Prolog basics

A little background on Prolog

Facts and queries

"Can you prove it?"

Atoms and numbers

Predicates, terms, and structures

Fact/query mismatches

More queries

Operators are structures

Alternative representations

Unification

Query execution

A little background on Prolog

The name comes from "programming in logic".

Developed at the University of Marseilles (France) in 1972.

First implementation was in FORTRAN and written by Alain Colmeraurer.

Originally intended as a tool for working with natural languages.

Achieved great popularity in Europe in the late 1970s.

Was picked by Japan in 1981 as a core technology for their "fifth generation" project.

Prolog is a commercially successful language. Many companies have made a business of supplying Prolog implementations, Prolog consulting, and/or applications in Prolog.

There are many free implementations of Prolog available. We'll be using SWI-Prolog.

Facts and queries

A Prolog program is a collection of *facts*, *rules*, and *queries*. We'll talk about facts first.

Here is a small collection of Prolog *facts*:

```
% cat foods.pl
food('apple').
food('broccoli').
food('carrot').
food('lettuce').
food('rice').
```

These facts enumerate some things that are food. We might read them in English like this: "An apple is food", "Broccoli is food", etc.

A fact represents a piece of knowledge that the Prolog programmer deems to be useful. The name `food` was chosen by the programmer. One alternative is `edible('apple').`

`'apple'` is an example of an *atom*. Note the use of single quotes, not double quotes. We'll learn more about atoms later.

Facts and queries, continued

On `lectura`, we can load a file of facts into Prolog like this:

```
% pl                (That's "PL")
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.20)
...

?- [foods].        (Note that ".pl" is assumed; DON'T specify it!)
% foods compiled 0.00 sec, 1,488 bytes
                   (To save space the slides usually wont' show this blank line.)
Yes
```

Once the facts are loaded we can perform *queries*:

```
?- food('carrot').      % Don't forget the trailing period!!
Yes

?- food('peach').
No
```

Prolog responds based on the facts it has been given. People know that peaches are food but Prolog doesn't know that because there's no fact that says that.

A query can consist of one or more *goals*. The queries above consist of one goal.

Facts and queries, continued

Here's a fact:

```
food('apple').
```

Here's a query:

```
food('apple').
```

Facts and queries have the same syntax. They are distinguished by the context in which they appear.

If a line is typed at the interactive `?-` prompt, it is interpreted as a query.

When a file is loaded with `[`*`filename`*`]`, its contents are interpreted as a collection of facts.

Loading a file of facts is also known as *consulting* the file.

We'll see later that files can contain "rules", too. Facts and rules are two types of *clauses*.

For the time being use all-lowercase filenames with the suffix `.pl` for Prolog source files.

Sidebar: `food` in ML

An ML programmer might represent the food information like this:

```
fun food "apple" = true
  | food "broccoli" = true
  | food "carrot" = true
  | food "lettuce" = true
  | food "rice" = true
  | food _ = false;

- food "apple";
val it = true : bool

- food "peach";
val it = false : bool
```

What's another way to implement `food` in ML?

How might we implement `food` in Ruby?

Facts and queries, continued

An alternative to specifying an atom, like `'apple'`, in a query is to specify a variable. **In Prolog an identifier is a variable iff it starts with a capital letter**.

```
?- food(Edible).
Edible = apple <cursor is here>
```

A query like `food('apple')` asks if it is known that apple is a food.

The above query asks, "Tell me something that you know is a food."

Prolog uses the first `food` fact and responds with `Edible = apple`, using the variable name specified in the query.

If the user is satisfied with the answer `apple`, pressing *<ENTER>* terminates the query. Prolog responds with "Yes" because the query was satisfied.

```
?- food(Edible).
Edible = apple <ENTER>
Yes

?-
```


Facts and queries, continued

If for some reason the user is not satisfied with the response `apple`, an alternative can be requested by typing a semicolon, without *<ENTER>*.

```
?- food(Edible).
Edible = apple ;
Edible = broccoli ;
Edible = carrot ;
Edible = lettuce ;
Edible = rice ;

No
```

In the above case the user exhausts all the facts by repeatedly responding with a semicolon. Prolog then responds with "No".

It is very important to recognize that a simple set of facts lets us perform two distinct computations: (1) We can ask if something is a food. (2) We can ask what all the foods are.

How could we construct an analog for the above behavior in Java, ML, or Ruby?

"Can you prove it?"

One way to think about a query is that we're asking Prolog if something can be "proven" using the facts (and rules) it has been given.

