Sally Dibbs, Dibbs Sally. 461-0192.

➤ *Raymond*

CHAPTER 4

Prolog

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

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-tutorial.html

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.

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:

Download prolog/friends.pl

```
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 102. 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...

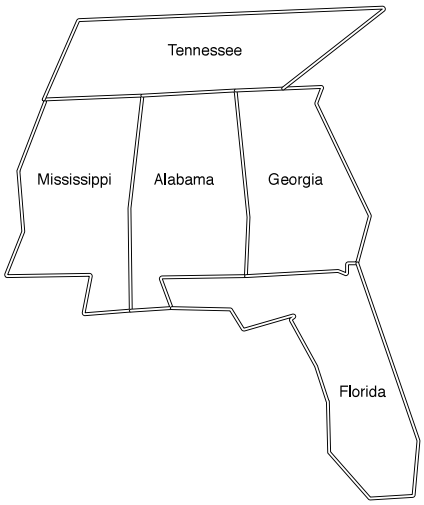

Figure 4—Map of some southeastern states

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:

Download prolog/ohmy.pl

```
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.

4.3 Day 2: Fifteen Minutes to Wapner

Grumpy Judge Wapner from *The People's Court* is an obsession of the central character in *Rain Man*. Like most autistics, Raymond obsesses over all things familiar. He latched on to Judge Wapner and *The People's Court*. As you're plowing through this enigmatic language, you might be ready for things to start to click. Now, you might be one of the lucky readers who has everything click for them right away, but if you don't, take heart. Today, there are definitely "fifteen minutes to Wapner." Sit tight. We will need a few more tools in the toolbox. You'll learn to use recursion, math, and lists. Let's get going.

Recursion

Ruby and Io were imperative programming languages. You would spell out each step of an algorithm. Prolog is the first of the declarative languages we'll look at. When you're dealing with collections of things such as lists or trees, you'll often use recursion rather than iteration. We'll look at recursion and use it to solve some problems with basic inferences, and then we'll apply the same technique to lists and math.

Take a look at the following database. It expresses the extensive family tree of the Waltons, characters in a 1963 movie and subsequent series. It expresses a father relationship and from that infers the ancestor relationship. Since an ancestor can mean a father, grandfather, or great grandfather, we will need to nest the rules or iterate. Since we're dealing with a declarative language, we're going to nest. One clause in the `ancestor` clause will use `ancestor`. In this case, `ancestor(Z, Y)` is a recursive subgoal. Here's the knowledge base:

Download prolog/family.pl

```
father(zeb,          john_boy_sr).
father(john_boy_sr, john_boy_jr).

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

`father` is the core set of facts that enables our recursive subgoal. The rule `ancestor/2` has two clauses. When you have multiple clauses that make up a rule, only one of them must be true for the rule to be true. Think of the commas between subgoals as `and` conditions and the periods between

clauses as or conditions. The first clause says "X is the ancestor of Y if X is the father of Y." That's a straightforward relationship. We can try that rule like this:

```
| ?- ancestor(john_boy_sr, john_boy_jr).

true ?

no
```

Prolog reports true, john_boy_sr is an ancestor of john_boy_jr. This first clause depends on a fact.

The second clause is more complex: ancestor(X, Y) :- father(X, Z), ancestor(Z, Y). This clause says X is an ancestor of Y if we can prove that X is the father of Z and we can also prove that same Z is an ancestor of Y.

Whew. Let's use the second clause:

```
| ?- ancestor(zeb, john_boy_jr).

true ?
```

Yes, zeb is an ancestor of john_boy_jr. As always, we can try variables in a query, like this:

```
| ?- ancestor(zeb, Who).

Who = john_boy_sr ? a

Who = john_boy_jr

no
```

And we see that zeb is an ancestor for john_boy_jr and john_boy_sr. The ancestor predicate also works in reverse:

```
| ?- ancestor(Who, john_boy_jr).

