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 won't 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

```
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 green things:

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

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

```
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).
```

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?

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 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, <u>all goals must succeed</u>.

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

```
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).
```

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?

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. <u>The result is the same regardless of the order.</u>

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

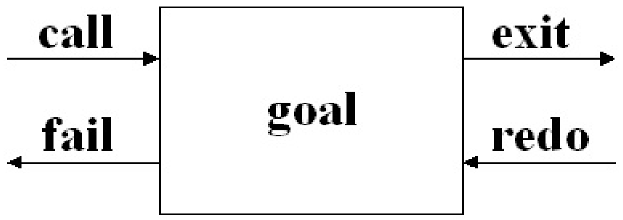
What we see is that unifying `Fact1` with `Query1` causes `F` to be instantiated to `lettuce`.

`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*:



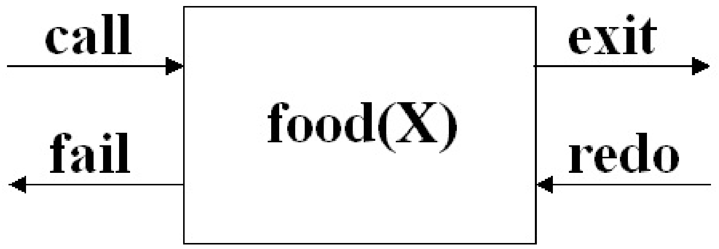
In the `Active Prolog Tutor`, Dennis Merritt describes the ports in this way:

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
```

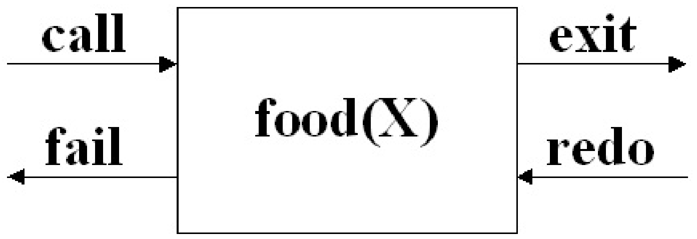


```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

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 ;
```



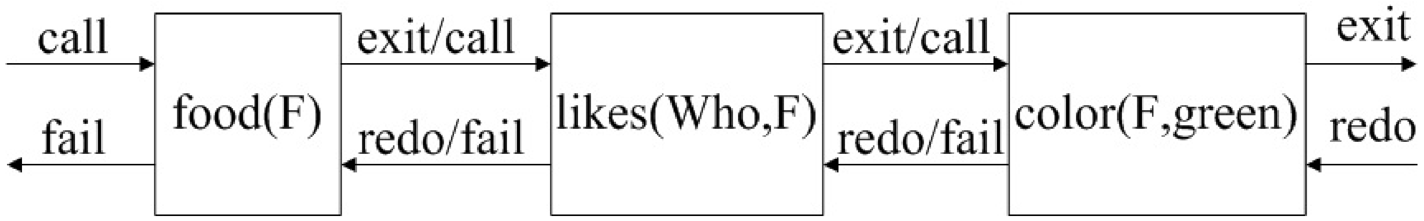
```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

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).       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!

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
```

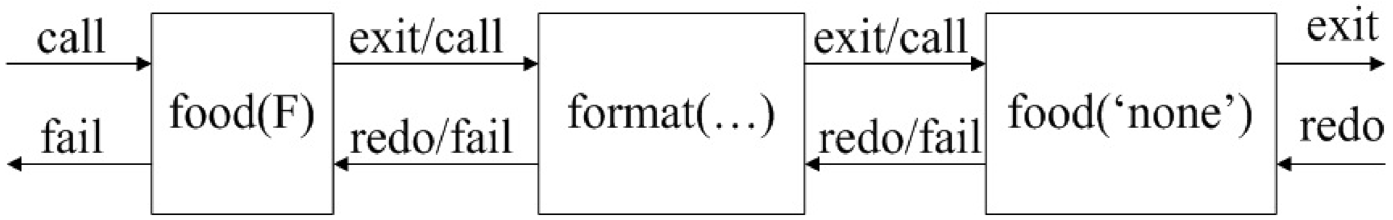
Ick—we have to type semicolons to cycle through the foods.

