

# Chapter 1

## Introducing Prolog

We have been learning a couple of languages so far: *Java*, *C*, ..., and *C++*. We might also have heard of a couple of others: *Fortran*, *COBOL*, *Visual Basic*, etc.. Almost all of them fall into the category of *imperative paradigm*, namely, use them to solve a problem, we have to provide all the operational details in the form of an algorithm. More specifically, we have to tell the computer exactly *how to solve that problem*. Thus, they are also referred to as the *procedural language*.

In contrast, the family of *declarative* languages only define *what* is the problem, and let the computer system decide how to solve this problem. Prolog is such a language.

# An example

One of the very first problems we learned to solve is the linear search problem, i.e., given a list of numbers, and a target value, we want to know if the target occurs in that list. Below is an algorithm for this problem in C(?).

```
int lsearch(int a[], int n, int t){
    int i=0;
    bool found=0;

    while(i<n && !found){
        if(a[i]==t)
            found=1;
        i++;
    }

    if(found)
        return 1;
    else return 0;
}
```

## A basic structure

Let's look at this problem from another perspective: given a list,  $L$ , either  $L$  is empty, or  $L$  contains two parts: the first element, or *Head* of  $L$ , together with the rest, or *Tail*, of  $L$ . Thus, we can express  $L$  as  $[\text{Head}|\text{Tail}]$ .

For example, for the list  $\{1, 2, 3, 4\}$ , its head is 1, and its tail is  $\{2, 3, 4\}$ . Thus,  $\{1, 2, 3, 4\} = [1|[2, 3, 4]]$ .

Now, given this expression,  $[\text{Head}|\text{Tail}]$ , and a value  $t$ , the problem of linear search becomes the follows: the target  $t$  occurs in  $[\text{Head}|\text{Tail}]$  iff (i)  $t = \text{Head}$  or (ii)  $t$  occurs in  $\text{Tail}$ .

## The first Prolog program

In Prolog, this problem is represented as a *relation*, `lsearch(T, L)`, which is true if the target `T` occurs as one of the elements in the list `L`.

Below is the Prolog program that we can derive from this thought:

```
lsearch(T, [T|Tail]).  
lsearch(T, [H|Tail]):-lsearch(T, Tail).
```

Let's try this out, `linearSearch.pl`, with SWI-Prolog.

# Start from the very beginning

What we saw in the last slide is two *rules* that specify the problem. Besides rules, a Prolog program usually also contains some facts.

For example, when studying a family tree, we can have the following fact: Tom is a parent of Bob. This can be represented as the following Prolog *fact*:

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

We can also have the following facts.

```
parent(pam, bob).
```

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

```
parent(bob, ann).
```

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

```
parent(pat, jim).
```

The above is the content of the `parent` relation.

## A very brief history

Prolog is the mainstream programming language of the logic programming paradigm. It was originally done by Alain Colmerauer at Marseilles in the 1970's.

There are quite different implementation of this language. The one we will use in this course is called SWI-Prolog, a one done by People in the University of Amsterdam, Holland. It has been installed in the lab, and can also be freely downloaded from the following site:

`http://www.swi-prolog.org/`

You can also find various documentation under Help.

## How to open a program?

The easiest way is to use an editor to type in the codes, i.e., the facts and the rules, and then, within the *Windows Explorer*, double click the file. It is now ready to be executed.

Another way, after entering all the codes, is to load SWI-Prolog, then *consult* the program under **File**.

For a quick start of the SWI-Prolog environment, you might want to quickly go through §1, and §2.1-§2.9 of the reference document, as found in the course page.

## How to run a program?

Given the above facts, we can ask the system some questions, or posing some *queries*, as follows:

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

Prolog will answer 'yes' since it is true according to the facts it has got. It will say 'no' to the following queries:

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

A more interesting one is the following:

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

It means that "who is the parent of liz?". Prolog will send back

```
X=tom
```

## Even more fun

**Question:** Who is a parent of whom?

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

The answers will be

```
X=pam
```

```
Y=bob;
```

```
X=tom
```

```
Y=bob; ...
```

The output will stop either we enter a return, rather than a ‘;’; or it has exhausted the facts it knows.

