Logic Programming in Prolog

Christian Rinderknecht

19 October 2008

Overview of artificial intelligence

Experts in AI would say that AI is a domain in itself, which is neither reducible to computer science nor biology, for instance.

Some experts would probably say that AI pretends to *solve complex problems the way humans do*.

Some other AI experts would disagree and say that the goal of AI is to solve so hard problems that, somehow, *programs may seem intelligent*.

Overview/A computer scientist's point of view

My presentation of AI is not based on history or philosophy or cognitive sciences or biology. I will only try to give some **intuitions** and some **techniques**, as algorithms and heuristics, that are useful in practice.

So what is AI, then?

From our point of view, it is a collection of techniques for solving or getting partial or suboptimal solutions to complex or intractable problems.

Plan

I. Overview

- A computer scientist's point of view
- **Applications**
- Symbolic approaches
- Connectionist approaches

Overview/Applications

One reason why some problems are hard to solve is that they may come from the interaction between the physical world and the computer, considered as a discrete model of the world.

For instance, how can the computer cope with shape or **face recognition** in signal processing (or **speech recognition**) and improve its success rate given successful and failed samples?

In this case, part of the difficulty lies in the passage from continuity (physical phenomenon) to discontinuity (logical abstraction).

Overview/Applications (cont)

Some problems are difficult because their computational complexity is too high (i.e. their solution requires too much computation to be realistic) and/or because the data is incomplete, hence the need for good approximations.

For instance, find the **shortest path** from one town to another, given the whole map, or play a **maze game**, without the whole map.

Overview/Applications (cont)

Some problems are difficult because they involve **automatic logical reasoning**.

For instance, is it possible to assist a physician in establishing a diagnosis or help to pinpoint a car failure?

Or, in security analysis: how can we prove this safety theorem about this train without pilot?

Overview/Applications (cont)

Another approach is more ideological, which consists in modeling some biological process, without any specific problem in mind, and then find the fruitful problems.

For instance, some techniques come from a model inspired by the brain, called **abstract neural networks**, or from genetics, called **genetic algorithms**.

These techniques are nevertheless applied, with various extents of success, to unexpected different kinds of problems.

Plan

I. Overview

- A computer scientist's point of view
- Applications
- **Symbolic approaches**
- Connectionist approaches

Overview/Symbolic approaches

Symbolic approaches are based on formal logic reasoning.
The rationale is to model objects of the problem, as well as their
relationships (in time or space), by **axioms** and **logical rules**.

- An axiom is a fact, something which is true by definition.
- A logical rule is an implication, like $A \Rightarrow B$, allowing to go from axioms to **theorems**.
- A theorem is a formula that can be deduced by logical rules from the axioms.

Overview/Symbolic approaches (cont)

A software assisting the task of proving theorems is called a **proof assistant** or a **knowledge-based system** (or **expert system**), depending on the abstraction level.

Proof assistants are used by computer theorists in the framework of very expressive logics, in order to prove for instance security property for embedded systems (i.e. autonomous vehicles like rockets, space robots, subway or plane without pilot, car ignition systems etc.).

Expert systems are more similar to a database of domain-specific informations and logic rules (simpler than within proof assistants) which allow queries to be formulated. Such systems are often used as an interactive tool to help an expert diagnose a failure. The **programming language Prolog** is most famous for such applications.

Plan

I. Overview

- A computer scientist's point of view
- Applications
- Symbolic approaches
- **Connectionist approaches**

Overview/Connectionist approaches

Connectionist approaches are based on the modeling of the brain as an **abstract neural network** whose properties are inspired by the electrical behaviour of biological neurons.

This approach tries to capture the transition between the continuity of physical phenomenon (electric waves) and discrete logical units (bits) by imitating brain-like structures.

An interesting question (not answered here) is how information pass from one model to another.

Overview/Connectionist approaches (cont)

Following the brain metaphor, the abstract neural networks are usually programmed through **learning** (or **supervised training**), i.e. by submitting inputs that must be accepted and inputs that must be rejected and, each time, adjusting the network parameters (called **weights**) to fit *all* the given examples (accept some and reject others).

Next, the network is expected to extrapolate accordingly to its learning phase to unknown examples. One application is **pattern recognition** (like bit-map character classification in noisy environments).

Defining relations with facts

Prolog is a programming language for symbolic (non-numeric) computation. Its name means **programming in logic**.

Let us approach it as if it were a **deductive database**. First, a database is populated by **relations**. A relation is what links two or more objects. This way we learn something about their relative meaning. For example, in "Tom is the parent of Bob," is a statement, the relation is parenthood and relates Tom and Bob.
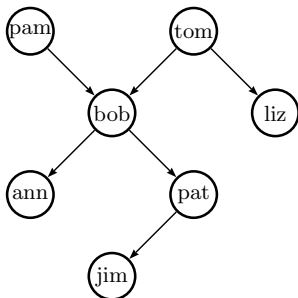
In Prolog, the simplest way of defining a relation is by enumerating **facts** (statements). For example

```
parent(tom,bob).
```

is a fact.

Defining relations with facts (cont)

A family tree encoded in Prolog


```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
```

Queries

This program is composed of six **clauses**, each declaring a fact about the parent relation. As we shall see later, a clause may not be a fact.

We say that, for example, parent(tom,bob) is an **instance** of the parent relation.

In general, a relation is defined as the set of all its instances.

After loading this relation, the Prolog interpreter allows the user to ask some questions about it. For example, the query "Is Bob a parent of Pat?" is written

?- parent(bob,pat).

Having found that this a fact, the interpreter answers

Yes

Ground queries

Another query can be

```
?- parent(liz,pat).
```

The interpreter answers

```
No
```

The same answers is given to query

```
?- parent(tom,ben).
```

because the interpreter never heard of ben.

A **query** starts with '?-', is followed by a **goal** and ended by a period.

