Programming Languages (Langages Evolués)

Roel Wuyts

Prolog

History of Prolog

- 1965: Unification and resolution principles
  - by Robinson
- 1973: Introduction of Prolog
  - by Colmerauer and Roussel, Université de Marseille
- 1980: Efficient implementation of Prolog
  - Edinburgh Prolog, University of Edinburgh

Meet Prolog

- Prolog
  - is interpreted
  - garbage collected
  - dynamically typed
- and... is a logic programming language

(Abstract) Prolog Interpreter

Input: A query Q and a logic program P

Output: *yes* if Q 'implied' by P, *no* otherwise

Initialise current goal set to {Q};
While the current goal is not empty do
  Choose a G from the current goal;
  Choose instance of a rule
    G :- B1, ... ,Bn from P;
    (if no such rule exists, exit while loop)
  Replace G by B1, ... ,Bn in current goal set;
  If current goal set is empty,
    output *yes*;
  else output *no*;

Programming Overview

- make database of facts and rules:

```
mother(inge, bram).
mother(nicole, inge).
grandMother(G, X) :- mother(G, M), mother(M, X)
```

- Then we can pose queries:

```
?- mother(Who, bram)
   Who = inge

?- grandMother(nicole, GrandChild)
   GrandChild = bram

?- mother(M, C)
   M = inge
   C = bram
   ...
```

Solving queries

- Prolog uses SLD resolution, backtracking and depth-first search to answer queries
- Important to know:
  - subgoals are treated left to right
  - the database is traversed top to bottom

Solving Queries: example

```
male(charles).
male(philip).

parent(charles, elizabeth).
parent(charles, philip).

father(X, M) :- parent(X,M), male(M).

?- trace(father(charles,F)).
 + 1 1 Call: father(charles,_67) ?
   + 2 2 Call: parent(charles,_67) ?
   + 2 2 Exit: parent(charles,elizabeth) ?
   + 3 2 Call: male(elizabeth) ?
   + 3 2 Fail: male(elizabeth) ?
   + 2 2 Redo: parent(charles,elizabeth) ?
   + 2 2 Exit: parent(charles,philip) ?
   + 3 2 Call: male(philip) ?
   + 3 2 Exit: male(philip) ?
   + 1 1 Exit: father(charles,philip) ? ...
```

Closed World Assumption

- Prolog adopts a closed world assumption: whatever cannot be proved to be true, is assumed to be false.
- Example

```
mother(inge, bram).
mother(nicole, inge).
grandMother(G, X) :- mother(G, M), mother(M, X)

?-mother(inge, roel).
  NO
```

- Has a lot of implications
  - especially regarding negation...

Prolog Concepts

- Facts, relations
- Queries, variables
- Clauses
- Recursion
- Functors and Lists
- Controlling Backtracking: Cut and negation
- Higher-order predicates
- Prolog Meta-Programming

Facts

```
female(anne).
male(filip).
male(pierre).
parent(anne,pierre).
parent(filip,pierre).
```

describe information
that is always true

Relations

- Collections of facts with the same name
  (*/x* represents the arity of the relation)

- Example

```
female/1
    { anne }
male/1
    { filip, pierre }
parent/2
    { (anne,pierre), (filip,pierre) }
```

Queries and variables

```
?- male(filip).
Yes

?- male(X).
X = filip
X = pierre
No

?- parent(P,pierre).
P = anne;
P = filip;
No

?- parent(P,E).
P = anne
E = pierre;

P = filip
E = pierre;

No
```

logic variables
(start with capital)

an answer of a query is a set of values that, when substituted for variables make the expression true

Anonymous variables

```
?- parent(P,E).
P = anne
E = pierre;

P = filip
E = pierre;

No

?- parent(P,_).
P = anne;
P = filip;
No

?- parent(P,_T).
P = anne;
P = filip;
No
```

anonymous logic variables
(start with underscore)

More on logic variables...