**Question:** Who is a grandparent of jim?

**Answer:** Call the grandparent X. X must be a parent of Y, where Y is a parent of jim.

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

**Question:** Who are Tom's grandchildren?

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

Prolog will send back

X=bob

Y=ann;

X=bob

Y=pat

**Question:** Do Ann and Pat have a common parent?

**Answer:** Call the parent of Ann X. Ann and Pat have a common parent, if X is also a parent of Pat.

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

Prolog will send back X=bob.

## A couple of points

1. A Prolog program contains a query and definition of relations, by specifying the  $n$ -tuple objects that satisfy them, in terms of *clauses*.

For example, `parent` is a 2-ary relation, we specify six objects, such as `parent(tom, bob)` that satisfy its definition.

2. A clause can be either a fact, or a rule. For example,

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

is a fact. We will discuss rules extensively later.

3. The arguments of a relations can be *concrete objects (constant)*, such as `pam`; or *general objects* such as `x`. The former are called *atoms*; and the latter are called *variables*.

4. A query to the program consists of one or more *goals*, which must all be satisfied.

For example, the following sequence

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

means that Y is a parent of jim, *and* X is a parent of Y. The word “goal” is used since Prolog accepts the query as goals to be satisfied.

Thus, when the answer to a question is “yes”, we say it is *satisfied*, or *succeeded*; otherwise, we say it is *unsatisfiable*, or *failed*.

5. If several answers satisfy a goal, Prolog will find as many as desired.

**Homework:** Exercises 1.1 and 1.2.

## Add in more facts...

Let's throw more stuff into the family tree example. For example, we can add in relations for *gender*, as follows:

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

These facts can be given in other ways, e.g.,

```
sex(pam, feminine).
```

## ...and rules

We can also extend our knowledge by adding in other things, such as *offspring*.

One way to do it is to add in a bunch of facts. However, since we already have the facts for parents, and we know that for all X and Y,

if X is a parent of Y then Y is an offspring of X.

Thus, the following rule:

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

In the above, the condition part, `parent(X, Y)` is called the *body* of a rule, while the conclusion part is called the *head*.

## Why do we need a rule?

Apparently, such a rule greatly simplifies things: we don't need to enter a bunch of facts which we can deduct from something we already know.

More importantly, it provides a piece of rather deep and general knowledge about the family tree.

One critical component of AI is how to represent such knowledge, effectively and efficiently, inside a computer so that it can use them in its own "reasoning" about the world.

## How does Prolog use a rule?

Assume we want to know if Liz is an offspring of Tom, by asking the following question:

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

Since there are no facts regarding offspring, the only way Prolog can proceed is to use the aforementioned rule for the offspring relation.

The process begins by *substituting* such concrete instances as `liz` and `tom` for the general variables `X` and `Y`.

After this *instantiation* of `X=tom`, `Y=liz`, we obtain the following special case of the rule.

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

Now, the conclusion part matches with the desired goal, the Prolog tries to satisfy the condition part.

Thus, the original goal `offspring(liz, tom)` is replaced with the new goal `parent(tom, liz)`. The latter is trivially satisfied.(?)

Hence, the original goal is also satisfied. We will see a “yes” sent back.

## More rule examples

**Question:** How to specify the *mother* relation?

**answer:** For all  $X$  and  $Y$ ,

$X$  is the mother of  $Y$  if  $X$  is a parent of  $Y$  and  $X$  is female.

Thus,

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

**Question:** How to specify the *sister* relation?

**Answer:** For all  $X$  and  $Y$ ,

$X$  is a sister of  $Y$  if  $X$  and  $Y$  share a parent,  $X$  is female, and  $X$  is not  $Y$ .