Existential queries

More interesting is asking "Who are the parents of Liz?" In this case
what we do not know is a set of persons. The usual way of speaking
about an unknown person is to name it with an arbitrary **variable**, e.g.
X. The corresponding Prolog query is then

```
?- parent(X,liz).
```

In this case, we expect *all* the parents of Liz to be found — or none if
no instance were defined. The interpreter answers

```
X = tom
Yes
```

A **ground query** is a query with no variables.

Existential queries on one variable

If asked who are Bob's children:

```
?- parent(bob,X).
```

we expect several answers. The interpreter gives one and wait for the user to either type in a semi-colon for more answers or the return key to stop:

```
X = ann ;
X = pat
Yes
```

If we had exhausted all the solutions, the interpreter would print

```
No
```

Existential queries on two variables

We can also query on two variables, like asking for all X and Y such that
X is a parent of Y. In other words, find all the instances of the parent
relation. This is formally written as

```
?- parent(X,Y).
```

The answer starts as

```
X = pam
Y = bob ;
X = tom
Y = bob ;
X = tom
Y = liz ;
```

Conjunctive queries and shared variables

We could ask: "Who is the grand-parent of Jim?" But, since, we did not define the grandparent relation, we have to break this question in two parts:

1. Who is a parent of Jim? Assume that (s)he is named Y.
2. Who is a parent of Y? Assume that (s)he is named X.

The answer to the original question is X. Formally, this is written

```
?- parent(Y,jim), parent(X,Y).
```

The answer is (final Yes omitted):

```
X = bob
Y = pat
```

Conjunctive queries and shared variables (cont)

Note that the Prolog interpreter returns the values for all the variables involved in the query (here X and Y), even if we are only interested in some of them (here X).

The previous query is made by tying two queries sharing a common variable, Y. It is a **conjunctive query**.

If we change the order of composition of the two queries, the logical meaning remains the same:

```
?- parent(X,Y), parent(Y,jim).
```

produces the same result. Similarly, we ask: "Who are Tom's grand-children?" by

```
?- parent(tom,X), parent(X,Y).
```

Conjunctive queries and shared variables (cont)

Let us ask: "Do Ann and Pat have a common parent?" Again, the question can be broken down in two sub-questions:

1. Who is a parent X of Ann?
2. Is (the same) X a parent of Pat?

In Prolog, this conjunctive query is written

```
?- parent(X,ann), parent(X,pat).
```

The Prolog queries are usually called **goals**.

Defining relations by rules

We can extend our example in many interesting ways. Let us first add
the information about the sex of the people. This can be easily done by
simply adding the following facts to our program:

```
female(pam).
female(liz).
female(pat).
female(ann).
male(jim).
male(tom).
male(bob).
```

The relations introduced are male and female. These relations only
have one argument, and are called **predicates**.

Defining relations by rules (cont)

Another way consists in defining a **binary relation** for the sex, i.e. a relation between two objects, like the parent relation. For example:

```
sex(pam,feminine).
sex(tom,masculine).
sex(bob,masculine).
...
```

Now let us introduce the **inverse relation** of parent, we call offspring. We can add new facts again. For example

```
offspring(liz,tom).
```

Defining relations by rules (cont)

However, the Prolog language offers a more compact and elegant way to define the relation `offspring` in terms of the already defined relation `parent`.

Indeed, what we want to say is:

For all X and Y, Y is an offspring of X if X is a parent of Y.

In Prolog this is expressed by means of a clause

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

Such clauses are called **rules**.

Defining relations by rules (cont)

There is an important difference between facts and rules. A fact like

```
parent(tom,liz).
```

is something that is unconditionally true. On the other hand, rules specify things that are true **if** some condition is satisfied.
Rules have a *condition* part (right-part of the rule) and a *conclusion* (left-hand side). The conclusion is also called the **head** and the condition the **body**. For example

$$\underbrace{\text{offspring(Y,X)}}_{\text{head}} \text{ :- } \underbrace{\text{parent(X,Y)}}_{\text{body}}.$$

Defining relations by rules (cont)

Let us ask now whether Liz is an offspring of Tom:

```
?- offspring(liz,tom).
```

Since there are no facts about offsprings in the program and there is a rule whose head is an instance of offspring, the Prolog interpreter tries to use the latter.
The head of this rule is offspring(Y,X), implicitly for all X and Y. So X can be replaced by tom and Y by liz. This process is called **instantiation**. We get a special case of the rule:

```
offspring(liz,tom) :- parent(tom,liz).
```

Defining relations by rules (cont)

Now the Prolog interpreter tries to prove the body `parent(tom,liz)`. This is actually a fact, so it answers `Yes`.

It does not give any variable information, like `X = tom`, because the original goal contained no variable.

Let us add more twists to our program by considering a `mother` relation. It could be informally defined by

For all X and Y, X is the mother of Y if X is a parent of Y and X is a female.

This is written in Prolog

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

Defining relations by rules (cont)

The grand-parent relation can be defined as

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

The interesting point here is that there is a variable, Z, that is shared between the two sub-goals in the body and that does not appear in the head.

The interpreter, when asked a question about two persons, on one being the grand-parent of the other, tries to find another person (Z) who is the child of the first and the parent of the second.

Defining relations by rules (cont)

The sister-hood relation can be defined as

```
sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X).
```

This definition seems right but something strange happens if we query

```
?- sister(X,pat).
```

which corresponds to the question: "Who are the sisters of Pat?" The answers are:

```
X = ann;
X = pat
```

Pat is her own sister!

Defining relations by rules (cont)

To fix this, we need a binary relation `different` and use it in the rule
defining the relation `sister`:

```
sister(X,Y) :- parent(Z,X), parent(Z,Y),
               female(X), different(X,Y).
```

The relation `different` can be defined by listing all the pairs of
different persons:

```
different(bob,pat).
different(bob,ann).
...
```

This approach does not scale. We will see later how to define a general
`different`, based on equality.

Recursive rules

Let us add another relation to our family program, the `ancestor` relation.

