Declarative problem solving methods

Lecture 1 & 2

Overview
Welcome
The master program
Presentation of the group
Course plan
Course evaluation and changes
Course structure
Introduction to Prolog
- facts
- rules
- recursion
- the connectives "and" and "or"
- Prolog's execution strategy
- different types of objects
- matching/unifying

References
Bratko 2012 Chapter 1.1-1.4, 2.1-2.4
Bratko 2001 Chapter 1, 2.1-2.4

Articles
Andrew, The commercial use of Prolog
Carlsson & Mildner, SICStus Prolog – The first 25 years. See section 2 and 6.
See articles in the folder Reference articles in the Student portal.

HOW something should be done – imperative languages

WHAT should be defined – declarative languages

We will use Prolog as programming language to investigate *how problems can be solved* in a declarative way.

Prolog
Prolog, PROgramming in LOGic, is an example of a logic programming system.

Some of Prolog's fields of applications:
- natural language processing
- databases
- expert systems & knowledge-based systems
- planning
- construction of compilators
- CAD (Computer Aided Design)

Some practical examples where Prolog is used:

• *The IBM Watson system* – the system's natural language processing with Prolog (Lally & Fodor, 2011)

• *Brutus* – a story telling program useful for preparation of commercial documents. (Andrew, 2005)

• *DealBuilder* – an intelligent document development and assembly tool for managing and generating end user license agreements in multiple languages. (Andrew, 2005)

• *AREZZO* – a clinical decision support software that enables the design, creation, and execution of clinical guidelines and patient care protocols. (Andrew, 2005)

• *In Flow* – a system that allows the modelling and analysis of social networks. (Andrew, 2005)

• *MAS* – intelligent agents that monitor, diagnose and solve problems utilised for e.g. railway circulations. (Briola et al, 2009)

• *CPW* – a Prolog-based intelligence server that generate drafts schedules automatically. The focus is to form construction plans by implementing construction principles and standard practice with the industry. (Trinidad, 2004)

• *FCM Assisting System* – a system that assists scientists to construct and simulate their own fuzzy cognitive maps (FCM). FCM is a method for making predictions and taking decisions. (Tsadiras, 2008)

• *HR006 Hyundai robot defect tracking system* – for fast and easy tracking of defects and generating proposals to remove these defects. (Daeinabi et al, 2007)

• *A knowledge-based scheduling system* – for reducing the time to create and maintain production schedules.
The framework has been used for
Protos – schedules production in the chemical industry, PSY schedules production in the metal industry, and Medicus schedules patients for heart surgery. (Sauer et al, 1997)

Prolog is a general programming language with a mechanism that can draw logical conclusions from the statements in a Prolog program.

A Prolog program consists of *Horn clauses*. The Horn clauses are a subset of the predicate calculus, a form of logic. A Horn clause can have the following formats:

in logic $\forall x1 ... \forall x_n (A \leftarrow B_1 \wedge \ldots \wedge B_i)$

in Prolog $A \leftarrow B_1, B_2, \ldots, B_i$	written as	A:- $B_1, B_2, \ldots, B_i$	(i)
$A \leftarrow$	written as	A.	(ii)
$\leftarrow A$	written as	?- A.	(iii)

In (i) the relation A is true if all the relations B_1 up to B_i are true. This means that the conclusion A is true if the conditions B_1 and B_2 and ... and B_i are satisfied. (ii) means that the relation A is always true. (iii) is the form for a question to the system whether the relation A is true. The answer is yes if A can be fulfilled and if not the answer is no.

The Horn clauses will also be called *clauses*.

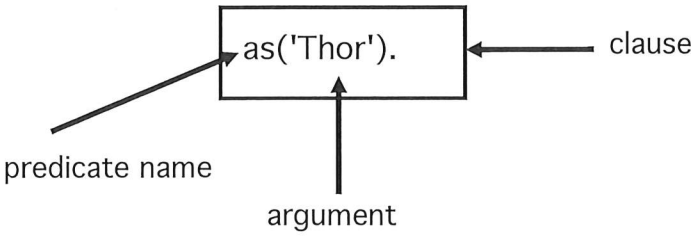
Prolog is a declarative programming language. This means that we declare what is valid for a world, i.e., what is true in our world. Most of the programming languages are imperative, which means that we define what should be performed when a condition is fulfilled. In Prolog we investigate what is true and false and we are not focused on a sequence of actions.

Introduction to Prolog

Let us start with an ancient legendary world, that of Norse mythology.

Facts

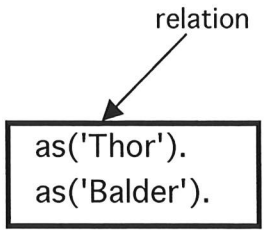
We would like to describe that Thor is an as, which is a Norse god.



Note: The clauses end with a full stop ".".