- In imperative languages, a variable is:
  - a name for a memory location that holds data of a certain type:
    - a variable always points to the same location
    - the contents of that position can change
- In logic programming languages, a variable is:
  - a substitutable parameter that can hold any kind of value
    - real variable, in a mathematical sense

Scoping

- variables do not have a scope outside the clause in which they occur
- Prolog does not support global datastructures
  - using meta-programming tricks one can *assert* and *retract* clauses from the database to have global information

Conjunction and disjunction

- Logic Conjunction: , (comma)

```
?- male(P), parent(P,E).
P = filip
E = pierre;
No
```

- Logic disjunction ; (semi-colon)

```
?- male(X); female(X).
X = filip;
X = pierre;
X = anne;
No
```

Clauses

HEAD :- BODY

```
father(P) :-
    male(P),
    parent(P,_).
mother(P) :-
    female(P),
    parent(P,_).
grandparent(X,Y) :-
    parent(X,Z),
    parent(Z,Y).
grandma(X,Y) :-
    grandparent(X,Y),
    female(X).
sibling(X, Y) :-
    parent(Z,X),
    parent(Z,Y), X\=Y.
```

Queries

```
?- father(X).
X = filip;
No

?- mother(X).
X = anne;
No

?- grandparent(X,Y).
No
```

Functors

- Functors are structured terms that can be nested
  - Example: a tree to represent the expression 5*3

```
tree('*',leaf(5),leaf(3))
```

  - Example: to represent (5+6)*(3-(2/2))

```
tree('*',
     tree('+',
          leaf(5),
          leaf(6)),
     tree('-',
          leaf(3),
          tree('/',
               leaf(2),
               leaf(2))))
```

Functors

- Recursive data structures can be manipulated
- Example:

```
operation(tree(Op,L,R),Op).
operation(tree(Op,L,R),T) :-
  L = tree(_,_,_), operation(L,T).
operation(tree(Op,L,R),T) :-
  R = tree(_,_,_), operation(R,T).


:- operation(tree('*', tree('+',leaf(5),leaf(6)),
 tree('-',leaf(3), tree('/',leaf(2),leaf(2)))), X)
    X = *;
    X = +;
    X= -;
    X = /;
    No
```

Functors and equality

- Three ways for testing equality:
  - identity

```
?- tree(X,Y) == tree(leaf(5),leaf(Z)).
 No
```

  - equality

```
?- tree(X,Y) = tree(leaf(5),leaf(Z)).
 X = leaf(5)
 Y = leaf(_G162)
 Z = _G162
```

  - mathematical equality

```
?- tree(X,Y) is tree(leaf(5),leaf(Z)).
 ERROR: Arithmetic 'tree/2' is not a function
```

Recursion

- Recursion is the sole control mechanism for loops
  - There is no other iteration mechanism
- Example: factorial:

```
factorial(0,1).
factorial(N,Fac) :-
    M is N-1,
    factorial(M,P),
    Fac is N*P.

?- factorial(3,X).
  X= 6
```

Recursion (ctd)

- Take care! Always put non-recursive clauses before recursive ones!
- Example:

```
factorial(N,Fac) :-
    N]0,
    M is N-1,
    factorial(M,P),
    Fac is N*P.
factorial(0,1).

?- factorial(5,X).
  ERROR: Out of local stack
```

Lists

- Finite list:

```
[monday,tuesday,wednesday,thursday,friday]
```

- Infinite list:

```
[ monday | Rest ]
[ monday,tuesday | Rest ]
```

  - after the | is a variable that represents the rest of the list

Lists: Internally

- Empty lists: special symbol: []
- Predefined functor .()

```
.(First,Rest)        =    [ First | Rest ]
.(a,.(b,.(c,[])))    =    [ a, b, c ]
```

- Following terms are all equal:

```
[ a, b, c ]
[ a | [b, c] ]
[ a | [b | [c|[]]] ]
```