If we consider again the family tree page 16, we model the `ancestor` binary relation by saying that X is an ancestor of Y if there is a chain of parenthood relationships from X to Y. For example, Pam is an ancestor of Jim and Tom is an ancestor of Liz.

The latter example actually illustrates a special case of the `ancestor` relation, when X is a parent of Y. This case is very easy to specify in Prolog:

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

But what if the chain of parents is longer? We could try to add another rule

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

Recursive rules (cont)

But what if the chain of ancestors is strictly longer than two parent relationships? We could try to add

```
ancestor(X,Y) :- parent(X,Z1), parent(Z1,Z2), parent(Z2,Y).
```

but we can never attain the general case, when the chain is arbitrarily long... The solution consists in designing a **recursive rule**. For example:

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

which translates "X is an ancestor of Y if X is the parent of some Z which is in turn the ancestor of Y."

Recursive rules (cont)

We also must keep the special case when ancestorship is parenthood, so all together we have

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

This pattern is typical of recursive rules: there is at least one rule which is not recursive, to cover base cases, and some others which are recursive. Let us query for example

```
?- ancestor(pam,X).
X = bob;
X = ann;
X = pat;
X = jim
```

How Prolog answers queries

To answer a query, the Prolog interpreter tries to satisfy all the goals.

Satisfying a goal means proving that a goal logically follows from the facts and rules in the program.

If the query contains variables, the interpreter must find particular objects in place of the the variables that entail the goal.

If it cannot prove the goal, the interpreter answers No.

How Prolog answers queries (cont)

For example, consider the famous syllogism about the philosopher Socrates. Given

All men are fallible [a rule], Socrates is a man [a fact].

a logical consequence is that

Socrates is fallible.

In Prolog, this is written

```
fallible(X) :- man(X).
man(socrates).
```

Then we have

```
?- fallible(socrates).
Yes
```

How Prolog answers queries (cont)

This query was answered by the interpreter by first looking up some fact that would match the goal `fallible(socrates)`.

Since there is none, the interpreter looked for rules such that the goal is an instance of the head, i.e. such that the goal can be formed by replacing variables in the head by some object.

If we set `X = socrates`, then the rule

`fallible(X) :- man(X).`

is instantiated into

`fallible(socrates) :- man(socrates).`

whose head matches exactly the query.

How Prolog answers queries (cont)

Now the interpreter tries to prove the body, i.e. the sub-goal

```
?- man(socrates).
```

just as it tried to prove the initial query.

It searches first for a fact which would be the sub-goal `man(socrates)`, and, indeed, there is such a fact in the program.

Therefore the sub-goal is true, so is the goal and the query is positively answered.

How Prolog answers queries (cont)

Consider a query about the family tree page 16 like

```
?- ancestor(tom,pat).
```

Let us recall the definition

```
ancestor(X,Y) :- parent(X,Y).                % Rule [anc1]
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y). % Rule [anc2]
```

where what follows a % until the end of the line is a commentary.

First, the interpreter tries to instantiate the first rule, [anc1], in such a way that the instance's head matches the goal. This can be achieved by letting X=tom and Y=pat. The instantiated rule is

```
ancestor(tom,pat) :- parent(tom,pat). % Instance of [anc1]
```

How Prolog answers queries (cont)

Next, the interpreter tries to prove the body of the rule's instance, i.e.

```
?- parent(tom,pat).
```

It searches among the facts defining the parent relation but finds not match. Since there is no rule for parent, the interpreter fails and parent(tom,pat) is false.

Hence, ancestor(tom,pat) cannot be proven using rule [anc1].

Before giving up, the interpreter tries again with the last remaining rule, [anc2]. The variable bindings are the same as before, and the rule instance is

```
ancestor(tom,pat) :- parent(tom,Z),
                     ancestor(Z,pat). % Instance of [anc2]
```

How Prolog answers queries (cont)

First, the interpreter tries to prove the sub-goal

```
?- parent(tom,Z).
```

It searches again the database defining `parent` and finds two matches:
Z=bob and Z=liz.

For each binding of Z, the interpreter substitutes Z by the associated
object into the second sub-goal and tries to prove it. First, it gets to
prove

```
?- ancestor(bob,pat).
```

How Prolog answers queries (cont)

The process for proving this goal is the same as before. Rule [anc1] is considered first. The variable binding X=bob and Y=pat leads to the following instance of [anc1]:

```
ancestor(bob,pat) :- parent(bob,pat). % Instance of [anc1]
```

whose head matches the current sub-goal. Now, the interpreter tries to prove

```
?- parent(bob,pat).
```

It searches the facts about the parent relation and finds a match. Therefore the sub-goal ancestor(bob,pat) is true, and, since

```
ancestor(tom,pat) :- parent(tom,bob), ancestor(bob,pat).
```

it proves the initial goal ancestor(tom,pat).

How Prolog answers queries (cont)

The execution is over, even if we left suspended the binding Z=liz, and in spite that Prolog interpreters always offer the possibility to find *all the solutions*.

The reason is that the initial goal contained no variable, so the interpreter will try to prove it only once, if there is at least one proof.

The technique that consists, when finding that a goal is false, to go back in history and try to prove an alternative goal is called **backtracking**.

Backtracking is also used in case of success but the user wants more solutions, if any.

How Prolog answers queries (cont)

Let us imagine now what would have happened if the interpreter had chosen to try the binding Z=liz, before Z=bob.

So it tries to prove

```
?- ancestor(liz,pat).
```

It uses the same strategy, and instantiate rule [anc1] with the bindings X=liz and Y=pat:

```
ancestor(liz,pat) :- parent(liz,pat).    % Instance of [anc1]
```

Since there is no fact parent(liz,pat), it fails and backtracks.

How Prolog answers queries (cont)

It tries now with rule [anc2], with the same variable bindings:

```
ancestor(liz,pat) :- parent(liz,Z), ancestor(Z,pat).
```

It searches all the facts of the shape parent(liz,Z) and finds none. Therefore it is useless to try to prove the second sub-goal parent(Z,pat), because the conjunction of false and any other boolean value is always false. In other words, for all x,

$$\text{false} \wedge x = x$$

Therefore, the interpreter backtracks further, because the binding Z=liz only leads to falsity, and then tries to prove the query with Z=bob, as we did in the first presentation.