Who = john_boy_sr ? a

Who = zeb

(1 ms) no
```

That's a beautiful thing, because we can use this rule in our knowledge base for two purposes, to find both ancestors and descendants.

A brief warning. When you use recursive subgoals, you need to be careful because each recursive subgoal will use stack space, and you can eventually run out. Declarative languages often solve this problem with a technique called *tail recursion optimization*. If you can position the recursive subgoal at the end of a recursive rule, Prolog can optimize the call to discard the call stack, keeping the memory use constant. Our call is tail recursive

because the recursive subgoal, `ancestor(Z, Y)`, is the last goal in the recursive rule. When your Prolog programs crash by running out of stack space, you'll know it's time to look for a way to optimize with tail recursion.

With that last bit of housekeeping out of the way, let's start to look at lists and tuples.

Lists and Tuples

Lists and tuples are a big part of Prolog. You can specify a list as `[1, 2, 3]` and a tuple as `(1, 2, 3)`. Lists are containers of variable length, and tuples are containers with a fixed length. Both lists and tuples get much more powerful when you think of them in terms of unification.

Unification, Part 2

Remember, when Prolog tries to unify variables, it tries to make both the left and right sides match. Two tuples can match if they have the same number of elements and each element unifies. Let's take a look at a couple of examples:

```
| ?- (1, 2, 3) = (1, 2, 3).

yes
| ?- (1, 2, 3) = (1, 2, 3, 4).

no
| ?- (1, 2, 3) = (3, 2, 1).

no
```

Two tuples unify if all the elements unify. The first tuples were exact matches, the second tuples did not have the same number of elements, and the third set did not have the same elements in the same order. Let's mix in some variables:

```
| ?- (A, B, C) = (1, 2, 3).

A = 1
B = 2
C = 3

yes
| ?- (1, 2, 3) = (A, B, C).

A = 1
B = 2
C = 3

yes
| ?- (A, 2, C) = (1, B, 3).
```

```
A = 1
B = 2
C = 3

yes
```

It doesn't really matter which sides the variables are on. They unify if Prolog can make them the same. Now, for some lists. They can work like tuples:

```
| ?- [1, 2, 3] = [1, 2, 3].

yes
| ?- [1, 2, 3] = [X, Y, Z].

X = 1
Y = 2
Z = 3

yes
| ?- [2, 2, 3] = [X, X, Z].

X = 2
Z = 3

yes
| ?- [1, 2, 3] = [X, X, Z].

no
| ?- [] = [].
```

The last two examples are interesting. `[X, X, Z]` and `[2, 2, 3]` unified because Prolog could make them the same with `X = 2`. `[1, 2, 3] = [X, X, Z]` did not unify because we used `X` for both the first and second positions, and those values were different. Lists have a capability that tuples don't. You can deconstruct lists with `[Head|Tail]`. When you unify a list with this construct, `Head` will bind to the first element of the list, and `Tail` will bind to the rest, like this:

```
| ?- [a, b, c] = [Head|Tail].

Head = a
Tail = [b,c]

yes
```

`[Head|Tail]` won't unify with an empty list, but a one-element list is fine:

```
| ?- [] = [Head|Tail].

no
| ?- [a] = [Head|Tail].

Head = a
Tail = []

yes
```

You can get complicated by using various combinations:

```
| ?- [a, b, c] = [a|Tail].

Tail = [b,c]

(1 ms) yes
```

Prolog matched the a and unified the rest with Tail. Or we can split this tail into the head and tail:

```
| ?- [a, b, c] = [a|[Head|Tail]].

Head = b
Tail = [c]

yes
```

Or grab the third element:

```
| ?- [a, b, c, d, e] = [_, _|[Head|_]].

Head = c

yes
```

_ is a wildcard and unifies with anything. It basically means "I don't care what's in this position." We told Prolog to skip the first two elements and split the rest into head and tail. The Head will grab the third element, and the trailing _ will grab the tail, ignoring the rest of the list.

That should be enough to get you started. Unification is a powerful tool, and using it in conjunction with lists and tuples is even more powerful.

Now, you should have a basic understanding of the core data structures in Prolog and how unification works. We're now ready to combine these elements with rules and assertions to do some basic math with logic.

Lists and Math

In our next example, I thought I'd show you an example of using recursion and math to operate on lists. These are examples to do counting, sums, and averages. Five rules do all the hard work.

Download prolog/list_math.pl

```
count(0, []).
count(Count, [Head|Tail]) :- count(TailCount, Tail), Count is TailCount + 1.

sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.

average(Average, List) :- sum(Sum, List), count(Count, List), Average is Sum/Count.
```

The simplest example is count. Use it like this:

```
| ?- count(What, [1]).

