Extracted from:

Seven Languages in Seven Weeks

A Pragmatic Guide to Learning Programming Languages

Seven Languages in Seven Weeks

A Pragmatic
Guide to
Learning
Programming
Languages

Bruce A. Tate

Edited by Jacquelyn Carter

Seven Languages in Seven Weeks

A Pragmatic Guide to Learning Programming Languages

Bruce A. Tate

Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jackie Carter (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
Steve Peter (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Ah, Prolog. Sometimes spectacularly smart, other times just as frustrating. You'll get astounding answers only if you know how to ask the question. Think *Rain Man.*[1] I remember watching Raymond, the lead character, rattle off Sally Dibbs' phone number after reading a phone book the night before, without thinking about whether he should. With both Raymond and Prolog, I often find myself asking, in equal parts, "How did he know that?" and "How didn't he know that?" He's a fountain of knowledge, if you can only frame your questions in the right way.

Prolog represents a serious departure from the other languages we've encountered so far. Both Io and Ruby are called *imperative languages*. Imperative languages are recipes. You tell the computer exactly how to do a job. Higher-level imperative languages might give you a little more leverage, combining many longer steps into one, but you're basically putting together a shopping list of ingredients and describing a step-by-step process for baking a cake.

It took me a couple of weeks of playing with Prolog before I could make an attempt at this chapter. I used several tutorials as I ramped up, including a tutorial by J. R. Fisher[2] for some examples to wade through and another primer by A. Aaby[3] to help the structure and terminology gel for me, and lots of experimentation.

Prolog is a declarative language. You'll throw some facts and inferences at Prolog and let it do the reasoning for you. It's more like going to a good baker. You describe the characteristics of cakes that you like and let the baker pick the ingredients and bake the cake for you, based on the rules you provided. With Prolog, you don't have to know *how*. The computer does the reasoning for you.

With a casual flip through the Internet, you can find examples to solve a Sudoku with fewer than twenty lines of code, crack Rubik's Cube, and solve famous puzzles such as the Tower of Hanoi (around a dozen lines of code). Prolog was one of the first successful logic programming languages. You make assertions with pure logic, and Prolog determines whether they are true. You can leave gaps in your assertions, and Prolog will try to fill in the holes that would make your incomplete facts true.

---

1. *Rain Man.* DVD. Directed by Barry Levinson. 1988; Los Angeles, CA: MGM, 2000.
2. http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html
3. http://www.lix.polytechnique.fr/~liberti/public/computing/prog/prolog/prolog-tu-torial.html

4.1 About Prolog

Developed in 1972 by Alain Colmerauer and Phillipe Roussel, Prolog is a logic programming language that gained popularity in natural-language processing. Now, the venerable language provides the programming foundation for a wide variety of problems, from scheduling to expert systems. You can use this rules-based language for expressing logic and asking questions. Like SQL, Prolog works on databases, but the data will consist of logical rules and relationships. Like SQL, Prolog has two parts: one to express the data and one to query the data. In Prolog, the data is in the form of logical rules. These are the building blocks:

- *Facts.* A fact is a basic assertion about some world. (Babe is a pig; pigs like mud.)

- *Rules.* A rule is an inference about the facts in that world. (An animal likes mud if it is a pig.)

- *Query.* A query is a question about that world. (Does Babe like mud?)

Facts and rules will go into a *knowledge base*. A Prolog compiler compiles the knowledge base into a form that's efficient for queries. As we walk through these examples, you'll use Prolog to express your knowledge base. Then, you'll do direct retrieval of data and also use Prolog to link rules together to tell you something you might not have known.

Enough background. Let's get started.

4.2 Day 1: An Excellent Driver

In *Rain Man*, Raymond told his brother he was an excellent driver, meaning he could do a fine job of handling the car at five miles per hour in parking lots. He was using all the main elements—the steering wheel, the brakes, the accelerator—he just used them in a limited context. That's your goal today. We're going to use Prolog to state some facts, make some rules, and do some basic queries. Like Io, Prolog is an extremely simple language syntactically. You can learn the syntax rules quickly. The real fun begins when you layer concepts in interesting ways. If this is your first exposure, I guarantee either you will change the way you think or you'll fail. We'll save the in-depth construction for a later day.

First things first. Get a working installation. I'm using GNU Prolog, version 1.3.1, for this book. Be careful. Dialects can vary. I'll do my best to stay on common ground, but if you choose a different version of Prolog, you'll need

to do a little homework to understand where your dialect is different. Regardless of the version you choose, here's how you'll use it.