How Prolog answers queries/Proof trees

There is a graphical representation of proofs that helps a lot to understand how the Prolog interpreter works. It is called a **proof tree**.

The idea consists in making a tree whose root is the goal to prove and the sub-trees correspond to the proofs of the sub-goals.

In other words, the inner nodes are made from rule instances and the leaves consist of facts.

How Prolog answers queries/Proof trees (cont)

For example, the successful proof of

```
?- ancestor(tom,pat).
```

can be graphically represented as the following proof tree.

$$\cfrac{\text{parent(tom,bob)} \qquad \cfrac{\text{parent(bob,pat)}}{\text{ancestor(bob,pat)}}\,\text{ANC}_1}{\text{ancestor(tom,pat)}}\,\text{ANC}_2$$

Note that all the leaves, parent(tom,bob) and parent(bob,pat), are facts; the name of the instantiated rule appears on the right of each inner node (horizontal line).

How Prolog answers queries/Proof trees (cont)

The Prolog interpreter starts from the root and tries to grow branches that all end in leaves which are facts. If not, it backtracks to try another rule instance, and if none matches the knot, it fails.

For instance, we saw that it tried first

$$\frac{\texttt{parent(tom,pat)}}{\texttt{ancestor(tom,pat)}} \text{ ANC}_1$$

but the leaf was not a fact, so it tried next

$$\frac{\texttt{parent(tom,bob)} \qquad \texttt{ancestor(bob,pat)}}{\texttt{ancestor(tom,pat)}} \text{ ANC}_2$$

Data objects/Atoms

Some objects are only identified by their name, called an **atom**. Thus, it is correct to say that, in such a case, an object *is* an atom. For example, we saw the atom bob.

An atom starts with a lower-case letter which can be followed by a string of characters out of lower-case letters, upper-case letters, digits and the underscore character ('_').

For example, the following are valid atoms:

```
anna            alpha_beta_proc
x25             call_Java
x_25            x_
x_25AB          x____y
```

Data objects/Numbers

Numbers in Prolog include integer numbers and floating-point numbers. The syntax of integer is as expected, for example

```
1    1234    0    -97
```

The lower and larger integers are limited by the actual Prolog system in use.

Floating-point numbers follow the usual syntax too, like

```
3.14    -0.06    100.5
```

The general syntax is not given here because Prolog is primarily intended for symbolic computations, not numerical computations.

Atoms and numbers define the group of **constants**.

Data objects/Variables

A **variable** is a name for an object, but, contrary to atoms, variables do not define any object. So a variable can denote several objects. They must start with an upper-case letter or an underscore and may be followed by any number of letters, digit and underscores, in any order. For example, the following are valid variables:

```
X    Obj_List    _23    Object2    Result    ObjList    _x23
```

If a variable appears only once in the body of a clause and not in the head, it is an **unknown variable** and can be replaced by an underscore. For example

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

can be replaced by the equivalent

```
has_a_child(X) :- parent(X,_).
```

Data objects/Variables (cont)

Several underscores can occur in a body. In this case, each of them should be considered an absolutely unique, despite unknown, variable.

For example, consider

```
someone_has_a_child :- parent(_,_).
```

Each underscore could be replaced by a unique variable, like

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

But certainly **not**

```
someone_has_a_child :- parent(X,X).
```

Data objects/Variables and lexical scopes

If an unknown variable appears in a query, then its value, found by the interpreter, is not displayed in the result.

For example, if one wants to know who are the people who have children, the Prolog query is

```
?- parent(X,_).
```

and only all the possible values of X that satisfy the query will be displayed.

Given an occurrence of a variable, the part of the program where this variable is usable is called the **scope**.

Data objects/Variables and lexical scopes (cont)

Prolog uses **lexical scoping**, which means

- the same variable always represents the same object inside a clause;
- the same variable in two different clauses represent different objects, in general.

For example, in clause

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

the tow occurrences of variable X represent the same object. But, if we also have

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

the occurrences of X denote, in general, objects different from the X in the previous clause.

Data objects/Functors and structures

Structured objects, or **structures**, are objects that are composed of several objects. These sub-objects are boxed together by means of a **functor**. For example

```
date(9,july,2006)
```

is a structure composed, in order, of the integer constant 1, the atom `july`, the number 2006 and tied together by the functor `date`. The sub-objects are called **arguments** and the number of arguments is the **arity**.

The syntax of functor is the same as the atoms. Functors are used to define facts (i.e. instances of a relation). For example,

```
female(pam).
```

# Data objects/Functors and structures (cont)

Just as it is possible to make a graphical representation of a successful run of the Prolog interpreter (see page 48), it is possible to draw a tree which represents a Prolog structure.

The idea here is to map a functor to a node whose label is the functor, and its arguments to sub-trees, recursively in the same order. The consequence is that the component objects that are not structures are mapped to the leaves. For example
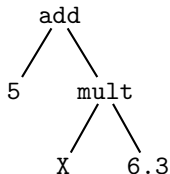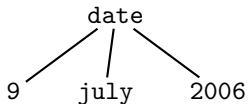
Data objects/Summary

It is also handy to represent all the kinds of Prolog data objects we reviewed until now in a tree. The idea consists in mapping a set of lexical elements to a node whose label is this set name, and the subsets are mapped to subtrees.
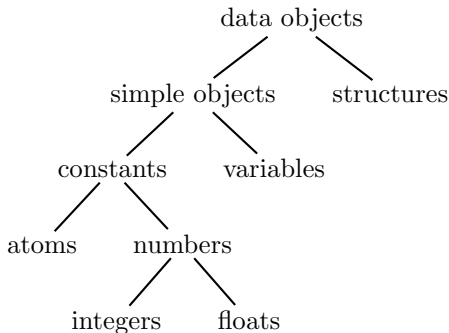
Matching

Once we have objects, it is useful to compare them. The mechanism to do so is called **matching** in Prolog, and is different from the equalities we find in other programming languages.