Lists and equality

```
?- [1,2] == [1,2].
   YES

?- [X,2] = [1,Y].
   X = 1
   Y = 2

?- [X|Y] = [1,2,3].
   X = 1
   Y = [2,3]

?- [X|[2|Y]] = [1,2,Z].
   X = 1
   Y = [_G169]
   Z = _G169

?- [[1,2],3,4] = [X|Y].
   X = [1,2]
   Y = [3,4]
```

Appending lists

- Appending lists:

```
append([],X,X).
append([H|T],Y,[H|Z]) :- append(T,Y,Z).
```

- Now how does this work ?!

Example: append

```
append([],X,X).
append([H|T],Y,[H|Z]) :- append(T,Y,Z).
```

append two lists

```
append([a,b,c], [x,y], L)
        H=a, T=[b, c], Y=[x,y], L=[a|Z]
    append([b,c], [x,y], Z)
            H'=b, T'=[c], Y'=[x,y], Z=[b|Z']
        append([c], [x,y], Z')
                H''=c, T''=[], Y''=[x,y], Z'=[c|Z'']
            append([], [x,y], Z'')
                    X=Z''=[x,y]

                H''=c, T''=[], Y''=[x,y], Z'=[c,x,y]

            H'=b, T'=[c], Y'=[x,y], Z=[b,c,x,y]

        H=a, T=[b, c], Y=[x,y], L=[a,b,c,x,y]
```

Example: append (ctd)

```
append([],X,X).
append([H|T],Y,[H|Z]) :- append(T,Y,Z).

append([a,b,c], L, [a,b,c,x,y])
    H=a, T=[b,c], Y=L, Z=[b,c,x,y]
  append([b,c], L, [b,c,x,y])
      H'=b, T'=[c], Y'=L, Z'=[c,x,y]
    append([c], L, [c,x,y])
        H''=c, T''=[], Y''=L, Z''=[x,y]
      append([], L, [x,y])
          X=L=[x,y]

        H''=c, T''=[], Y''=L=[x,y], Z''=[x,y]

      H'=b, T'=[c], Y'=L=[x,y], Z'=[c,x,y]

    H=a, T=[b,c], Y=L=[x,y], Z=[a,b,c,x,y]
```

what list can be appended to [a,b,c] to produce [a,b,c,x,y] ?

Example: append (ctd)

```
append([],X,X).
append([H|T],Y,[H|Z]) :- append(T,Y,Z).

append(L, [x,y], [a,b,c,x,y])
      L=[H|T], Y=[x,y], H=a, Z=[b,c,x,y]
   append([T, [x,y], [b,c,x,y])
         T=[H'|T'], Y'=[x,y], H'=b, Z'=[c,x,y]
      append(T', [x,y], [c,x,y])
            T'=[H''|T''], Y''=[x,y], H''=c, Z''=[x,y]
         append(T'', [x,y], [x,y])
               T''=[]


            T'=[c], Y''=[x,y], H''=c, Z''=[x,y]


         T=[b,c], Y'=[x,y], H'=b, Z'=[c,x,y]


      L=[a,b,c], Y=[x,y], H=a, Z=[b,c,x,y]
```

what list can be prepended to [x,y] to produce [a,b,c,x,y] ?

member/2

- X is a member of a list whose first element is X.
  X is a member of a list whose tail is R *if* X is a member of R:

```
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R).
```