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
```

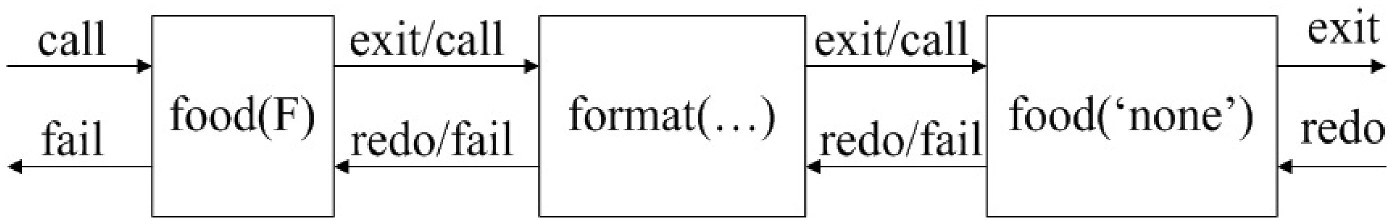


In essence, this query is a loop. `food(F)` unifies with the first food fact and instantiates `F` to its term, the atom `apple`. Then `format` is called, printing a string with the value of `F` interpolated. `food('none')` unifies with nothing, and fails. Control then moves left, into the `redo` port 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 <u>not</u> 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. <u>This is a key idiom of Prolog programming.</u>

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.

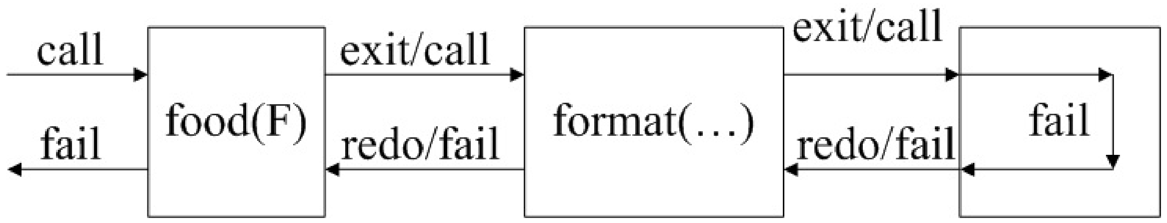
A geometry manager might force a collection of predicates representing windows to produce a configuration that is mutually acceptable.

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
```

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

Extra credit:

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

Rules

`showfoods`: a simple "rule"

Horn clauses

Rules with arguments

Instantiation as "return"

Computing with facts

Sidebar: Describing predicates

Arithmetic

Comparisons

`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!
```

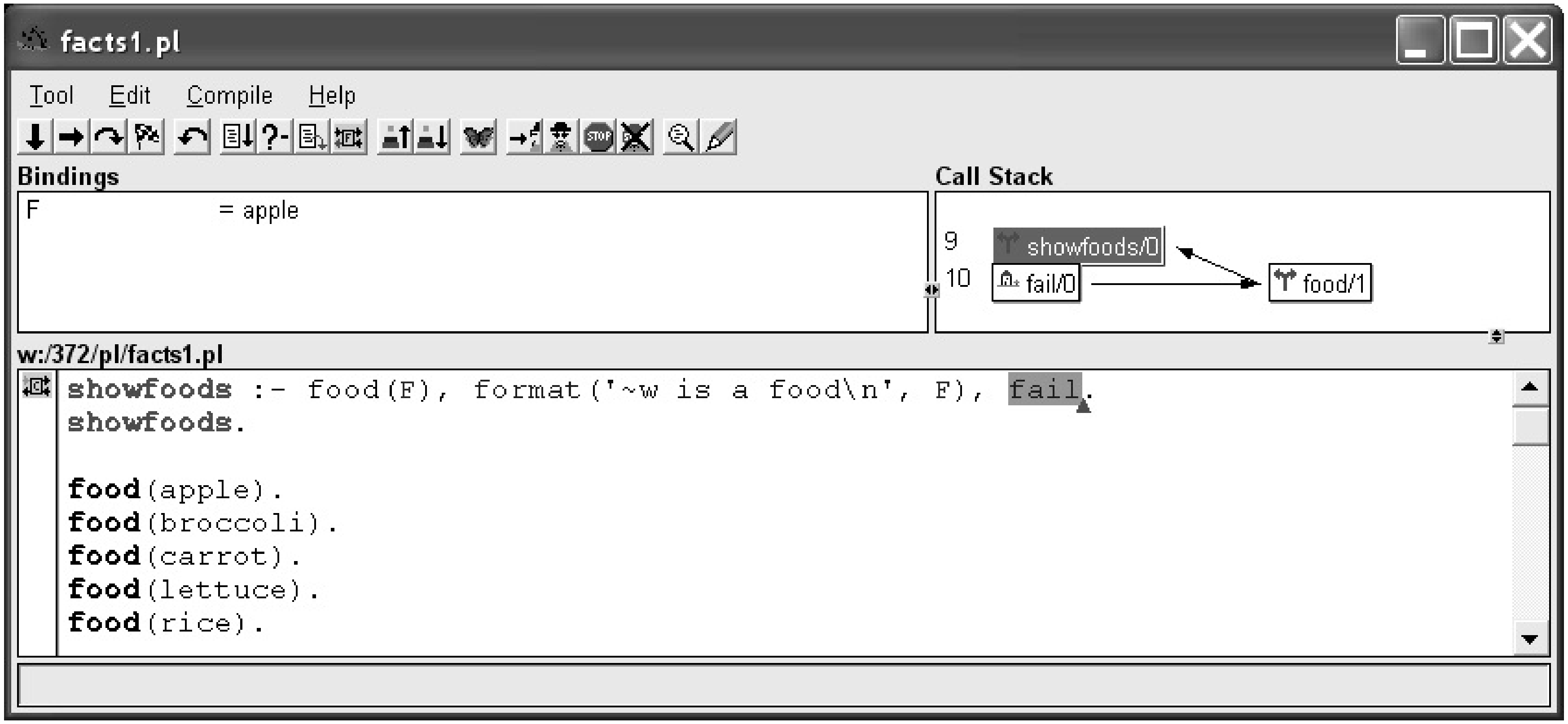
Prolog tried the two clauses for the predicate `showfoods/0` in turn. The first clause, a rule, was ultimately a failure but printed the foods as a side-effect. Because the first clause failed, Prolog tried the second clause, a fact which is trivially proven.

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.



