, b, c]
  - [4, 8, 15, 16, 23, 42]
- Lists elements can be anything, including other lists
  - [2, [a, b, c], [3, [x, y], 4], 5]

Lists

- The internal representation uses the . functor and two arguments – the head and tail of the list
  - Ex.: [1, 2, 3] = .(1, .(2, .(3, []))))

```
| ?- A = .(1, .(2, .(3, []) ) ).
A = [1,2,3] ?
yes
```

```
  .
 / \
1   .
   / \
  2   .
     / \
    3   []
```

- Strings are a representation of lists of character ASCII codes

```
| ?- A = "Hello".
A = [72,101,108,108,111] ?
yes
```

Lists

- Easily separate the head of the list from the rest of the list
  - The head of the list can separate more than one element

```
[ H | T ] % where T is a list with the remaining elements of the list
[ 4 ] = [ 4 | [ ] ]        % tail of list with one element is empty list
[4, 8, 15, 16, 23, 42] = [4 | [8, 15, 16, 23, 42] ]
[4, 8, 15, 16, 23, 42] = [ 4, 8 | [ 15, 16, 23, 42] ]
```

- Definition of what is a list
  - An empty list
  - A list construct where tail is a list

```
is_list( [ ] ).
is_list( [H|T] ):- is_list(T).
```

List Length

- There are several useful built-in predicates to work with lists
  - ***length(?List, ?Size)***
    - Size of a list (very flexible)

```
| ?- length([1,2,3], 3).
yes
| ?- length([1,2,3], L).
L = 3 ?
yes
| ?- length(L, 3).
L = [_A,_B,_C] ?
yes
| ?- length(L, S).
L = [],
S = 0 ? ;
L = [_A],
S = 1 ? ;
L = [_A,_B],
S = 2 ? ;
L = [_A,_B,_C],
S = 3 ?
yes
```

Can also be easily implemented recursively

```
length( [ ], 0 ).
length( [_|T], L ):-
        length(T, L1),
        L is L1+1.
```

Actually, there's a small caveat with this solution. Can you find it? (homework)

List Membership

- ***member(?Elem, ?List)***
  - List member (very flexible)
- ***memberchk(?Elem, ?List)***
  - Member verification (determinate)

Can also be easily implemented recursively

```
member( X, [X|_] ).
member( X, [_|T] ):-
        member(X, T).

memberchk( X, [X|_] ).
memberchk( X, [Y|T] ):-
        X \= Y,
        memberchk(X, T).
```

```
| ?- member(2, [1,2,3]).
yes
| ?- member(2, L).
L = [2|_A] ? ;
L = [_A,2|_B] ?
yes
| ?- member(M, [1,2]).
M = 1 ? ;
M = 2 ? ;
no
| ?- member(M, L).
L = [M|_A] ? ;
L = [_A,M|_B] ?
yes
```

Appending Lists

- ***append(?L1, ?L2, ?L3)***
  - Appends two lists into a third (very flexible)

Can also be easily implemented recursively

```
append( [ ], L2, L2 ).
append( [H|T], L2, [H|T3] ):-
        append( T, L2, T3).
```

```
| ?- append([1,2], [3,4], [1,2,3,4]).
yes
| ?- append([1,2], [3,4], L).
L = [1,2,3,4] ?
yes
| ?- append([1,2], L, [1,2,3,4]).
L = [3,4] ?
yes
| ?- append(L, [3,4], [1,2,3,4]).
L = [1,2] ?
yes
| ?- append(L1, L2, [1,2,3]).
L1 = [],
L2 = [1,2,3] ? ;
L1 = [1],
L2 = [2,3] ? ;
L1 = [1,2],
L2 = [3] ? ;
L1 = [1,2,3],
L2 = [] ? ;
no
```

Sorting Lists

- ***sort(+List, -SortedList)***
  - Sorts a (proper) list
- ***keysort(+PairList, -SortedList)***
  - Sorts a (proper) key-value pair list
  - If a key appears more than once, elements retain original order

```
| ?- sort([4,2,3,1], [1,2,3,4]).
yes
| ?- sort([4,2,3,1], SL).
SL = [1,2,3,4] ?
yes
| ?- keysort([2-1, 1-2, 4-3, 3-4], SL).
SL = [1-2,2-1,3-4,4-3] ?
yes
| ?- keysort([2-1, 1-2, 4-3, 3-4, 1-1], SL).
SL = [1-2,1-1,2-1,3-4,4-3] ?
yes
```

Can also be implemented recursively
Homework!

