More queries, continued

Are any two foods the same color?

```
 ?-  food(F1),food(F2),color(F1,C),color(F2,C).
F1 = apple
F2 = apple
C = red ;

F1 = broccoli
F2 = broccoli
C = green ;
```

To avoid foods matching themselves we can specify "not equal" with \==.

```
?- food(F1), food(F2), F1 \== F2, color(F1,C), color(F2,C).
F1 = broccoli
F2 = lettuce
C = green
```

Remember that in order for a query to produce an answer for the user, all goals must succeed.

Etymology: \== symbolizes a struck-through "equals".

Sidebar: Predicates in operator form

The `op/3` predicate, which we may discuss later, allows a predicate to be expressed as an operator. These two queries are equivalent:

```
?- abc \== xyz.
Yes

?- \==(abc,xyz).
Yes
```

In fact, the sequence `abc \== xyz` causes Prolog to create a structure. `display/1` can be used to show the structure:

```
?- display(abc \== xyz).
\==(abc, xyz)
```

Ultimately, `abc \== xyz` means "invoke the predicate named \== and pass it two terms, `abc` and `xyz`".

Nested sidebar: `help/1` displays the documentation for a predicate. To learn about the `op` predicate, do this:

```
?- help(op).
```


Alternative representations

A given body of knowledge may be represented in a variety of ways using Prolog facts. Here is another way to represent the food and color information:

```
thing(apple, red, yes).
thing(broccoli, green, yes).
thing(carrot, orange, yes).
thing(dirt, brown, no).
thing(grass, green, no).
thing(lettuce, green, yes).
thing(rice, white, yes).
thing(sky, blue, no).
```

What is a food?

```
?- thing(X, _, yes).
X = apple ;
X = broccoli ;
X = carrot ;
...
```

The underscore designates an anonymous logical variable. It indicates that any value matches and that we don't want to have the value associated with a variable (and displayed).

Alternate representation, continued

Practice:

What is green that is not a food?

What color is lettuce?

What foods are orange?

What foods are the same color as lettuce?

```
thing(apple, red, yes).
thing(broccoli, green, yes).
thing(carrot, orange, yes).
thing(dirt, brown, no).
thing(grass, green, no).
thing(lettuce, green, yes).
thing(rice, white, yes).
thing(sky, blue, no).
```

Is `thing/3` a better or worse representation of the knowledge than the combination of `food/1` and `color/2`?

Answers:

```
thing(lettuce, Color, _).
thing(X, green, no).
thing(F, orange, yes).
thing(lettuce, Color, _), thing(Food, Color, yes).
```

Unification

Prolog has a more complex notion of equality than conventional languages.

The operators == and \== test for equality and inequality. They are roughly analogous to = / <> in ML and == / != in Ruby:

```
?- abc == 'abc'.
Yes

?- 3 \== 5.
Yes

?- abc(xyz) == abc(xyz).
Yes

?- abc(xyz) == abc(xyz,123).
No
```

Just like comparing tuples and lists in ML, and arrays in Ruby, structure comparisons in Prolog are "deep". Two structures are equal if they have the same functor, the same number of terms, and the terms are equal. Later we'll see that deep comparison is used with lists, too.

Think of == and \== as asking a question: is one thing equal (or not equal) to another.

Unification, continued

The = operator, which we'll read as "unify" or "unify with", can be used in a variety of ways.

If both operands are variables then `A = B` specifies that `A` must have the same value as `B`. Examples:

```
?- A = 1, B = abc, A = B.
No

?- A = 1, B = 1, A = B.
A = 1
B = 1 <CR>
Yes
```

Unification is not a question; it is a demand! Consider the following:

```
?- A = B, B = 1.
A = 1
B = 1
```

There are two unifications. The first unification demands that `A` must equal `B`. The second unification demands that `B` must equal `1`. In order to satisfy those two demands, Prolog says that `A` must be 1.