- Test membership:

```
?- member(2,[1,2,3]).
Yes
```

- Generate members of a list:

```
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

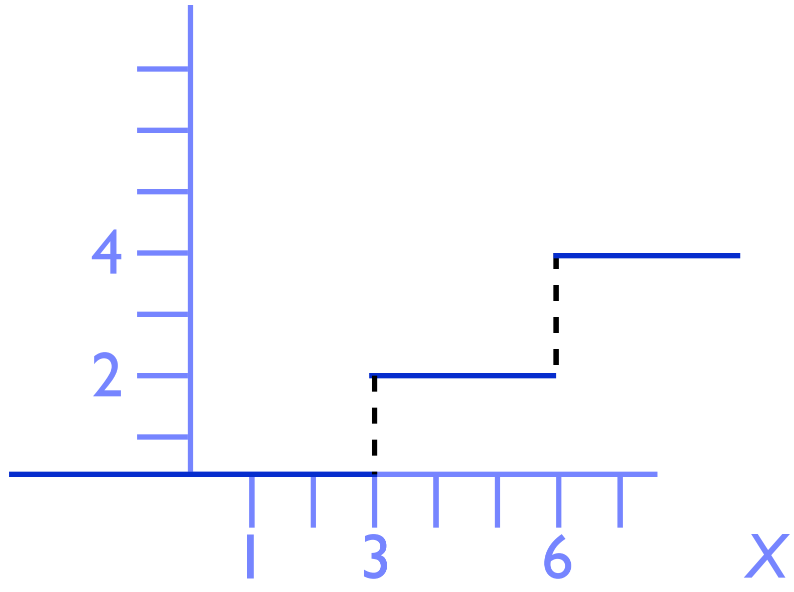
Controlling Backtracking

- Automatic backtracking is very handy
  - implicit control structure
- But can be very costly
  - unnecessary search branches are traversed
- Mechanisms to control backtracking manually
  - cut (*green cut* and *red cut*)
  - negation and explicit failure

Example program

- Double-step function:

```
f(X, 0) :- X [ 3.              %Rule 1
f(X, 2) :- 3 =[ X, X [ 6.      %Rule 2
f(X,4) :- 6 =[ X.              %Rule 3
```

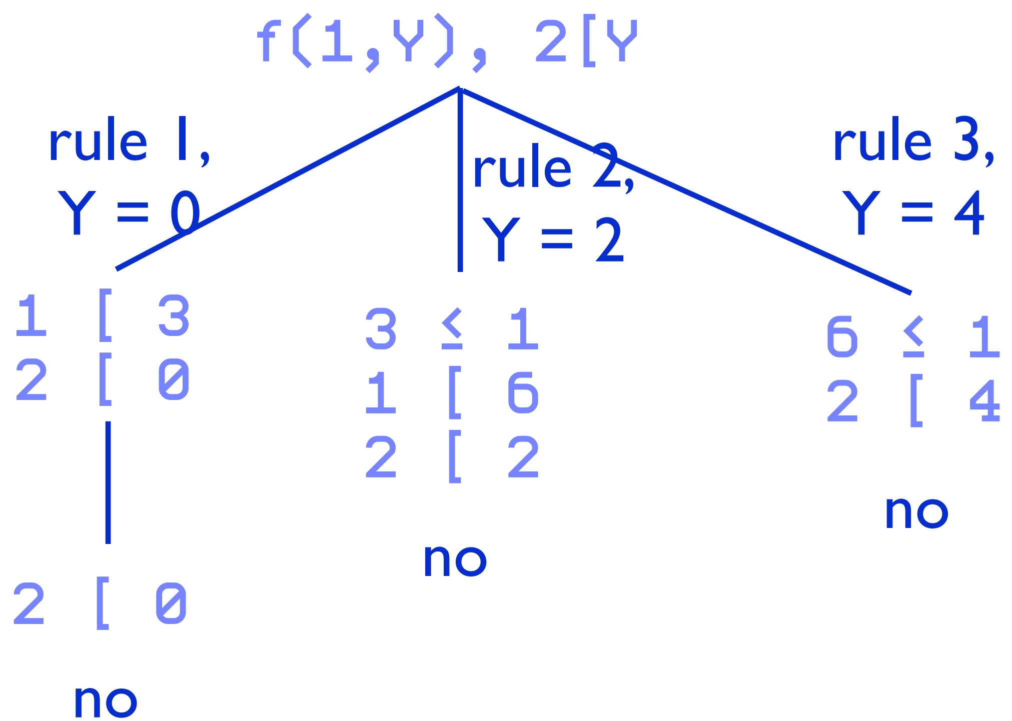


Green cut

- Suppose we ask:

