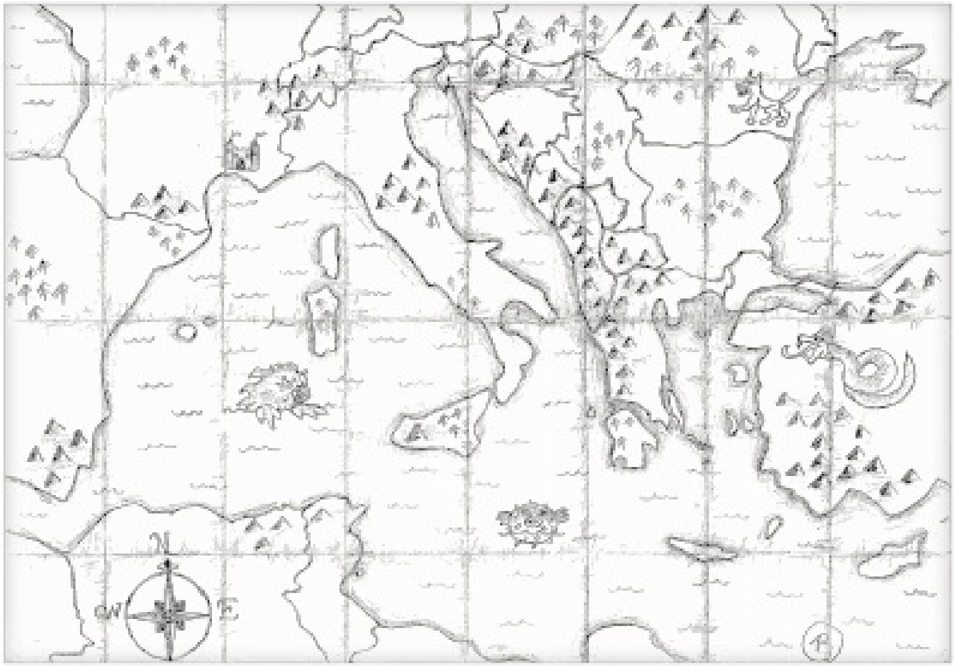
The Pragmatic Programmers

grox.io Learning

Programmer Passport

Prolog



Bruce A. Tate

Edited by Jacquelyn Carter

Prolog

Bruce Tate

The Pragmatic Bookshelf
Raleigh, North Carolina

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

Contents

Change History

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

B4.0: Mar 1, 2020

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

B3.0: Feb 17, 2020

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

B2.0: Feb 3, 2020

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

B1.0: January 15, 2020

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

CHAPTER 1

Logic Programming Basics

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

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

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

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

What is Prolog, Anyway?

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

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

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

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

Important Prolog Features

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

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

backtracking
: The next major contribution is backtracking. When Prolog is working on a problem, it doesn't look for a single solution to a problem. It looks for all of them. When Prolog is solving a complex query, sometimes it looks

for all possible partial solutions. When one partial solution is wrong, Prolog can backtrack and try another.

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

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

Prolog Programming Basics

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

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

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

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

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

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

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

The SWI Prolog Console

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

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

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

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

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

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

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

Prolog responds with:

```
hello, world
true.
```

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

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

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

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

Since the comma is an and, Prolog must run both predicates to determine whether the predicate is true. Prolog dutifully prints out the results of both write predicates.

Or you can join them with or, with semicolons between them:

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

We get the first write, true, and Prolog waits because there may be more solutions to our query. We can ask for the next value with a ; in the console, like this:

```
true ;
 and that's a wrap.
true.
```

We type a semicolon. Then, we get the results of the second write followed by the value of the second predicate, a true. Often, running certain commands will return many responses. We'll explore these different commands as our problems get more sophisticated and our problems get more advanced.

You're probably itching to get to the good stuff, and there's plenty of good stuff. More than anything else, Prolog is about making inferences from data, and we'll need to type in a database to see those concepts in action. Let's create one now!

Databases, Facts, and Rules

If you spend much time with SQL databases, Prolog databases are not what you're used to. The databases we'll be using describe logical models with facts and rules. In truth, the traditional "Hello, World" in Prolog is not write("hello, world");. It's a database with a few facts and rules about who likes who.

To tread some new ground, let's build a quick database about the Twitter social networking application. Rather than grinding through a lot of syntax, we'll establish a quick database and try it out. Then we'll circle back and tighten up the rules you've used.

We're going to use an abbreviated syntax that makes it easy to load our database. That means we'll name our database db.pl, and starting swipl from the directory that has the database. Let's create a database that describes a few things about a tweet.

For those who don't, um, follow Twitter, here are a couple of simple rules. We'll call a twitter user a tweep, and those people write tiny posts called tweets.

Rather than working with actual names like “Bruce Tate” and actual tweet contents, we’ll use atoms such as bruce to name tweeps and political to name tweet. We’re more interested in the logical relationships between them than actual contents.

Create and navigate to a directory called twitter. Open up the database file called db.pl, and add some tweeps, like this:

```
tweep(jill).
tweep(eric).
tweep(jose).
tweep(joe).
tweep(anna).
```

To us, those are the people in our Twitter universe. After those tweeps, add some tweets, like this:

```
tweet(gossip).
tweet(policy).
tweet(declaration).
tweet(greeting).
tweet(screed).
```

Those are the tweets in our Twitter universe. Now, the only things in our database are facts. We’re going to read the first fact as “jill is a tweep”, and the last fact as “screed is a tweet.” Notice that all of the *arguments* in this database work with specific atoms. Alone, think of an atom as a *value*, like 1 is a value. The atom jill gets its significance not from Prolog, but from the programmer that defines and names the atom. Prolog only knows that jill is different from eric. Prolog code is much more readable if you give your atoms meaningful names.

The expression tweet(greeting). is a fact. Greeting is a tweet. Now, let’s open up swipl from the same directory that has your database and the database like this:

```
?- [db].
true.
```

We load the database. Notice the last true. If it fails to load, you’ll get a message, and you’ll need to find your syntax errors before continuing. Once it loads, we can have some fun.

Queries

Now we can ask some basic questions about our dataset. We’ll start with some queries to verify some individual facts, like this:

```
?- tweep(joe).
true.

?- tweep(robert).
false.
```

We're asking Prolog about individual facts we've added. Right now, the only significance of those queries is that we know tweep(joe) is in the database and tweep(robert) is not.

Now, let's make things a bit more interesting. Let's ask Prolog for all of the tweeps. We'll use a variable instead of an atom, and ask Prolog to fill in the details. Remember, variables start with an upper case character:

```
?- tweep(Who).
Who = jill ;
Who = eric ;
Who = jose ;
Who = joe ;
Who = anna.
```

Notice we type a semicolon after each of Prolog's suggestion. Remember, semicolons mean "Or", so we're merely asking for another suggestion. Also notice that Prolog adds a . after the last possible suggestion. Say we don't want any more suggestions after the first. We'd do this:

```
?- tweet(What).
What = gossip .
```

We gave Prolog a period, meaning "We're done." Whenever Prolog is working through a result set, you can type h for help.

So far, we've provided only a database of basic facts, so we're only using the most basic of Prolog's capabilities. The next step is to add some more sophistication to our database so Prolog can build some inferences.

Inferences

Where facts are generally *specific*, rules are *generic*. A *specific* fact might be that Bruce likes Joe. A rule might be that people who like the same things like each other.

A rule is a logical conclusion. A rule comes in the form hypothisis :- facts. We read it as "hypothesis if facts." Let's add a few more facts and some rules now. To the bottom of your db.pl file, add these lines:

```
tweets(jill, greeting).
tweets(anna, gossip).

follows(eric, jill).
follows(joe, anna).
follows(eric, joe).
follows(jill, joe).
```

We add a few facts. “tweets(jill, greeting)” means “Jill sends the tweet named greeting.” Now that tweets have been sent, we can look at who can see them. In the Twitter universe, that means we look at the follow relationship. In Twitter, followers get the tweets sent by those they follow.

Let’s code that rule. Remember, our rules will move from the specific to the general, so we’ll move from atoms to variables. Add this rule to your database:

```
receives(Tweep, What) :-
  tweets(Who, What),
  follows(Tweep, Who).
```

This inference means Tweep receives Tweet if Who tweets What, and Tweep follows Who. Take a moment to think about that rule until you understand it.

Remember, inferences are mostly *general*. The inference uses variables instead of atoms and lets Prolog fill in the details. Let’s try out our inference with some specific questions:

```
?- receives(joe, greeting).
false.
```

joe doesn’t receive greeting because joe doesn’t follow jill. That makes sense. How about this one?

```
?- receives(joe, gossip).
true.
```

joe receives gossip because anna tweets gossip, and joe follows anna. Now, let’s make the queries slightly more general.

```
?- receives(joe, What).
What = gossip.

?- receives(Who, gossip).
Who = joe.
```

The first query is What tweets does joe receive? Prolog walks through the associations, trying out possible values for each of the variables. If no value can be found, Prolog backs up and tries additional values. This process is called *backtracking* and we’ll look at it in depth later. Prolog correctly determines that joe receives gossip because joe follows anna and anna tweets gossip.

We can also ask this question in the most general way, Who receives What, like this:

```
?- receives(Who, What).
Who = eric,
What = greeting ;
Who = joe,
What = gossip.
```

Prolog follows the inference, trying out the possible values that would make the inference true. The possible values for the tweets predicates on the left hand side are jill and anna. The possible values on the right hand side are greeting and gossip. And the possible left hand values for the follows predicate are eric, joe, and jill. So using those values, let's go down a decision tree, stopping when a clause is not true:

```
receives(Tweep, What) :-
  tweets(Who, What),
  follows(Tweep, Who).

tweets(Who, What):
  Who = jill
    What = greeting
    tweets(jill, greeting): true (continue)
      follows(eric, Who=jill): true (SUCCESS: next)
      follows(joe, Who=jill): false (STOP: next)
      follows(jill, Who=jill): false (STOP: next)

    What = gossip
    tweets(jill, gossip): false (STOP: next)

  Who = anna
    What = greeting
    tweets(anna, greeting): false (STOP: next)

    What = gossip
    tweets(anna, gossip): true continue
      follows(eric, Who=anna): false (STOP: next)
      follows(joe, Who=anna): true (SUCCESS: next)
      follows(jill, Who=anna): false (STOP: next)
```

Take a look at this decision tree. It's not exactly how Prolog would process it, but it's close. We get the possible values for each variable by looking at the facts in the database. Next, we try out the first possible fact in the inference, and all of the possible facts. If we can't make a fact true, we quit, and move on to the next possible value.

Depending on how you think about the data, there's a potential glitch in our database, so let's fix it now. To do so, we're going to need to use the semicolon.

Inferences with Or

Ideally, a tweep receives her own tweet. You might make a case for receives working another way, but for now, let's consider receives as everyone who has seen a particular tweet, even the one who wrote it. Extending our receives inference to handle this case is easy. We just need to add the case that a receiver can also be the one tweets, like this:

```
receives(Tweep, What) :-
  tweets(Tweep, What);
  tweets(Who, What),
  follows(Tweep, Who).
```

This inference means "Tweep receives What if *either* Tweep tweets What, *or* Who tweets What and Tweep follows Who." The commas are "ands" and the semicolons are "ors". Prolog will group the and clauses together and execute them before executing the or clauses. That means you'll get something like this:

```
tweets(Tweep, What) OR (tweets(Who, What) AND follows(Tweep, Who)).
```

Try it out, and you'll find that the senders now receive their own tweets, like this:

```
?- [db].
true.

?- receives(Who, What).
Who = jill,
What = greeting ;
Who = anna,
What = gossip ;
Who = eric,
What = greeting ;
Who = joe,
What = gossip.
```

Now jill receives greeting, and anna receives gossip. They're the original senders so that makes sense. The programming model takes some getting used to, but inferences have many of the properties of functional programs. Let's build some more advanced inferences.