- Two numbers match if they represent the same mathematical number.
- Two atoms match if they are made of the same characters.
- A variable matches any object.
- Two structures match if
  - their functors match (as atoms),
  - all their corresponding arguments match.

Matching (cont)

Matching and equality agree on **ground terms**, i.e. objects containing no variables. In this case, they both either return true (Yes in Prolog) or false (No).

The difference between matching and equality concerns **non-ground terms**, i.e. objects containing variables.

Consider the following fragment of a C program:

```
if (x == 5) x++;
```

Here, the run-time has to compare *the value of* x with 5, i.e. it looks up the value to which x is bound and then matches it against 5 (remember that equality and matching agree on ground terms).

Matching (cont)

In Prolog, the closest clause, as far as comparison is concerned, is the query

```
?- X = 5.
```

But the scoping rules of Prolog say that this occurrence of variable X is visible only in this clause. Therefore it is unbound, i.e., it is not associated to any "previous" value.

Instead, because X is a variable, it must match 5, so the interpreter answers

```
X = 5
Yes
```

Here, the successful matching returns a **binding** for X, before answering Yes.

Matching (cont)

Imagine now a matching involving a structure, like

```
?- date(D,july,2006) = date(9,M,2006).
```

The interpreter first checks whether the functors match (are equal), which is true. Next, it matches the corresponding arguments against each other: D against 9, july against M and 2006 against 2006.

The first matching involves a variable and a number, so the interpreter chooses the binding D = 9.

The second matching involves an atom and a variable, so the interpreter chooses the binding M = july.

The last matching is trivial, 2006 = 2006, and does not require any binding.

Matching (cont)

So the answer is

```
D = 9
M = july
Yes
```

In other words, a successful matching returns bindings for all variables in the terms being matched, such as the corresponding **instantiation** leads to equal ground terms:

```
date(9,july,2006) = date(9,july,2006)
```

Instantiation means to replace all the variables by the object to which they are bound.

Matching/Failure

A failed matching consists of only No. For example

```
?- 1 = 2.
No
?- date(9,july,2006) = date(9,july,2007).
No
?- date(X,july,2006) = date(9,july,X).
No
```

In the last case, the interpreter finds two bindings for X which are different: X=9 and X=2006, which leads to failure.

Matching/Most general substitution

In general, a successful matching returns several bindings. A set of bindings is called a **substitution**. So, an instantiation consists in applying a substitution to a clause.

Sometimes there can be several possible substitutions that make the matching a success. For example the matching

```
?- X = Y.
```

can be satisfied by X = -3, Y = -3 or X = 7, Y = 7 and so on.

Matching/Most general substitution (cont)

In such cases, Prolog ensures that the most general substitution will be retained. In our example, `X = Y` is the most general, because all the instances can be obtained from it by replacing `X` and `Y` by the same arbitrary object.

In other words, when matching a variable `A` against another variable `B`, the matching succeeds with the most general substitution `A = B`, or

```
A = _G187
B = _G187
```

where `_G187` is a variable generated by the interpreter. In these slides we prefer to write

$$A = \alpha$$
$$B = \alpha$$

Matching/Most general substitution (cont)

Consider the special case

```
?- X = X.
X = α
Yes
```

The danger here is that Prolog uses the same syntactical convention, the = character, to denote both variable bindings and matchings: `A = 2` can be a matching or a binding, same for `A = B`.

But `2 = A` is a matching but **not** a variable binding. Same for

```
date(9,july,2006) = date(9,july,2006)
```

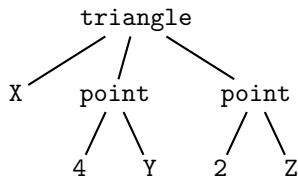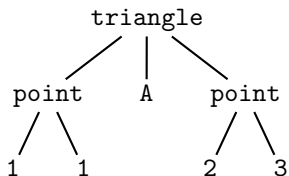## Matching/Tree representation

It is useful to use the tree representation of Prolog terms (page 58) to understand how a matching is performed.

Consider the two terms

```
triangle(point(1,1),A,point(2,3))
triangle(X,point(4,Y),point(2,Z))
```

These terms are represented by the trees

Matching/Tree representation (cont)

The interpreter traverse the two trees from the root to the leaves, following the same order when visiting the sub-trees. Let us assume that order between siblings is from left to right.

It matches first the two roots: if one of them is a variable, it stops and declares success, otherwise it matches the subtrees. Here, triangle = triangle, so, next, it matches the first subtree of the first tree with the first subtree of the second tree, i.e. `?- point(1,1) = X`. This is a success with the substitution `X = point(1,1)`.

Then, the second subtrees are matched, i.e.,

`?- A = point(4,Y). A = point(4,`α`) Y = `α

Matching/Tree representation (cont)

Next, the last subtrees are matched, i.e. the interpreter tries to answer the query

```
?- point(2,3) = point(2,Z).
```

The roots are the same: point = point. So, it then matches the subtrees, i.e. answers now the queries

```
?- 2 = 2.     ?- 3 = Z.
```

successfully with substitution Z = 3. Finally the answer is the substitution which is the union of all the others:

X = point(1,1)
A = point(4,α)
Y = α
Z = 3

Matching/Tree representation (cont)

The proof of the matching can be displayed by means of a proof tree:

$$
\cfrac{
\text{point(1,1) = X} \qquad \text{A = point(4,Y)} \qquad
\cfrac{2 = 2 \qquad 3 = Z}{\text{point(2,3) = point(2,Z)}}
}{
\begin{array}{c}
\text{triangle(point(1,1),A,point(2,3))} \\
\text{= triangle(X,point(4,Y),point(2,Z))}
\end{array}
}
$$

Declarative versus procedural meaning

There are two ways of understanding a Prolog program.