```
?- f(1, Y), 2 [ Y.
No
```

- In order to do this, the three rules were tried!



Making the rules exclusive

- Add a cut to prevent backtracking:

```
f(X, 0) :- X [ 3, !.              %Rule 1
f(X, 2) :- 3 =[ X, X [ 6, !.      %Rule 2
f(X,4) :- 6 =[ X.                 %Rule 3
```

```
                f(1,Y), 2[Y
     rule 1,
     Y = 0
         1 [ 3
         2 [ 0
CUT ——— |
         2 [ 0
          no
```

Optimize further: red cut

- If the rules are exclusive, then we can optimize more, since some conditions are no longer needed

```
f(X, 0) :- X [ 3, !.                    %Rule 1
f(X, 2) :- X [ 6, !.                    %Rule 2
f(X,4).                                 %Rule 3
```

Red vs. Green cut

- The cut operator (!) commits Prolog to a particular search path.
  - Says to Prolog: "This is the right answer to this query. If later you are forced to backtrack, please do not consider any alternatives to this decision.
- Green cut: does not change declarative semantics
  - if you remove it, the program does the same
- Red cut: changes declarative semantics
  - remove it, and you have a different program

Problems with *cut*

- Have to take care of procedural aspects
  - With a cut, the order of clauses can change the declarative semantic of a program
- Example

```
p :- a, b.
p :- c.
```

$p \Leftrightarrow (a \wedge b) \vee c$

```
p :- a, !, b.
p :- c.
```

$p \Leftrightarrow (a \wedge b) \vee (\sim a \wedge c)$

```
p :- c.
p :- a, !, b.
```

$p \Leftrightarrow c \vee (a \wedge b)$

Failure

- How do we express
  "mary likes all animals but snakes"
- We can if we use *fail*, a special goal that always fails, thus forcing the parent goal to fail as well.
- The implementation becomes:

```
likes(mary, X) :-
   snake(X), !, fail.
likes(mary, X) :-
   animal(X).
```

Negation as failure

- Prolog defines not as:

```
not(P) :-
    P, !, fail
   ;
    true
```

- We can rephrase our example more cleanly as:

```
likes(mary, X) :-
    not(snake(X)), fail.
likes(mary, X) :-
    animal(X).
```

Negation: failure vs. mathematical

- negation as failure does *not* correspond to negation in mathematical logic
  - remember the close world assumption...
- Example

```
?- expensive(X)
```

$\exists x$: expensive(X)

```
?- not( expensive(X) )
```

not($\exists x$: expensive(X))

=

$\forall x$: not(expensive(X))

$\neq$

$\exists x$: not(expensive(X))

Second-order predicates

- In Prolog we have second-order predicates
  - Predicates taking other predicates as argument
- Two well-known examples:
  - forall(+Cond, +Action) : For all alternative bindings of Cond Action can be proven.
  - findall(+Template, +Goal, -Bag) : Creates a list of the instantiations Template gets successively on backtracking over Goal and unifies the result with Bag.

forall example

```
posList(L) :- forall( member(E,L), (number(E),E]0) ).

?- posList([1,2,3]).
Yes
?- posList([1,a,3]).
No
?- posList([1,2,-3]).
No
?-posList([]).
Yes
```

findall example

```
getNegatives(L1,L2) :-
  findall(E,
         (member(E,L1),number(E),E[0),
         L2).