What = 1 ? ;

no
```

The rules are trivially simple. The count of an empty list is 0. The count of a list is the count of the tail plus one. Let's talk about how this works, step by step:

- We issue the query count(What, [1]), which can't unify with the first rule, because the list is not empty. We move on to satisfying the goals for the second rule, count(Count, [Head|Tail]). We unify, binding What to Count, Head to 1, and Tail to [].
- After unification, the first goal is count(TailCount, []). We try to prove that subgoal. This time, we unify with the first rule. That binds TailCount to 0. The first rule is now satisfied, so we can move on to the second goal.
- Now, we evaluate Count is TailCount + 1. We can unify variables. TailCount is bound to 0, so we bind Count to 0 + 1, or 1.

And that's it. We did not define a recursive process. We defined logical rules. The next example is adding up the elements of a list. Here's the code for those rules again:

```
sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.
```

This code works precisely like the count rule. It also has two clauses, a base case and the recursive case. The usage is similar:

```
| ?- sum(What, [1, 2, 3]).

What = 6 ? ;

no
```

If you look at it imperatively, sum works exactly as you would expect in a recursive language. The sum of an empty list is zero; the sum of the rest is the Head plus the sum of the Tail.

But there's another interpretation here. We haven't really told Prolog how to compute sums. We've merely described sums as rules and goals. To satisfy some of the goals, the logic engine must satisfy some subgoals. The declarative interpretation is as follows: "The sum of an empty list is zero, and the sum of a list is Total if we can prove that the sum of the tail plus the

head is Total." We're replacing recursion with the notion of proving goals and subgoals.

Similarly, the count of an empty list is zero; the count of a list is one for the Head plus the count of the Tail.

As with logic, these rules can build on each other. For example, you can use sum and count together to compute an average:

```
average(Average, List) :- sum(Sum, List), count(Count, List), Average is Sum/Count.
```

So, the average of List is Average if you can prove that

- the sum of that List is Sum,
- the count of that List is Count, and
- Average is Sum/Count.

And it works just as you'd expect:

```
| ?- average(What, [1, 2, 3]).

What = 2.0 ? ;

no
```

Using Rules in Both Directions

At this point, you should have a fairly good understanding of how recursion works. I'm going to shift gears a little bit and talk about a tight little rule called append. The rule append(List1, List2, List3) is true if List3 is List1 + List2. It's a powerful rule that you can use in a variety of ways.

That short little bit of code packs a punch. You can use it in many different ways. It's a lie detector:

```
| ?- append([oil], [water], [oil, water]).

yes
| ?- append([oil], [water], [oil, slick]).

no
```

It's a list builder:

```
| ?- append([tiny], [bubbles], What).

What = [tiny,bubbles]

yes
```

It does list subtraction:

```
| ?- append([dessert_topping], Who, [dessert_topping, floor_wax]).

Who = [floor_wax]

yes
```

And it computes possible splits:

```
| ?- append(One, Two, [apples, oranges, bananas]).

One = []
Two = [apples,oranges,bananas] ? a

One = [apples]
Two = [oranges,bananas]

One = [apples,oranges]
Two = [bananas]

One = [apples,oranges,bananas]
Two = []

(1 ms) no
```

So, one rule gives you four. You may think that building such a rule will take a lot of code. Let's find out exactly how much. Let's rewrite the Prolog append, but we'll call it concatenate. We'll take it in several steps:

1. Write a rule called concatenate(List1, List2, List3) that can concatenate an empty list to List1.
2. Add a rule that concatenates one item from List1 onto List2.
3. Add a rule that concatenates two and three items from List1 onto List2.
4. See what we can generalize.

Let's get started. Our first step is to concatenate an empty list to List1. That's a fairly easy rule to write:

Download prolog/concat_step_1.pl

```
concatenate([], List, List).
```

No problem. concatenate is true if the first parameter is a list and the next two parameters are the same.

It works:

```
| ?- concatenate([], [harry], What).

What = [harry]

yes
```

Onto the next step. Let's add a rule that concatenates the first element of List1 to the front of List2:

Download prolog/concat_step_2.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
```

For concatenate(List1, List2, List3), we break List1 into the head and tail, with the tail being an empty list. We'll break our third element into the head and tail, using List1's head and List2 as the tail. Remember to compile your knowledge base. It works just fine:

```
| ?- concatenate([malfoy], [potter], What).

What = [malfoy,potter]

yes
```

Now, we can define another couple of rules to concatenate lists of lengths 2 and 3. They work in the same way:

Download prolog/concat_step_3.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
concatenate([Head1|[Head2|[]]], List, [Head1, Head2|List]).
concatenate([Head1|[Head2|[Head3|[]]]], List, [Head1, Head2, Head3|List]).

| ?- concatenate([malfoy, granger], [potter], What).