Thus,

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

## A couple of points

1. During computation, a variable in a clause can be substituted, or instantiated, by another object.

2. Variables occurring in the head are *universally quantified*, while those that occur only in the body are *existentially quantified*. For example, the following rule

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

can be read as “for all  $X$ ,  $X$  has a child if for some  $Y$ ,  $X$  is a parent of  $Y$ .”

**Homework:** Exercises 1.4–1.5.

## Recursive rules

**Question:** How to specify the *ancestor* relation?

**A preliminary answer:** For all  $X$  and  $Y$ ,  $X$  is an ancestor of  $Y$  if  $X$  is a parent of  $Y$ , or  $X$  is a parent of a parent of  $Y$ , .... Thus,

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

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

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

```
...
```

This program does not work.(?)

## A better answer

If we think a bit more, we will come up with the following alternative answer: For all  $X$  and  $Y$ ,

$X$  is an ancestor of  $Y$  if  $X$  is a parent of  $Y$ ; or  $X$  is a parent of  $Z$  and  $Z$  is an ancestor of  $Y$ .

Thus,

$\text{Ancestor}(X, Y) :- \text{parent}(X, Y).$

$\text{Ancestor}(X, Y) :- \text{parent}(X, Z), \text{Ancestor}(Z, Y).$

## How does Prolog do it?

Given a sequence of goals, Prolog tries to satisfy all of them, by demonstrating that each and every goal is true, assuming that all the relations contained in the program are true. More specifically, if the goal contains variables, Prolog will find all the objects for which the goal is true.

A bit more formally, Prolog tries to show that the given goal *logically follows* from the facts and the rules contained in the program.

Thus, what really happens is that Prolog, accepting the facts and the rules contained in the program as axioms, tries to prove the given goal as a conjectured theorem.

## An example

Given the following Prolog program:

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

and the query

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

Prolog tries to show that the goal is true based on the given facts.

## What is to happen?

Logically, the process goes backwards:

`fallible(socrates)` is satisfied if `man(socrates)` is. The latter is indeed true, since we have it as a fact.

Technically, it needs a bit more: it needs a way to instantiate the variable  $X$  in the rule with `socrates`.

## A more detailed example

Given the family-tree program, when we want to know if Tom is an ancestor of Pat, we have the following query:

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

In trying to satisfy this goal, Prolog tries to find a clause such that this goal matches with its head. There are two such clauses, both of which are rules.

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

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

## Top-down among clauses

There are two options here. Prolog starts *from the top and go to the bottom*, and instantiates the first rule with (X=tom, Y=pat) to derive a specific instance

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

then tries to satisfy `parent(tom, pat)`. The latter fails.(?)

It then tries the other option, by instantiating the second rule to derive the following:

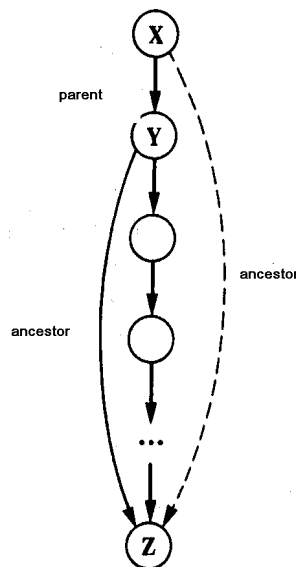
```
ancestor(tom, pat):-parent(tom, Y),  
                    ancestor(Y, pat).
```

then tries to satisfy both `parent(tom, Y)` and `ancestor(Y, pat)`.

# From left to right in one clause

Now, there are two goals to satisfy, Prolog proceeds *from left to right*.

Prolog trivially satisfies `parent(tom, Y)` with `Y=bob`, based on a given fact. This instantiation will be carried over to the second goal to derive `ancestor(bob, pat)`. After yet another round, Prolog proves this sub-goal, thus, also the original goal.



**Homework:** Exercises 1.7.