?- getNegatives([-1,a,2,-3],L).
L = [-1, -3]
?- getNegatives([], L).
L = []

getWrappedPositives(L1, L2) :-
  findall(posNumber(E),
         (member(E, L1), number(E), E]0),
         L2).
?- getWrappedPositives([-1,a,2,-3],L).
L = [posNumber(2)]
```

Prolog Meta-programming

- One can write Prolog programs that manipulate Prolog programs
- Often-used predicates for meta-programming:
  - call
  - assert
  - retract

call

- call(Term) : succeeds if and only if the execution of Term succeeds
  - needed for treating functors as procedures
- Example: validate syntax of abstract syntax tree

```
tree(Op,X,Y) :-
   arithmeticOp(Op), treeOrLeaf(X), treeOrLeaf(Y).
arithmeticOp(Op) :-
   member(Op,['*','+','-','*','/']).
treeOrLeaf(X) :-
   X = tree(_,_,_), call(X).
treeOrLeaf(leaf(L)) :-
   member(L,[0,1,2,3,4,5,6,7,8,9]).

?- call(tree('*',leaf(5),leaf(6))).
Yes
```

Lists and functors

- =.. : Convert between functors and lists
- Example

```
?- X =.. [tree, '*', leaf(5), leaf(6)]
   X = tree(*, leaf(5), leaf(6))

?- tree('*',X,Y) =.. Z.
   X = _G158
   Y = _G159
   Z = [tree, *, _G158, _G159]
```

assert and *retract*

- allow a program to change itself dynamically by adding new facts or rules, or remove existing ones.
- Definitions:
  - assert(Clause): add Clause in the database as the last fact or clause of the corresponding predicate.
  - retract(Clause): The first fact or clause in the database that unifies with Clause is removed from the database.
  - Also exists: *asserta*, *assertz*, *retractall*

dynamic

- Clauses that are used with assert or retract have to be marked dynamic
  - Otherwise an error message is generated

```
ERROR: No permission to modify
 static_procedure `foo/2'
```

- Example: to mark foo/2 as dynamic:

```
:- dynamic mere/2.
```

Example

```
:-dynamic cachedProduct/1.

computer(g5-d2).
computer(amd64).
keyboard(diNuovo).

product(X) :- cachedProduct(X).

product(X) :-
        not(cachedProduct(X)),
        (computer(X) ; keyboard(X)),
        assert(cachedProduct(X)).

resetProductCache :- retractall(cachedProduct(_P)).

?- cachedProduct(g5-d2).
  No
?- product(g5-d2).
  Yes
?- cachedProduct(g5-d2).
  Yes
?- resetProductCache.
  Yes
```

Some Prolog Examples

- Some string manipulation examples
  - playing with whitespace
- Reasoning on class hierarchies
  - extracted from a Smalltalk system

Array of char vs. string

- Array of char codes: between double quotes
- string: between single quotes
- conversion between two: *name* predicate
- Example

```
?- name('hello', CCList).
  N = [104, 101, 108, 108, 111]

?- name(A, "hello").
  A = hello.

?- name(A, CCList).
  ERROR: name/2: Arguments are not sufficiently
   instantiated
```

whitespace

```
Let's write a predicate that relates two arrays of characters, Rest and
String, such that Rest is String with all possible whitespace characters
removed.

whitespaceChar(C) :-
   member(C, " \t\n\r").


strippedspace([], []).

strippedspace([S | Srest], Rrest) :-
   whitespaceChar(S),
   strippedspace(Srest, Rrest).

strippedspace([S|Srest], [S|Rrest]) :-
   not(whitespaceChar(S)),
   strippedspace(Srest, Rrest).

?- strippedspace("  hello", R).
   [104, 101, 108, 108, 111]
?- strippedspace("  hel    lo  ", R).
   [104, 101, 108, 108, 111]
```

whitespace

Let's write a predicate that relates two arrays of characters, Rest and String, such that Rest is a proper suffix of String with one or more whitespace characters removed.

```
whitespace0([], []).

whitespace0([S|Srest], Rrest) :-
   whitespaceChar(S),
   whitespace0(Srest, Rrest).

whitespace0([S|Rest], [S|Rest]) :-
   not(whitespaceChar(S)).