What = [malfoy,granger,potter]

yes
```

So, what we have is a base case and a strategy where each subgoal shrinks the first list and grows the third. The second stays constant. We now have enough information to generalize a result. Here's the concatenate using nested rules:

Download prolog/concat.pl

```
concatenate([], List, List).
concatenate([Head|Tail1], List, [Head|Tail2]) :-
  concatenate(Tail1, List, Tail2).
```

That terse little block of code has an incredibly simple explanation. The first clause says concatenating an empty list to List gives you that List. The second clause says concatenating List1 to List2 gives you List3 if the heads of List1 and List3 are the same, and you can prove that concatenating the tail of List1 with List2 gives you the tail of List3. The simplicity and elegance of this solution are a testament to the power of Prolog.

Let's see what it would do with the query concatenate([1, 2], [3], What). We'll walk through unification at each step. Keep in mind that we're nesting the rules, so each time we try to prove a subgoal, we'll have a different copy of the variables. I'll mark the important ones with a letter so you can keep them

straight. With each pass, I'll show what happens when Prolog tries to prove the next subgoal.

- Start with this:

  ```
  concatenate([1, 2], [3], What)
  ```

- The first rule doesn't apply, because [1, 2] is not an empty list. We unify to this:

  ```
  concatenate([1|[2]], [3], [1|Tail2-A]) :- concatenate([2], [3], [Tail2-A])
  ```

  Everything unifies but the second tail. We now move on to the goals. Let's unify the right side.

- We try to apply the rule `concatenate([2], [3], [Tail2-A])`. That's going to give us this:

  ```
  concatenate([2|[]], [3], [2|Tail2-B]) :- concatenate([], [3], Tail2-B)
  ```

  Notice that `Tail2-B` is the tail of `Tail2-A`. It's not the same as the original `Tail2`. But now, we have to unify the right side again.

- `concatenate([], [3], Tail2-C) :- concatenate([], [3], [3]) .`

- So, we know `Tail2-C` is [3]. Now, we can work back up the chain. Let's look at the third parameter, plugging in `Tail2` at each step. `Tail2-C` is [3], which means `[2|Tail2-2]` is [2, 3], and finally `[1|Tail2]` is [1, 2, 3]. `What` is [1, 2, 3].

Prolog is doing a lot of work for you here. Go over this list until you understand it. Unifying nested subgoals is a core concept for the advanced problems in this book.

Now, you've taken a deep look at one of the richest functions in Prolog. Take a little time to explore these solutions, and make sure you understand them.

What We Learned in Day 2

In this section, we moved into the basic building blocks that Prolog uses to organize data: lists and tuples. We also nested rules, allowing us to express problems that you might handle with iteration in other languages. We took a deeper look at Prolog unification and how Prolog works to match up both sides of a :- or =. We saw that when we're writing rules, we described logical rules instead of algorithms and let Prolog work its way through the solution.

We also used math. We learned to use basic arithmetic and nested subgoals to compute sums and averages.

Finally, we learned to use lists. We matched one or more variables within a list to variables, but more importantly, we matched the head of a list and the remaining elements with variables using the [Head|Tail] pattern. We used this technique to recursively iterate through lists. These building blocks will serve as the foundations of the complex problems we solve in day 3.

Day 2 Self-Study

Find:

- Some implementations of a Fibonacci series and factorials. How do they work?
- A real-world community using Prolog. What problems are they solving with it today?

If you're looking for something more advanced to sink your teeth into, try these problems:

- An implementation of the Towers of Hanoi. How does it work?
- What are some of the problems of dealing with "not" expressions? Why do you have to be careful with negation in Prolog?

Do:

- Reverse the elements of a list.
- Find the smallest element of a list.
- Sort the elements of a list.

4.4 Day 3: Blowing Up Vegas

You should be getting a better understanding of why I picked the Rain Man, the autistic savant, for Prolog. Though it's sometimes difficult to understand, it's amazing to think of programming in this way. One of my favorite points in *Rain Man* was when Ray's brother realized he could count cards. Raymond and his brother went to Vegas and just about broke the bank. In this section, you're going to see a side of Prolog that will leave you smiling. Coding the examples in this chapter was equal parts maddening and exhilarating. We're going to solve two famous puzzles that are right in Prolog's comfort zone, solving systems with constraints.

You may want to take a shot at some of these puzzles yourself. If you do, try describing the rules you know about each game rather than showing Prolog a step-by-step solution. We're going to start with a small Sudoku and then give you a chance to build up to a larger one in the daily exercises. Then, we'll move on to the classic Eight Queens puzzle.

Solving Sudoku

Coding the Sudoku was almost magical for me. A Sudoku is a grid that has rows, columns, and boxes. A typical puzzle is a nine-by-nine grid, with some spaces filled in and some blank. Each cell in the grid has a number, from 1–9 for a nine-by-nine square. Your job is to fill out the solution so that each row, column, and square has one each of all of the digits.

We're going to start with a four-by-four Sudoku. The concepts are exactly the same, though the solutions will be shorter. Let's start by describing the world, as we know it. Abstractly, we'll have a board with four rows, four columns, and four squares. The table shows squares 1–4:

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

The first task is to decide what the query will look like. That's simple enough. We'll have a puzzle and a solution, of the form sudoku(Puzzle, Solution). Our users can provide a puzzle as a list, substituting underscores for unknown numbers, like this:

```
sudoku([_, _, 2, 3,
        _, _, _, _,
        _, _, _, _,
        3, 4, _, _],
        Solution).
```

If a solution exists, Prolog will provide the solution. When I solved this puzzle in Ruby, I had to worry about the algorithm for solving the puzzle. With Prolog, that's not so. I merely need to provide the rules for the game. These are the rules:

- For a solved puzzle, the numbers in the puzzle and solution should be the same.
- A Sudoku board is a grid of sixteen cells, with values from 1–4.
- The board has four rows, four columns, and four squares.
- A puzzle is valid if the elements in each row, column, and square has no repeated elements.

Let's start at the top. The numbers in the solution and puzzle should match:

Download prolog/sudoku4_step_1.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle.
```

We've actually made some progress. Our "Sudoku solver" works for the case where there are no blanks:

```
| ?- sudoku([4, 1, 2, 3,
             2, 3, 4, 1,
             1, 2, 3, 4,
             3, 4, 1, 2], Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

The format isn't pretty, but the intent is clear enough. We're getting sixteen numbers back, row by row. But we are a little too greedy:

```
| ?- sudoku([1, 2, 3], Solution).

Solution = [1,2,3]

yes
```

Now, this board isn't valid, but our solver reports that there is a valid solution. Clearly, we have to limit the board to sixteen elements. We have another problem, too. The values in the cells can be anything:

```
| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

Solution = [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6]

yes
```

For a solution to be valid, it should have numbers from 1–4. This problem will impact us in two ways. First, we may allow some invalid solutions. Second, Prolog doesn't have enough information to test possible values for each cell. In other words, the set of results is not *grounded*. That means that we have not expressed rules that limit possible values of each cell, so Prolog will not be able to guess what the values are.

Let's solve these problems by solving the next rule to the game. Rule 2 says a board has sixteen cells, with values from 1–4. GNU Prolog has a built-in predicate to express possible values, called fd_domain(List, LowerBound, UpperBound). This predicate is true if all the values in List are between LowerBound and UpperBound, inclusive. We just need to make sure all values in Puzzle range from 1 to 4.

Download prolog/sudoku4_step_2.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle,
        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],
        fd_domain(Puzzle, 1, 4).
```

We unified Puzzle with a list of sixteen variables, and we limited the domain of the cells to values from 1–4. Now, we fail if the puzzle is not valid:

```
| ?- sudoku([1, 2, 3], Solution).

no


| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

no
```

Now, we get to the main piece of the solution. Rule 3 says a board consists of rows, columns, and squares. We're going to carve the puzzle up into rows, columns, and squares. Now, you can see why we named the cells the way we did. It's a straightforward process to describe the rows:

```
Row1 = [S11, S12, S13, S14],
Row2 = [S21, S22, S23, S24],
Row3 = [S31, S32, S33, S34],
Row4 = [S41, S42, S43, S44],
```

Likewise for columns:

```
Col1 = [S11, S21, S31, S41],
Col2 = [S12, S22, S32, S42],
Col3 = [S13, S23, S33, S43],
Col4 = [S14, S24, S34, S44],
```

And squares:

```
Square1 = [S11, S12, S21, S22],
Square2 = [S13, S14, S23, S24],
Square3 = [S31, S32, S41, S42],
Square4 = [S33, S34, S43, S44].
```

Now that we've chopped the board into pieces, we can move on to the next rule. The board is valid if all rows, columns, and squares have no repeated elements. We'll use a GNU Prolog predicate to test for repeated elements. fd_all_different(List) succeeds if all the elements in List are different. We need to

build a rule to test that all rows, columns, and squares are valid. We'll use a simple rule to accomplish this:

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).
```

This predicate is valid if all the lists in it are different. The first clause says that an empty list is valid. The second clause says that a list is valid if the first element's items are all different and if the rest of the list is valid.

All that remains is to invoke our valid(List) rule:

