Fundamental Prolog (Part 1)

Written by Thomas Linder Puls
Prolog Development Center A/S

Last updated: 10-06-2004

In this tutorial you will learn about the very fundamental ideas of Prolog programming.

Visual Prolog 6 is object oriented, strictly typed and mode checked. You will of course have to master all this to write Visual Prolog 6 programs. But here we will focus on the *core* of the code, i.e. the code when disregarding classes, types and modes.

For this purpose we will use the PIE example that is included in the Visual Prolog 6 distribution. PIE is a "classical" Prolog interpreter, by using this you can learn and experiment with Prolog without at all being concerned with classes, types, etc.

Horn Clause Logic

Visual Prolog and other Prolog dialects are based on Horn Clause logic. Horn Clause logic is a formal system for reasoning about things and the way they relate to each other.

In natural language I can express a statement like:

> John is the father of Bill.

Here I have two "things": John and Bill, and a "relation" between these, namely that one is the father of the other. In Horn Clause Logic I can formalize this statement in the following way:

```
father("Bill", "John").
```

father is a predicate/relation taking two arguments, where the second is the father of the first.

Notice that ***I*** have chosen that the **second** person should be the father of the **first**. I might as well have chosen it the other way around: The order of the arguments is the choice of the "designer" of the formalization. However, once you have chosen, you must be consistent. So in **my** formalization the father must **always** be the second person.

I have chosen to represent the persons by their names (which are string literals). In a more complex world this would not be sufficient because many people have same name. But for now we will be content with this simple formalization.

With formalizations like the one above I can state any kind of family relation between any persons. But for this to become really interesting I will also have to formalize ***rules*** like this:

> X is the grandfather of Z, if X is the father of Y and Y is the father of Z

where X, Y and Z are persons. In Horn Clause Logic I can formalize this rule like this:

```
grandFather(Person, GrandFather) :-
    father(Person, Father), father(Father, GrandFather).
```

I have chosen to use ***variable*** names that help understanding better than X, Y and Z. I have also introduced a predicate for the grandfather relation. Again I have chosen that the grandfather should be the second argument. It is wise to be consistent like that, i.e. that the arguments of the different predicates follow some common principle. When reading rules you should interpret :- as ***if*** and the comma that separates the relations as ***and***.

Statements like "John is the father of Bill" are called ***facts***, while statements like "X is the grandfather of Z, if X is the father of Y and Y is the father of Z" are called ***rules***.

With facts and rules we are ready to formulate theories. A ***theory*** is a collection of facts and rules. Let me state a little theory:

```
father("Bill", "John").
father("Pam", "Bill").
grandFather(Person, GrandFather) :-
    father(Person, Father),
    father(Father, GrandFather).
```

The purpose of the theory is to answer questions like these:

```
Is John the father of Sue?
Who is the father of Pam?
Is John the grandfather of Pam?
...
```

Such questions are called goals. And they can be formalized like this (respectively):

```
?- father("Sue", "John").
?- father("Pam", X).
?- grandFather("Pam", "John").
```

Such questions are called ***goal clauses*** or simply ***goals***. Together *facts*, *rules* and *goals* are called Horn clauses, hence the name Horn Clause Logic.

Some goals like the first and last are answered with a simple *yes* or *no*. For other goals like the second we seek a ***solution***, like X = "Bill".

Some goals may even have many solutions. For example:

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

has two solutions:

```
X = "Bill", Y = "John".
X = "Pam", Y = "Bill".
```

A Prolog program is a theory and a goal. When the program starts it tries to find a solution to the goal in the theory.