For example, consider the abstract query

$\mathcal{P}$:- $\mathcal{Q}$, $\mathcal{R}$.

where $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ are objects.

This clause can be read in two ways:
1. $\mathcal{P}$ is true if $\mathcal{Q}$ and $\mathcal{R}$ are true.
2. To prove $\mathcal{P}$, *first* prove $\mathcal{Q}$ and *next* prove $\mathcal{R}$.

Declarative versus procedural meanings (cont)

Note the difference in the wording and the ordering.

In the first case, we speak about **truth** and the order in which the truth values are obtained is actually not completely defined. For example, in this context, "$\mathcal{Q}$ and $\mathcal{R}$ are true" is equivalent to "$\mathcal{Q}$ and $\mathcal{R}$ are true", because $\mathcal{Q}$ must be true and, separately, $\mathcal{R}$ too.

In the second case, we speak about the **process** of obtaining the truth values in details. In this context, "$\mathcal{P}$ and $\mathcal{Q}$ are true" may **not** be equivalent to "$\mathcal{Q}$ and $\mathcal{P}$ are true" because one may be more efficient (e.g. if $\mathcal{P}$ is false, there is no need to prove $\mathcal{Q}$) or one may not terminate (e.g. if $\mathcal{P} = \mathcal{Q}$).

Declarative meaning

Informally, the declarative meaning of a Prolog program is as follows.

> A goal $\mathcal{G}$ is true (or logically follows from the program) if
>
> 1. there is a clause C in the program
> 2. of which an instance I can be deduced such that
>   - 2.1 the head of I is identical to $\mathcal{G}$,
>   - 2.2 all the goals of the body of I are true.

Note that it is not said how to find C and I, and no ordering of the goals is imposed (they just all must be true).

Procedural meaning

Given a list $\mathcal{G}$ of goals, a list $\mathcal{C}$ of clauses and the identity substitution σ,

1. *if $\mathcal{G}$ is empty, then end with σ and* **success***;*
2. *if $\mathcal{C}$ is empty then* **fail***; let G_1 be the first goal and C_1 the first clause;*
3. *let $\overline{C}_1$ be an instance of C_1 containing no variable in common with $\mathcal{G}$;*
4. *if the head of $\overline{C}_1$ does not match the head of G_1, restart with the remaining clauses;*
5. *let σ' be the resulting substitution, $\mathcal{G}'$ the remaining goals and $\mathcal{B}$ the goals in the body of $\overline{C}_1$; restart with the list of goals made of $\sigma'(\mathcal{G}')$ and $\sigma'(\mathcal{B})$ and substitution $\sigma' \circ \sigma$;*
6. *if it failed, restart with $\mathcal{G}$ and the remaining clauses in $\mathcal{C}$ (backtracking).*

Procedural meaning/Example

The declarative meaning can be seen as an *abstraction* of the
procedural meaning, i.e. it hides certain aspects of it. Consider

```
big(bear).          % Clause 1
big(elephant).      % Clause 2
small(cat).         % Clause 3
brown(bear).        % Clause 4
black(cat).         % Clause 5
gray(elephant).     % Clause 6
dark(Z) :- black(Z). % Clause 7
dark(Z) :- brown(Z). % Clause 8

?- dark(X), big(X).  % What is dark and big?
```

Procedural meaning/Example

The list of goals is $\mathcal{G} = (G_1, G_2)$, where $G_1 = \mathtt{dark(X)}$ and $G_2 = \mathtt{big(X)}$.

There is no match between G_1 and the facts, until clause 7.

Clause 7 has no variable in common with G_1.

The matching between the head of clause 7, $\mathtt{dark(Z)}$, and G_1 succeeds with substitution $\sigma' = \{\mathtt{X} = \alpha, \mathtt{Z} = \alpha\}$.

Let us start again with the list of goals $(\mathtt{black}(\alpha), \mathtt{big}(\alpha))$, and the substitution σ'.

Actually, it is enough to restrict σ' to the variables in G_1, so let us take instead
$$\sigma' = \{\mathtt{X} = \alpha\}$$

Procedural meaning/Example (cont)

Now the list of goals is $\mathcal{G} = (G_1, G_2)$, where $G_1 = \text{black}(\alpha)$ and $G_2 = \text{big}(\alpha)$, and $\sigma = \{\text{X} = \alpha\}$.

The first clause whose head matches $\text{black}(\alpha)$ is clause 5, which contains no variable. The substitution resulting of the matching is $\sigma' = \{\alpha = \text{cat}\}$.

Let us start again with the list of goals reduced to big(cat) (since clause 5 is a fact, so has no body) and substitution $\sigma' \circ \sigma = \{\text{X} = \text{cat}\}$.

But no clause has a head matching big(cat), so let us backtrack and try another match below clause 5.

There is none.

Procedural meaning/Example (cont)

So we must backtrack further and reconsider the list of goals is
$\mathcal{G} = (G_1, G_2)$, where $G_1 = \texttt{dark(X)}$ and $G_2 = \texttt{big(X)}$, and the identity
substitution σ.

The clause after clause 7, whose head matches G_1 is clause 8. It does
not have common variables with G_1. The resulting substitution is
$\sigma' = \{\texttt{X} = \beta\}$.

Let us start again with the list of goals $(\texttt{brown}(\beta), \texttt{big}(\beta))$ and the
substitution

$$\sigma' \circ \sigma = \sigma'$$

Procedural meaning/Example (cont)

Now the list of goals is $\mathcal{G} = (G_1, G_2)$, where $G_1 = \mathtt{brown}(\beta)$ and $G_2 = \mathtt{big}(\beta)$, and $\sigma = \{\mathtt{X} = \beta\}$.

The first clause whose head matches G_1 is clause 4, which contains not variable (it is a fact).

The matching leads to substitution $\sigma' = \{\beta = \mathtt{bear}\}$.