Basic Facts

In some languages, capitalization is entirely at the programmer's discretion, but in Prolog, the case of the first letter is significant. If a word begins with a lowercase character, it's an *atom*—a fixed value like a Ruby symbol. If it begins with an uppercase letter or an underscore, it's a *variable*. Variable values can change; atoms can't. Let's build a simple knowledge base with a few facts. Key the following into an editor:

```
likes(wallace, cheese).
likes(grommit, cheese).
likes(wendolene, sheep).

friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

The previous file is a knowledge base with facts and rules. The first three statements are facts, and the last statement is a rule. Facts are direct observations of our world. Rules are logical inferences about our world. For now, pay attention to the first three lines. These lines are each facts. wallace, grommit, and wendolene are atoms. You can read them as wallace likes cheese, grommit likes cheese, and wendolene likes sheep. Let's put the facts into action.

Start your Prolog interpreter. If you're using GNU Prolog, type the command gprolog. Then, to load your file, enter the following:

```
| ?- ['friends'].
compiling /Users/batate/prag/Book/code/prolog/friends.pl for byte code...
/Users/batate/prag/Book/code/prolog/friends.pl compiled, 4 lines read -
997 bytes written, 11 ms

yes

| ?-
```

Unless Prolog is waiting on an intermediate result, it will respond with yes or no. In this case, the file loaded successfully, so it returned yes. We can start to ask some questions. The most basic questions are yes and no questions about facts. Ask a few:

```
| ?- likes(wallace, sheep).

no
| ?- likes(grommit, cheese).

yes
```

These questions are pretty intuitive. Does wallace like sheep? (No.) Does grommit like cheese? (Yes.) These are not too interesting: Prolog is just parroting your facts back to you. It starts to get a little more exciting when you start to build in some logic. Let's take a look at inferences.

Basic Inferences and Variables

Let's try the friend rule:

```
| ?- friend(wallace, wallace).
no
```

So, Prolog is working through the rules we gave it and answering yes or no questions. There's more here than meets the eye. Check out the friend rule again:

In English, for X to be a friend of Y, X cannot be the same as Y. Look at the first part to the right of :-, called a *subgoal*. \+ does logical negation, so \+(X = Y) means X is not equal to Y.

Try some more queries:

```
| ?- friend(grommit, wallace).
yes
| ?- friend(wallace, grommit).
yes
```

In English, X is a friend of Y if we can prove that X likes some Z and Y likes that same Z. Both wallace and grommit like cheese, so these queries succeed.

Let's dive into the code. In these queries, X is not equal to Y, proving the first subgoal. The query will use the second and third subgoals, likes(X, Z) and likes(Y, Z). grommit and wallace like cheese, so we prove the second and third subgoals. Try another query:

```
| ?- friend(wendolene, grommit).
```

```
no
```

In this case, Prolog had to try several possible values for X, Y, and Z:

- wendolene, grommit, and cheese
- wendolene, grommit, and sheep

Neither combination satisfied both goals, that wendolene likes Z and grommit likes Z. None existed, so the logic engine reported no, they are not friends.

Let's formalize the terminology. This…

```
friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

…is a Prolog rule with three variables, X, Y, and Z. We call the rule friend/2, shorthand for friend with two parameters. This rule has three subgoals, separated by commas. All must be true for the rule to be true. So, our rule means X is a friend of Y if X and Y are not the same and X and Y like the same Z.

Filling in the Blanks

We've used Prolog to answer some yes or no questions, but we can do more than that. In this section, we'll use the logic engine to find all possible matches for a query. To do this, you will specify a *variable* in your query.

Consider the following knowledge base:

```
Download prolog/food.pl
food_type(velveeta, cheese).
food_type(ritz, cracker).
food_type(spam, meat).
food_type(sausage, meat).
food_type(jolt, soda).
food_type(twinkie, dessert).

flavor(sweet, dessert).
flavor(savory, meat).
flavor(savory, cheese).
flavor(sweet, soda).

food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z).
```

We have a few facts. Some, such as food_type(velveeta, cheese), mean a food has a certain type. Others, such as flavor(sweet, dessert), mean a food type has a characteristic flavor. Finally, we have a rule called food_flavor that infers the flavor of food. A food X has a food_flavor Y if the food is of a food_type Z and that Z also has that characteristic flavor. Compile it:

```
| ?- ['food'].
compiling /Users/batate/prag/Book/code/prolog/food.pl for byte code...
/Users/batate/prag/Book/code/prolog/food.pl compiled,
12 lines read - 1557 bytes written, 15 ms

(1 ms) yes
```

and ask some questions:

```
| ?- food_type(What, meat).

What = spam ? ;

What = sausage ? ;

no
```