Unification, continued

At hand:

```
?- A = B, B = 1.
A = 1
B = 1
```

Variables are initially uninstantiated. After `A = B` we can say that `A` is unified with `B` but both `A` and `B` are still uninstantiated.

The unification `B = 1` instantiates `B` to the value `1`. Because `A` and `B` are unified, `B = 1` also causes `A` to be instantiated to `1`.

Another way to think about it is that unifications create constraints that Prolog must honor in order for a query to succeed. If the constraints can't be honored, the query fails.

```
?- A = 1, A = 2.
No

?- B = C, A = B, B = 1, C = 2.
No
```

DO NOT think of unification as assignment or comparison. It is neither!

Unification, continued

Here's how we might say that `S` must be a structure with functor `f` and term `A`, and that `A` must be `abc`:

```
?- S = f(A), A = abc.
S = f(abc)
A = abc
Yes
```

As a result of the unifications, `S` is instantiated to `f(abc)` and `A` is instantiated to `abc`.

A series of unifications can be arbitrarily complex. Here's a more complicated sequence:

```
?- Term1 = B, S = abc(Term1,Term2), B = abc, Term2=g(B,B,xyz).
Term1 = abc
B = abc
S = abc(abc, g(abc, abc, xyz))
Term2 = g(abc, abc, xyz)
```

Remember that a query specifies a series of goals. The above goals can be placed in any order. The result is the same regardless of the order.

Unification, continued

We can think of the query

```
?- food(carrot).
```

as a search for facts that can be unified with `food(apple)`.

Here's a way to picture how Prolog considers the first fact, which is `food(apple)`.

```
?- Fact = food(apple), Query = food(carrot), Fact = Query.
No
```

The demands of the three unifications cannot be satisfied simultaneously . The same is true for the second fact, `food(broccoli)`.

The third fact produces a successful unification:

```
?- Fact = food(carrot), Query = food(carrot), Fact = Query.
Fact = food(carrot)
Query = food(carrot)
Yes
```

The instantiations for `Fact` and `Query` are shown, but are no surprise.

Unification, continued

Things are more interesting when the query involves a variable, like `?- food(F).`

```
?- Fact = food(apple), Query = food(F), Query = Fact.
Fact = food(apple)
Query = food(apple)
F = apple
```

The query succeeds and Prolog shows that `F` has been instantiated to `apple`.

Unification, continued

Consider again this interaction:

```
?- food(F).
F = apple ;
F = broccoli ;
F = carrot ;
F = lettuce ;
F = rice ;
No
```

Prolog first finds that `food(apple)` can be unified with `food(F)` and shows that `F` is instantiated to `apple`.

When the user types semicolon `F` is uninstantiated and the search for another fact to unify with `food(F)` resumes.

`food(broccoli)` is unified with `food(F)`, `F` is instantiated to `broccoli`, and the user is presented with `F = broccoli`.

The process continues until Prolog has found all the facts that can be unified with `food(F)` or the user is presented with a value for `F` that is satisfactory.

Unification, continued

Following an earlier example, here's how we might view successful unifications with the query `?- food(F), color(F,C)`:

```
?- Fact1 = food(lettuce), Fact2 = color(lettuce,green),
   Query1 = food(F), Query2 = color(F,C),
   Fact1 = Query1, Fact2 = Query2.

C = green
F = lettuce
...
```

Only the interesting instantiations, for `F` and `C`, are shown above

What we see is that unifying `Fact1` with `Query1` causes `F` to be instantiated to `lettuce`.

`Query2`, which due to the value of `F` is effectively `color(lettuce,C)`, can be unified with `Fact2` if `C` is instantiated to `green`.

Unification and variable instantiation are cornerstones of Prolog.