?- whitespace0("  hello", R).
   [104, 101, 108, 108, 111]
?- whitespace0(" hello ", S).
  S = [104, 101, 108, 108, 111, 32]
?- whitespace0("  hel  lo  ", R).
   [104, 101, 108, 32, 32, 108, 111, 32, 32]
```

Reasoning over class hierarchies

- Let's reason over classes and inheritance:

  1. dump some information in a file

     ```
     classes
     inheritance relationships
     ```

  2. write predicates to query this information:

     ```
     classList
     superclassOfList
     classChain
     inverseClassChain
     classInHierarchyOf
     sibling
     ```

Dumping information

```
| class superclassOf classes inheritance selectBlock |
class := 'class(''<1s>'').<n>'.
superclassOf := 'superclassOf(''<1s>'',''<2s>'').<n>'.

classes := WriteStream on: String new.
inheritance := WriteStream on: String new.

classes nextPutAll: 'rootclass(''Object'').'; cr; cr.

selectBlock := [:cl |
   classes nextPutAll: (class expandMacrosWith: cl name).
   cl subclasses do: [:each |
      inheritance nextPutAll: (superclassOf expandMacrosWith: cl name
                                                          with: each name).
      selectBlock value: each]].
selectBlock value: Object.

'classData.pl' asFilename writeStream
   nextPutAll: classes contents;
   nextPutAll: inheritance contents;
   close
```

Dump result

```
rootclass('Object').

class('Object').
class('Layout').
class('LayoutOrigin').
class('LayoutFrame').
class('AlignmentOrigin').
class('LayoutSizedOrigin').
class('NameScope').
class('LocalScope').
...

superclassOf('Object','Layout').
superclassOf('Layout','LayoutOrigin').
superclassOf('LayoutOrigin','LayoutFrame').
superclassOf('LayoutOrigin','AlignmentOrigin').
superclassOf('LayoutOrigin','LayoutSizedOrigin').
superclassOf('Object','NameScope').
superclassOf('NameScope','LocalScope').
superclassOf('LocalScope','HintedScope').
superclassOf('NameScope','NullScope').
....
```

Consulting a file

```
% To read in a file with Prolog code: the consult predicate.

:- consult('~/Documents/Development/swiProlog/classData').

% Or shorter:
:- consult(classData).

% Once this is done, we can query!
?- class(C)
      C = 'Object' ;
      C = 'Layout' ;
      C = 'LayoutOrigin' ;
      C = 'LayoutFrame' ;
      ...
```

Assembling facts into lists

```
classList(L) :-
    findall(  C,
              class(C),
              L).

superclassOfList(L) :-
    findall(  superclassOf(Super,Sub),
              superclassOf(Super,Sub),
              L).


?- classList(L)
      L = ['Object', 'Layout', 'LayoutOrigin', 'LayoutFrame',
       'AlignmentOrigin', 'LayoutSizedOrigin', 'NameScope', 'LocalScope',
       'HintedScope'|...]

?- superclassOfList(L)
      L = [superclassOf('Object', 'Layout'), superclassOf('Layout',
       'LayoutOrigin'), superclassOf('LayoutOrigin', 'LayoutFrame'),
       superclassOf(..., ...)|...]

?- classList(CL), superclassOfList(SCL),
      length(CL, NCL), length(SCL, NSCL), NCL is NSCL + 1
```

classChain

```
% Class is a class. The classchain is the list consisting of class,
% and the subsequent superclasses all the way until a root class.

classChain(Class, [Class]) :-
   rootclass(Class).

classChain(Class, [Class | Rest]) :-
   superclassOf(Super, Class),
   classChain(Super, Rest).


