Extracted from:

Prolog

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

grox.io
Learning

Programmer Passport

Prolog

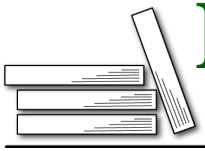
Bruce A. Tate

Edited by Jacquelyn Carter

Prolog

Bruce A. Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: L. Sakhi MacMillan
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-935-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 16, 2022

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. 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 toolbox: 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.