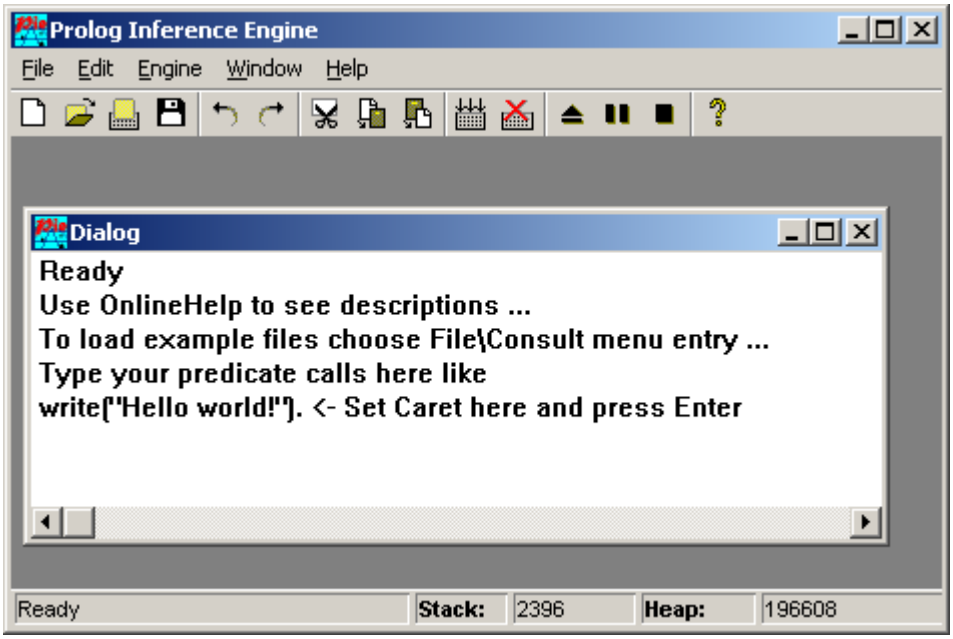
PIE: Prolog Inference Engine

Now we will try the little example above in PIE, the Prolog Inference Engine. That comes with Visual Prolog 6.

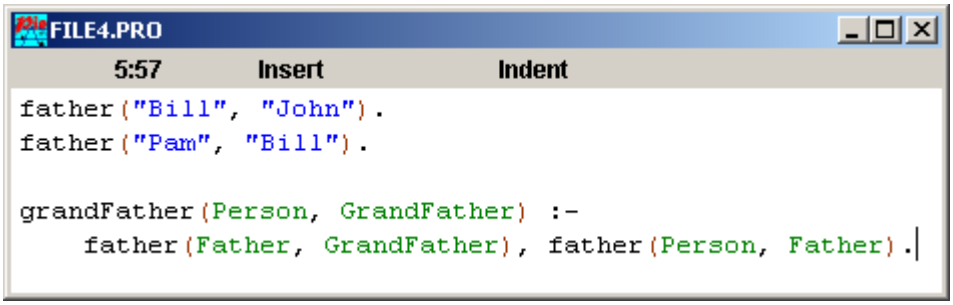
Before we start you should install and build the PIE example.

- Select "Install Examples" in the Windows start menu (**Start -> Visual Prolog 6 -> Install Examples**).
- Open the PIE project in the VDE and **run** the program, as it is described in Tutorial 01: Environment Overview.

When the program starts it will look like this:



Select **File -> New** and enter the father and grandFather clauses above:



While the editor window is active choose **Engine -> Reconsult**. This will load the file into the engine. In the ***Dialog*** window you should receive a message like this:

```
Reconsulted from: ....\pie\Exe\FILE4.PRO
```

Reconsult loads whatever is in the editor, ***without*** saving the contents to the file, if you want to save the contents use **File -> Save**.

File -> Consult will load the disc contents of the file regardless of whether the file is opened for editing or not.

Once you have "consulted" the theory, you can use it to answer goals.

On a blank line in the ***Dialog*** window type a goal (without the **?-** in front).