?- classChain('Object', ['Object']).
   Yes

?- classChain('Array', Chain).
   Chain = ['Array', 'ArrayedCollection',
                            'SequenceableCollection', 'Collection', 'Object']

?- classChain(C, ['Collection', 'Object']).
     C = 'Collection'

?- classChain('ExceptionSet', ['Collection', 'Object'])
   No
```

inverseClassChain

```
% Class is a class. The inverse classchain is the list consisting of the
root class and the inheritance chain down to Class.

inverseClassChain(Class, InverseChain) :-
   inverseClassChain(Class, [], InverseChain).


inverseClassChain(Class, Result, [Class | Result]) :-
   rootclass(Class).

inverseClassChain(Class, CurrentChain, Result) :-
   superclassOf(Super, Class),
   inverseClassChain(Super, [Class | CurrentChain], Result).


?- inverseClassChain('Array', Chain)
     Chain = ['Object', 'Collection', 'SequenceableCollection',
                                       'ArrayedCollection', 'Array']
```

classInHierarchyOf

```
Sub is a class somewhere in the class hierarchy of Root (it is a direct or
indirect subclass of Root).

classInHierarchyOf(Sub, Root) :-
   superclassOf(Root, Sub).

classInHierarchyOf(Sub, Root) :-
   superclassOf(Super, Sub),
   classInHierarchyOf(Super, Root).


?- classInHierarchyOf('OrderedCollection', 'Object')
     Yes

?- classInHierarchyOf('OrderedCollection', C).
      C = 'SequenceableCollection' ;
      C = 'Collection' ;
      C = 'Object';
      No
```

lastInCommon

```
% Given two lists L1 and L2, E is the last element where the two
% lists are the same.

lastInCommon(L1, L2, E) :-
   lastInCommon(L1, L2, _, E),
   nonvar(E).

lastInCommon([], [], R, R).

lastInCommon([S | Rest1], [S | Rest2], _Result, T) :-
   lastInCommon(Rest1, Rest2, S, T).

lastInCommon([S | _Rest1], [T | _Rest2], Result, Result) :-
   S \= T.


?- lastInCommon([1, 2, 3, 4], [1, 2, 3, 5], 3).
      Yes

?- lastInCommon([1, 2, 3, 4], [1, 2, 3, 4], E).
      E = 4

?- lastInCommon([1, 2], [4, 2], E).
      No
```

sibling

```
%Two classes C1 and C2 are sibling if they share a common ancestor Common

sibling(C1, C2) :-
   sibling(C1, C2, _).


sibling(C1, C2, Common) :-
   inverseClassChain(C1, L1),
   inverseClassChain(C2, L2),
   lastInCommon(L1, L2, Common).


?-sibling('SortedCollection', 'Graph').
      Yes

?- sibling('SortedCollection', 'Graph', 'OrderedCollection').
      Yes

?- sibling('SortedCollection', 'LinkedList', C).
      C = 'SequenceableCollection')).
```

Cherry on the cake..

```
check(Goal) :- Goal, !.
check(Goal) :-
   write('TEST FAILED: '),
   write(Goal), nl.

classChainTests :-
   check(classChain('Object', ['Object'])),
   check(classChain('Collection', ['Collection', 'Object'])),
   check((classChain(C, ['Collection', 'Object']), C = 'Collection')),
   check(not(classChain('ExceptionSet', ['Collection', 'Object']))).

classInHierarchyOfTests :-
   check(classInHierarchyOf('Collection', 'Object')),
   check(classInHierarchyOf('OrderedCollection', 'Collection')),
   check(classInHierarchyOf('OrderedCollection', 'Object')).

test :-
   write('TESTING ...\n'),
   classChainTests,
   classInHierarchyOfTests,
      %...
   write('DONE'), nl,
   true.
```

Wrap-up

- You now know:
  - what this course is about
  - who I am and how to contact me

References

- http://www.ulb.ac.be/di/rwuyts/INFO020_2003/
- The Ciao Prolog System Reference Manual, Technical Report CLIP 3/97.1, www.clip.dia.fi.upm.es