The Pragmatic Programmers

grox.io Learning

Programmer Passport

Prolog

Bruce A. Tate

Edited by Jacquelyn Carter

Prolog

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

ISBN-13: pending
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—Monthname, yyyy

Contents

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B4.0: Mar 1, 2020

- This is the last Prolog chapter, the second language in our Joe Armstrong tribute and celebration. This chapter is focused on practical Prolog, namely writing schedulers. Finish strong!

B3.0: Feb 17, 2020

- Welcome to the third Prolog chapter, the second language in our Joe Armstrong tribute and celebration. We'll focus on solving problems with graphs. Graph theory is becoming increasingly important in computer science, and you're about to find out why. Have fun!

B2.0: Feb 3, 2020

- Welcome to the second Prolog chapter, the second language in our Joe Armstrong tribute and celebration. We'll solve the eight queens problem and the map coloring problem. If you're on the Programmer Passport site, you can also find the video for solving a Sudoku puzzle. Have fun!

B1.0: January 15, 2020

- Welcome to the first Prolog chapter. Prolog is the first language in our Joe Armstrong tribute and celebration. Prolog was one of his favorite languages, and was used to make the first compiler for the Erlang language.

CHAPTER 1

Logic Programming Basics

The next three releases in the Programmer Passport are part of our tribute to Joe Armstrong, one of the creators of Erlang. We're going to focus on languages that inspired Joe and a few other technologies that Joe inspired, starting with Prolog. Many developers may not know that the earliest implementations of Erlang were written in Prolog. Eventually, Erlang was moved from Prolog to C, but you can still see a heavy Prolog influence.

Prolog is at once fascinating and maddening because it's so different from many of the other languages you might have experienced. Instead of giving Prolog a rote program describing exactly what to do, you'll give it a database of facts such as "Bruce follows José". Then, you'll add some basic inferences to your database such as "A user receives a tweet if that user follows someone, and that person tweets something." Once you have that database, you can ask Prolog questions, called queries, such as "Who receives tweets from José?" Prolog will tie the facts and inferences together to solve some pretty demanding questions.

In this chapter, we'll start with what makes Prolog different from other languages. Then, we'll move into the Prolog environment and the basics of logic programming. Finally, we'll look at making inferences with Prolog before we conclude.

As you work through this module, look for ways that Prolog can help you think about problem solving. Many programming languages can benefit from the way Prolog establishes rules and inferences. Some programming languages even have implementations built in, especially Lisp dialects like Clojure. Keep your eyes open and you're sure to find something that you find interesting. Let's get started!

What is Prolog, Anyway?

As we start each language, the first thing we strive to do together is understand a language's reason for being. So far, the languages we've covered have both been general purpose languages. Though they're each built to play in a focused niche, both Crystal and Pony are built to solve general purpose problems within their niche. Java, Ruby, and Python are examples of mainstream general purpose languages. Lisp, Erlang, and Haskell are examples of mainstream languages that establish a niche among general purpose languages.

Other languages exist to solve problems in a specific genre. For example, SQL is built to retrieve data from databases and HTML is built to provide structure to documents, particularly web pages. Like these languages, Prolog is a problem-specific language built to handle logic programming. As such, it's an important language for artificial intelligence.

Since Prolog is not a general purpose language, its goals and strategies for processing programs vary significantly from other languages. Let's look at some of the features that define it.

Prolog is a language that relies on databases. Each databases uses data in the form of facts and rules. Facts are specific axioms about a problem set, and rules express generalizations. Then, Prolog ties the data in the database together to make more sophisticated inferences. These factors make Prolog a good language for problems that involve logic, especially when a program must establish a series of facts to get to a more sophisticated inference.

Important Prolog Features

Prolog made at least three huge contributions to programming languages: unification, backtracking, and inference programming. Let's form a rough definition of each, and then we'll tighten them up as we go.

unification

Most languages you're used to probably work on one side of an expression, like an assignment, at a time. With Prolog's unification, a program can solve for variables on both sides of a single expression. We'll look at many unification problems through the course of these four chapters.

backtracking

The next major contribution is backtracking. When Prolog is working on a problem, it doesn't look for a single solution to a problem. It looks for all of them. When Prolog is solving a complex query, sometimes it looks for all possible partial solutions. When one partial solution is wrong, Prolog can backtrack and try another.

inferences
Prolog is able to combine multiple rules in a database to build its own inferences. These can happen recursively, and these inferences can be quite complex. For example, when we solve a map coloring problem, we'll tell Prolog that colors come from a list, that certain countries share borders, and that our map must not put similar colors together. Prolog will infer valid colorings!

These different features make Prolog a good choice for some artificial intelligence problems. Before we install Prolog, let's look into some of the basics.

Prolog Programming Basics

At its core, Prolog programs have three parts: facts, rules, and queries. A *fact* is a piece of information about the world. In this section, we'll look at simple facts made up of predicates and atoms. Here are some examples of facts, and their potential meanings.

fact	meaning
`red(car).`	The `car` is red
`fast(car).`	The `car` is fast
`fun(car).`	The `car` is fun
`owns(car, jim).`	`jim` owns `car`
`blue(prius).`	The `prius` is `blue`

In these facts, the atoms are `car`, `jim`. and `prius`, the words inside the parentheses. Each of these expressions is a predicate.

Our predicates can get more sophisticated. Our logic gets more interesting once we use *variables* within logical *rules*, and let Prolog fill out those variables. A variable starts with an uppercase letter. Eventually variables will hold values, such as the `car` and `jim` atoms. We can specify *rules*. A rule specifies a type of if-then condition. Here are some examples.

rule	meaning
`red(Thing) :- fun(Thing).`	If a `thing` is `red`, it's `fun`
`fast(Thing) :- fun(Thing).`	If a `thing` is `fast`, it's `fun`

The above rules will let Prolog infer that a red `car` is `fun`, but a `blue prius` is not `fun`.

With the basics out of the way, let's install Prolog and write some code!

The SWI Prolog Console

Before we can solve problems in Prolog, we'll need to install the Prolog console and learn to navigate it. We'll be working with SWI Prolog. This implementation was built by Jan Wielemaker. SWI comes from the name of a University of Amsterdam group Sociaal-Wetenschappelijke Informatica. The English translation is "Social Science Informatics."

This open source Prolog implementation has binaries that run on Windows, OS X, and several Unix dialects. It's one of the most broadly used Prolog implementations today. Follow the installation instructions.[1]

After working with a couple of languages that don't have first class consoles, you may be pleased to learn that we'll be doing much of our work in consoles that allow us to load and manipulate data directly. Before we create a database, let's issue a few commands to get a sense for how the console works.

Each command we type is a *query*, one that returns either true or false, and Prolog will do its best to answer that query. If our query has variables, Prolog will do its best to fill in the values that make the query true.

Once you've installed, bring it up by typing swipl (on most environments), and then type this query:

```
?- write("hello, world").
```

Notice we terminated the statement with . and that's important. It tells Prolog that the query is done, and it's time to start computing.

Prolog responds with:

```
hello, world
true.
```

Each query has a value of true or false. A *predicate* is part of a query. The write predicate is always true, and it prints a value to the console. The second line is the value of the query we just entered, true.

If you want to join two statements together with and, type a comma between them, like this:

```
?- write("hello, world"), write(" and that's a wrap.").
hello, world and that's a wrap.
true.
```

1. https://wwu-pi.github.io/tutorials/lectures/lsp/010_install_swi_prolog.html