```
valid([Row1, Row2, Row3, Row4,
       Col1, Col2, Col3, Col4,
       Square1, Square2, Square3, Square4]).
```

Believe it or not, we're done. This solution can solve a four-by-four Sudoku:

```
| ?- sudoku([_, _, 2, 3,
             _, _, _, _,
             _, _, _, _,
             3, 4, _, _],
             Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

Breaking that into a friendlier form, we have the solution:

4	1	2	3
2	3	4	1
1	2	3	4
3	4	1	2

Here's the completed program, beginning to end:

Download prolog/sudoku4.pl

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).

sudoku(Puzzle, Solution) :-
        Solution = Puzzle,
        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],
```

```
fd_domain(Solution, 1, 4),

Row1 = [S11, S12, S13, S14],
Row2 = [S21, S22, S23, S24],
Row3 = [S31, S32, S33, S34],
Row4 = [S41, S42, S43, S44],

Col1 = [S11, S21, S31, S41],
Col2 = [S12, S22, S32, S42],
Col3 = [S13, S23, S33, S43],
Col4 = [S14, S24, S34, S44],

Square1 = [S11, S12, S21, S22],
Square2 = [S13, S14, S23, S24],
Square3 = [S31, S32, S41, S42],
Square4 = [S33, S34, S43, S44],

valid([Row1, Row2, Row3, Row4,
       Col1, Col2, Col3, Col4,
       Square1, Square2, Square3, Square4]).
```

If you haven't had your Prolog moment, this example should give you a nudge in the right direction. Where's the program? Well, we didn't write a program. We described the rules of the game: a board has sixteen cells with digits from 1–4, and none of the rows, columns, or squares should repeat any of the values. The puzzle took a few dozen lines of code to solve and no knowledge of any Sudoku solving strategies. In the daily exercises, you'll get the chance to solve a nine-row Sudoku. It won't be too difficult.

This puzzle is a great example of the types of problems Prolog solves well. We have a set of constraints that are easy to express but hard to solve. Let's look at another puzzle involving highly constrained resources: the Eight Queens problem.

Eight Queens

To solve the Eight Queens problem, you put eight queens on a chess board. None can share the same row, column, or diagonal. It may appear to be a trivial problem on the surface. It's just a kid's game. But on another level, you can look at the rows, columns, and diagonals as constrained resources. Our industry is full of problems that solve constrained systems. Let's look at how we can solve this one in Prolog.

First, we'll look at what the query should look like. We can express each queen as `(Row, Col)`, a tuple having a row and a column. A `Board` is a list of

tuples. eight_queens(Board) succeeds if we have a valid board. Our query will look like this:

```
eight_queens([(1, 1), (3, 2), ...]).
```

Let's look at the goals we need to satisfy to solve the puzzle. If you want to take a shot at this game without looking at the solution, just look at these goals. I won't show the full solution until later in the chapter.

- A board has eight queens.
- Each queen has a row from 1–8 and a column from 1–8.
- No two queens can share the same row.
- No two queens can share the same column.
- No two queens can share the same diagonal (southwest to northeast).
- No two queens can share the same diagonal (northwest to southeast).

Rows and columns must be unique, but we must be more careful with diagonals. Each queen is on two diagonals, one running from the lower left (northwest) to the upper right (southeast) and the other running from the upper left to the lower right as in Figure 5, *Eight Queens rules*, on page 125. But these rules should be relatively easy to encode.

Once again, we'll start at the top of the list. A board has eight queens. That means our list must have a size of eight. That's easy enough to do. We can use the count predicate you saw earlier in the book, or we can simply use a built-in Prolog predicate called length. length(List, N) succeeds if List has N elements. This time, rather than show you each goal in action, I'm going to walk you through the goals we'll need to solve the whole problem. Here's the first goal, then:

```
eight_queens(List) :- length(List, 8).
```

Next, we need to make sure each queen from our list is valid. We build a rule to test whether a queen is valid:

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).
```

The predicate member does just what you think; it tests for membership. A queen is valid if both the row and column are integers from 1–8. Next, we'll build a rule to check whether the whole board is made up of valid queens:

```
valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).
```

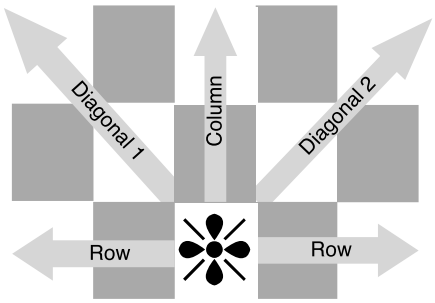

Figure 5—Eight Queens rules

An empty board is valid, and a board is valid if the first item is a valid queen and the rest of the board is valid.