The number of arguments decides the *arity* of the predicate. The predicate or relation as has arity 1.

Balder is also an as.



Our Prolog program should be extended with the fact that Freya is a van.

```
as('Thor').                                    (1)
as('Balder').                                  (2)

van('Freya').                                  (3)
```

Questions to the program or goals

Is Freya a van?
Is Freya an as?
Is Balder an as and is Thor an as too?
Is Thor a van or an as?

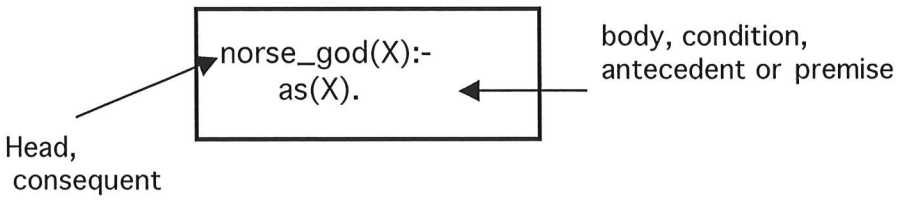
Questions with variables

Who is an as?
Who is a van?
Who are the Norse gods?

When a variable gets a value, e.g., when X gets the value Thor, you say that the variable gets *instantiated*.

Rules

We would like to express that a person who is an as also is a norse_god.



The clause is read as X is a norse_god if X is an as.

Moreover, all vans are norse_god.

```
norse_god(X):-
        as(X).                                 (4)
norse_god(X):-
        van(X).                                (5)
```

Questions
Is Freya a Norse god?
Who are Norse gods?
Is Odin a Norse god?

The Prolog system presupposes that the world is closed world, i.e., everything that is relevant is defined in the program. Even if we know that Odin is a norse_god the Prolog system cannot know this until we define that in our program.

Recursion

Let us look at the characteristics for an as to be able to extend our description, i.e., our program.

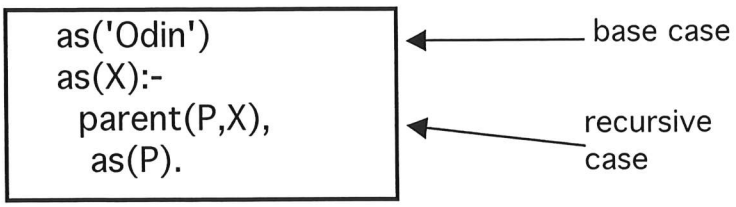
Who is an as? The forefather of every as is Odin; Odin is the ancestor. Everybody that has a descent to Odin is an as. This means that Odin's children, grandchildren, grand-grand children and so on, are aesirs (as in plural).

```
as('Odin').
as(X):-
        parent('Odin',X).
as(X):-
        parent('Odin',P),
        parent(P,X).
as(X):-
        parent('Odin',P1),
        parent(P1,P2),
        parent(P2,X).
and so on.
```

Since we don't know how many generations we need to define the relation as for we need to define the relation in a more general form – and also in a more effective way. What we describe above is that someone is an as if the parent is an as. This can be described in the following way:

```
as('Odin').                                    (6)
as(X):-
        parent(P,X),
        as(P).                                 (7)
```

(6) describes that Odin is an as. (7) expresses that X is an as if P is a parent to X and if P is also an as. (7) is a *recursive definition* since we use the definition for as to define what is true for as. In recursive definitions there has to be at least one *base case,* in this case (6) is our base case, to prohibit an infinite evaluation of the predicate.



To be able to use the predicate as as it is defined in (6) and (7) we need to define the relation parent. A person is a parent to a child if the person is the mother or the father to the child.

```
parent(P,C):-
        mother(P,C).                                    (8)
parent(P,C):-
        father(P,C).                                    (9)
```

(8) can be read as P is a parent to C if P is a mother to C.

```
father('Odin','Thor').                                  (10)
father('Thor','Trud').                                  (11)
father('Thor','Magne').                                 (12)
```

Questions
Is Odin the father to Thor?
Who are the children to Thor?
Has Trud and Thor the same father?
Is Magne an as?
Who are the aesirs?

The connective "and" and "or"

If you don't want to use the definition of parent in the predicate as you can use the predicates mother and father in the statement (7).

```
as('Odin').                                             (6)
as(X):-
        mother(P,X),
        as(P).                                          (7a1)
as(X):-
        father(P,X),
        as(P).                                          (7a2)
```

(7a1) and (7a2) can be put together in a clause if you use the connective for "or",i.e., ";".

```
as(X):-
        (mother(P,X); father(P,X)),
        as(P).                                          (7b)
```

Since the connective "and" binds harder than "or" you have to use parenthesis to get a right definition. If you don't use the parenthesis (7b) will be:

```
as(X):-
        mother(P,X);
        father(P,X),
        as(P).
```