Layering Inferences

Sometimes, an inference might get too complex. When that happens, we do the same thing we do in other programming languages. We break down the complex inference into several similar ones. Let's say we wanted to build in the notion of fans. Fans will always retweet the stars they follow, and retweets will contribute to received tweets. We could build a complex notion of tweets and retweets, but that code would be difficult to understand.

Instead, we'll build the notion of says by using both tweets and retweets. Let's add a couple of inferences to our database to handle retweets and says, like this:

```
fan(joe, anna).
fan(eric, joe).
```

The first things we need are fans. joe is a fan of anna and eric is a fan of joe. We want the fans to always retweet everything from those they follow. We can do so with this simple inference:

```
retweets(Fan, What) :-
  tweets(Star, What),
  fan(Fan, Star).
```

Fans retweet the tweets their stars tweet. But now, the receives inference is incomplete because it won't receive retweets. We can fix that by adding a new inference called says that encompasses both tweets and retweets, like this:

```
says(Tweep, What) :-
  tweets(Tweep, What);
  retweets(Tweep, What).
```

Beautiful. We combine these two basic facts with a semicolon, meaning only one of either tweet or retweet needs to be true for the inference to be true. All that remains is to tie says into receive, like this:

```
receives(Tweep, What) :-
  tweets(Tweep, What);
  says(Who, What),
  follows(Tweep, Who).
```

Two subclauses make up this inference. Thee first is that tweeps can see their own tweets. Second, tweeps see retweets and tweets from those they follow.

That's a lot of complexity, but we're ready to use these inferences.

```
?- [db].
true.

?- [db].
true.

?- receives(Who, gossip).
Who = anna ;
Who = joe ;
Who = joe ;
Who = eric ;
Who = jill.
```

anna receives gossip because she tweeted it. joe receives gossip because joe follows anna (and one other reason—can you find it?).

eric gets gossip because joe is a fan of eric, and eric is a fan of joe. And jill gets gossip because jill follows joe, and joe retweets anna, and eric retweets joe.

And *that's* interesting. We can extend these inferences indefinitely, and Prolog will do the heavy lifting to determine what's true and what's not.

This chapter has been pretty dense, so it's time to wrap up. We'll ask a couple of questions about the twitter database, and introduce some of the basic Exercism problems.

Try It Yourself

Prolog is the first language in our Joe Armstrong celebration. It's a language made up of databases. Each databases uses data to make facts and rules, and combining these concepts can lead to some pretty sophisticated inferences. These factors make Prolog a good language for problems that involve logic, especially when a program must establish a series of facts to get to a more sophisticated inference.

The best way to explore Prolog is to dive in, and now it's your turn.

Your Turn

In this section, we'll work some of the basic Exercism problems and ask a few questions about the previous Twitter database above. Let's start with Exercism.

Exercism

Once you've gone to Exercism[2] and installed the Prolog track, you can look at the way the problems are organized. Follow the directions at Exercism prolog tests[3] for a little background.

If you decide to, um, attack the Queen Attack problem, you'll need a couple of tricks.

First, you can use the ins/2 function to assert that variables are in a range. We'll use it like this:

```
:- use_module(library(clpfd)).
```

2. exercism.io
3. https://exercism.io/tracks/prolog/tests

```
create((A, B)) :- [A, B] ins 1..7.
```

The first directive includes the library we need. The second rule means both A and B are in the domain of integers from 1..7.

Next, you'll also need to check identity with #= (not ==), and you'll need absolute values: abs(-1) returns 1.

Finally, you needed to merge tests into a single grouping of tests in the queen attack exercise. It might be a problem related to my environment, but that trick got me over the hump. These two exercises should get you started.

These easy exercises gets you used to dealing with the Prolog database:

- Hello (hello-world) sets up the environment.
- Queen Attack (queen-attack) tells whether two queens can attack each other in chess.
- Leap (leap) tells whether a year is a leap year.

Twitter Database

The following questions will help you work within a database and to determine whether facts about your database are true. Use the db.pl data set defined in this chapter./ed>

- Who are the tweeps?
- Who can see tweets by anna?
- Who's tweets can eric see?

These problems will let you extend the db.pl database.

- Keep the rules the same, but build a database that will demonstrate Prolog's recursive problem solving nature. For example, luke is a fan of obie, and obie is a fan of yoda. Show that luke can see tweets by yoda.
- Extend the rules to allow blocks. A blocked follower doesn't see tweets.

Next Time

In the next chapter, we'll take on some classic logic problems. We'll solve the states coloring problem and also the eight queens problem, and some others, too. These problems will give us some grounding for dealing with lists and help us establish the toolkit necessary to solve eight queens, and other list-based problems.

We'll see you in two weeks!

CHAPTER 2

Logic Problem Solving

In the first Prolog chapter, we focused on the basics of establishing databases with facts and rules, and then we queried that database. In this chapter, we're going to focus on problem solving techniques. In it, we're going to work through some functional programming basics. You'll see the Erlang and Elixir standbys of pattern matching, deconstruction, and recursion. Fans of Joe Armstrong, the creator of Erlang, will find plenty of influence here.

Even if you're familiar with functional programming, this chapter is going to take you to interesting places. Prolog's twist on programming is decidedly different. We'll start this chapter with my first Prolog problem, one I explored with Joe Armstrong himself. He called this solution my first Prolog moment. We'll work through it several different ways, and explore ways to express the problem. We'll also do some refactoring along the way, and examine a Prolog concept called *domains* as part of that solution.

Next, we'll tackle list-based processing. Before moving on to our final logic puzzle, we'll need a few problem solving techniques: pattern matching, list deconstruction, recursion, and accumulators.

Finally, we'll look at another iconic problem, the Prolog eight queens problem. It involves placing eight queens on a chess board with no two queens sharing the same row, column, or diagonal.

We have a lot of work to do, so let's get busy.

Map Coloring

In geometry, there's a theorem that no more than four colors are required to fill in regions on a map so that all connected regions have different colors. Let's build a Prolog program to pick colors for countries in a map. Let's think a bit about how to represent the problem.

My first inclination for solving such problems was to think about building an algorithm to get to a *solution*. That's rarely the best way to think about a Prolog problem. Instead, it's best to think about building rules and facts that describe the *problem domain*.

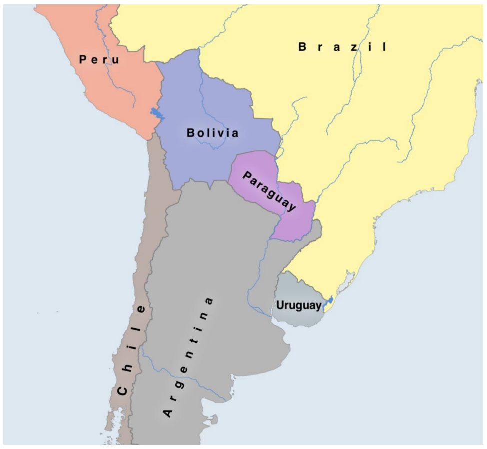
As a developer who is used to thinking in terms of types, my next inclination is to assign atoms to each dimension. Our problem domain has two types of entities, countries and colors. You might be tempted to build rules based on bordering countries, like this:

```
border(Country1, Color1, Country2, Color2) :-
  border(Country2, Color2, Country1, Color1),
  dif(Country1, Country2),
  dif(Color1, color2).
```

Then, we'd have a valid_coloring rule with all of the potential borders. This solution might work, but we'd have to work harder than we should. There's a better way.

Both Variables and Data Are Important

A better approach is to have variables for one dimension of the solution and values for the other. Then, the problem becomes filling in country variables with the right colors. Doing so makes the problem stunningly simple. Let's say we're solving for this map:



We want to establish that the colors are different from each other. They seem basic, but they're important because they let Prolog determine all of the possible values on either side of different. Navigate to a directory called colors. Let's key in a problem, piece by piece. The first step is to establish that colors are different from each other. Let's bludgeon the problem into submission with brute force now, and we'll tweak the program to use a little finesse later:

```
different(yellow, red).
different(red, yellow).
different(yellow, blue).
different(blue, yellow).
different(blue, red).
different(red, blue).
```

We'll start with three colors, as they'll be enough to color the proposed map. Open up a swipl instance and let's make sure our program is working. Load the database and try a couple of rules, like this:

```
?- [db].
true.

?- different(yellow, yellow).
false.

?- different(yellow, red).
true.
```

So good, so far. The rest of the problem is shockingly easy. Define a rule for a valid coloring of the countries in the map. Remember, each border must be different, like this:

```
coloring(Chile, Argentina, Bolivia, Uruguay, Paraguay, Peru) :-
  different(Chile, Argentina),
  different(Chile, Bolivia),
  different(Chile, Peru),
  different(Peru, Bolivia),
  different(Bolivia, Paraguay),
  different(Argentina, Bolivia),
  different(Argentina, Uruguay),
  different(Argentina, Paraguay),
  different(Argentina, Bolivia).
```

We'll forget Brazil for now so we can debug with a lesser list of states. Look at the simplicity of the solution! Because we let variables represent countries and values color atoms, the solution is laughably simple. We can run it and get a solution:

```
?- [db].
true.

?- coloring(Chile, Argentina, Bolivia, Uruguay, Paraguay, Peru).
Chile = Uruguay, Uruguay = Paraguay, Paraguay = yellow,
Argentina = Peru, Peru = red,
Bolivia = blue ;
```

Nice! Chile, Uruguay, and Paraguay are all yellow; Argentina and Peru are red, and Bolivia is blue. Throw in Brazil, though, and our solution starts to break down. We'll have to specify that all *four* colors are different from each other, so the code will get more and more tedious. Adding one country will force us to add nearly ten lines of code! Let's refactor this solution to make our intentions clearer.

Think About Domains

Truth be told, our beautiful solution has a couple of significant flaws. The first is that our specification of different. It's error prone and tedious. The second is that we don't label all of our concepts in code. Sure we could solve this problem with copious comments, but there's a better way. borders is a better name than different. Finally, it would also be nice to be able to label colors more concisely, without having to go through all of the permutations of different colors.

Two of these problems, colors and the definition of different, are problems of *domain*. We need a way to tell Prolog that certain variables belong in a domain. The third is just a software design problem.

Let's attack the first problem by tweaking our definition of different, like this:

```
different(CountryColor1, CountryColor2) :-
  member(CountryColor1, [red, yellow, blue]),
  member(CountryColor2, [red, yellow, blue]),
  dif(CountryColor1, CountryColor2).
```

That's a little bit better. We label the countries as CountryColorN to avoid confusion. We also do something very important for a language like Prolog. *We explicitly specify the domain the countries can come from.* By clearly expressing the pool that Prolog can draw from for each variable, the interpreter can cycle through the valid combinations in a logical manor.

Then, we use a Prolog function called dif to verify that the country colors are in fact different. You can run the program to verify that it still works.

We still don't have to be satisfied, though. There's still a little bit of duplication in the two different member functions, and we can do better with the rule name. Let's turn the different function into two, like this:

```
border(Country1, Country2) :-
  color(Country1),
  color(Country2),
  dif(Country1, Country2).

color(Color) :-
```

```
  member(Color, [red, yellow, blue]).
```

Remember to change the different functions in the coloring rule, like this:

```
coloring(Chile, Argentina, Bolivia, Uruguay, Paraguay, Peru) :-
  border(Chile, Argentina),
  border(...),
  ...
```

And now, the program is better. The border function beautifully describes what we're doing, and the color function sets the domain for each of the variables using member. Now, adding Brazil is much easier. You'll need to only add another color, and add the countries bordered by Brazil to the coloring rule.

If you would like to break away from the text to write some code, go ahead and find a valid coloring for the map, but with Brazil! If you don't want to look it up, Brazil borders all countries on that map except Chile. It's the Tennessee of South America!

Let's step away from the coloring problem for now, and toward another iconic problem, the eight queens problem. To solve it, we'll need to look at pattern matching and lists first.

Unification, Lists, and Pattern Matching

If you're an object oriented or procedural programmer used to languages like Java, Ruby, Swift, or C++, you're going to have some fun. Lists in Prolog are processed differently than they are in the other languages that use arrays. The reason is that Prolog is declarative, so it needs to be able to express facts about lists rather than iterate over them.