Let us start again with the list of goals $\mathtt{big(bear)}$ and the substitution

$$\sigma' \circ \sigma = \{\mathtt{X} = \mathtt{bear}\}$$

Procedural meaning/Example (cont)

Now the the list of goals is $\mathcal{G} = (G_1)$, where $G_1 = \text{big(bear)}$, and the substitution
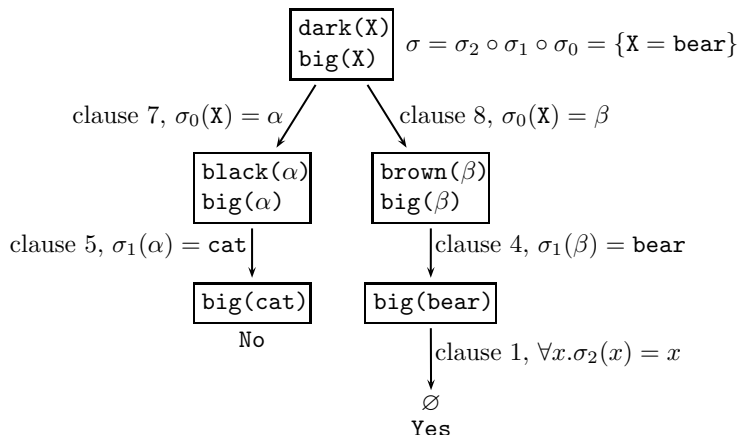
$$\sigma = \{X = \text{bear}\}$$

The first clause whose head matches G_1 is clause 1. It has no variable. The resulting substitution is the identity. Since it is a fact, it is proven and there is no new (sub-)goals.

This means that the interpreters ends with the positive result

```
?- dark(X), big(X).    % What is dark and big?
X = bear
Yes
```

Procedural meaning/Example (cont)

The corresponding proof tree is

$$
\langle 8, \{\text{Z = BEAR}\} \rangle \quad \cfrac{\cfrac{\langle 4 \rangle}{\text{brown(bear)}}}{\text{dark(bear)}} \quad \cfrac{\langle 1 \rangle}{\text{big(bear)}}
$$
$$
\overline{\text{dark(bear),big(bear)}}
$$

Declarative versus procedural meaning (resumed)

Given a Prolog program, it is possible to provide several programs which have the same declarative meaning but potentially different procedural meanings by playing on the order of the clauses and the order of the goals in the bodies. For example, consider again the relation `ancestor` page 36:

```
ancestor(X,Y) :- parent(X,Y).                    % Version 1
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).    % Version 2
ancestor(X,Y) :- parent(X,Y).

ancestor(X,Y) :- parent(X,Y).                    % Version 3
ancestor(X,Y) :- ancestor(Z,Y), parent(X,Z).

ancestor(X,Y) :- ancestor(Z,Y), parent(X,Z).    % Version 4
ancestor(X,Y) :- parent(X,Y).
```

Declarative versus procedural meaning (resumed)

Let x and y be some data object. Given a query

```
?- ancestor(x, y).
```

The procedural behaviours of the different versions are as follows:

- versions 1 and 2 always allow an answer to be found;
- versions 3 and 4 always loop forever.

Consider the examples:

```
?- ancestor(liz,jim).
?- ancestor(tom,pat).
```

Declarative versus procedural meaning (resumed)

What about the following variations?

```
ancestor(X,Y) :- parent(X,Y).                    % Version 3bis
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).

ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y). % Version 4bis
ancestor(X,Y) :- parent(X,Y).
```
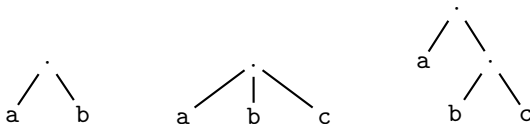
Tuples

Tuples and **lists** are a very useful data structure that aggregates objects in a specific order.

For example

```
.(a, b)       % Pair made of 'a' and 'b'
.(a, b, c)    % Triple made of 'a', 'b' and 'c'
.(a, .(b, c))
```

Tuples are common in mathematics, e.g. "Let p a point of coordinates (x, y) such as ..." We saw page 58 that Prolog objects can be represented as trees:

Lists

Lists are similar to tuples

```
[a, b]         % List made of 'a' and 'b'
[a, b, c]      % List made of 'a', 'b' and 'c'
[a, [b, c]]
```

So, where is the difference? List have a special syntax that allows to distinguish and extract the first elements, called **head**, while the remaining elements are called the **tail**:

```
[a, b, c]       % List made of 'a', 'b' and 'c'.
[a | [b,c]]     % Idem. Head is 'a' and tail is [b,c].
[a,b | [c]]     % Idem but head is [a, b] and tail is [c].
[a,b,c | []]    % Idem but head is [a, b, c] and tail is [].
```

Lists (cont)

Actually, list can be coded by means of tuples; all what is needed is a special atom for representing the empty list. Tradition notes it []. Then

```
[a, b, c]              % List made of 'a', 'b' and 'c'
.(a, .(b, .(c, []))))   % Same
```

Remember that lists can be **heterogeneous**, i.e. elements can be of any kind (see [a, [b, c]] above).

Lists/Membership

Let us write a Prolog program that checks if a given object belongs to a given list. Let member be this binary relation.

For example, we want

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

This suggests the recursive definition

X is a member of a list L if either

- *X is the head of L, or*
- *X is a member of the tail of L.*

Lists/Membership (cont)

This informal definition can straightforwardly be translated to Prolog:

```
member(X, [X | Tail]).
member(X, [Head | Tail]) :- member(X, Tail).
```

or

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

Remember that order matters for the procedural meaning!