Query execution

Goals, like `food(fries)` or `color(What, Color)` can be thought of as having four *ports*:

In the `Active Prolog Tutor`, Dennis Merritt describes the ports in this way:

call: Using the current variable bindings, begin to search for the facts which unify with the goal.

exit: Set a place marker at the fact which satisfied the goal. Update the variable table to reflect any new variable bindings. Pass control to the right.

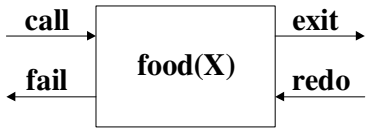
redo: Undo the updates to the variable table [that were made by this goal]. At the place marker, resume the search for a clause which unifies with the goal.

fail: No (more) clauses unify, pass control to the left.

Query execution, continued

Example:

```
?- food(X).
X = apple ;
X = broccoli ;
X = carrot ;
X = lettuce ;
X = rice ;
No
```

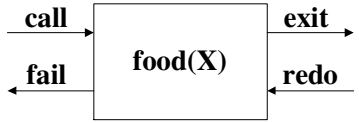



```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```


Query execution, continued

The goal `trace/0` activates "tracing" for a query. Here's what it looks like:

```
?- trace, food(X).
   Call: food(_G410) ? <CR>
   Exit: food(apple) ? <CR>
X = apple ;
   Redo: food(_G410) ? <CR>
   Exit: food(broccoli) ? <CR>
X = broccoli ;
   Redo: food(_G410) ? <CR>
   Exit: food(carrot) ? <CR>
X = carrot ;
```



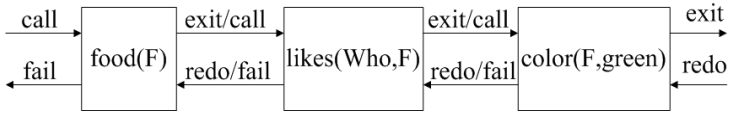

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

Tracing shows the transitions through each port. The first transition is a call on the goal `food(X)`. The value shown, `_G410`, stands for the uninstantiated variable `X`. We next see that goal being exited, with `X` instantiated to `apple`. The user isn't satisfied with the value and by typing a semicolon forces the redo port to be entered, which causes X, previously bound to `apple`, to be uninstantiated. The next food fact, `food(broccoli)` is tried, instantiating `X` to `broccoli`, exiting the goal, and presenting `X = broccoli` to the user. (etc.)

Query execution, continued

Query: Who likes green foods?

```
?- food(F), likes(Who,F), color(F, green).
```



Facts:

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).

likes(bob, carrot).
likes(bob, apple).
likes(joe, lettuce).
likes(mary, broccoli).
likes(mary, tomato).
likes(bob, mary).
likes(mary, joe).
likes(joe, baseball).
likes(mary, baseball).
likes(jim, baseball).

color(sky, blue).
color(dirt, brown).
color(grass, green).
color(broccoli,green).
color(lettuce, green).
color(apple, red).
color(tomato, red).
color(carrot, orange).
color(rose, red).
color(rice, white).
```

Try tracing it!

Producing output

The predicate `write/1` always succeeds and as a side effect prints the term it is called with. `writeln/1` is similar, but appends a newline.

```
?- write('apple'), write(' '), write('pie').
apple pie
Yes

?- writeln('apple'), writeln('pie').
apple
pie
Yes

?- writeln('apple\npie').
apple
pie
Yes
```


Producing output, continued

The predicate `format/2` is much like `printf` in other languages.

```
?- format('x = ~w\n', 10).
x = 10

Yes

?- format('label = ~w, value = ~w, x = ~w\n',
          ['abc', 10, 3+4]).
label = abc, value = 10, x = 3+4

Yes
```

If more than one value is to be output, the values must be in a list. We'll see more on lists later but for now note that we make a list by enclosing zero or more terms in square brackets. Lists are heterogeneous, like Ruby arrays

`~w` is one of many format specifiers. The "w" indicates to use `write/1` to output the value.

Use `help(format/2)` to see the documentation on `format`.