In this section, we're going to write rules that use pattern matching, and that's going to use a Prolog concept called unification.

Unification

Loosely defined, unification seeks to make the left and right sides match, filling in the variables it can. Here's a simple example:

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

That result is not surprising at all. It works like an assignment, but that's not what's happening at all. Unification seeks to match the expressions on either side of the = operator. Let's turn up the volume. Unlike assignment, = works in either direction:

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

```
List = [1, 2, 3].
```

OK. That's more surprising. Now, try this expression that unifies in both directions:

```
?- [A, 2, 3] = [1, B, C].
A = 1,
B = 2,
C = 3.
```

Now, that's interesting. What happens when Prolog doesn't have enough information?

```
?- [A, B, 3] = [1, B, C].
A = 1,
C = 3.
```

Prolog fills in the ones it can. Now let's try lists that we can't logically match:

```
?- [A, B, 3] = [1, B, 4].
false.
```

That fact can't be resolved, so Prolog correctly tells us the statement is false. We'll use this unification concept frequently as we reach goals. Each Prolog query returns a true or a false, and also *supplies the variables that would make the goal true*. Take this example of the Prolog `append` predicate:

```
?- append([[1, 2], [3, 4]], [1, 2, 3, 4]).
true.

?- append([[1, 2], [3, 4]], What).
What = [1, 2, 3, 4].

?- append([[1, 2], What], [1, 2, 3, 4]).
What = [3, 4] .

?- append([Thing1, Thing2], [1, 2, 3, 4]).
Thing1 = [],
Thing2 = [1, 2, 3, 4] ;
Thing1 = [1],
Thing2 = [2, 3, 4] ;
Thing1 = [1, 2],
Thing2 = [3, 4] ;
Thing1 = [1, 2, 3],
Thing2 = [4] ;
Thing1 = [1, 2, 3, 4],
Thing2 = [] ;
false.
```

Rather than providing a function that appends the elements of a list, Prolog provides a `true` or `false` predicate that checks whether the list on the left can be joined to form the list on the right. It's a powerful tool, but one that takes a bit to understand.

Let's add a tool to the tool box: list deconstruction.

Deconstructing Lists

Here's the next set of concepts we're going to need. To make rules relying on lists, we're going to have to address items in the list. In Prolog, you can access the first element of a list and everything but the first element, like this:

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

Nice. Prolog took apart that list for us. Since we're using unification, the process also works in reverse, like this:

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

Unification can also fill in missing data from both sides, like this:

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

Lists and Inferences

Now, we have enough information to write some programs. Let's say we wanted to build some functions to get the first and last elements of a list. We have all of the tools: rules, list deconstruction, and unification. Create a directory called `math`, and drop this code into `db.pl`:

```
first(First, List) :- [First|_] = List.
```

Ah. That makes sense. Rather than creating a function to return an answer, we create a rule that expresses the question and answer on one side, and the facts that must be true to solve the problem on the other side. Now, we can use that rule like this:

```
?- first(What, [1, 2, 3]).
What = 1.

?- first(1, [A, 2, 3]).
A = 1.

?- first(1, List).
List = [1|_22334].
```

With unification, the first two solutions make sense, but the third is a little strange. Prolog is telling us it doesn't have enough information to solve the problem, but it knows the general shape of the answer.

That problem doesn't require much reasoning. Let's go a little further. Let's get the last element of the list.

We're going to break this rule into multiple different clauses. When processing a rule, Prolog will pick the first one that matches the inbound arguments. Our last rule will have two clauses: one for lists of one, and one for greater lists. The first clause is easy:

```
final([Final], Final).
```

The last element of a list with one item is that item. Start the next clause, like this:

```
final(List, ...
```

Hm. We're stuck, because Prolog doesn't let us get any element explicitly except the first. Getting just the list isn't going to be enough for us. Let's break the list into pieces.

```
final([First|Rest], ...
```

Now we're getting somewhere. Let's assume there's a final answer named Final, like this:

```
final([First|Rest], Final) :-
```

And we can start to fill in the rest of the program. At this point, we're going to use *recursion*. We'll set a clause in the inference that depends on the rule we're writing. If Rest is a list with one element, it will match the first clause we've already built. If it's not, we can try final again, stripping away the head of the list, and trying again. The solution looks like this:

```
final([Final], Final).
final([First|Rest], Final) :-
  final(Rest, Final).
```

And that gives us exactly what we need. Try it out:

```
?- [db].
true.

?- final([1, 2, 3], What).
What = 3 .
```

Bingo. We're done. We can solve many different problems this way. For example, computing the sum of a list looks like this:

```
total([], 0).
total([Number|Rest], Sum) :-
  total(Rest, PartialSum),
  Sum is PartialSum + Number.
```

Notice the is. Typically, we'll want to use is when the left hand side is *unbound*, or not yet assigned. Think assignment instead of unification. The sum of an empty list is zero. The second total of a list made up of Number and Rest is Sum IF:

- the total of all numbers except the first is PartialSum, AND
- the Sum is PartialSum plus the first item.

And that's a wrap! Only, it's not. Sometimes we need another problem solving strategy with lists. List processing uses something called tail recursion optimization. That means *if the last thing a rule does is invoke a recursive call*, the compiler can translate inefficient recursive calls into efficient loops. To build code in this way, we'll need accumulators.

Accumulators and Lists

An accumulator in Prolog, and in other functional languages, amounts to an intermediate value. We'll use these values in recursive assertions. Think about logically solving a problem of sums to a list, with an intermediate value. We'll build the head of our rule like sum(List, Intermediate, Goal). The Intermediate is the sum of the list to this point in the recursion. Here are the variable values we're shooting for at each step in the recursion:

List	Intermediate	Goal
[1, 2, 3]	0	6
[2, 3]	1	6
[3]	3	6
[]	6	6

That looks a lot like iteration, and solutions often feel like iteration. The down side is that there's a bit more ceremony. The plus side is that these solutions are often to be much more efficient. Here's how we'd express a sum rule with an accumulator:

```
sum(List, Total) :- sum(List, 0, Total).

sum([], Final, Final).
sum([Number|Rest], Acc, Final) :-
  PartialSum is Acc + Number,
  sum(Rest, PartialSum, Final).
```

The first step is the goal, our *base case*. In it, the recursion has completed and there's no more work to do. To use it, we'd call `sum([1, 2, 3], 0, What)`. The `0` value is the intermediate one. When we make the initial call, no values from the list have yet been processed. The last value is the goal. Presumably we'll call it with a variable for Prolog to figure out.

The next clause is the interesting one. The `PartialSum` is the accumulator. Said another way, the `sum` of a `List` made up of `Number` and `Rest` is the sum of the rest, plus the `PartialSum + Number` processed so far, and an end result of `Final`.

If you've not seen code like this yet, take a while to see what's going on.

When you're ready, think about the API. We can improve on it by adding another case that mixes in the accumulator. Here's the whole solution, with the new clause:

```
sum(List, Total) :- sum(List, 0, Total).

sum([], Final, Final).
sum([Number|Rest], PartialSum, Final) :-
  sum(Rest, PartialSum + Number, Final).
```

Nice! Since Prolog will pick the first clause that matches, when we call the `sum` function without an accumulator, we simply invoke the rule `sum(List, 0, Total)`, so our clients will never need to remember the accumulator!

Let's look at another solution that uses accumulators. This time, we'll implement our version of `reverse`, like this:

```
rev(List, Final) :-
  rev(List, [], Final).

rev([], Final, Final).
rev([First|Rest], Reversed, Final) :-
  rev(Rest, [First|Reversed], Final).
```

The first clause sets a goal of our rule with the accumulator, passing an empty accumulator to start. The base clause for our recursion says reversing an empty list is an empty list.

The last clause calls `rev` with the progressively smaller `Rest`, adding `First` to the accumulator as it goes.

Let's tighten up the idea of tail recursion optimization. Tail recursion can be transformed into a loop if *the last thing a rule does is invoke a recursive rule.*

Now, we have more tools in our toolbox to attack the eight-queens problem.

Eight Queens

We've been dancing around Prolog's core capabilities as we gather enough information to solve a nontrivial problem. It's time to attack one. Eight Queens is an iconic programming problem involving the queen piece in a game of chess. Chess is played on an eight by eight grid, and the queen is the strongest piece that can attack along rows, columns, or diagonals. In this problem, we're looking for a solution that places eight queens on a board without any two attacking each other. That means no two queens can share rows, columns, or diagonals.

We're going to solve the problem in a naive way first, and then refactor our solution to be more efficient. Let's first think about the problem with some exploratory code. Then, we'll hone in on an approach that's efficient and understandable.

Check the Domain

To represent a queen on a board, we're going to start with tuples. In Prolog, tuples are represented as a list of elements, separated by parentheses. Our elements are going to be integers. A queen in the upper left corner of the board will be (1, 1), lower right will be (8, 8). Our first two jobs are to establish the domains for our queens as integers from 1 to 8, and establish two queen placements as safe.

To do so, we're going to need a couple of new tools. First, we'll need the library that checks domains. We'll use the :- use_module(library(clpfd)). directive. Next, we'll need an operator from that library, the in operator. It establishes that a variable must have values within the specified domain.

Create a new directory called queens, and a db.pl database in it so we can start to play:

```
:- use_module(library(clpfd)).

queen((Row, Col)) :-
  in(Row, 1..8),
  in(Col, 1..8).
```

Now, we've defined that a queen must be in the specified domain. We can try that much out:

```
?- [db].
true.

?- queen((0, 5)).
false.
```

```
?- queen((8, 8)).
true.
```

So we can determine whether something is a queen or not. Next, we can think about the concept of an attack.

Queen Attack Logic

Two queens attack one another if they share a row, column, or diagonal. Let's determine whether two queens attack, and then we can generalize that solution to determine if two queens are safe. Before we do that, we need to do a little exploration. In your console, check for equality:

```
?- 1 + 1 = 2.
false.
```

Whoa! What's up?

The = operator does not do arithmetic computation, or check for equality. It does unification. If you want to test equality, use =:=:

```
?- 1 + 1 =:= 2.
true.
```

There's another bit of Prolog that takes some getting used to. The console is for *evaluating queries*, and those are all true and false expressions. So you can't, for example, do this:

```
?- abs(1-2).
ERROR: Undefined procedure: abs/1 (DWIM could not correct goal)
```

`abs` is the absolute value[1] of an integer. But you can do this:

```
?- Abs is abs(1-2).
Abs = 1.
```

Now, we have enough to code the first part of our attack problem. Add an `attack` rule, like this:

```
attack((R1, C1), (R2, C2)) :-
  R1 =:= R2;
  C1 =:= C2;
  abs(R1-C1) =:= abs(R2-C2).
```

Notice we use semicolons instead of commas. The semicolons mean the rule succeeds if *any* of the clauses are true, meaning we get *or* logic. Try it out:

```
?- attack((1, 1), (1, 4)).
true .
```

1. https://www.dictionary.com/browse/absolute-value

```
?- attack((1, 1), (4, 1)).
true .

?- attack((1, 1), (4, 2)).
false.

?- attack((1, 1), (4, 4)).
true.
```

We're not going to belabor the diagonal attack math, but break out a chess board and play with the values. You'll see it works. We really don't need unsafe pairings, though. We need to test safe pairings. To be safe, *all* of the clauses must be false. Change the name of the clause to safe. Then, trade semicolons for commas and =:= for =\= (not-equal):

```
safe((R1, C1), (R2, C2)) :-
  R1 =\= R2, C1 =\= C2, abs(R1-C1) =\= abs(R2-C2).
```

Test it if you want to convince yourself that it's correct. We're one step closer to a solution. Next, let's think about the greater problem. One of the biggest problems we'll need to solve is whether a whole list of queens are safe.