Moving on, the next rule is that two queens can't share the same row. To solve the next few constraints, we're going to need a little help. We will break down the program into pieces that can help us describe the problem: what are the rows, columns, and diagonals? First up is rows. We'll build a function called `rows(Queens, Rows)`. This function should be true if `Rows` is the list of `Row` elements from all the queens.

```
rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).
```

This one takes a little imagination, but not much. `rows` for an empty list is an empty list, and `rows(Queens, Rows)` is `Rows` if the `Row` from the first queen in the list matches the first element of `Rows` and if `rows` of the tail of `Queens` is the tail of `Rows`. If it's confusing to you, walk through it with a few test lists. Luckily, columns works exactly the same way, but we're going to use columns instead of rows:

```
cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).
```

The logic works exactly the same as `rows`, but we match the second element of a queen tuple instead of the first.

Moving on, we're going to number diagonals. The easiest way to number them is to do some simple subtraction and addition. If north and west are 1, we're going to assign the diagonals that run from northwest to southeast a value of `Col – Row`. This is the predicate that grabs those diagonals:

```
diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).
```

That rule worked just like rows and cols, but we had one more constraint: `Diagonal is Col -- Row`. Note that this is not unification! It's an `is` predicate, and it will make sure that the solution is fully grounded. Finally, we'll grab the southeast to northwest like this:

```
diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).
```

The formula is a little bit tricky, but try a few values until you're satisfied that queens with the same sum of row and col are in fact on the same diagonal. Now that we have the rules to help us describe rows, columns, and diagonals, all that remains is to make sure rows, columns, and diagonals are all different.

So you can see it all in context, here's the entire solution. The tests for rows and columns are the last eight clauses.

Download prolog/queens.pl

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).
```

```
eight_queens(Board) :-
  length(Board, 8),
  valid_board(Board),

  rows(Board, Rows),
  cols(Board, Cols),
  diags1(Board, Diags1),
  diags2(Board, Diags2),

  fd_all_different(Rows),
  fd_all_different(Cols),
  fd_all_different(Diags1),
  fd_all_different(Diags2).
```

At this point, you could run the program, and it would run… and run… and run. There are just too many combinations to efficiently sort through. If you think about it, though, we know that there will be one and only queen in every row. We can jump start the solution by providing a board that looks like this:

```
| ?- eight_queens([(1, A), (2, B), (3, C), (4, D), (5, E), (6, F), (7, G), (8, H)]).

A = 1
B = 5
C = 8
D = 6
E = 3
F = 7
G = 2
H = 4 ?
```

That works just fine, but the program is still working too hard. We can eliminate the row choices quite easily and simplify the API while we're at it. Here's a slightly optimized version:

Download prolog/optimized_queens.pl

```
valid_queen((Row, Col)) :- member(Col, [1,2,3,4,5,6,7,8]).
valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).
```

```
diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
  Board = [(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)],
  valid_board(Board),