For example:

When the caret is placed at the end of the line, press the **Enter** key on your keyboard. PIE will now consider the text from the beginning of the line to the caret as a goal to execute. You should see a result like this:

Extending the family theory

It is straight forward to extend the family theory above with predicates like mother and grandMother. You should try that yourself. You should also add more persons. I suggest that you use persons from your own family, because that makes it lot easier to validate, whether some person is in deed the grandMother of some other person, etc.

Given mother and father we can also define a parent predicate. You are a parent if you are a mother; you are also a parent if you are a father. Therefore we can define parent using two clauses like this:

```
parent(Person, Parent) :- mother(Person, Parent).
parent(Person, Parent) :- father(Person, Parent).
```

The first rule reads (recall that the second argument corresponds to the predicate name):

> Parent is the parent of Person, if Parent is the mother of Person

You can also define the parent relation using semicolon ";" which means or, like this:

```
parent(Person, Parent) :-
   mother(Person, Parent);
   father(Person, Parent).
```

This rule reads:

> Parent is the parent of Person, if Parent is the mother of Person ***or*** Parent is the father of Person

I will however advise you to use semicolon as little as possible (or actually not at all). There are several reasons for this:

- The typographical difference "," and ";" is very small, but the semantic difference is rather big. ";" is often a source of confusion, since it is easily misinterpreted as ",", especially when it is on the end of a long line.
- Visual Prolog only allows to use semicolon on the outermost level (PIE will allow arbitrarily deep nesting).

Try creating a sibling predicate! Did that give problems?

You might find that siblings are found **twice**. At least if you say: Two persons are siblings if they have same mother, two persons are also siblings if they have same father. I.e. if you have rules like this:

```
sibling(Person, Sibling) :- mother(Person, Mother), mother(Sibling, Mother).
sibling(Person, Sibling) :- father(Person, Mother), father(Sibling, Mother).
```

The first rule reads:

> Sibling is the sibling of Person, if Mother is the mother of Person ***and*** Mother is the mother of Sibling

The reason that you receive siblings twice is that most siblings both have same father and mother, and therefore they fulfill both requirements above. And therefore they are found twice.

We shall not deal with this problem now; currently we will just accept that some rules give too many results.

A fullBlodedSibling predicate does not have the same problem, because it will require that both the father and the mother are the same:

```
fullBlodedSibling(Person, Sibling) :-
    mother(Person, Mother),
    mother(Sibling, Mother),
    father(Person, Father),
    father(Sibling, Father).
```

Prolog is a Programming Language

From the description so far you might think that Prolog is an expert system, rather than a programming language. And indeed Prolog can be used as an expert system, but it is designed to be a programming language.

We miss two important ingredients to turn Horn Clause logic into a programming language:

- Rigid search order/program control
- Side effects

Program Control

When you try to find a solution to a goal like:

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

You can do it in many ways. For example, you might just consider at the second fact in the theory and then you have a solution.

But Prolog does not use a "random" search strategy, instead it always use the same strategy. The system maintains a ***current goal***, which is always solved from **left to right**.

I.e. if the current goal is:

```
?- grandFather(X, Y), mother(Y, Z).
```

Then the system will always try to solve the sub-goal grandFather(X, Y) before it solves mother(Y, Z), if the first (i.e. left-most) sub-goal cannot be solved then there is no solution to the overall problem and then the second sub-goal is not tried at all.

When solving a particular sub-goal, the facts and rules are always tried from **top to bottom**.

When a sub-goal is solved by using a rule, the right hand side replaces the sub-goal in the current goal.

I.e. if the current goal is:

```
?- grandFather(X, Y), mother(Y, Z).
```

And we are using the rule

```
grandFather(Person, GrandFather) :- father(Person, Father), father(Father, GrandFather).
```

to solve the first sub-goal, then the resulting current goal will be:

```
?- father(X, Father), father(Father, Y), mother(Y, Z).
```

Notice that some variables in the rule have been replaced by variables from the sub-goal. I will explain this in details later.