Now that we're getting to the greater problem, we should think about the math of the overall solution. Prolog is going to work by working through possible combinations in some fashion. If you look at the total number of combinations, there's a lot of work to do. Eight queens times eight rows times eight columns gives us 512 total combinations. That's not too outrageous but we can do better. We can restrict the total number of possible solutions significantly.

Queens as Integers

We can decide to put the first queen on the first row, the second queen on the second row, and so on. Then, Prolog wouldn't have to calculate the row for any queen. We'd have a list like this:

```
[(1, Q1), (2, Q2), (3, Q3), ...]
```

But for any solution to be valid, no two queens can share a row, or they would be attacking. Further, they can't even share the same *column* either. If we can infer the queen's row from the position in the list, we really don't have to track a row at all. Our queen board would look like this:

```
[Q1, Q2, Q3...]
```

That's much better. Go ahead and scrap the code you've done so far. Throwing away code is good when things aren't right. We know more about the problem domain than we did. This new domain will make things much easier. Now, we know that our list of queens:

- is a list of integers
- with each integer in 1..8
- where all integers are different
- where all integers are safe in terms of their diagonal placement

We can take care of the first three requirements almost immediately. Discard your database and start over with the following rule:

```
:- use_module(library(clpfd)).

queens(Queens) :-
  length(Queens, 8),
  Queens ins 1..8,
  all_different(Queens).
```

We build a clause, and tell Prolog that the list will be of length 8, and each member of the list will be in the domain from 1..8. (Think of this rule as many in clauses, one for each list element, or ins). While we're at it, we use another predicate to enforce that all integers in the list must be different.

And let's try it out.

```
?- [db].
true.

?- queens(What).
What = [_6306, _6312, _6318, _6324, _6330, _6336, _6342, _6348],
_6306 in 1..8,
all_different([_6306, _6312, _6318, _6324, _6330, _6336, _6342, _6348]),
_6312 in 1..8,
_6318 in 1..8,
_6324 in 1..8,
_6330 in 1..8,
_6336 in 1..8,
_6342 in 1..8,
_6348 in 1..8.
```

That's… disappointing.Prolog is providing an efficient intermediate form, but we can do better. We can ask for some more human-friendly labeling. Let's do so:

```
?- queens(What), label(What).
What = [1, 2, 3, 4, 5, 6, 7, 8] ;
What = [1, 2, 3, 4, 5, 6, 8, 7] ;
What = [1, 2, 3, 4, 5, 7, 6, 8] ;
What = [1, 2, 3, 4, 5, 7, 8, 6] .
```

Marvelous! Prolog is going though potential solutions! We just need a way to express whether a diagonal is safe.

Safe Diagonals with Distance

This diagram represents four queens on a board. You can see that each queen gets her row from the position in the array, and the column from the integer value. The other thing that's important is the *distance between the queens*.

```
   1  2  3  4  5  6  7  8
1     Q4 : 2
2
3  Q1 : 1
4     Q2 : 2
5        Q3 : 3
6
7
8
```

Queens Q1, Q2, and Q3 share a diagonal. If you look at them carefully, you can tell that a queen shares a diagonal if the distance between them in rows is the same as the distance between them in columns. Q4 does not share a diagonal. The differences between the rows are 2, 3, and 4 respectively, but the differences between the columns are 1, 0, and 1. So we have the math to check for safety, given two queen integers and a distance. Let's do that now. Drop this rule in:

```
different_diagonals(Queen1, Queen2, Distance) :-
  abs(Queen1 - Queen2) #\= Distance.
```

We get the absolute value, and use the #\= predicate to specify distinct integer values. This predicate does not do unification; it merely tests whether Queen1, Queen2, and Distance satisfy the test.

Now you can try it out:

```
?- [db].
true.

?- different_diagonals(3, 1, 2).
false.

?- different_diagonals(3, 1, 3).
true.

?- different_diagonals(1, 3, 3).
true.
```

Beautiful. So good so far. Now, we need a plan of attack for the hardest part of the problem. It's time to check *all* of the diagonals on the whole board.

Recursive Row Check

Remember, we're not really building a problem solver, per se. We're describing the rules of the game. To describe the rest of the problem, we need to make sure no queen conflicts with any other.

Think about how you might check a queen setup if you were placing queens on a chess board. One way is to go to the queen on the first row of the board, and make sure it couldn't attack any other queen, checking each one in turn. When you were done with that one, you'd discard that queen, and check the rest. You'd finish when no more queens remain.

Our solution will closely resemble that approach. First, we'll pick a queen and check it against all of the other queens on the board. When we're done with that one, we'll discard that queen and check the rest of the board. When we're out of queens, we'll be done.

Let's code up the first part. We need to check one queen against all of the other queens on the board. We'll choose the first queen as our initial target, since it's the easiest one to grab, and we'll check it against the remaining queens, one by one. That test looks like this:

```
safe_queen([], _, _).
safe_queen([Queen|FollowingQueens], Target, Difference) :-
        different_diagonals(Queen, Target, Difference),
        safe_queen(FollowingQueens, Target, Difference + 1).
```

It's a bit tricky, so let's take it a clause at a time. These clauses have three arguments. The first is the list of queens we're checking against. The second is the target queen. The third is the distance between the target and the rest of the queens. Our goal is to make sure that none of the queens on the list can attack our target.

First, we make sure there's no collision between the target and the first queen on the list, and once we're done with that one, we check the rest of the queens, making sure to keep up with the distance as we cycle through queens by adding one each time.

We're done when there are no more queens to check. That's the base case for our recursion.

The next clause is the big one. Remember, we're taking a single queen and asking, "Does this queen conflict with any of the queens in the list?" The queen we're checking is called Target. We check the target and the first queen on the list with different_diagonals, providing the distance. If it fails that check, we're done and Prolog backtracks and tries the next list of queens. If it passes that test, we run the same test on the rest of the queens on the list, increasing the distance by one for each trip through!

We can try it out:

```
?- [db].
true.

?- safe_queen([2], 1, 1).
false.

?- safe_queen([2], 1, 2).
true.

?- safe_queen([2, 5], 7, 1).
false.
```

Bingo! Queens at 1 and 2 will conflict if the distance between them is one, as they should. Queens at 2 and 5 will not conflict with a queen at 7, if we give it a distance of one. If it's confusing to you, put queens on a chess board at position 7 on the first row, 2 on the second, and 5 on the third:

```
   1  2  3  4  5  6  7  8
1     Q
2              Q
3                    Q
...
```

Notice the distance between row 1 and row 2 is one row. You can see they don't share the same diagonal, so we're good! Then, we'd check the first queen against the third one, with a new distance of 2, and so on.

When we've gone through this goal, we know a single queen has no conflicts. All that remains is to make sure none of the other queens have conflicts! You may even be able to see how that rule will look.

First, we'll drop in the additional rule called queens(), like this:

```
safe([]).
safe([Queen|Queens]) :-
  safe_queen(Queens, Queen, 1),
  safe(Queens).
```

That code is short, but there's a lot going on. A board with no queens has no conflicts. That's the base recursion case. Then, in the recursive clause, we break the list down to the first queen (which will become our target) and the rest. First, we establish the safe_queen goal with the queens, the target queen, and an initial distance of one. Once this clause has been run, it's firmly established that the first queen has no conflicts. We can move on to check the rest of the board. We do so with our recursive call, safe(Queens).

All that remains is to call this clause from our initial queens clause, like this:

```
queens(Queens) :-
  length(Queens, 8),
  Queens ins 1..8,
  all_different(Queens),
  safe(Queens).
```

Now everything is tied together. All that we've done is write a rule book for the eight queens problem. Reading our "rulebook" goes something like this:

- queens(Queens) is true if:
- Queens is a list with a length of 8, and
- every item in Queens is an integer from 1 to 8, and
- every item in Queens is distinct
- the configuration is safe, meaning no queen shares diagonals

Let's try it out:

```
?- [db].
true.

?- queens(Queens), label(Queens).
Queens = [1, 5, 8, 6, 3, 7, 2, 4] ;
Queens = [1, 6, 8, 3, 7, 4, 2, 5] ;
Queens = [1, 7, 4, 6, 8, 2, 5, 3] .

?-
```

And we're done! Look at the first solution on a board and you can see that they don't share rows, columns, or diagonals.

```
   1  2  3  4  5  6  7  8
1  1
2              5
3                       8
4                 6
5        3
6                    7
7     2
8           4
```

The nice thing is that our solution can give us one eight-queens solution, several of them, or all of them. We solved the problem not by describing a solution in code, but by describing the rules of the problem.

Problem Constraint Programming

Let's take a step back and look at our program in broad strokes. These are the steps you'll take when solving a family of solutions known as constraint-based programming.

1. Establish a rule with an end goal. Our queens rule did this.
2. Restrict the problem set. We did so with the length and ins predicates.
3. Build in the constraints that describe your solution. We did this with all_different to restrict columns, and safe to restrict diagonals.

We didn't have to spend any time describing the algorithm Prolog would need to solve the problem. Prolog simply tried solutions, one at a time, and quit working on a particular solution once a constraint failed.

This family of programming is highly lucrative and interesting to those who can break into it. It's called constraint logic programming. Think scheduling problems, delivery routes, delivery schedules, and the like. We'll start to look at some tools you can use to solve such problems, like programming routes and schedules, in the next chapter.

It's been a full chapter, so it's time to wrap up.

Try It Yourself

As we continue through our Joe Armstrong celebration, we've come to two of his favorite problems for Prolog beginners. The map coloring problem is an iconic problem that requires a mental leap. It's best to model the problem domain in two dimensions: regions and colors. Represent one with values such as atoms, and the other with variables. Then, let Prolog fill in the details.

The eight-queens problem is a constraint-based logic problem. In Prolog, we set the goal with a rule, establish a domain with list and ins, and then set the constraints. When Prolog works with these kinds of problems, it works through possibilities in the domain until it breaks a constraint, and then moves on to the next possible goal. This process happens recursively, as constraints establish goals of their own.

Now, you can try some of these solutions yourself. We're going to suggest some changes to the eight-queens problem.

Your Turn

In this section, we're going to suggest some classic problems for you to solve. We'll stay away from Exercism, since those problems really aren't the constraint-based problems requiring logic. Instead, we'll first make some enhancements to the eight-queens program, and then solve a sudoku, with some help from the Groxio[2] video series.

2. grox.io\languages\prolog#videos

Easy: Extend the eight-queens problem to take on N queens on an N by N chess board. It will involve making several changes to the first queens clause we coded in our chapter.

Medium: Provide a friendlier output for eight queens. You can add a clause to the queens clause called print_queens that takes a list of queens, and prints out those queens in a prettier format. For example, you might print out this:

```
Q
  Q
    Q
      Q
```

for the list [1, 3, 5, 7]. To solve this problem, you're going to need:

- recursion for each row and for the whole board
- the write('char') predicate to write out a character, either a Q or a space
- the nl predicate to write a new line

Easy: Build a map coloring algorithm for the southern countries of Africa, with all countries below and even to Zambia.

Easy: Add Brazil and the color green to the map coloring program we wrote together in this chapter.

Hard: Express the map coloring problem using three lists: a list of region variables, a list of color atoms, and a list of borders. For example:

```
colorings([Tennessee, Alabama], [red, blue], [[Tennessee, Alabama]]) :-
  your_solution_goes_here(...)
```

Pick a map that interests you.

Next Time

In Chapter 3, Graphs, on page 35, we'll take a crack at some more practical problems. We'll look at graphs. They can be used to find paths between points using facts and recursion. That problem will take us through aggregating results and even sorting solutions. When we're done, we'll be able to think about many types of problems related to maps and social networks.

We'll see you in two weeks!

CHAPTER 3

Graphs

As we delve deeper into Prolog, we've begun to crack some problems that may be a bit more difficult in other languages. Beginning with this chapter, we're going to be working increasingly practical problems.

We'll spend this entire chapter working on the immensely important concept called graphs. Be careful. *Graph* refers to the mathematical concept[1], not charts based on numbers.