Producing output, continued

First attempt: print all the foods

```
?- food(F), format('~w is a food\n', F).
apple is a food

F = apple ;
broccoli is a food

F = broccoli ;
carrot is a food
```

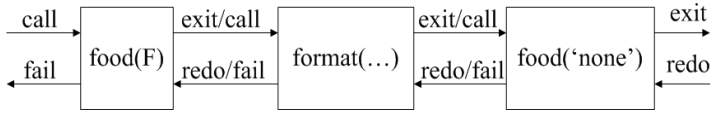
Ick—we have to type semicolons to cycle through the foods.

Any ideas?

Producing output, continued

Second attempt: Force alternatives by specifying a goal that always fails.

```
?- food(F), format('~w is a food\n', F), food('none').
apple is a food
broccoli is a food
carrot is a food
...
No
```

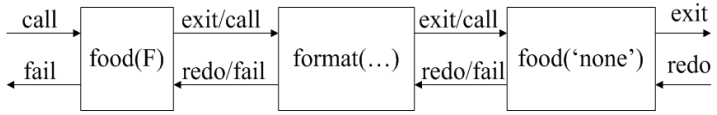


In essence, this query is a loop. `food(F)` unifies with the first food fact and instantiates `F` to its term, the atom `apple`. Then `format` is called, printing a string with the value of `F` interpolated. `food('none')` unifies with nothing, and fails. Control then moves left, into the `redo` port of `format`. `format` doesn't erase the output but it doesn't have an alternatives either, so it fails, causing the `redo` port of `food(F)` to be entered. `F` is uninstantiated and `food(F)` is unified with the next food fact in turn, instantiating `F` to `broccoli`. The process continues, with control repeatedly moving back and forth until all the food facts have been tried.

Backtracking

At hand:

```
?- food(F), format('~w is a food\n', F), food('none').
apple is a food
broccoli is a food
...
No
```



The activity of moving leftwards through the goals is known as *backtracking*.

We might say, "The query gets a food `F`, prints it, fails, and then backtracks to try the next food."

By design, Prolog does not analyze things far enough to recognize that it will never be able to "prove" what we're asking. Instead it goes through the motions of trying to prove it and as side-effect, we get the output we want. This is a key idiom of Prolog programming.

Backtracking, continued

At hand:

```
?- food(F), format('~w is a food\n', F), food('none').
apple is a food
broccoli is a food
...
No
```

It's important to note that different predicates respond to "redo" in various ways. With only a collection of facts for `food/1`, redo amounts to advancing to the next fact, if any. If there is one, the goal exits (goes to the right). If not, it fails (goes to the left).

A predicate might create a file when called and delete it on redo.

A sequence of redos might cause a predicate to work through a series of URLs to find a current data source.

A geometry manager might force a collection of predicates representing windows to produce a configuration that is mutually acceptable.

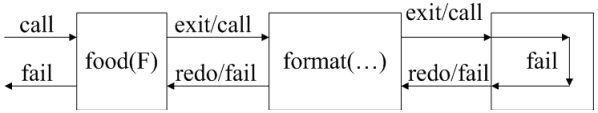
The predicate `fail`

The predicate `fail/1` always fails. It's important to understand that an always-failing goal like `food('none')` or `1==2` produces exhaustive backtracking but in practice we'd use `fail` instead:

```
?- food(F), format('~w is a food\n', F), fail.
apple is a food
broccoli is a food
carrot is a food
lettuce is a food
rice is a food

No
```

In terms of the four-port model, think of `fail` as a box whose call port is "wired" to its fail port:



Another way to think about it: `fail/1` causes control to make a U-turn.