The interpretation of this statement is that if P is a mother to X then X is an as, alternatively X is an as if P is the father till X and P is an as. This interpretation can also be described by the following two clauses:

```
as(X):-
        mother(P,X).
as(X):-
        father(P,X),
        as(P).
```

How is the following statement interpreted by Prolog?

```
P:- Q, R ; S, T.
```

Prolog's way of reasoning

Let us look at the following problem: How are animals behaving?

```
mammal(X):-                                    (13)
        viviparous(X),
        carnivore(X).
viviparous(lion).                              (14)
viviparous(tiger).                             (15)
carnivore(Y):-                                 (16)
        has_sharp_teeth(Y),
        has_claws(Y).
has_sharp_teeth(lion).                         (17)
has_sharp_teeth(tiger).                        (18)
has_claws(tiger).                              (19)
has_claws(lion).                               (20)
```

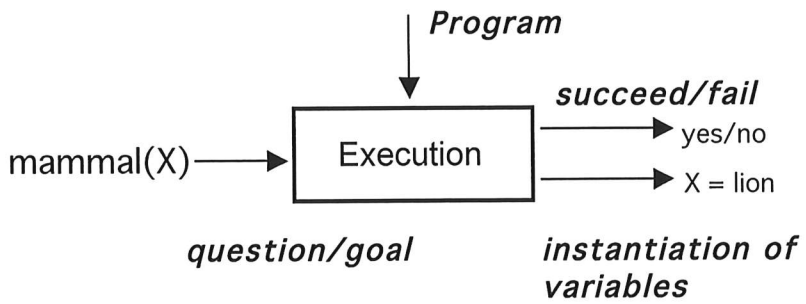
How is the following question answered?

```
?- mammal(tiger).
```

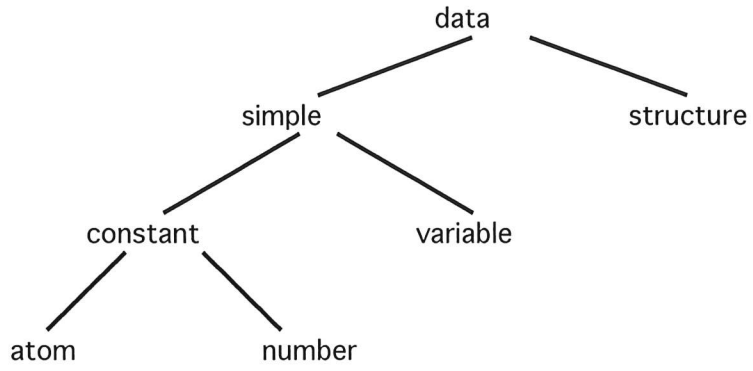
Prolog is searching for the answers

- upside and down (from question to answer)
- from left to right (in questions comprising several conditions)
- the depth first (investigates a branch first)

<u>The proof procedure from input to output</u>



Different types of objects



Constants

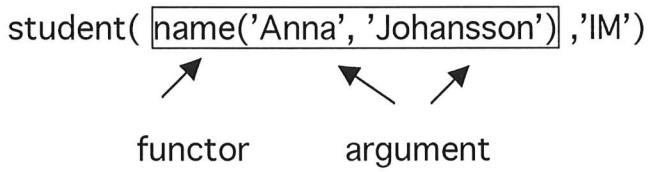
- atoms
  - starts with a lower case letter, e.g., anna, tiger, viviparous
  - strings of special characters, e.g., <-->
  - strings within ' ', e.g., 'Thor', 'Has sharp teeth'
- numbers, include integer and real numbers
  1 121 3.13 -573

Variables

Starts with an upper-case letter or _ ,e.g., X, Father, Person, _t1, _4

Structures

Have the form functor(arg1, arg2, ..., argN)
e.g., date(2016,1,21), personal_number(860807,-,4567)



Matching/unification

If a question could *match* a fact or a rule's head they have to be able to be *unified*. The rules for the *unification* are:

- Two predicates can be unified if
  - they have the same predicate name
  - they have the same arity (equal number of arguments)
  - the arguments can be unified
- Two terms can be unified if either
  - one is an variable
  - both are the same constants
  - both are structures and
    - they have the same functor
    - they have the same arity
    - the arguments can be unified

S	T	Unification
abc	abc	
X	a	
X	Y	
date(2013,X,31)	date(2013,12,Y)	
date(Year,12,Day)	date(2018,D,D)	
date(Y,M,D)	Date	
d(Year)	d(year(1,9,9,9))	
a	b	
d(1999)	d(year(1,9,9,9))	
d(X,Y)	d(1789)	
X	date(X,Y,Z)	

Example using a functor

Let us define natural numbers as recursive definitions.
We uses a structure with the functor s (denoting successor)
s(0) denotes the number 1
s(s(0)) denotes the number 2 etc.

Write a program that computes the sum of two natural numbers using the given functor.