Graphs represent the concept of *connection*. In math, graphs are collections of nodes (also called vertices) and the connections, called edges, between them. A node might be a city on a map or a person in a social network. An edge might be the route between two subway stations, or a follows relationship in social media.

Representing connections is important, but what makes graph theory so powerful are the analysis tools a knowledgable programmer can bring to bear. Think of it this way. Knowing all of the airports a carrier uses is interesting. Knowing the data about the flights between them is infinitely more important. Scheduling flights to minimize fuel costs and maximize customer load is the difference between profitability and bankruptcy.

Prolog is the ideal language for dealing with graphs *because the database works that way*. You can often think of a Prolog fact as an edge in a database. That makes it easy to envision what's happening in a graph. In this chapter, we'll use Prolog answer several kinds of questions:

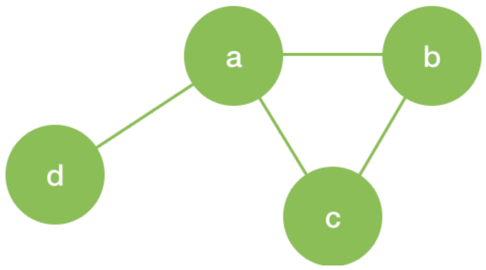
- What connections exist from a given node?
- Is one node connected to another, either directly or indirectly, via other connected nodes?

1. https://en.wikipedia.org/wiki/Graph_theory

- What is the shortest distance between nodes?
- How can we represent data about connections, such as the costs or value of following a connection?

In most cases, we'll write our own algorithms from scratch to answer these questions. In others, we'll use Prolog libraries to solve problems instead.

Let's take a look at a graph, and then the Prolog database we might use to represent it. The following image shows a graph. The point of a graph is *not the data represented in each node*. A graph *is about the relationships between nodes.*



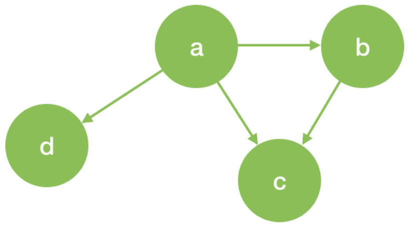
We're going to prefer the less-rigid nomenclature of *nodes* instead of *vertices*, and *edges* for the connections between them. We'll label the edges in several different ways including `a-b`, `(a, b)`, or even `edge(a, b)`. In the section that follows, we'll build some basic algorithms with graphs like the previous one to reason about the graphs.

Most modern Prolog implementations, including our SWI Prolog, have libraries for dealing with graphs. We're going spend a good deal of time focusing on basic algorithms that let us build our own versions of graph-like features. As we get further into the chapter, we'll focus on special cases of graph theory, including weighted graphs and undirected graphs.

In fact, you've already created a couple of Prolog graphs, albeit tiny ones. Both the Twitter follower database in Chapter 1, Logic Programming Basics, on page 1 and the map coloring database in Chapter 2, Logic Problem Solving, on page 15 are graphing problems. Let's create a more formal database graph, with the most basic of representations. You've used Prolog facts in the first two chapters, and we'll use a separate fact to represent each edge of a graph.

Directed Graphs With Edges as Facts

In a directed graph, each edge has a *direction*. In Prolog, edges also imply directions because `fact(a, b)` is not the same as `fact(b, a)`. Consider the same image as the previous one, but with a tiny tweak. We've put an arrow on each of the lines:



That means a is connected to b, but b is not connected to a. We can describe the graph like this. a is connected to b, c, and d, and b is connected to c. Directed graphs are ideal for representing roads (which may be one way), *follows*-style relationships in a social media network, or dependencies.

We can label the edges in the graphs with Prolog facts. Create a new directory called directed and the usual db.pl database. Key this much in:

```
edge(a, b).
edge(a, c).
edge(a, d).
edge(b, c).
```

This rule represents our graph. Let's try it out in swipl:

```
?- [db].
true.
?- edge(b, d).
false.
?- edge(a, d).
true.
?- edge(d, a).
false.
```

It's working, and the last query demonstrates that our graph is connected. An edge between a and d does not show that d and a are also connected, and that's the behavior we want. If we wanted to, we could represent a two-way connection by adding an edge twice, like edge(a, b). edge(b, a).

Now, let's see what kinds of questions we can derive directly from that graph. Here are four:

- Are two specific nodes connected?
- What are the neighbors of a node?
- What are the sources sources leading to a node?
- What are all of the connections?

Let's use the database to answer questions like the ones above:

```
?- [db].
true.

?- edge(d, c).
false.

?- edge(a, Neighbor).
Neighbor = b ;
Neighbor = c ;
Neighbor = d.

?- edge(Source, c).
Source = a ;
Source = b.

?- edge(Source, Destination).
Source = a,
Destination = b ;
Source = a,
Destination = c ;
Source = a,
Destination = d ;
Source = b,
Destination = c.
```

Each of our edge facts represent one of the arrows on our directed graph. Look closer at the results of the four queries. The first asks the question "Is d connected to c?" We don't have to lock in both of the nodes, though. The second and third queries ask "What are the neighbors of this node?" and "What sources feed into this node?". These queries lock in one argument and let Prolog fill in the other. The final query asks Prolog to find *all* of the combinations that fulfill the query.

As with many Prolog predicates, our edge predicate is quite flexible. Sometimes, we use it to fill in the left side or the right side. Let's build a predicate that collects all neighbors in a single list.

Find Neighbors with findall

The Prolog clauses findall, bagof, and setof are clauses that take goals. In a sense, they are like *higher order functions* in other languages, or functions that take functions as arguments. In Prolog, goals that take goals are *meta predicates*. We're going to use findall to find all of the neighbors of a node.

The purpose of findall is to find all data that satisfies a goal. Each invocation will look like this:

```
findall(Template, Goal, List).
```

This predicate finds a List of all objects in the shape of Template that satisfy a Goal. That description is a bit abstract, so let's use it to find all sources or destinations for a node. Add these rules to your database:

```
sources(Destination, List) :-
  findall(Source, edge(Source, Destination), List).

destinations(Source, List) :-
  findall(Destination, edge(Source, Destination), List).
```

In the first rule, we're finding a List of all Sources for a Destination. We're using findall. The first argument is the element we want to add to the list. The second is the goal, edge in our case. The final one is the List we're collecting.

The second rule works in exactly the same way, but looks for a Destination instead. Our solution works like this:

```
?- [db].
true.

?- sources(c, Set).
Set = [a, b].

?- destinations(a, Set).
Set = [b, c, d].
```

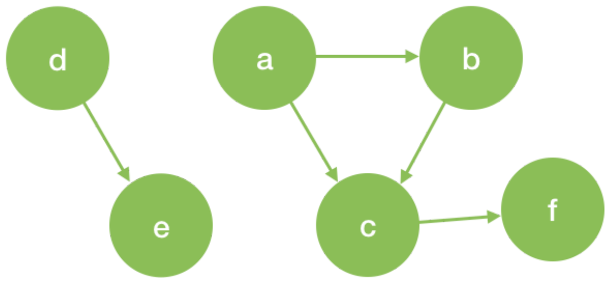
Nice! We can use our edge facts to set individual goals, and our other rules to get a whole list of solutions. Let's crank up the complexity, and the power.

edges can show whether two nodes are connected through an edge, but can't show whether two nodes are connected through a series of intermediate hops. Let's solve that next.

Reachability

With our edge facts, we can answer the question "Can you reach node b from node a, through a single connection?" Sometimes, we'd like to be able to tell if you can reach one node from another directly, or through a series of intermediate hops between connected nodes.

That problem is called reachability. To solve it, we need a more sophisticated graph, like this one:



In this graph, nodes a, b, c, and f are connected to each other, one way. d and e are also connected, but there are no connections between those clusters. More importantly, this graph has a pair of points—a and f—that are not connected, but reachable. After all, that's the problem we're solving. Tweak the edges in your database to look like this:

```
edge(a, b).
edge(a, c).
edge(b, c).
edge(c, f).
edge(d, e).
```

Now, the database reflects the previous image. There's an interesting problem represented in nodes a and f. These nodes are not connected through a single edge, but several paths exist from a to f through connected edges. This problem is a common one within maps, social networks, and distributed systems.

To find whether node a can reach b, we'll do this:

- If a is connected to b, the answer is yes.
- Otherwise, pick a node c that's a destination for a, and check whether c can reach b.

Unsurprisingly, it's a recursive rule. Let's translate it to code.

```
reachable(A, B) :-
  edge(A, B).
reachable(A, B) :-
  edge(A, C),
  reachable(C, B).
```

We start with a base clauses. A node is reachable from a neighbor if there's an edge (in the right direction) between them. Then, we add a recursive rule. A node A is reachable from another node B if they are connected, or if B is reachable from an intermediate node C, and there's an edge from A to C.

That's a surprisingly short program. It does work, but imperfectly:

```
?- [db].
```

```
true.

?- reachable(a, f).
true ;
true ;
false.
```

Prolog confusingly reports three solutions, two true ones and a false one. To see what's happening, look at the graph. You can see two paths that make this query work, a-c-f and a-b-c-f. Each one of the true solutions is for one of the paths, and the third solution reports "there are no more solutions".

Our solution will break down if we add cycles to the graph. If you want to see this solution fail, add edge(c, a) to your database. You'll find Prolog can't exhaust all possibilities.

As we touch on more sophisticated solutions, let's also shift away from directed graphs and toward graphs that are *not* directed.

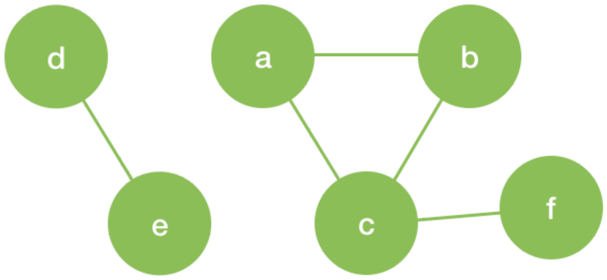
Bi-Directional Graphs

In a bidirectional graph, also called a graph, all of the edges flow in both directions. Social networks like LinkedIn and Facebook are graphs. So are highway maps where roads go both ways. Let's express one in code. Open up a new database called edges and key this much in.

```
edge(a, b).
edge(a, c).
edge(b, c).
edge(c, f).
edge(d, e).

connected(A, B) :-
  edge(A, B); edge(B, A).
```

Except for the final rule, this graph should look familiar to you. It has the same edges as the previous graph. Since we have the connected rule letting edges flow in both directions, our graph shows no arrows:



We can use the same techniques to get the neighbors for a node, but we'd use the connected rule rather than the edge facts, like this:

```
neighbors(A, Set) :-
  findall(B, connected(A, B), Set).
```

This code is almost identical to the sources code, but it uses our intermediate connected rule instead of the edge fact. Try it out:

```
?- [db].
true.

?- neighbors(c, What).
What = [f, a, b].
```

It returns neighbors, regardless of the direction of the edge. That's exactly what we expect.

In this kind of graph, computing reachability is a bit more difficult than it is in directed graphs. Simple recursive rules like our connected one won't work because of cycles. Let's look at why.

Compute Reachability With Paths

Think about possible paths from a to f. a-c-f is a valid path. So is a-b-c-f. Those aren't the only rules, though. a-b-c-b-c-f is valid, as is a-b-c-a-b-c-f. These cycles can go on indefinitely. We need an algorithm that tracks not only where we're going, but *where we've already been*. We need an accumulator.

If you think about it, the accumulator of visited nodes *is the path*. That information is incredibly useful!

In this section, we'll find a path. I'm presenting a solution inspired by this tutorial[2] from California Polytechnic State University. I reshaped the code and renamed variables to fit this book, but otherwise, the solution is the same.

We'll solve the problem with two rules. The first rule, called go, seeks to visit each node in the path, collecting visited nodes in the Seen accumulator along the way. The second rule, called path, is a friendly presentation of our go rule, one that populates an initial accumulator.