Given this evaluation strategy you can interpret clauses much more ***procedural***. Consider this rule:

```
grandFather(Person, GrandFather) :- father(Person, Father), father(Father, GrandFather).
```

Given the strict evaluation we can read this rule like this:

> To solve grandFather(Person, GrandFather) **first** solve father(Person, Father) **and then** solve father(Father, GrandFather).

Or even like this:

> When grandFather(Person, GrandFather) is ***called***, first ***call*** father(Person, Father) and then ***call*** father(Father, GrandFather).

With this ***procedural*** reading you can see that predicates correspond to procedures/subroutines in other languages. The main difference is that a Prolog predicate can return several solutions to a single invocation or even ***fail***. This will be discussed in details in the next sections.

Failing

A predicate invocation might not have any solution in the theory, for example calling parent

("Hans", X) has no solution as there are no parent facts or rules that applies to "Hans". We say that the predicate call ***fails***. If the goal fails then there is simply no solution to the goal in the theory. The next section will explain how failing is treated in the general case, i.e. when it is not the goal that fails.

Backtracking

In the procedural interpretation of a Prolog program "or" is treated in a rather special way. Consider the clause

```
parent(Person, Parent) :-
    mother(Person, Parent);
    father(Person, Parent).
```

In the logical reading we interpreted this clause as:

> Parent is the parent of Person **if** Parent is the mother of Person **or** Parent is the father of Person.

The "or" introduces two possible solutions to an invocation of the parent predicate. Prolog handles such multiple choices by first trying one choice and later (if necessary) ***backtracking*** to the next alternative choice, etc.

During the execution of a program a lot of alternative choices (known as ***backtrack points***) might exist from earlier predicate calls. If some predicate call fails, then we will backtrack to the last backtrack point we met and try the alternative solution instead. If no further backtrack points exists then the overall goal has failed, meaning that there was no solution to it.

With this in mind we can interpret the clause above like this:

> When parent(Person, Parent) is called first record a backtrack point to the second alternative solution (i.e. to the call to father(Person, Parent)) and then call mother(Person, Parent)

A predicate that has several classes behave in a similar fashion. Consider the clauses:

```
father("Bill", "John").
father("Pam", "Bill").
```

When father is invoked we first record a backtrack point to the second clause, and then try the first clause.

If there are three or more choices we still only create one backtrack point, but that backtrack point will start by creating another backtrack point. Consider the clauses:

```
father("Bill", "John").
father("Pam", "Bill").
father("Jack", "Bill").
```

When father is invoked, we first record a backtrack point. And then we try the first clause. The backtrack point we create points to some code, which will itself create a backtrack point (namely to the third clause) and then try the second clause. Thus all choice points have only two choices, but one choice might itself involve a choice.

Example To illustrate how programs are executed I will go through an example in details. Consider these clauses:

```
mother("Bill", "Lisa").
father("Bill", "John").
father("Pam", "Bill").
father("Jack", "Bill").
parent(Person, Parent) :-
   mother(Person, Parent);
   father(Person, Parent).
```

And then consider this goal:

```
?- father(AA, BB), parent(BB, CC).
```

This goal states that we want to find three persons AA, BB and CC, such that BB is the father of AA and CC is a parent of BB.

As mentioned we always solve the goals from left to right, so first we call the father predicate. When executing the father predicate we first create a backtrack point to the second clause, and then use the first clause.

Using the first clause we find that AA is "Bill" and BB is "John". So we now effectively have the goal:

```
?- parent("John", CC).
```

So we call parent, which gives the following goal:

```
?- mother("John", CC); father("John", CC).
```

You will notice that the variables in the clause have been replaced with the actual parameters of the call (exactly like when you call subroutines in other languages).

The current goal is an "or" goal, so we create a backtrack point to the second alternative and pursuit the first. We now have two active backtrack points, one to the second alternative in the parent clause, and one to the second clause in the father predicate.

After the creation of this backtrack point we are left with the following goal:

```
?- mother("John", CC).
```

So we call the mother predicate. The mother predicate fails when the first argument is "John" (because it has no clauses that match this value in the first argument).