Lists Library

- The Lists library has numerous predicates to work with lists

- Libraries can be imported using the *use_module* directive:

```
:-use_module(library(lists)).
```

See section 10.24 of the SICStus Manual for a complete description of available predicates

Lists Library

- Some useful predicates from the lists library
  - nth0(?Pos, ?List, ?Elem) / nth1(?Pos, ?List, ?Elem)
  - nth0(?Pos, ?List, ?Elem, ?Rest) / nth1(?Pos, ?List, ?Elem, ?Rest)
  - select(?X, ?XList, ?Y, ?YList)
  - delete(+List, +ToDel, -Rest) / delete(+List, +ToDel, + Count, -Rest)
  - last(?Init, ?Last, ?List)
  - segment(?List, ?Segment)
  - sublist(+List, ?Part, ?Before, ?Length, ?After)

Lists Library

- append(+ListOfLists, -List)
- reverse(?List, ?Reversed)
- rotate_list(+Amount, ?List, ?Rotated)
- transpose(?Matrix, ?Transposed)
- remove_dups(+List, ?PrunedList)
- permutation(?List, ?Permutation)
- sumlist(+ListOfNumbers, ?Sum)
- max_member(?Max, +List) / min_member(?Min, +List)
- max_member(:Comp, ?Max, +List) / min_member(:Comp, ?Min, +L)