2. https://www.cpp.edu/~jrfisher/www/prolog_tutorial/2_15A.pl

Compute a Path With an Accumulator

We'll start with the go rule. Our predicate will have a node we're coming From, a node we're traveling To, the accumulated nodes we've seen so far, and the ultimate result. Here's what the clauses will do.

If From and To are connected, then the path is To plus the nodes Seen so far.

Otherwise, try to go to From from To by choosing an intermediate node called Between. Given the nodes Seen so far and the final Path, we can go from From to To if all of the following are true:

- The source node From and the intermediate node Between are connected
- The intermediate node Between is not the destination node To, and Between has not yet been seen
- We can get from Between to To in some way

Now, we need to translate those two clauses of go into code. Add the first one, our base clause, to your database:

```
go(From, To, Seen, [To|Seen]) :-
  connected(From, To).
```

Go from From to To, where Path is [To|Seen].

Now, let's see what the complex case looks like. Add this clause to your database:

```
go(From, To, Seen, Path) :-
  connected(From, Between),
  Between \== To,
  \+member(Between, Seen),
  go(Between, To, [Between|Seen], Path).
```

The connected clause picks an arbitrary point connected to From called Between. The next two clauses restrict the intermediate node Between, making sure it's not To or any other node on the Seen list.

We finish the go rule off with a recursive clause. We add Between to the Seen accumulator, and go from Between to To. Each recursive call adds one to the accumulator each time through. The call will terminate when Prolog runs out of different nodes to try, or when it reaches a Between that's connected to To through an edge.

Whew. That is the recursive clause. As usual, we'll consume that cumbersome clause with a prettier API. Let's build the tiny friendlier version next.

Present a Friendly API

We're doing the lion's share of the work with go. All that's left are two tiny bits of work in path. First, we'll need to provide an initial accumulator, since our solution never considers the initial node. Next, we'll need to reverse the path since the answer we get is built by appending parts of the path to the *front* of the list as we visit them.

Add this code to invoke our go clause:

```
path(From, To, Path) :-
       go(From, To, [From], Result),
       reverse(Path, Result).
```

There's a Path from From to To if:

- we can go to To from From through path Result, starting with accumulator [From], and
- the Result is the reverse of that Path.

It pays to read through this code a few times if you've not seen this kind of recursion before. It takes a while to internalize the concepts. We're going to build on these concepts going forward through this chapter.

Now, let's try it out.

```
?- [db].
true.

?- path(a, f, Path).
Path = [a, b, c, f] ;
Path = [a, c, f] ;
false.
```

That's more like it!

We query for the paths from a to f, and find two: a-b-c-f and a-c-f. Look at the graph and you'll see that these answers are right. We have a working solution!

Optimizing Paths and Weighted Paths

So far, we've spent a good amount of time working on solutions that return accurate solutions. That's interesting, but much better is getting the *best* solution! Be careful, though. With large datasets, it's easy to chew up plenty of memory and time if your algorithm is not efficient.

In this section, we're going to look at some built-in Prolog features that let us roll up many potential results and choose the best one. As we plan a minimal solution, we'll unlock the secret to dealing with *weighted graphs*, which are graphs having edges with a cost or benefit. Using these techniques, we can count the solutions to a problem, build a collection of solutions, or work out the combined cost or value of all possible solutions.

Let's focus on picking a best solution first. We'll define the best solution as the one with the fewest number of nodes in the path.

Plan the Minimal Solution

We already have the primary tool we need to build a minimal solution, the path rule. We'll wrap path in a couple of other rules to make our task easier. These are the wrappers we're going to build.

First, we'll use path to help compute a path_with_length. Then, we'll build a rule called minimum to return the shortest solution, given many paths with lengths. Finally, we'll use aggregate predicate to build a list of possible paths with lengths, and choose the best one. In text, here's what our plan looks like.

- Compute a valid path with a length.
- Write a function to pick the minimal solution

You might want to try to solve the problem without any Prolog predicates! If you decide to do so and have trouble, you can find a working solution in this tutorial[3] we've been relying on so far. That code works on our Prolog compiler just fine.

We're going to choose to save a little bit of time and go straight to the built-in aggregation, so we have plenty of time to deal with other graph forms. Let's start with a path length.

Compute a Path With Length

Getting a path with a length is straightforward, using the techniques we've already used. We simply use the length predicate, feeding it our path. Add this rule to our database:

```
path_with_length(From, To, Path, Length) :-
  path(From, To, Path),
  length(Path, Length).
```

A path from From to To goes through a Path that's Length nodes long if a Path exists from From to To, and the length of that Path is Length. Not bad. You can try it out to verify that it returns paths with a length:

```
?- path_with_length(a, f, [Path, Length]).
```

3. https://www.cpp.edu/~jrfisher/www/prolog_tutorial/2_15A.pl

```
Path = [a, b, c, f],
Length = 4 ;
Path = [a, c, f],
Length = 3 ;
false.
```

Nice! It works exactly like we expect it to. We've also begun to compute a *path* for reaching a node, and also a *cost* for doing so. We'll use the same aggregation technique we use in the rest of this section to build weighted graphs.

Now, we need to pick the path with the smallest length from a list. This solution is going to come together quickly!

Use aggregate to Find the Best Solution

As you might imagine, Prolog users frequently must choose the best solution to a problem. That means there are some good libraries for doing what we want. We're going to try the aggregate[4] predicate. As you might expect, it's another meta-predicate, meaning a predicate that takes a predicate as an argument. Open up the documentation and give it a brief look. The predicate takes three different arguments:

Template
: A pattern match defining the shape of each Goal result.

Goal
: The goal we want to aggregate.

Result
: Th end result.

Both the first and third arguments will use one of the aggregate goals from min, max, count, sum, set, or bag. You'd use min or max to pick the best goal, bag and set to capture all goals, count to count the total number of solutions, and sum to aggregate the total size of all solutions.

To use the library, we'll need to use the library. Add this line to the top of your database:

```
:- use_module(library(aggregate)).
```

That's the only new module we need. Now, let's use the code. Because we want a single minimal solution, add this code to your database:

```
shortest_path(From, To,  Path, Length) :-
  aggregate(min(L, P), path_with_length(From, To, P, L), min(Length, Path)).
```

4. https://www.swi-prolog.org/pldoc/man?section=aggregate

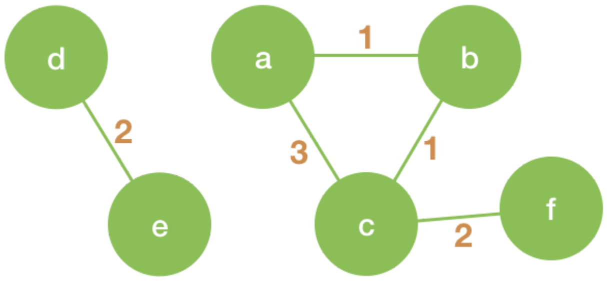
Beautiful! We can now compute the single shortest path with one call, like this:

```
?- [db].
true.
?- shortest_path(a, f, Path, Length).
Path = [a, c, f],
Length = 3.
```

We get one path, the shortest of the two! That's exactly what we want. Let's see how we might apply that solution to a weighted predicate.

Weighted Graphs

To transform a graph to a weighted one, all you need is to add a *weight* to each edge. A weight is the quantifiable price or benefit of visiting a node by way of an edge. Let's add weights to our graph, like this:



Notice that each edge has a weight. It's just a number that might stand for time, miles, gallons of fuel, or anything else representing the cost or benefit of traveling from one node to another.

We can represent that weighted graph. Since our code is going to change to accommodate weights, let's create a whole new database in a directory called weights. We'll create the edges with the weights in the previous image. While we're at it, we'll define connected and use the aggregate module we need. Drop this into db.pl:

```
:- use_module(library(aggregate)).

edge(a, b, 1).
edge(a, c, 3).
edge(b, c, 1).
edge(c, f, 2).
edge(d, e, 2).

connected(A, B, Weight) :-
  edge(A, B, Weight); edge(B, A, Weight).
```

Easy enough. We drop an additional attribute with the weight into the database. Let's deal with the sticky part of this problem, working with path and go.

Carry an Extra Accumulator for Weight

Now we come to the fun part. Tracking a weight is not really much different from tracking a length. We can just tack on a weight argument for the final result, and drop in an accumulator to track weight as we go.

It's a little noisy, but works just fine. Work the extra weight argument into path, like this:

```
path(From, To, Path, Weight) :-
       go(From, To, [From], Result, 0, Weight),
       reverse(Path, Result).
```

That's not bad at all. We have the extra argument, and we must pass the initial accumulator of 0 and the final Weight argument through to the go clause. We'll need to modify go to handle the extra information:

```
go(From, To, Seen, [To|Seen], PartialWeight, Weight) :-
  connected(From, To, W),
  Weight is PartialWeight + W.

go(From, To, Seen, Path, PartialWeight, Weight) :-
  connected(From, Between, W),
  Between \== To,
  \+member(Between, Seen),
  IncrementalWeight is PartialWeight + W,
  go(Between, To, [Between|Seen], Path, IncrementalWeight, Weight).
```

This rule is tedious and noisy, but not overly complicated. The base clause works just like our previous go version, but with a weight and an accumulator. PartialWeight holds the weight of Seen nodes, and the final Weight tracks with the weight of the full Path.

The recursive clause is similar. It picks up the incremental weight W from connected, and adds that weight to the accumulator as we recurse. Take a few moments to check out the logic, then try it out:

```
?- [db].
true.

?- path(d, e, Path, Weight).
Path = [d, e],
Weight = 2 ;
false.

?- path(a, f, Path, Weight).
Path = [a, b, c, f],
Weight = 4 ;
Path = [a, c, f],
Weight = 5 ;
false.
```

Wonderful! Prolog correctly filters out the longer solution. Let's see what it takes to aggregate the solution.

Aggregate Based On Weights

Keep in mind that we don't need a path_with_length to compute our weighted clause. We merged the weights into the path predicate. We'll roll up the final value based on those weights rather than lengths. All we need is to roll the result up in our accumulator, like this:

```
minimum_path(From, To,  Path, Weight) :-
  aggregate(min(W, P), path(From, To, P, W), min(Weight, Path)).
```

Also keep in mind that you might want to optimize the longest, or most profitable routes. Let's say we wanted to maximize the weights:

```
maximum_path(From, To,  Path, Weight) :-
  aggregate(min(W, P), path(From, To, P, W), min(Weight, Path)).
```

And Prolog will do the rest:

```
?- [db].
true.

?- maximum_path(a, f, Path, Weight).
Path = [a, c, f],
Weight = 5.

?- minimum_path(a, f, Path, Weight).
Path = [a, b, c, f],
Weight = 4.
```

You can see the possibilities. Our accumulated weight based on a simple formula is only one possible way we can calculate the value of a route. Weights can represent any formula we want. With the variety of aggregations available, we have a rich set of options available for expressing our graphs.

This chapter has been long and demanding, but I should point out a couple of resources. Make sure you watch the part 3 video for Prolog and work through the project. The video will walk you through some SWI Prolog APIs for working with graphs. This video will show you some alternative ways for thinking about graphs. The project will walk you through solving a maze! This problem is an iconic problem of programming contests.

We've done enough for this unit so it's time to wrap up.

Try It Yourself

The Joe Armstrong celebration is in full swing! Joe often talked about the hidden computing languages, those used to do constraint-logic programming. Many times at conferences, I heard him attempt to steer novices toward this lucrative field!

Graphs and Prolog are inextricably linked. Directed graphs show connections in one direction, like a twitter follower network or a map of downtown one-way roads. Other graphs are not directed, and represent distributed networks and friend-based social networks like Facebook or LinkedIn.

The problems we can solve range from finding how to get from Austin to Amarillo, to showing how to best deliver packages to eight customers in the minimum time, or while expending minimal fuel.

Now, it's time to flex your own programming muscles. Let's find some problems.

Your Turn

Once again, we're going to abandon Exercism to find the Prolog problems that best reflect the language. Consider this map.

city	city	miles
Austin	San Antonio	80
Austin	Dallas	200
Austin	Houston	170
Houston	San Antonio	200
Dallas	Houston	240

Use weighted graphs to answer the following questions.