Now, that's interesting. We're asking Prolog, "Find some value for What that satisfies the query food_type(What, meat)." Prolog found one, spam. When we typed the ;, we were asking Prolog to find another, and it returned sausage. They were easy values to find since the queries depended on basic facts. Then, we asked for another, and Prolog responded with no. This behavior can be slightly inconsistent. As a convenience, if Prolog can detect that there are no more alternatives remaining, you'll see a yes. If Prolog can't immediately determine whether there are more alternatives without doing more computation, it will prompt you for the next and return no. The feature is really a convenience. If Prolog can give you information sooner, it will. Try a few more:

```
    | ?- food_flavor(sausage, sweet).

    no
| ?- flavor(sweet, What).

What = dessert ? ;

What = soda

yes
```

No, sausage is not sweet. What food types are sweet? dessert and soda. These are all facts. But you can let Prolog connect the dots for you, too:

```
| ?- food_flavor(What, savory).

What = velveeta ? ;

What = spam ? ;

What = sausage ? ;

no
```

Remember, food_flavor(X, Y) is a rule, not a fact. We're asking Prolog to find all possible values that satisfy the query, "What foods have a savory flavor?" Prolog must tie together primitive facts about food, types, and flavors to reach the conclusion. The logic engine has to work through possible combinations that could make all the goals true.

Map Coloring

Let's use the same idea to do map coloring. For a more spectacular look at Prolog, take this example. We want to color a map of the southeastern United States. We'll cover the states shown in Figure 4, *Map of some southeastern states, on page 12*. We do not want two states of the same color to touch.

We code up these simple facts:

```
Download prolog/map.pl
different(red, green). different(red, blue).
different(green, red). different(green, blue).
different(blue, red). different(blue, green).

coloring(Alabama, Mississippi, Georgia, Tennessee, Florida) :-
  different(Mississippi, Tennessee),
  different(Mississippi, Alabama),
  different(Alabama, Tennessee),
  different(Alabama, Mississippi),
  different(Alabama, Georgia),
  different(Alabama, Florida),
  different(Georgia, Florida),
  different(Georgia, Tennessee).
```

We have three colors. We tell Prolog the sets of different colors to use in the map coloring. Next, we have a rule. In the coloring rule, we tell Prolog which states neighbor others, and we're done. Try it:

```
| ?- coloring(Alabama, Mississippi, Georgia, Tennessee, Florida).

Alabama = blue
Florida = green
Georgia = red
Mississippi = red
Tennessee = green ?
```

Sure enough, there is a way to color these five states with three colors. You can get the other possible combinations too by typing a. With a dozen lines of code, we're done. The logic is ridiculously simple—a child could figure it out. At some point, you have to ask yourself...
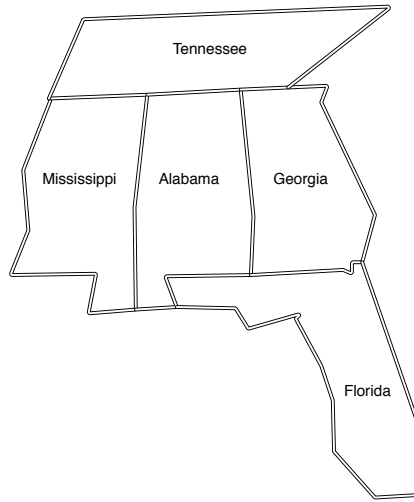
Figure 4—Map of some southeastern states

Where's the Program?

We have no algorithm! Try solving this problem in the procedural language of your choice. Is your solution easy to understand? Think through what you'd have to do to solve complex logic problems like this in Ruby or Io. One possible solution would be as follows:

1. Collect and organize your logic.

2. Express your logic in a program.

3. Find all possible solutions.

4. Put the possible solutions through your program.

And you would have to write this program over and over. Prolog lets you express the logic in facts and inferences and then lets you ask questions. You're not responsible for building any step-by-step recipe with this language. Prolog is not about writing algorithms to solve logical problems. Prolog is about describing your world as it is and presenting logical problems that your computer can try to solve.

Let the computer do the work!

Unification, Part 1

At this point, it's time to back up and provide a little more theory. Let's shine a little more light on unification. Some languages use variable assignment.

In Java or Ruby, for example, x = 10 means assign 10 to the variable x. Unification across two structures tries to make both structures identical. Consider the following knowledge base:

```
cat(lion).
cat(tiger).

dorothy(X, Y, Z) :- X = lion, Y = tiger, Z = bear.
twin_cats(X, Y) :- cat(X), cat(Y).
```

In this example, = means unify, or make both sides the same. We have two facts: lions and tigers are cats. We also have two simple rules. In dorothy/3, X, Y, and Z are lion, tiger, and bear, respectively. In twin_cats/2, X is a cat, and Y is a cat. We can use this knowledge base to shed a little light on unification.