  cols(Board, Cols),
  diags1(Board, Diags1),
  diags2(Board, Diags2),
  fd_all_different(Cols),
  fd_all_different(Diags1),
  fd_all_different(Diags2).
```

Philosophically, we've made one major change. We matched the `Board` with `(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)` to reduce the total permutations significantly. We also removed all rules related to rows, and the results show. On my ancient MacBook, all solutions compute inside of three minutes.

Once again, the end result is quite pleasing. We built in very little knowledge of the solution set. We just described the rules to the game and applied a little logic to speed things up a little. Given the right problems, I could really find myself getting into Prolog.

What We Learned in Day 3

Today, you put together some of the ideas we've used in Prolog to solve some classic puzzles. The constraint-based problems have many of the same characteristics as classic industrial applications. List constraints, and crunch out a solution. We would never think of doing a SQL nine-table join imperatively, yet we don't even blink at solving logical problems in this way.

We started with a Sudoku puzzle. Prolog's solution was remarkably simple. We mapped sixteen variables onto rows, columns, and squares. Then, we described the rules of the game, forcing each row, column, and square to be unique. Prolog then methodically worked through the possibilities, quickly arriving at a solution. We used wildcards and variables to build an intuitive API, but we didn't provide any help at all for solution techniques.

Next, we used Prolog to solve the Eight Queens puzzle. Once again, we encoded the rules of the game and let Prolog work into a solution. This classic problem was computationally intensive, having 92 possible solutions, but even our simple approach could solve it within a handful of minutes.

I still don't know all of the tricks and techniques to solve advanced Sudokus, but with Prolog, I don't need to know them. I only need the rules of the game to play.

Day 3 Self-Study

Find:

- Prolog has some input/output features as well. Find print predicates that print out variables.
- Find a way to use the print predicates to print only successful solutions. How do they work?

Do:

- Modify the Sudoku solver to work on six-by-six puzzles (squares are 3x2) and 9x9 puzzles.
- Make the Sudoku solver print prettier solutions.

If you're a puzzle enthusiast, you can get lost in Prolog. If you want to dive deeper into the puzzles I've presented, Eight Queens is a good place to start.

- Solve the Eight Queens problem by taking a list of queens. Rather than a tuple, represent each queen with an integer, from 1–8. Get the row of a queen by its position in the list and the column by the value in the list.

4.5 Wrapping Up Prolog

Prolog is one of the older languages in this book, but the ideas are still interesting and relevant today. Prolog means programming with logic. We used Prolog to process rules, composed of clauses, which were in turn composed with a series of goals.

Prolog programming has two major steps. Start by building a knowledge base, composed of logical facts and inferences about the problem domain. Next, compile your knowledge base, and ask questions about the domain. Some of the questions can be assertions, and Prolog will respond with `yes` or `no`. Other queries have variables. Prolog fills in these gaps that makes those queries true.

Rather than simple assignment, Prolog uses a process called *unification* that makes variables on both sides of a system match. Sometimes, Prolog has to try many different possible combinations of variables to unify variables for an inference.

Strengths

Prolog is applicable for a wide variety of problems, from airline scheduling to financial derivatives. Prolog has a serious learning curve, but the demanding problems that Prolog solves tend to make the language, or others like it, worthwhile.

Think back to Brian Tarbox's work with the dolphins. He was able to make simple inferences about the world and make a breakthrough with a complex inference about dolphin behavior. He was also able to take highly constrained resources and use Prolog to find schedules that fit among them. These are some areas where Prolog is in active use today:

Natural-Language Processing

Prolog was perhaps first used to work with language recognition. In particular, Prolog language models can take natural language, apply a knowledge base of facts and inferences, and express that complex, inexact language in concrete rules appropriate for computers.

Games

Games are getting more complex, especially modeling the behavior of competitors or enemies. Prolog models can easily express the behavior of other characters in the system. Prolog can also build different behaviors into different types of enemies, making a more lifelike and enjoyable experience.

Semantic Web

The semantic Web is an attempt to attach meaning to the services and information on the Web, making it easier to satisfy requests. The resource description language (RDF) provides a basic description of resources. A server can compile these resources into a knowledge base. That knowledge, together with Prolog's natural-language processing, can provide a rich end user experience. Many Prolog packages exist for providing this sort of functionality in the context of a web server.

Artificial Intelligence

Artificial intelligence (AI) centers around building intelligence into machines. This intelligence can take different forms, but in every case, some "agent" modifies behavior based on complex rules. Prolog excels in this arena, especially when the rules are concrete, based on formal logic. For this reason, Prolog is sometimes called a *logic programming language*.

Scheduling

Prolog excels in working with constrained resources. Many have used Prolog to build operating system schedulers and other advanced schedulers.

Weaknesses

Prolog is a language that has held up over time. Still, the language is dated in many ways, and it does have significant limitations.

Utility

While Prolog excels in its core domain, it's a fairly focused niche, logic programming. It is not a general-purpose language. It also has some limitations related to language design.

Very Large Data Sets

Prolog uses a depth-first search of a decision tree, using all possible combinations matched against the set of rules. Various languages and compilers do a pretty good job of optimizing this process. Still, the strategy is inherently computationally expensive, especially as data sets get very large. It also forces Prolog users to understand how the language works to keep the size of data sets manageable.

Mixing the Imperative and Declarative Models

Like many languages in the functional family, particularly those that rely heavily on recursion, you must understand how Prolog will resolve recursive rules. You must often have tail-recursive rules to complete even moderately large problems. It's relatively easy to build Prolog applications that cannot scale beyond a trivial set of data. You must often have a deep understanding of how Prolog works to effectively design rules that will scale at acceptable levels.

Final Thoughts

As I worked through the languages in this book, I often kicked myself, knowing that over the years, I've driven many screws with a sledgehammer. Prolog was a particularly poignant example of my evolving understanding. If you find a problem that's especially well suited for Prolog, take advantage. In such a setting, you can best use this rules-based language in combination with other general-purpose languages, just as you would use SQL within Ruby or Java. If you're careful with the way you tie them together, you're likely to come out ahead in the long run.