- Easy: What's the shortest distance from San Antonio to Dallas?
- Easy: What's the longest path from Austin to San Antonio that doesn't visit the same city twice?
- Medium: If a Tesla gets 140 MPG and a Hummer gets 20 MPG, given weighted edges for the above table and the facts like `mpg(tesla, 140)`, how do you compute weighted paths for fuel costs?
- Hard: What's the shortest delivery path that reaches all cities, and winds up in the same place? (Hint: You may need to recursively exercise shortest-paths, removing all visited cities from your deliveries path.)

Next Time

In Chapter 4, Schedules and Code Organization, on page 53, we'll wrap things up by building a scheduler. We'll build a schedule for a sports team, but the same techniques can build schedules for conferences, project teams, or classrooms. I hope you're enjoying Prolog as much as I am.

We'll see you in two weeks!

CHAPTER 4

Schedules and Code Organization

In a world obsessed with time, schedules are the centerpiece for any business selling services, transportation, or education. Schedules are demanding for most languages because the programmer must balance so many different elements at the same time. When dealing with the inevitable conflicts that arise, things go from bad to worse.

Prolog is a fantastic language for building schedules because of its ability to deal with all of the variables. As we express constraints, Prolog can backtrack to avoid them.

In this chapter, we're going to be dealing with schedules. We'll start with pools of resources using the same kinds of strategies we used to solve the map coloring problem. Then we'll make some trivial changes to our program to deal with conflicts.

We'll learn some interesting Prolog techniques to return only a single result rather than all of them. When we've done so, we'll shift our attention to presenting clean output for our readers.

We'll solve a basic scheduling problem. We'll schedule ball games for a children's sports league, putting teams in time slots. The problem will be small enough to tackle in a single chapter, but complex enough to present a few interesting constraint problems. After all, we don't want little Maya to have to play two matches in the same day. We'll also allow for scheduling around the Otter's inevitable birthday party that's right at match time.

You'll be able to use the same techniques to solve more advanced scheduling problems. College class, exams, project teams, and more use the same techniques we'll present here. As a bonus, we'll pay special attention to how to map Prolog rules onto data you can get from input files.

There's a lot to cover so let's get busy!

Schedule Teams on a Field

This chapter's problem is a little league team scheduler. Our league will have five teams. Each team will play all of the other teams twice, with each team getting a chance to be the home team. The teams will play on one field that has both morning and afternoon slots available.

Let's think about the general landscape of scheduling. This problem is a *logic constraint* problem. Every scheduling problem has at least four elements. *Actors* need access to a *resource* at a *time*, despite *constraints*.

Represent Possibilities

All of the teams must play each other in some order. Let's think about the list of possibilities as a deck of cards. We'll let Prolog build our deck of possible matches, and then we'll deal from that deck into hands representing our schedule.

We can't expect our league managers to sling code, so let's make it easy to customize the program. For now, we'll settle for including a separate Prolog file that has the teams and the days we'll play. We'll start with two leagues, the Tots and the Munchkins.

Create a directory called teams and create a file for tots.pl that looks like this:

```
teams(["comets", "asteroids", "rockets", "stars"]).
days(["July 2", "July 9", "July 16", "July 23", "July 30", "August 6"]).
```

And now munchkins.pl:

```
teams(["otters", "monkeys", "parrots", "penguins"]).
days(["May 2", "May 9", "May 16", "May 23", "May 30", "June 6"]).
```

With the leagues set up, now we can start to work on the basic predicates that organize those teams into matches. We'll use a simple team predicate, and then roll those up into a match predicate, and then use those to find all possible matches.

A common way to solve a scheduling problem is to explore possible combinations of two of the three from resources, time slots, and actors. Then, you can look for conflicts, schedule the slot, and backtrack if necessary. The findall predicate is a good way to find all possibilities for such an approach.

Open up a third file, db.pl, and key this in:

```
:- consult(tots).

team(Team) :-
```

```
  teams(Teams),
  member(Team, Teams).

match(A, B) :-
  team(A),
  team(B),
  dif(A, B).

potential_matches(Matches) :-
  findall((A, B), match(A, B), Matches).
```

First we import our league. That gives us access to the teams and days predicates from the previous file. From here on, we're describing our solution.

The next job is to define teams and matches. A team is a member of the teams, and a match is two teams that are different from one another.

Finally, we find all valid potential matches, and roll them up with findall. We can try that much:

```
Matches = [("comets", "asteroids"),  ("comets", "rockets"),
           ("comets", "stars"),  ("asteroids", "comets"),
           ("asteroids", "rockets"),  ("asteroids", "stars"),
           ("rockets", "comets"),  ("rockets", "asteroids"),
           (..., ...)|...].
```

That does everything we need. We can't see all of the potential solutions, but we can tell that Prolog is creating a list of matches. We'll declare victory and move on. With the teams and matches fully represented, we can think of ways to deal with time.

Represent Time Concepts

In our team scheduler, *teams* want to play matches on *fields* at a *time*, and avoiding *conflicts*. In our example, each day we'll have two games played. One game will be in a morning slot, and one will be in an afternoon slot. We'll only have one field.

First, we need to decide how to represent each of our concepts. The time concepts we need to account for are:

- A game day is in a valid list of days
- A game day schedule has a morning match, an afternoon match, and a game day
- A game slot is represented by the position of a match in a day schedule
- A schedule is a list of all game day schedules.

Most of these concepts are built into the way we're representing our schedule. A day is an integer in a day's schedule, and a morning or afternoon slot is just a match's place in a day's schedule.

Still, our days have a potential inconvenience. It will be easier for our *program* to specify a day as an integer. It will be easier for our *users* to see a full date string. Let's provide a day predicate for tying both concepts together by labeling an integer as a day from the days list, like this:

```
day(Day, Label) :-
  days(Days),
  nth1(Day, Days, Label).
```

That simple predicate unifies Days with our list, and then unifies the label with the nth item. nth1 tells Prolog to start indexes at element one instead of element zero.

With the team, match, and time concepts firmly in hand, we can safely move on. We're ready to recursively build our schedule.

Schedule Possibilities

Many Prolog schedulers work by scheduling one slot from a list of possible combinations, then reducing the list of possible combinations, and scheduling the next one. That's the approach we'll use here. As we work through our potential_matches, we'll schedule a day's worth of two games and then remove those two matches from the PotentialMatches list.

One of the trickiest parts of the scheduling problem is finding and representing constraints in code.

Establish the Core Constraints

Our primary constraints are putting a single team on a field, putting two teams in a day, and making sure we don't schedule the same team twice. Constraining the field and slots will be easy because we'll ask Prolog for one match for each slot. The other constraint is a little bit trickier. We need to make sure no team has to play twice:

```
different_teams((A, B), (C, D)) :-
  dif(A, C),
  dif(A, D),
  dif(B, C),
  dif(B, D).
```

We use pattern patching to extract the teams for the two matches for a given day. A and B can't match either of the teams in the second match. We build similar restrictions for C and D by comparing them to A and B from the first match. We'll be able to use this constraint directly in our schedule predicate.

Schedule One Day

It's time to build the heart of our solution, the schedule predicate. You've probably guessed that it will need an accumulator of games scheduled so far. It will also need a predicate with the matches yet to schedule and the schedule day. Add this code to your database:

```
schedule(Scheduled,  [], Scheduled, _).
```

This is our base clause, and it takes a lot of arguments! Here's what they do:

Acc
: The teams scheduled so far

PotentialMatches
: The matches we've not yet scheduled

Scheduled
: The result of scheduled matches

Day
: An integer representing the day

This clause means if there are no more teams to schedule, we're done with this clause, and the result is in Scheduled. Here's the recursive case:

```
schedule(Acc, PotentialMatches, Scheduled, Day) :-
  member(MatchA, PotentialMatches),
  member(MatchB, PotentialMatches),
  different_teams(MatchA, MatchB),
  subtract(PotentialMatches, [MatchA, MatchB], WithoutAB),
  NextDay is Day + 1,
  schedule([(MatchA, MatchB, Day)|Acc], WithoutAB, Scheduled, NextDay).
```

That's a bit trickier, but it's not too bad. MatchA and MatchB both come from PotentialMatches, and they must involve different teams. To prepare our recursion, we remove the matches MatchA and MatchB from our list, leaving WithoutAB. Then, we compute the next day, and we call our recursive clause, tacking on the day's schedule.

We already have a program that works, with a couple of quirks:

```
?- [db].
true.

?- potential_matches(PM), schedule([], PM, Scheduled, 1).
PM = [("comets", "asteroids"),  ...],
Scheduled = [
((rockets, asteroids), (stars, comets), 6),
((rockets, comets), (stars, asteroids), 5),
((asteroids, comets), (stars, rockets), 4),
((comets, stars), (asteroids, rockets), 3),
((comets, rockets), (asteroids, stars), 2),
((comets, asteroids), (rockets, stars), 1)].
```

I've tweaked the output by extracting some extraneous data and adding a couple of line feeds, but everything else is exactly what you'll see when you run it. It's a great foundation for our scheduler, but there are issues.

The results show up backwards, and that's awkward for our intended users. The scheduler also returns many solutions, but we only need one. Finally, invoking our clause is pretty demanding.

We can make a few improvements:

- Add a pretty API that calls the ugly API with an accumulator.
- Reverse the result.
- Show only a single result.
- Format the results.
- Deal with constraints.

We can deal with the first three of those problems with a public API. The other two can wait until later in the chapter.

Simplify the Invocation with an API

By now, you're likely familiar with the concept of calling our complex APIs having an accumulator with a helper function that smooths out the user's experience. In that helper, we can specify the accumulator, reverse the solution, and focus on a single result.

Let's add a schedule(Result) predicate that looks up potential members, and seeds the accumulators for matches scheduled, matches to schedule, and the number of days, like this:

```
schedule(Final) :-
  potential_matches(PM),
  schedule([], PM, Final, 1).
```

Now we can call it like this:

```
?- schedule(Final).
Final = [((rockets, asteroids), (stars, comets), 6),
         ((rockets, comets), (stars, asteroids), 5),
         ((asteroids, comets), (stars, rockets), 4),  ...].
 ...]
```

That's already better. Now, we can tweak the API to use a Prolog concept called a *point cut*, represented with a !. The point cut is a predicate that always succeeds, and keep Prolog from backtracking to try more solutions:

```
schedule(Final) :-
  potential_matches(PM),
  schedule([], PM, Final, 1),
  !.
```

Even better. The point cut succeeds, but turns off backtracking so that Prolog checks no more solutions after this one. If you run the program, you'll find that Prolog stops after representing the first solution, which is the exact behavior we want. We need one solution, not all of them.

Finally, we can tweak the program to reverse the result, like this:

```
schedule(Final) :-
  potential_matches(PM),
  schedule([], PM, Scheduled, 1),
  reverse(Scheduled, Final),
  !.
```

We're getting there! Now Prolog returns the results, in order:

```
?- schedule(Final).
Final = [
(("comets", "asteroids"), ("rockets", "stars"), 1),
(("comets", "rockets"), ("asteroids", "stars"), 2),
(("comets", "stars"), ("asteroids", "rockets"), 3),
(("asteroids", "comets"), ("stars", "rockets"), 4),
(("rockets", "comets"), ("stars", "asteroids"), 5),
(("rockets", "asteroids"), ("stars", "comets"), 6)].
```

That's an API we can work on. We still have two major items on our wish list, and we'll start working on them next. We need to provide a pretty representation of our schedule, and we need to deal with constraints.

Let's focus on the constraints!

Establish Constraints

Our program already has a few constraints of two types. Based on the structure of the program, the first constraint allows the program to only schedule two games per day, and the second constraint only allows one game per slot. We also have another constraint, one we call different_teams, that the same team can't play twice in a day. Let's take a deeper look at how our schedule predicate works, and how we might generalize it to handle more sophisticated requirements.