First, let's use the first rule. I'll compile and then do a simple query with no parameters:

```
| ?- dorothy(lion, tiger, bear).

yes
```

Remember, unification means "Find the values that make both sides match." On the right side, Prolog binds X, Y, and Z to lion, tiger, and bear. These match the corresponding values on the left side, so unification is successful. Prolog reports yes. This case is pretty simple, but we can spice it up a little bit. Unification can work on both sides of the implication. Try this one:

```
| ?- dorothy(One, Two, Three).

One = lion
Three = bear
Two = tiger

yes
```

This example has one more layer of indirection. In the goals, Prolog unifies X, Y, and Z to lion, tiger, and bear. On the left side, Prolog unifies X, Y, and Z to One, Two, and Three and then reports the result.

Now, let's shift to the last rule, twin_cats/2. This rule says twin_cats(X, Y) is true if you can prove that X and Y are both cats. Try it:

```
| ?- twin_cats(One, Two).

One = lion
Two = lion ?
```

Prolog reported the first example. lion and lion are both cats. Let's see how it got there:

1. We issued the query twin_cats(One, Two). Prolog binds One to X and Two to Y. To solve these, Prolog must start working through the goals.

2. The first goal is cat(X).

3. We have two facts that match, cat(lion) and cat(tiger). Prolog tries the first fact, binding X to lion, and moves on to the next goal.

4. Prolog now binds Y to cat(Y). Prolog can solve this goal in exactly the same way as the first, choosing lion.

5. We've satisfied both goals, so the rule is successful. Prolog reports the values of One and Two that made it successful and reports yes.

So, we have the first solution that makes the rules true. Sometimes, one solution is enough. Sometimes, you need more than one. We can now step through solutions one by one by using ;, or we can get all of the rest of the solutions by pressing a.

```
Two = lion ? a

One = lion
Two = tiger

One = tiger
Two = lion

One = tiger
Two = tiger

(1 ms) yes
```

Notice that Prolog is working through the list of all combinations of X and Y, given the information available in the goals and corresponding facts. As you'll see later, unification also lets you do some sophisticated matching based on the structure of your data. That's enough for day 1. We're going to do a little more heavy lifting in day 2.

Prolog in Practice

It has to be a little disconcerting to see a "program" presented in this way. In Prolog, there's not often a finely detailed step-by-step recipe, only a description of the cake you'll take out of the pan when you're done. When I was learning Prolog, it helped me tremendously to interview someone who had used the language in practice. I talked to Brian Tarbox who used this logic language to create schedules for working with dolphins for a research project.

An Interview with Brian Tarbox, Dolphin Researcher

Bruce: *Can you talk about your experiences learning Prolog?*

Brian: *I learned Prolog back in the late 1980s when I was in graduate school at the University of Hawaii at Manoa. I was working at the Kewalo Basin Marine Mammal Laboratory doing research into the cognitive capabilities of bottlenosed dolphins. At the time I noticed that much of the discussion at the lab concerned people's theories about how the dolphins thought. We worked primarily with a dolphin named Akeakamai, or Ake for short. Many debates started with "Well, Ake probably sees the situation like this."*

I decided that my master's thesis would be to try to create an executable model that matched our beliefs about Ake's understanding of the world, or at least the tiny subset of it that we were doing research on. If our executable model predicted Ake's actual behavior, we would gain some confidence in our theories about her thinking.

Prolog is a wonderful language, but until you drink the Kool-Aid, it can give you some pretty weird results. I recall one of my first experiments with Prolog, writing something along the lines of $x = x + 1$. Prolog responded "no." Languages don't just say "no." They might give the wrong answer or fail to compile, but I had never had a language talk back to me. So, I called Prolog support and said that the language had said "no" when I tried to change the value of a variable. They asked me, "Why would you want to change the value of a variable?" I mean, what kind of language won't let you change the value of a variable? Once you grok Prolog, you understand that variables either have particular values or are unbound, but it was unsettling at the time.