In case of failure we backtrack to the last backtrack point we created. So we will now pursuit the goal:

```
?- father("John", CC).
```

When calling father this time, we will again first create a backtrack point to the second father clause.

Recall that we also still have a backtrack point to the second clause of the father predicate, which corresponds to the first call in the original goal.

We now try to use the first father clause on the goal, but that fails, because the first arguments do not match (i.e. "John" does not match "Bill").

Therefore we backtrack to the second clause, but before we use this clause we create a backtrack point to the third clause.

The second clause also fails, since "John" does not match "Pam", so we backtrack to the third clause. This also fails, since "John" does not match "Jack".

Now we must backtrack all the way back to the first father call in the original goal; here we created a backtrack point to the second father clause.

Using the second clause we find that AA is "Pam" and BB is "Bill". So we now effectively have the goal:

```
?- parent("Bill", CC).
```

When calling parent we now get:

```
?- mother("Bill", CC); father("Bill", CC).
```

Again we create a backtrack point to the second alternative and pursuit the first:

```
?- mother("Bill", CC).
```

This goal succeeds with CC being "Lisa". So now we have found a solution to the goal:

```
AA = "Pam", BB = "Bill", CC = "Lisa".
```

When trying to find additional solutions we backtrack to the last backtrack point, which was the second alternative in the parent predicate:

```
?- father("Bill", CC).
```

This goal will also succeed with CC being "John". So now we have found one more solution to the goal:

```
AA = "Pam", BB = "Bill", CC = "John".
```

If we try to find more solutions we will find

```
AA = "Jack", BB = "Bill", CC = "John".
AA = "Jack", BB = "Bill", CC = "Lisa".
```

After that we will experience that everything will eventually fail leaving no more backtrack points. So all in all there are four solutions to the goal.

Improving the Family Theory

If you continue to work with the family relation above you will probably find out that you have problems with relations like *brother* and *sister*, because it is rather difficult to determine the sex of a person (unless the person is a father or mother).

The problem is that we have chosen a bad way to formalize our theory.

The reason that we arrived at this theory is because we started by considering the ***relations*** between the entities. If we instead first focus on the ***entities***, then the result will naturally

become different.

Our main entities are persons. Persons have a name (in this simple context will still assume that the name identifies the person, in a real scale program this would not be true). Persons also have a sex. Persons have many other properties, but none of them have any interest in our context.

Therefore we define a person predicate, like this:

```
person("Bill", "male").
person("John", "male").
person("Pam", "female").
```

The first argument of the person predicate is the name and the second is the sex.

Instead of using mother and father as facts, I will choose to have parent as facts and mother and father as rules:

```
parent("Bill", "John").
parent("Pam", "Bill").
father(Person, Father) :- parent(Person, Father), person(Father, "male").
```

Notice that when father is a "derived" relation like this, it is impossible to state *female* fathers. So this theory also has a built-in consistency on this point, which did not exist in the other formulation.

Recursion

Most family relations are easy to construct given the principles above. But when it comes to "infinite" relations like *ancestor* we need something more. If we follow the principle above, we should define ancestor like this:

```
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), parent(P1, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), parent(P1, P2), parent(P2, Ancestor).
...
```

The main problem is that this line of clauses never ends. The way to overcome this problem is to use a recursive definition, i.e. a definition that is defined in terms of itself. like this:

```
ancestor(Person, Ancestor) :- parent(Person, Ancestor).
ancestor(Person, Ancestor) :- parent(Person, P1), ancestor(P1, Ancestor).
```

This declaration states that a parent is an ancestor, and that an ancestor to a parent is also an ancestor.

If you are not already familiar with recursion you might find it tricky (in several senses). Recursion is however fundamental to Prolog programming. You will use it again and again, so eventually you will find it completely natural.