Here's the code we have in our program, this time in three chunks:

```
schedule(Acc, PotentialMatches, Scheduled, Day) :-
  member(MatchA, PotentialMatches),
  member(MatchB, PotentialMatches),

  different_teams(MatchA, MatchB),
  subtract(PotentialMatches, [MatchA, MatchB], WithoutAB),

  NextDay is Day + 1,
  schedule([(MatchA, MatchB, Day)|Acc], WithoutAB, Scheduled, NextDay).
```

The code in these three chunks describes our explicit schedule, but you can build many different kinds of schedules with chunks like these. Here's the purpose of each of the three pieces:

- First, we *choose an item to schedule*, based on possible solutions.
- Next, we *apply constraints to our chosen items*, reducing the pool of possible solutions.
- Finally, we *recursively schedule the remaining possibilities*.

That's the general shape that many scheduling solutions will take. Let's start our constraint programming by reshaping our code a bit.

Define a Constraints Block

If you've followed Programmer Passport or *Seven Languages in Seven Weeks [Tat10]*, you know that we love to take advantage of function names and structure to name concepts in code. The apply_constraints predicate is a great example. We can build a predicate whose explicit job is to apply constraints. In our case, applying constraints should do two things:

- It should fail if the chosen items to schedule fail to meet our constraints.
- It should reduce the pool of available teams.

That means our predicate will have the items we choose, the pool of potential solutions, and the reduced pool of solutions. It will apply our existing constraint to the two matches, and then reduce the remaining matches as we prepare to call the recursive clause. Add this new apply_constraints predicate, below different_teams:

```
apply_constraints(MatchA, MatchB, PotentialMatches, RemainingMatches, Day) :-
  different_teams(MatchA, MatchB),
  subtract(PotentialMatches, [MatchA, MatchB], RemainingMatches).
```

We're basically doing the same work we did in a separate predicate. We make sure the same teams can't play on the same day, and subtract the two matches from our pool of potential matches to schedule. Now, we can tweak the main clause of schedule, like this:

```
schedule(Scheduled,  [], Scheduled, _).
schedule(Acc, PotentialMatches, Scheduled, Day) :-
  member(MatchA, PotentialMatches),
  member(MatchB, PotentialMatches),
  apply_constraints(MatchA, MatchB, PotentialMatches, RemainingMatches, Day),
  NextDay is Day + 1,
  schedule([(MatchA, MatchB, Day)|Acc], RemainingMatches, Scheduled, NextDay).
```

That change alone makes our program easier to understand. We'll go ahead and pass the day through because we'll need it to process the constraints. The rest of the arguments are exactly what we'll expect. The main benefit is that we have a great place to process additional constraints. Our next job is to allow our individual leagues to add conflicts to schedules. That's the next topic.

Add Individual cant_play Constraints

Our constraints will have a positive constraint that's easier to reason about in the program, and a negative version that makes more sense when it's packaged with the schedule.

Add the following to both tots.pl and munchkins.pl:

```
cant_play(team, morning_or_afternoon, date).
```

Our league files will need to specify a cant_play predicate or our program won't work. We take the opportunity to add a little bit of documentation for whoever is building the file. We could use this clause directly, but it would be better to build a more general clause that represents our constraint.

Add a General can_play Clause

Next, we build our positive constraint called can_play. That rule should take a match, a game, and a slot, and make sure no cant_play clauses exist. This clause goes in db.pl, like this:

```
can_play((A, B), Slot, Day) :-
  day(Day, DayLabel),
  \+cant_play(A, Slot, DayLabel),
  \+cant_play(B, Slot, DayLabel).
```

This clause has two purposes. First, it flips us from a negative constraint to a positive one that will be easier to understand. Second, it has a hidden purpose. It converts from the integer our programs use to the string dates that are easier for us to read. The rule is trivial. There's a Label that matches

Day; there’s no cant_play clause for team A in the given Slot on Day; and there’s no cant_play clause for team A in the given Slot on Day.

Our constraint is ready to plug in. Let’s do that now.

Integrate the Solution

Now that our constraint is in place, we can integrate it into the overall solution. We know we’ll put it in the apply_constraints rule. Let’s add it now:

```
apply_constraints(MatchA, MatchB, PotentialMatches, RemainingMatches, Day) :-
  different_teams(MatchA, MatchB),
  can_play(MatchA, morning, Day),
  can_play(MatchB, morning, Day),
  subtract(PotentialMatches, [MatchA, MatchB], RemainingMatches).
```

The rule is exactly the same, but with one exception. We invoke the can_play constraints for the individual tasks.

Now, let’s run our program once without adding constraints:

```
((comets, asteroids), (rockets, stars), 1),
((comets, rockets), (asteroids, stars), 2),
((comets, stars), (asteroids, rockets), 3),
((asteroids, comets), (stars, rockets), 4),
((rockets, comets), (stars, asteroids), 5),
((rockets, asteroids), (stars, comets), 6)
```

We’re ready to shoot our schedules out to the parents of the tots. But wait! We get an emergency phone call that the comets team is all going to the same birthday party, and they can’t play a morning match! That’s ok. We can add a constraint in tots.pl:

```
cant_play(comets, morning, "July 2").
```

And rerun the schedule:

```
((asteroids, rockets), (comets, stars), 1),
((comets, asteroids), (rockets, stars), 2),
((comets, rockets), (asteroids, stars), 3),
((asteroids, comets), (stars, rockets), 4),
((rockets, comets), (stars, asteroids), 5),
((rockets, asteroids), (stars, comets), 6)].
```

We can add other constraints to tots.pl too, and they’re cumulative:

```
cant_play(comets, morning, "July 2").
cant_play(stars, afternoon, "July 23").
```

And rerun:

```
((asteroids, rockets), (comets, stars), 1),
((comets, asteroids), (rockets, stars), 2),
((comets, rockets), (asteroids, stars), 3),
((stars, comets), (rockets, asteroids), 4),
((asteroids, comets), (stars, rockets), 5),
((rockets, comets), (stars, asteroids), 6)
```

And if we add too many constraints to tots.db for a successful schedule:

```
?- schedule(Final).
false.
```

Beautiful!

The schedule won't unify until we relax a constraint! The ability to provide both a generic framework and specific exceptions makes Prolog a powerful tool for describing nuanced problems. In the next section, we'll work on the last major feature. We'll touch up the output.

Write a Pretty Solution

We have one more task before our schedule is complete. We need to make the output prettier for the non-programmers who will read it. Here's our plan.

- Build a rule to print a schedule for a match.
- Build a rule to print the schedule for a whole day.
- Build a rule to print the entire schedule.
- Call the rule from the friendly schedule rule.

Let's get to it!

Print Matches

Printing in Prolog is easy, but tedious. We'll use two predicates with side effects. The write predicate will write data. The nl predicate will write a new line. Here's what it looks like to print a schedule for a match in db.pl:

```
print_match((Away, Home), Slot) :-
  write(Away),
  write(" play at "),
  write(Home),
  write(" in the "),
  write(Slot),
  nl().
```

We'll dodge the more powerful but more complicated format predicate for now and simply print the results with the basics. We write the names of the Home and Away teams with the Slot, with some prose interspersed. We close this section with a newline using nl.

Print Both Matches

Next in db.pl, let's recursively call print_match, like this:

```
print_matches([], _).
print_matches([Match|Teams], [Slot|Slots]) :-
  print_match(Match, Slot),
  print_matches(Teams, Slots).
```

The base case does nothing. the main case prints the first Match, and then recursively prints the rest of the matches. We're almost done. Now, let's print a whole schedule.

Print the Schedule

With the lions share of the complexity done, we just need to roll it all up. We can make use of the predicates we've already written. Add this code to db.pl:

```
print_schedule([]).
print_schedule([(Morning, Evening, MatchDay)|Rest]) :-
  day(MatchDay, DayLabel),
  write("On "),
  write(DayLabel),
  write(":"),
  nl(),
  print_matches([Morning, Evening], [morning, afternoon]),
  nl(),
  print_schedule(Rest).
```

Printing an empty schedule does nothing. Printing a schedule unifies the MatchDay to the user friendly DayLabel, then writes some heading prose, then prints the matches for a day, and then prints a new line followed by the rest of the schedule. None of this is complex, especially since we're using the two old-style Prolog predicates of write and nl to print. It's time to try it out.

Test Drive the Full Scheduler

Let's take it for a test drive:

```
?- [db].
true.

?- schedule(Final).
On May 2:
otters play at monkeys in the morning
parrots play at penguins in the afternoon

On May 9:
otters play at parrots in the morning
monkeys play at penguins in the afternoon
```

```
On May 16:
otters play at penguins in the morning
monkeys play at parrots in the afternoon

On May 23:
monkeys play at otters in the morning
penguins play at parrots in the afternoon

On May 30:
parrots play at otters in the morning
penguins play at monkeys in the afternoon

On June 6:
parrots play at monkeys in the morning
penguins play at otters in the afternoon

Final = [..].
```

And that's a schedule that will work just fine. Each match is presented in clear English. We could tweak the schedule to add a tabular form and tweak the capitalization of our teams, but we are approaching the point of diminishing returns. It's time to claim victory and bounce the remaining problems to you, the reader to figure out.

It's time to wrap up and try out the concepts you've learned.

Try It Yourself

The Joe Armstrong continues! Joe loved talking about radical ideas for companies. I can remember Joe's unbounded energy from a conference in Sweden when we kicked around several company ideas related directly to building schedules or solving similar constraint problems in Prolog, just for the fun of it. Now, you can see what made him so excited.

Schedules and Prolog go together like peas and carrots. Prolog's ability to backtrack when constraints limit choices allow the programmer to build code that doesn't have to consider such sticky details. Our scheduler works like many others. First, we build a list of possible resources and users. Then, we schedule one set of them and apply constraints. Finally, we reduce the pool of possible solutions and build the rest of the schedule recursively.

We organize our code by considering the possible users of each block of code. Doing so let us separate the concerns of the program. In our case, we put the rules for each league in its own file and the general solver in another. The generic scheduler had pieces to iterate through possibilities. For each scheduling unit, we apply constraints, and schedule one unit. Then, we reduce the possibilities and schedule the rest of the list.

Prolog schedulers can cover sports, like our example problem. Other applications of schedulers are project teams with constrained skills and resources or classrooms or exams with rooms of various sizes and teachers. The sky is the limit.

Now, it's time to get down to the business of scheduling. Let's find some problems.

Your Turn

Once again, we're going to abandon Exercism to focus on problems dealing with schedules.

Modify our program to handle:

- Easy: A shortened league. Teams play each other exactly once.
- Medium: Six teams on three fields.
- Medium: A season that's arbitrarily long. Try to keep the number of times teams play relatively even.

Hard: Try to build this scheduler:

- A classroom scheduler that balances teachers, classrooms, and time slots. Use the following kinds of rules:

```
teaches(subject, teacher).

% Examples:
teaches(algebra, stephens).
teaches(geometry, stephens).
teaches(english, perez).
teaches(spanish, perez).

room(room, occupancy).

% Examples:
room(101, 20).
room(102, 20).
room(103, 20).
room(104, 30).

class(subject, size)

% Examples:

class(english, 30).
class(spanish, 20).
class(algebra, 30).
class(geometry, 20).

time_slots([mw_morning, mw_afternoon, tt_morning, tt_afternoon]).
```

Consider how you might add other constraints as well.

Next Time

It's hard to believe, but this final chapter in Prolog wraps up the first half of Programmer Passport! In the next language, we continue our Joe Armstrong tribute. We'll move on to Elixir, the functional language built on top of Erlang infrastructure. You'll find several groundbreaking ideas:

- Elixir is the first fully functional language we'll cover. The evolution from Prolog is natural.
- Because of it's Erlang ancestry, Elixir is a great language for building distributed infrastructure that's extremely reliable and scales well, both horizontally and vertically.
- Elixir is an excellent platform for building interactive web applications without JavaScript using a framework called Phoenix LiveView.
- Elixir is an ideal language for the Internet of Things based on reliability, high-level programming model, and tooling.

We'll see you in two weeks!