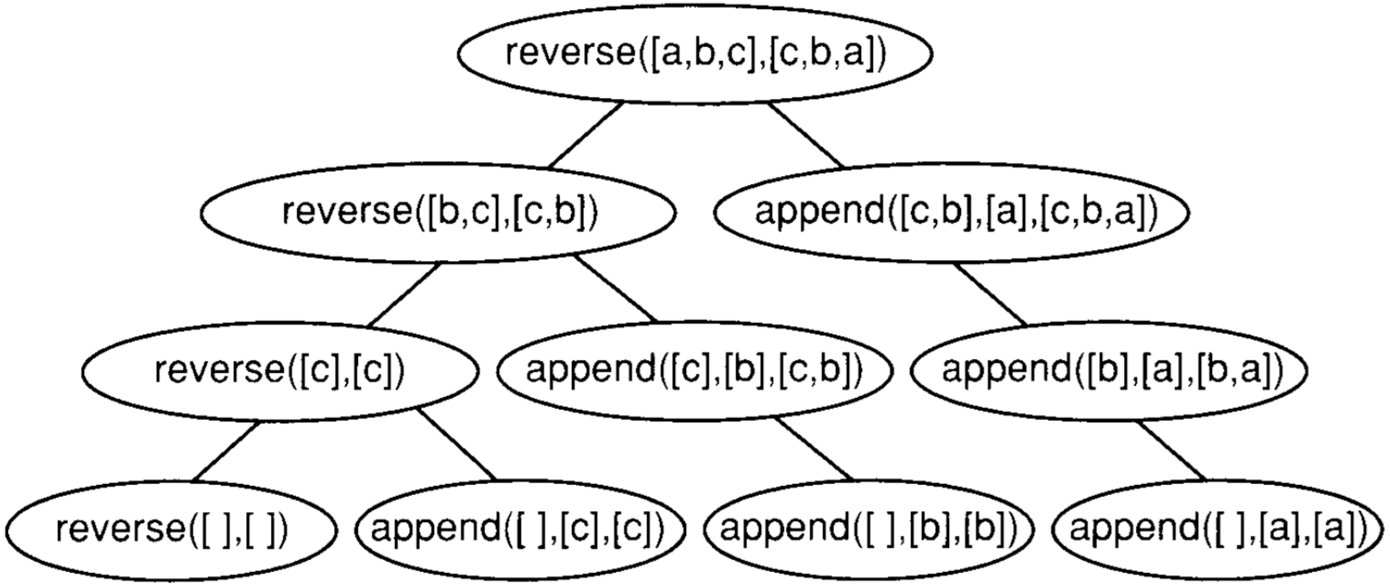
Lists Library

- maplist(:Pred, +L) / maplist(:Pr, +L1, ?L2) / maplist(:Pr, +L1, ?L2, ?L3)
- map_product(:Pred, +Xs, +Ys, ?List)
- scanlist(:Pred, +Xs, ?Start, ?Final)
- cumlist(:Pred, +Xs, ?Start, ?List)
- some(:Pred, +List) / some(:Pred, +Xs, ?Ys) / some(:Pr, +Xs, ?Ys, ?Zs)
- include(:P, +X, ?L) / include(:P, +X, +Y, ?L) / include(:P, +X, +Y, +Z, ?L)
- exclude(:P, +X, ?L) / exclude(:P, +X,+Y, ?L) / exclude(:P, +X,+Y,+Z, ?L)
- group(:Pred, +List, ?Front, ?Back)

Lists

- Several of these predicates can be implemented using append
  - However, sometimes we can find more efficient versions
- Example: list reverse

```
reverse([], []).
reverse([X|Xs], Rev):-
        reverse(Xs, Ys),
        append(Ys, [X], Rev).
```

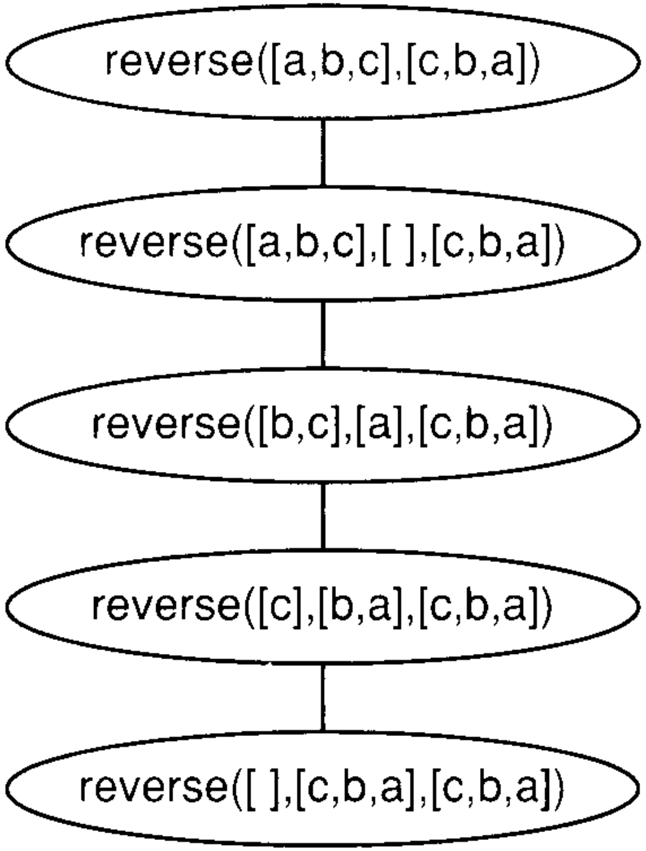


  - Size of proof tree is <u>quadratic</u> to the number of elements in the list

Lists

- We can use an accumulator (tail recursion) to reverse the list

```
reverse(Xs, Rev):- reverse(Xs, [], Rev).
reverse([X|Xs], Acc, Rev):-
        reverse(Xs, [X|Acc], Rev).
reverse([], Rev, Rev).
```



  - The accumulator holds the reversed list in the last step of the recursion
  - Now the process is <u>linear</u> to the number of elements in the list

Additional Readings

- Prolog
  - Leon Sterling and Ehud Shapiro (1994). The Art of Prolog. The MIT Press (2nd ed). ISBN: 978-0262691635
  - Krzysztof R. Apt (1996). From Logic Programming to Prolog. Prentice Hall. ISBN: 978-0132303682
  - Patrick Blackburn, Johan Bos and Kristina Striegnitz (2006). Learn Prolog Now! College Publications. ISBN: 978-1904987178
  - Max Bramer (2013). Logic Programming with Prolog. Springer (2nd ed). ISBN: 978-1447154860

Q & A