Let us try to execute an ancestor goal:

```
?- ancestor("Pam", AA).
```

We create a backtrack point to the second ancestor clause, and then we use the first, finding the new goal:

```
?- parent("Pam", AA).
```

This succeeds with the solution:

```
AA = "Bill".
```

Then we try to find another solution by using our backtrack point to the second ancestor clause. This gives the new goal:

```
?- parent("Pam", P1), ancestor(P1, AA).
```

Again "Bill" is the parent of "Pam", so we find P1= "Bill", and then we have to goal:

```
?- ancestor("Bill", AA).
```

To solve this goal we first create a backtrack point to the second ancestor clause and then we use the first one. This gives the following goal

```
?- parent("Bill", AA).
```

This goal has the gives the solution:

```
AA = "John".
```

So now we have found two ancestors of "Pam": "Bill" and "John".

If we use the backtrack point to the second ancestor clause we get the following goal:

```
?- parent("Bill", P1), ancestor(P1, AA).
```

Here we will again find that "John" is the parent of "Bill", and thus that P1 is "John". This gives the goal:

```
?- ancestor("John", AA).
```

If you pursuit this goal you will find that it will not have any solution. So all in all we can only find two ancestors of "Pam".

Recursion is very powerful but it can also be a bit hard to control. Two things are important to remember:

- the recursion must make progress
- the recursion must terminate

In the code above the first clause ensures that the recursion can terminate, because this clause is not recursive (i.e. it makes no calls to the predicate itself).

In the second clause (which is recursive) we have made sure, that we go one ancestor-step further back, before making the recursive call. I.e. we have ensured that we make some progress in the problem.

Side Effects

Besides a strict evaluation order Prolog also has side effects. For example Prolog has a number of predefined predicates for reading and writing.

The following goal will write the found ancestors of "Pam":

```
?- ancestor("Pam", AA), write("Ancestor of Pam : ", AA), nl().
```

The ancestor call will find an ancestor of "Pam" in AA.

The write call will write the string literal "Ancestor of Pam : ", and then it will write the value of AA.

The nl call will shift to a new line in the output.

When running programs in PIE, PIE itself writes solutions, so the overall effect is that your output and PIE's own output will be mixed. This might of course not be desirable.

A very simple way to avoid PIE's own output is to make sure that the goal has no solutions. Consider the following goal:

```
?- ancestor("Pam", AA), write("Ancestor of Pam : ", AA), nl(), fail.
```

fail is a predefined call that always fails (i.e. it has no solutions).

The first three predicate calls have exactly the same effect as above: an ancestor is found (if such one exists, of course) and then it is written. But then we call fail this will of course *fail.* Therefore we must pursuit a backtrack point if we have any.

When pursuing this backtrack point, we will find another ancestor (if such one exists) and write that, and then we will fail again. And so forth.

So, we will find and write all ancestors. and eventually there will be no more backtrack points, and then the complete goal will fail.

There are a few important points to notice here:

- The goal itself did not have a single solution, but nevertheless all the solutions we wanted was given as side effects.
- Side effects in failing computations are not undone.

These points are two sides of the same thing. But they represent different level of optimism. The first optimistically states some possibilities that you can use, while the second is more pessimistic and states that you should be aware about using side effects, because they are not undone even if the current goal does not lead to any solution.

Anybody, who learns Prolog, will sooner or later experience unexpected output coming from failing parts of the program. Perhaps, this little advice can help you: Separate the "calculating" code from the code that performs input/output.

In our examples above all the stated predicate are "calculating" predicates. They all calculate some family relation. If you need to write out, for example, "parents", create a separate predicate for writing parents and let that predicate call the "calculating" parent predicate.

Conclusion

In this tutorial we have looked at some of the basic features of Prolog. You have seen

facts, ***rules*** and ***goals***. You learned about the ***execution strategy*** for Prolog including the notion of ***failing*** and ***backtracking***. You have also seen that backtracking can give ***many results*** to a single question. And finally you have been introduced to ***side effects***.