

# EDAN20

## Language Technology

<http://cs.lth.se/edan20/>  
A Short Introduction to Prolog

Pierre Nugues

Lund University  
[Pierre.Nugues@cs.lth.se](mailto:Pierre.Nugues@cs.lth.se)  
[http://cs.lth.se/pierre\\_nugues/](http://cs.lth.se/pierre_nugues/)

September 2, 2015



# Facts

```
character(priam, iliad).      character(ulysses, odyssey).  
character(hecuba, iliad).    character(penelope, odyssey).  
character(achilles, iliad).  character(telemachus, odyssey).
```

% Male characters

```
male(priam).  
male(achilles).  
male(agamemnon).  
male(patroclus).  
male(hector).  
male(rhesus).  
male(ulysses).  
male(menelaus).  
male(telemachus).  
male(laertes).  
male(nestor).
```

% Female characters

```
female(hecuba).  
female(andromache).  
female(helen).  
female(penelope).
```



# More Facts

```
% Fathers                % Mothers
father(priam, hector).    mother(hecuba, hector).
father(laertes,ulysses).  mother(penelope,telemachus).
father(atreus,menelaus).  mother(helen, hermione).
father(menelaus, hermione).
father(ulysses, telemachus).
```

```
king(ulysses, ithaca, achaeon).
king(menelaus, sparta, achaeon).
king(agamemnon, argos, achaeon).
king(priam, troy, trojan).
```

A Prolog fact corresponds to:

```
relation(object1, object2, ..., objectn).
```



# Terms

Terms	Graphical representations
<code>male(ulysses)</code>	<pre>graph TD     male[male] --- ulysses[ulysses]</pre>
<code>father(ulysses, telemachus)</code>	<pre>graph TD     father[father] --- ulysses[ulysses]     father --- telemachus[telemachus]</pre>
<code>character(ulysses, odyssey, king(ithaca, achaeon))</code>	<pre>graph TD     character[character] --- ulysses[ulysses]     character --- odyssey[odyssey]     character --- king[king]     king --- ithaca[ithaca]     king --- achaeon[achaeon]</pre>



# Queries

*Is Ulysses a male?*

```
?- male(ulysses).
```

Yes

*Is Penelope a male?*

```
?- male(penelope).
```

No

*Is Menelaus a male and is he the king of Sparta and an Achaean?*

```
?- male(menelaus), king(menelaus, sparta, achaeen).
```

Yes



# Variables

## Characters of the Odyssey

```
?- character(X, odyssey).
```

X = ulysses

*What is the city and the party of king Menelaus? etc.*

```
?- king(menelaus, X, Y).
```

X = sparta, Y = achaeen

```
?- character(menelaus, X, king(Y, Z)).
```

X = iliad, Y = sparta, Z = achaeen

```
?- character(menelaus, X, Y).
```

X = iliad, Y = king(sparta, achaeen)



# Multiple Solutions

All the males:

```
?- male(X).
```

```
X = priam ;
```

```
X = achilles ;
```

```
...
```

```
No
```



# Shared Variables

*Is the king of Ithaca also a father?*

```
?- king(X, ithaca, Y), father(X, Z).  
X = ulysses, Y = achaeon, Z = telemachus
```

The anonymous variable `_`:

```
?- king(X, ithaca, _), father(X, _).  
X = ulysses
```





# Rules

Derive information from facts:

```
son(X, Y) :- father(Y, X), male(X).
```

```
son(X, Y) :- mother(Y, X), male(X).
```

```
HEAD :- G1, G2, G3, ... Gn.
```

```
?- son(telemachus, Y).
```

```
Y = ulysses;
```

```
Y = penelope;
```

```
No
```

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

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



# Recursive Rules

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

```
grand_grandparent(X, Y) :-  
    parent(X, Z), parent(Z, W), parent(W, Y).
```

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

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

```
?- ancestor(X, hermione).
```

```
X= menelaus;
```

```
X = helen;
```

```
X = atreus;
```

```
No
```