The query

```
?- food('apple').
```

can be thought of as asking, "Can you prove that apple is a food?" It is trivially proven because we've supplied a fact that says that apple is a food.

The query

```
?- food('pickle').
```

produces "No" because based on the facts we've supplied, Prolog can't prove that pickle is a food.

"Can you prove it?", continued

Consider again a query with a variable:

```
?- food(F).     % Remember that an initial capital denotes a variable.
F = apple ;
F = broccoli ;
F = carrot ;
F = lettuce ;
F = rice ;
No
```

The query asks, "For what values of `F` can you prove that `F` is a food? By repeatedly entering a semicolon we see the full set of values for which that can be proven.

The collection of knowledge at hand, a set of facts about what is food, is trivial but Prolog is capable of finding proofs for an arbitrarily complicated body of knowledge.

Atoms

It was said that `'apple'` is an *atom*.

One way to specify an atom is to enclose a sequence of characters in single quotes. Here are some examples:

```
'   just testing   '
'!@#$%^&()'
'don\'t'               % don't
```

An atom can also be specified by a sequence of letters, digits, and underscores that begins with a lowercase letter. Examples:

```
apple          % Look, no quotes!
twenty2
getValue
term_to_atom
```

Is it common practice to avoid quotes and use atoms that start with a lowercase letter:

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


Atoms, continued

We can use `atom` to query whether something is an atom:

```
?- atom('apple').
Yes

?- atom(apple).
Yes

?- atom(Apple).     % Uppercase "A". It's a variable, not an atom!
No

?- atom("apple").
No
```

Numbers

Integer and floating point literals are *numbers*.

```
?- number(10).
Yes

?- number(123.456).
Yes

?- number('100').
No
```

Some things involving numbers don't work as you might expect:

```
?- 3 + 4.
ERROR: Undefined procedure: (+)/2

?- a = 5.
No

?- Y = 4 + 5.
Y = 4+5
Yes
```

We'll learn why later.

Predicates, terms, and structures

Here are some more examples of facts:

```
color(sky, blue).

color(grass, green).

odd(1). odd(3). odd(5).

number(one, 1, 'English').

number(uno, 1, 'Spanish').

number(dos, 2, 'Spanish').
```

We can say that the facts above define three *predicates*: `color/2`, `odd/1`, and `number/3`. The number following the slash is the number of *terms* in the predicate.

Predicates, terms, and structures, continued

A term is one of the following: atom, number, structure, variable.

Structures consist of a *functor* (always an atom) followed by one or more *terms* enclosed in parentheses.

Here are examples of structures:

```
color(grass, green)

odd(1)

number(uno, 1, 'Spanish')

equal(V, V)

lunch(sandwich(ham), fries, drink(coke))
```

The structure functors are `color`, `odd`, `number`, `equal`, and `lunch`, respectively.

Two of the terms of the last structure are structures themselves.

Note that a structure can be interpreted as a fact or a goal, depending on the context.

Fact/query mismatches

Here is a predicate `x/1`:

```
x(just(testing,date(7,4,1776))).
x(10).
```

Here are some queries:

```
?- x(abc).
No
?- x([1,2,3]).      % A list!
No
?- x(a(b)).
No
```

The predicate consists of two facts, one with a term that's a structure and another that's an integer. That inconsistency is not considered to be an error. The goals in the queries have terms that are an atom, a list, and a structure. There's no indication that those queries are fundamentally mismatched with respect to the terms in the facts.

Prolog says "No" in each case because nothing it knows about aligns with anything it's being queried about.

Fact/query mismatches, continued

At hand:

```
x(just(testing,date(7,4,1776))).
x(10).
```

It is an error if there's no predicate with as many terms as specified in a goal:

```
?- x(1,2).
ERROR: Undefined procedure: x/2
ERROR:     However, there are definitions for:
ERROR:         x/1
```

Note that a correction is suggested.

More queries

A query that requests green things:

```
?- color(Thing, green).
Thing = grass ;
Thing = broccoli ;
Thing = lettuce ;
No
```

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

A query that requests each thing and its color:

```
?- color(Thing, Color).
Thing = sky
Color = blue ;

Thing = dirt
Color = brown ;

Thing = grass
Color = green ;
...
```

We're essentially asking this: For what pairs of `Thing` and `Color` can you prove `color`?