`fail`, continued

The predicate `between/3` can be used to instantiate a variable to a sequence of values:

```
?- between(1,3,X).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

Problem: Print this sequence:

```
000
001
010
011
100
101
110
111
```

Extra credit:

```
10101000
10101001
10101010
10101011
10111000
10111001
10111010
10111011
```

No, you can't do `write('000\n001\n....')`!

Rules

`showfoods`: a simple "rule"

Horn clauses

Rules with arguments

Instantiation is how Prolog "returns" values

`showfoods`: a simple "rule"

Facts are one type of Prolog *clause*. The other type of clause is a *rule*. Here is a rule:

```
showfoods :- food(F), format('~w is a food\n', F), fail.
```

Just like facts, rules are not entered at the query prompt (it would be interpreted as a query!). Instead, they put into a file along with facts. The rules are loaded by consulting the file.

```
% cat facts1.pl
showfoods :- food(F), format('~w is a food\n', F), fail.

food(apple).
food(broccoli).
...

% pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.12)
?- [facts1].
...
```

The order doesn't matter—even though `showfoods/0` uses `food/1`, it can precede it.

"`facts1.pl`" is now a misnomer because it now contains rules, too. We'll ignore that.

Sidebar: Horn Clauses

Prolog borrows from the idea of *Horn Clauses* in symbolic logic. A simplified explanation is that a Horn Clause represents logic like this:

If Q_1, Q_2, Q_3, ..., Q_n, are all true, then P is true.

In Prolog we might represent a three-element Horn clause with this rule:

```
p :- q1, q2, q3.
```

The query

```
?- p.
```

which asks Prolog to "prove" `p`, causes Prolog to try and prove `q1`, then `q2`, and then `q3`. If it can prove all three, and can therefore prove `p`, Prolog will respond with `Yes`. (If not, then `No`.)

Note that this is an abstract example—we haven't defined the predicates `q1/0` et al.

`showfoods`, continued

At hand are the following rules:

```
p :- q1, q2, q3.

showfoods :- food(F), format('~w is a food\n', F), fail.
```

We can print all the foods with this query:

```
?- showfoods.
apple is a food
broccoli is a food
carrot is a food
lettuce is a food
rice is a food

No
```

In its unsuccessful attempt to "prove" `showfoods`, and thus trying to prove all three goals in the body of the `showfoods` rule, Prolog ends up doing what we want: all the foods are printed.

In other words, we send Prolog on a wild goose chase to get our work done!

showfoods, continued

Let's print all the foods three times:

```
 ?- showfoods, showfoods, showfoods.
apple is a food
broccoli is a food
carrot is a food
lettuce is a food
rice is a food

No
```

What's wrong?

showfoods, continued

At hand:

```
showfoods :- food(F), format('~w is a food\n', F), fail.

?- showfoods, showfoods, showfoods.
apple is a food
...
rice is a food
[Just one listing of the foods]

No
```

The showfoods rule above ALWAYS fails—there's NO WAY to get past the fail/0 at the end. We get the output we want but because the first showfoods goal ultimately fails (when the food facts are exhausted) Prolog doesn't try the second two goals—it can't even get past the first goal!

You might have noticed that Prolog concludes with No after printing the foods. That's because it fails to prove showfoods.

What can we do?

showfoods, continued

We've seen Prolog try all facts in turn for predicates like food/1, and color/2 in order to satisfy a query. Let's add a second clause to the predicate showfoods/0. The second clause is a fact:

```
showfoods :- food(F), format('~w is a food\n', F), fail.
showfoods.
```

Result:

```
 ?- showfoods.
apple is a food
broccoli is a food
carrot is a food
lettuce is a food
rice is a food