Arithmetic

Arithmetic operators are special functors. The following are available:

+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Power
//	Integer division
mod	Modulo (remainder of integer division)

Arithmetic (cont)

If the Prolog interpreter is naively asked

```
?- X = 1 + 2.
X = 1+2
Yes
```

This because + is a functor and functors trigger no computation. Prolog builds the tree

```
      +
     / \
    1   2
```

In order to force the arithmetic interpretation, we use

```
?- X is 1 + 2.
X = 3
```

Arithmetic (cont)

We have

```
?- X is 5/2,
   Y is 5//2,
   Z is 5 mod 2.
X = 2.5
Y = 2
Z = 1
Yes
```

Arithmetic/Comparison

The comparison operators force the evaluation of their argument, as `is` does:

```
?- 5 * 3 > 2.
Yes
```

Assume we have a relation `born` in a program, that relates people's name to their birth years. We can find all the persons born between 1980 and 1990 by the query

```
?- born(Name, Year),
   Year >= 1980,
   Year =< 1990.
```

Arithmetic/Comparison (cont)

The comparison operators are

```
X > Y     X is greater than Y
X < Y     X is smaller than Y
X >= Y    X is greater than or equal to Y
X =< Y    X is smaller than or equal to Y
X =:= Y   the values of X and Y are equal
X =\= Y   the values of X and Y are not equal
```

Arithmetic/Comparison (cont)

Beware! The goals X = Y (matching) and X =:= Y (arithmetic comparison) are completely different.
Consider

```
?- 1 + 2 =:= 2 + 1.
Yes
?- 1 + 2 = 2 + 1.
No
?- 1 + A = B + 2.
A = 2
B = 1
Yes
```

Arithmetic/Example

Let us define a relation `length` which associate a list and its length.

```
length([],0).
length([_ | Tail],N) :- length(Tail,M), N is 1 + M.
?- length([a,b,[c,d],e],N).
N = 4
```

Note that "N is 1 + M" *must* be the second goal of the body. And what if

```
length([],0).
length([_ | Tail],N) :- length(Tail,M), N = 1 + M.
?- length([a,b,[c,d],e],N).
N = ???
```

Arithmetic/Example (bis)

Answer:

```
?- length([a,b,[c,d],e],N).
N = 1 + (1 + (1 + 0))
```

This, again, because + is just a functor. Therefore, we can equivalently write

```
length([],0).
length([_ | Tail],N) :- N = 1 + M, length(Tail,M).
?- length([a,b,[c,d],e],N).
N = 1 + (1 + (1 + 0))
```

Arithmetic/Example (bis)

Or even shorter

```
length([],0).
length([_ | Tail],1 + M) :- length(Tail,M).

?- length([a,b,[c,d],e],N).
N = 1 + (1 + (1 + 0))
Yes

?- length([a,b,[c,d],e],N), Length is N.
N = 1 + (1 + (1 + 0))
Length = 3
Yes
```

The eight queens problem

Let us now use Prolog to solve a more difficult kind of problem. One of these famous problems is **the eight queens problem**.

It consists in placing on a (European) chess board eight queens such that they do not attack each other.

We would like to define a predicate `solution` such that

```
?- solution(Pos).
```

returns a substitution for `Pos` which corresponds to a chess board position satisfying the problem's constraints.

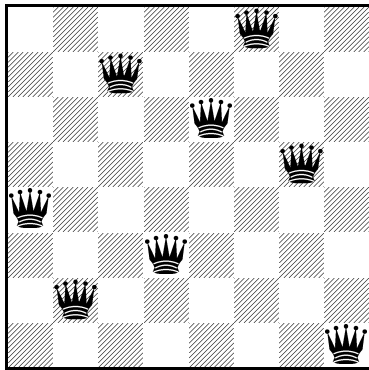# The eight queens problem (cont)

Figure: A solution to the eight queens problem.

The eight queens problem (cont)

First, we have to choose a representation for the board positions. One possibility is to model a square by two coordinates, the leftmost, down-most square being $(1, 1)$ and the rightmost, uppermost $(8, 8)$.

The example in the previous slide can be modeled by the list of queens

`[.(1,4),.(2,2),.(3,7),.(4,3),.(5,6),.(6,8),.(7,5),.(8,1)]`

Keeping this idea, we choose a template solution of the form

```
template([.(1,Y1),.(2,Y2),.(3,Y3),.(4,Y4),
          .(5,Y5),.(6,Y6),.(7,Y7),.(8,Y8)]).
```

because there must be a queen on each column.

The eight queens problem (cont)

There are two cases:

1. the list of queens is empty: the empty list is certainly a solution since there is no attack;

2. the list of queens is not empty: then it has the shape `[.(X,Y) | Others]`, that is, the first queen is on the square `.(X,Y)` and the others in the sub-list `Others`. If this is a solution, then the following constraints must hold.

   2.1 there must be no attack between the queens in `Others`, i.e. `Others` must be a solution;
   2.2 `X` and `Y` must be integers between 1 and 8;
   2.3 a queen at square `.(X,Y)` must not attack any of the queens in the list `Others`.

The eight queens problem (cont)

This is written in Prolog

```prolog
solution([]).

solution([.(X,Y) | Others]) :-
  solution(Others),
  member(Y, [1,2,3,4,5,6,7,8]),
  no_attack(.(X,Y), Others).

member(Item, [Item | _]).

member(Item, [_ | Tail]) :- member(Item, Tail).
```

It remains to define the relation no_attack.

The eight queens problem (cont)

Given no_attack(Q, Qlist), there are two cases.

1. if the list of queens Qlist is empty, then the relationship is true because there is no queen to attack or to be attacked by;

2. if the list Qlist is not empty, it must be of the shape [Q1 | Qsublist], with the following conditions holding:

   2.1 the queen at Q must not attack the queen at Q1,

   2.2 the queen at Q must not attack the queens in Qsublist.

The eight queens problem (cont)

A queen does not attack another queen if they are on different columns, rows and diagonals.

Finally:

```
no_attack(_, []).

no_attack(.(X,Y), [.(X1,Y1) | Others]) :-
  Y =\= Y1,
  Y1 - Y =\= X1 - X,
  Y1 - Y =\= X - X1,
  no_attack(.(X,Y), Others).
```

The eight queens problem (cont)

The query has the shape

```
?- template(S), solution(S).
```

Note that

```
?- solution(S), template(S).
```

is **wrong**. Why?