More queries, continued

A query can contain more than one goal. This a query that directs Prolog to find a food `F` that is green:

```
?- food(F), color(F, green).
F = broccoli ;
F = lettuce ;
No
```

The query has two goals separated by a comma, which indicates conjunction—both goals must succeed in order for the query to succeed.

We might state it like this: "Is there an `F` for which you can prove both `food(F)` and `color(F, green)`?

Let's see if any foods are blue:

```
?- color(F, blue), food(F).
No
```

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

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

Note that the ordering of the goals was reversed. In this case the order doesn't matter.

Goals are always executed from left to right.

More queries, continued

Write these queries:

Who likes baseball?

Who likes a food?

Who likes green foods?

Who likes foods with the same color as foods that Mary likes?

```
food(apple).
...

color(sky,blue).
...

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).
```

Answers:

```
likes(Who,baseball).
likes(Who,X), food(X).
likes(Who,X), food(X), color(X,green).
likes(mary,F), food(F), color(F,C), likes(Who,F2), food(F2), color(F2,C).
```

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 equals" 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: Operators are structures

Expressions such as `3 + 4` and `a \== b` are simply convenient ways to specify a structure. `display/1` can be used to show a term as a structure:

```
?- display(3 + 4).
+(3, 4)
Yes

?- display(3 + 4 * 5).
+(3, *(4, 5))
Yes

?- display(a \== b).
\==(a, b)
Yes

?- display(1 + 2 \== 3 * 4).
\==(+(1, 2), *(3, 4))
Yes
```

Note that `display` has the side-effect of printing a line.

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

Formally, we say that the query

```
?- food(apple).
```

searches for facts that can be *unified* with `food(apple).` There is such a fact (`food(apple)`) and the goal succeeds because a unification is possible.

The query

```
?- food(F).
```

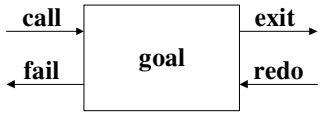
searches for facts that can be unified with `food(F).` The first food fact is unified with the query and the variable `F` is *instantiated* to the term (`apple`) of that fact.

If the user requests an alternative (via a semicolon) then `F` is *uninstantiated* and the search for a unifiable fact resumes, starting with the food fact that follows `food(apple).` The fact `food(broccoli)` is next and `F` is instantiated to `broccoli`.

If the user keeps typing semicolons the process continues until the food facts are exhausted.

Query execution

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



We think of control moving through the ports as follows:

call—Start searching for clauses (facts and rules) that unify with the goal.

exit—Goal is satisfied; mark the clause that unified and instantiate (or "bind") any logical variables used in the goal.

redo—Unbind variables bound at last exit and resume search, starting with the next clause.

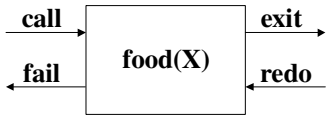
fail—Goal could not be satisfied with clauses left to search.

Remember that we haven't seen any rules yet, only facts.

Query execution, continued

Example:

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



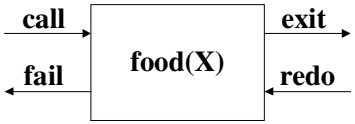

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```


Query execution, continued

Querying with `trace/0` activates "tracing" for the next query. Here's what it looks like:

```
?- trace.

Yes
?- 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 ;
```



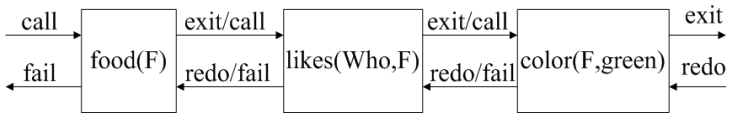

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

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 ("bound") 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 unbound. The next food fact, `food(broccoli)` is tried, binding `X` to `broccoli`, exiting the goal, and presenting `X = broccoli` to the user.

Query execution, continued

Query: Who likes green foods?

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



Facts:

```
food(apple).      likes(bob, carrot).      color(sky, blue).
food(broccoli).   likes(bob, apple).       color(dirt, brown).
food(carrot).     likes(joe, lettuce).     color(grass, green).
food(lettuce).    likes(mary, broccoli).   color(broccoli,green).
food(rice).       likes(mary, tomato).     color(lettuce, green).
                  likes(bob, mary).        color(apple, red).
                  likes(mary, joe).        color(tomato, red).
                  likes(joe, baseball).    color(carrot, orange).
                  likes(mary, baseball).   color(rose, red).
                  likes(jim, baseball).    color(rice, white).
```

Try tracing it!