The Prolog Programming Language

Prolog stands for PROgramming in LOGic. It was developed from a foundation of logical theorem proving and originally used for research in natural language processing. Although its popularity has sprung up mainly in the artificial intelligence (AI) community, where it has been used for applications such as *expert systems*, *natural language*, and *intelligent databases*, it is also useful for more conventional types of applications. It allows for more rapid development and prototyping than most languages because it is semantically close to the logical specification of a program.

Programming in Prolog is significantly different from conventional procedural programming and requires a readjustment in the way one thinks about programming. Logical relationships are asserted, and Prolog is used to determine whether or not certain statements are true, and if true, what variable bindings make them true. This leads to a very declarative style of programming.

One of the Prolog's tutorials by Dennis Merritt that I found on the WEB starts from the story that gives some idea what this computer language is good for. Here it is.

Dennis was working for an aerospace company in the 1970s when someone got a copy of the original Adventure game - a simulated world the player explores, at that time purely text-based, with natural language and installed it on mainframe computer. For the next month his lunch hours, evenings and weekends, as well as normal work hours, were consumed with fighting the green dragon and escaping from

the twisty little passages. Finally he had proudly earned all the points in the game.

This was time for performance review. His boss was a stern man, who was more comfortable with machines than with people. He opened up a large computer printout containing a log of the hours each of his programmers spent on the mainframe computer. He noticed that recently Dennis had been working evenings and weekends and that he admired that type of dedication in his employees. He gave Denis the maximum raise and told him to keep up the good work. Years later, when he got first home computer, he immediately started to write his own adventure game in 'C'. First came the tools, a simple dynamic database to keep track of the game state and pattern matching functions to search that database. Then came a natural language parser for the front end. Functions implemented the various rules of the game.

At around the same time Dennis joined the Boston Computer Society and attended a lecture of the newly formed Artificial Intelligence group. The lecture was about Prolog. He was amazed - here was a language that included all of the tools needed for building adventure games and more. It had a much richer dynamic database and more powerful pattern matcher than the one he had written, plus its syntax was rules, which are much more natural for coding the specification of the game. It had a built-in search engine and, to top it all off, had tools for natural language processing.

To get some idea of what the Logic Programming is let’s look at few short examples. In classical logic we might say, “All people are mortal”, or rephrase for Prolog, “For all X, X is mortal if X is a person.”

```
mortal(X) :- person(X).
```

Similarly, we can assert the simple fact that Socrates is a person.

```
person(socrates).
```

From these two logical assertions, Prolog can now prove whether or not Socrates is mortal.

```
?- mortal(socrates).
```

The listener responds

```
Yes
```

We could also ask "Who is mortal?" like this

```
?- mortal(X).
```

and receive the response

```
X = Socrates
```

This declarative style of programming is one of Prolog's major strengths. For the most part, the programmer is freed from having to worry about control structures and transfer of control mechanisms. Prolog does it automatically. However, logical theorems prove is not a practical programming tool. A programmer needs to do things that have nothing to do with logic, such as read and write terms. A programmer also needs to manipulate the built-in control structure of Prolog in order for the program to execute as desired. The following example illustrates a Prolog program that prints a report of all the known mortals. It is a mixture of pure logic from before, extra-logical I/O, and forced control of the Prolog execution behavior. We add some more philosophers to the 'mortal' source.

```
person(plato)
person(zeno).
person(aristotle).
```

Next we add the report-writing code.

```
mortal_report:-
write('Known mortals are:'),nl,
mortal(X),
write(X),nl,
fail.
```

The execution:

```
?- mortal_report
Known mortals are:
socrates
plato
aristotle
```

A Prolog program is a Prolog database. The program is composed of **predicates** (procedures, record, types, relations). In the example above, there are three predicates. They are *person, mortal_report, and mortal*. Each of these three predicates has a distinctly different flavor. *person* looks like multiple data records with one data field for each, *mortal_report* looks like a procedure with no arguments, ***mortal*** is a logical **assertion** or **rule** that is somewhere in between data and procedure. Each predicate in a program is defined by the existence of one or more **clauses** in the database. In our example, the predicate person has four clauses. The other predicates have only one clause. A clause can be either a **fact** or a **rule**. The three clauses of the person predicate are all facts.

The single clauses of *mortal_report* and *mortal* are both rules. Facts are the simplest form of Prolog predicates, and are similar to records in a relational

database. They can be queried as a database. Prolog queries work by pattern matching. The query pattern is called a **goal**. If there is a fact that matches the goal, then the query succeeds and the listener responds with 'yes.' If there is no matching fact, then the query fails and the listener responds with 'no.'

Many newcomers to Prolog find that the task of writing a Prolog program is not like specifying an algorithm in the same way as in a conventional programming language. Instead, the Prolog programmer asks more what formal relationships and objects occur in his problem, and what relationships are "true" about the desired solution. So, Prolog can be viewed as a descriptive language. The Prolog approach is rather to describe known facts and relationships about a problem, than to prescribe the sequence of steps taken by a computer to solve the problem. When a computer is programmed in Prolog, the actual way the computer carries out the computation is specified partly by the logical declarative semantics of Prolog, partly by what new facts Prolog can 'infer' from the given ones, and only partly by explicit control information supplied by the programmer.

Some of Prolog’s terms.

Atom	The most fundamental element in Prolog made up of a string of characters, numbers, and some special characters.
Backtracking	A control method used to search backwards for solutions.

Backward-Chaining	A process used to find the solution by searching backwards from the solution towards the initial conditions thus verifying the specified goal.
Binding	The process of assigning a variable a value.
Clauses	Either a Prolog fact or rule.
Cut	An operator used to terminate backtracking in areas that will not give useful solutions.
Declarative Language	A language that allows programming by defining the boundary conditions and constraints and letting the computer determine a solution that meets these requirements.
Fact	A statement about the relationship between objects.
Fail	A Prolog operator that causes backtracking to occur.
Forward-chaining	A process used to find the solution by starting with an assumption and working toward a final goal.
Goal	The solution that the Prolog program is trying to reach or prove correct.
Predicate	A function that returns either a true or false value.
Proposition	An expression about an object which can have either a true or false value.
Rule	A clause that defines the relationship or relationships between facts and objects.

Terms	An object, compound object, variable, or list.
Unification	The pattern matching technique used by Prolog to match goals and sub-goals in a program.