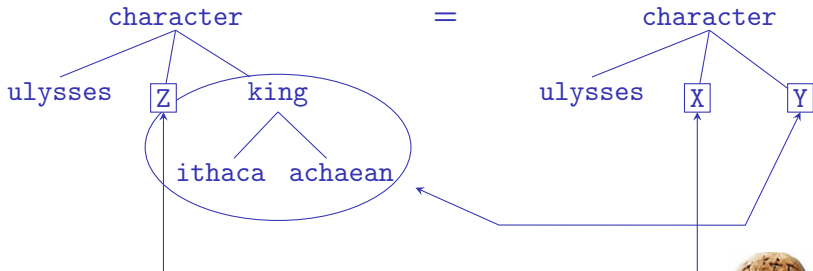


# Unification

Prolog uses unification in queries to match a goal and in term equation  $T1 = T2$ .

$T1 = \text{character}(\text{ulysses}, Z, \text{king}(\text{ithaca}, \text{achaeen}))$

$T2 = \text{character}(\text{ulysses}, X, Y)$



# Lists

Lists are useful data structures

Examples of lists:

- `[a]` is a list made of an atom
- `[a, b]` is a list made of two atoms
- `[a, X, father(X, telemachus)]` is a list made of an atom, a variable, and a compound term
- `[[a, b], [[[father(X, telemachus)]]]]` is a list made of two sublists
- `[]` is the atom representing the empty list.



# Head and Tail of a List

It is often necessary to get the head and tail of a list:

?- [a, b] = [H | T].

H = a, T = [b]

?- [a] = [H | T].

H = a, T = []

?- [a, [b]] = [H | T].

H = a, T = [[b]]

The empty list can't be split:

?- [] = [H | T].

No



# The member/2 List Predicate

`member/2` checks whether an element is a member of a list:

```
?- member(a, [b, c, a]).
```

Yes

```
?- member(a, [c, d]).
```

No

`member/2` can be queried with variables to generate elements member of a list as in:

```
?- member(X, [a, b, c]).
```

`X = a ;`

`X = b ;`

`X = c ;`

No.



# The member/2 Definition

member/2 is defined as

```
member(X, [X | Y]). % Termination case  
member(X, [Y | YS]) :- % Recursive case  
    member(X, YS).
```

We could also use anonymous variables to improve legibility and rewrite member/2 as

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



# The append/3 List Predicate

append/3 appends two lists and unifies the result to a third argument:

```
?- append([a, b, c], [d, e, f], [a, b, c, d, e, f]).
```

Yes

```
?- append([a, b], [c, d], [e, f]).
```

No

```
?- append([a, b], [c, d], L).
```

L = [a, b, c, d]

```
?- append(L, [c, d], [a, b, c, d]).
```

L = [a, b]

```
?- append(L1, L2, [a, b, c]).
```

L1 = [], L2 = [a, b, c] ;

L1 = [a], L2 = [b, c] ; etc.

with all the combinations.





# The append/3 Definition

append/3 is defined as

```
append([], L, L).  
append([X | XS], YS, [X | ZS]) :-  
    append(XS, YS, ZS).
```



# Searching the Minotaur

```
link(r1, r2). link(r1, r3).  
link(r1, r4). link(r1, r5).  
link(r2, r6). link(r2, r7).  
link(r3, r6). link(r3, r7).  
link(r4, r7). link(r4, r8).  
link(r6, r9).
```



Since links can be traversed both ways, the `s/2` predicate is:

```
s(X, Y) :- link(X, Y).  
s(X, Y) :- link(Y, X).
```

And `minotaur(r8).`



# Depth-First Search

```
%% depth_first_search(+Node, -Path)
depth_first_search(Node, Path) :-
    depth_first_search(Node, [], Path).

%% depth_first_search(+Node, +CurrentPath, -FinalPath)
depth_first_search(Node, Path, [Node | Path]) :-
    goal(Node).
depth_first_search(Node, Path, FinalPath) :-
    s(Node, Node1),
    \+ member(Node1, Path),
    depth_first_search(Node1, [Node | Path], FinalPath).
```

The goal is expressed as: `goal(X) :- minotaur(X).`