Yes     % IMPORTANT: Now it says Yes, not No!
```

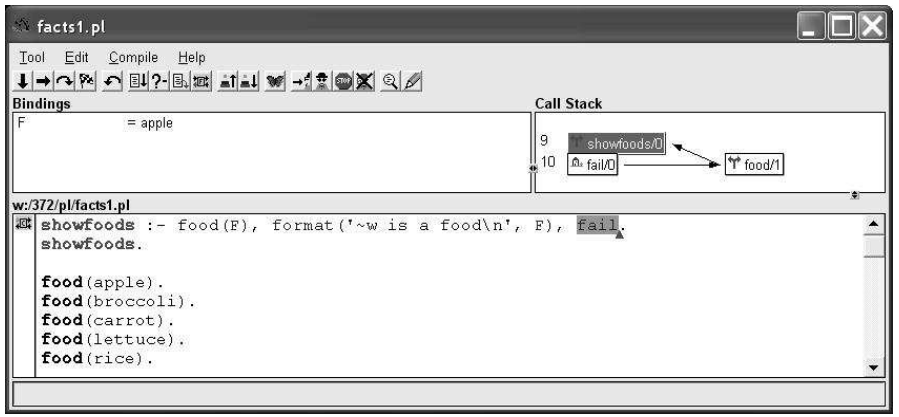
Prolog tried the two clauses for the predicate showfoods/0 in turn. The first clause, a rule, was ultimately a failure but printed the foods as a side-effect. Because the first clause failed, Prolog tried the second clause, a fact which is trivially proven.

Sidebar: Tracing execution with gtrace

On a Windows machine the predicate gtrace/0 activates a graphical tracer in a separate window. This query,

```
?- gtrace, showfoods.
apple is a food
```

and a few clicks on the right-arrow button, produces the output above and the display below.



Rules with arguments

Here is a one-rule predicate that asks if there is a food with a particular color:

```
foodWithColor(Color) :- food(F), color(F,Color).

?- foodWithColor(green).
Yes
```

To prove the goal `foodWithColor(green)`, Prolog first searches its clauses for one that can be unified with the goal. It finds a rule (above) whose *head* can be unified with the goal. Then the variable `Color` in the clause is instantiated to the atom `green`. It then attempts to prove `food(F)`, and `color(F, green)` for some value of `F`. The `Yes` response tells us that at least one green food exists, but that's all we know.

A failure:

```
?- foodWithColor(blue).
No
```

Ignoring the facts that you know are present, what are two distinct possible causes for the failure?

Can we do anything with it other than asking if there is a food with a particular color?

Rules with arguments, continued

```
foodWithColor(Color) :- food(F), color(F,Color).
```

If instead of an atom we supply a variable to `foodWithColor` the variable is instantiated to the color of each food in turn:

```
?- foodWithColor(C).
C = red ;
C = green ;
...
```

A very important rule:

> When a variable is supplied in a query and it matches a fact or the head of a rule with variable in the corresponding term, the two variables are unified. (Instantiating one instantiates the other.)

In the above case the variable `C` first has the value `red` because `C` in the query was unified with `Color` in the head of the rule, AND the goals in the body of the rule succeeded, AND `Color` was instantiated to `red`.

When we type a semicolon in response to `C = red`, Prolog backtracks and then comes up with another food color, `green`.

Instantiation is how Prolog "returns" values

```
foodWithColor(Color) :- food(F), color(F,Color).
```

Prolog has no analog for "`return x`" as is found in most languages. In Prolog there is no way to say something like this,

```
?- Color = foodWithColor(), writeln(Color), fail.
```

or this,

```
?- writeln(foodWithColor()), fail.
```

Instead, predicates "return" values by instantiating logical variables.

```
?- foodWithColor(C), writeln(C), fail.
red
green
...
```


Instantiation as "return", continued

Recall this predicate: **`between(1,10,X)`**

Here is what `help(between)` shows:

```
between(+Low, +High, ?Value)
  Low and High are integers, High >= Low. If Value is an integer,
  Low =< Value =< High. When Value is a variable it is successively
  bound to all integers between Low and High.
```

If an argument has a plus prefix, like `+Low` and `+High`, it means that the argument is an input to the predicate. A question mark indicates that the argument can be input or output. Thus, `between` can both generate values and test for membership in a range.

```
?- between(1,100,X), between(50,52,X), writeln(X), fail.
50
51
52

No
```