Bruce: *How have you used Prolog?*

Brian: *I developed two main systems: the dolphin simulator and a laboratory scheduler. The lab would run four experiments a day with each of four dolphins. You have to understand that research dolphins are an incredibly limited resource. Each dolphin was working on different experiments, and each experiment required a different set of personnel. Some roles, such as the actual dolphin trainer, could be filled by only a few people. Other roles such as data recorder could be done by several people but still required training. Most experiments required a staff of six to a dozen people. We had graduate students, undergraduates, and Earthwatch volunteers. Every person had their own schedule and their own shift set of skills. Finding a schedule that utilized everyone and made sure all tasks were done had become a full-time job for one of the staff.*

I decided to try to build a Prolog-based schedule builder. It turned out to be a problem tailor-made for the language. I built a set of facts describing each person's skill set, each person's schedule, and each experiment's requirements. I could then basically tell Prolog "make it so." For each task listed in an experiment, the language would find an available person with that skill and bind them to the task. It would continue until it either satisfied the needs of the experiment or was unable to. If it could not find a valid binding, it would start undoing previous bindings

and trying again with another combination. In the end, it would either find a valid schedule or declare that the experiment was over-constrained.

Bruce: *Are there some interesting examples of facts, rules, or assertions related to dolphins that would make sense to our readers?*

Brian: *There was one particular situation I remember where the simulated dolphin helped us understand Ake's actual behavior. Ake re-sponded to a gestural sign language containing "sentences" such as "hoop through" or "right ball tail-touch." We would give her instructions, and she would respond.*

Part of my research was to try to teach new words such as "not." In this context, "touch not ball" meant touch anything but the ball. This was a hard problem for Ake to solve, but the research was proceeding well for a while. At one point, however, she started simply sinking underwater whenever we gave her the instruction. We didn't understand it all. This can be a very frustrating situation because you can't ask a dolphin why it did something. So, we presented the training task to the simulated dolphin and got an interesting result. Although dolphins are very smart, they will generally try to find the simplest answer to a problem. We had given the simulated dolphin the same heuristic. It turns out that Ake's gestural language included a "word" for one of the windows in the tank. Most trainers had forgotten about this word because it was rarely used. The simulated dolphin discovered the rule that "window" was a successful response to "not ball." It was also a successful response to "not hoop," "not pipe," and "not frisbee." We had guarded against this pattern with the other objects by changing the set of objects in the tank for any given trial, but obviously we could not remove the window. It turns out that when Ake was sinking to the bottom of the tank she was positioned next to the window, though I could not see the window!

Bruce: *What do you like about Prolog the most?*

Brian: *The declarative programming model is very appealing. In general, if you can describe the problem, you have solved the problem. In most languages I've found myself arguing with the computer at some point saying, "You know what I mean; just do it!" C and C++ compiler errors such as "semicolon expected" are symbolic of this. If you expected a semicolon, how about inserting one and seeing whether that fixes it? In Prolog, all I had to do in the scheduling problem was basically say, "I want a day that looks like this, so go make me one" and it would do it.*

Bruce: *What gave you the most trouble?*

Brian: *Prolog seemed to be an all-or-nothing approach to problems, or at least to the problems I was working on. In the laboratory scheduling problem, the system would churn for 30 minutes and then either give us a beautiful schedule for the day or simply print "no." "No" in this case meant that we had over-constrained the day, and there was no full solution. It did not, however, give us a partial solution or much of any information about where the over-constraint was.*

What you see here is an extremely powerful concept. You don't have to describe the solution to a problem. You have only to describe the problem. And the language for the description of the problem is logic, only pure logic. Start from facts and inferences, and let Prolog do the rest. Prolog programs are at a higher level of abstraction. Schedules and behavior patterns are great examples of problems right in Prolog's wheelhouse.

What We Learned in Day 1

Today, we learned the basic building blocks of the Prolog language. Rather than encoding steps to guide Prolog to a solution, we encoded knowledge using pure logic. Prolog did the hard work of weaving that knowledge together to find solutions. We put our logic into knowledge bases and issued queries against them.

After we built a few knowledge bases, we then compiled and queried them. The queries had two forms. First, the query could specify a fact, and Prolog would tell us whether the facts were true or false. Second, we built a query with one or more variables. Prolog then computed all possibilities that made those facts true.

We learned that Prolog worked through rules by going through the clauses for a rule in order. For any clause, Prolog tried to satisfy each of the goals by going through the possible combinations of variables. All Prolog programs work this way.

In the sections to come, we're going to make more complex inferences. We're also going to learn to use math and more complex data structures such as lists, as well as strategies to iterate over lists.

Day 1 Self-Study

Find:

- Some free Prolog tutorials

- A support forum (there are several)

- One online reference for the Prolog version you're using

Do:

- Make a simple knowledge base. Represent some of your favorite books and authors.

- Find all books in your knowledge base written by one author.

- Make a knowledge base representing musicians and instruments. Also represent musicians and their genre of music.

- Find all musicians who play the guitar.