```
F                = apple
```

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

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

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 ;
...
```

<u>A very important rule:</u>

> When a variable is supplied in a query and it matches a fact or the head of a rule with a 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 as "return"

```
    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

Here's a one-rule predicate that produces structures with food/color pairs:

```
get_fwc(Color, Result) :-
    food(F), color(F,Color), Result = fwc(F,Color).
```

Usage:

```
?- get_fwc(green, R).
R = fwc(broccoli, green) ;
R = fwc(lettuce, green) ;
No

?- get_fwc(_, R).
R = fwc(apple, red) ;
R = fwc(broccoli, green) ;
R = fwc(carrot, orange) ;
R = fwc(lettuce, green) ;
R = fwc(rice, white) ;
No
```

How could we make the predicate more concise?

Instantiation as "return", continued

First version and usage:

```
get_fwc(Color, Result) :-
    food(F), color(F,Color), Result = fwc(F,Color).

?- **get_fwc(green,R).**
R = fwc(broccoli, green)
```

Above, `R` is unified with `Result`, and `Result` is unified with `fwc(F,Color)`, (transitively) unifying `R` with `fwc(broccoli, green)`.

Instead of unifying `Result` with a structure in the body of the rule, we can eliminate the "middle-man" `Result` and specify the structure in the head, unifying it with whatever is specified in the query:

```
get_fwc(Color, fwc(F,Color)) :- food(F), color(F,Color).
```

Computing with facts

Consider these two nearly-identical predicates:

```
    equal_e(X,Y) :- X == Y.    % equality

    equal_u(X,Y) :- X = Y.     % unification
```

How does their behavior differ? What's something we could do with one that we couldn't do with the other?

Computing with facts, continued

```
equal_e(X,Y) :- X == Y.    % equality

equal_u(X,Y) :- X = Y.     % unification
```

Usage:

```
?- equal_e(abc,abc).
Yes

?- equal_e(abc,X).
No

?- equal_u(abc,X).
X = abc

?- equal_u(X,abc).
X = abc
```

What is the result of the following queries?

```
?- A = 1, equal_u(A,B).
?- equal_u(B, A), A = 1, equal_u(B, 2).
```

Can `equal_u` be shortened?

Computing with facts, continued

At hand:

```
    equal_u(X,Y) :- X = Y.
```

The right way to do it:

```
    equal_u(X,X).
```

`equal_u` is a fact that performs computation via unification and instantiation.

Below are some more facts that perform computation. Describe what they do.

```
    square(rectangle(S, S)).

    rotate(rectangle(W, H), rectangle(H, W)).

    width(rectangle(W, _), W).

    height(rectangle(_, H), H).
```

Computing with facts, continued

```
square(rectangle(S, S)).
rotate(rectangle(W, H), rectangle(H, W)).
width(rectangle(W, _), W).
```

Usage:

```
?- square(rectangle(3,3)).
Yes

?- rotate(rectangle(3,5),R2).
R2 = rectangle(5, 3)
Yes

?- rotate(rectangle(3,5),R2), width(R2, W).
R2 = rectangle(5, 3)
W = 5

?- rotate(muffler,M2).
No
```

Problem: Using only `rotate/2`, write `square/1`.

Computing with facts, continued

At hand:

```
    rotate(rectangle(W, H), rectangle(H, W)).
```

One way to think about a square is that it's a rectangle whose 90-degree rotation equals itself:

```
    square(R) :- rotate(R,R).
```

The thing to appreciate is that `rotate` describes a relationship between two `rectangle` structures. With a single fact Prolog can create a rotated rectangle, determine if one rectangle is a rotation of another, and more.

Sidebar: Describing predicates

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.

The documentation indicates that `between` can (1) generate values and (2) test for membership in a range.

```
?- between(1,10,X).
X = 1
...

?- between(1,10,5).
Yes
```

Describing predicates, continued

How would the arguments of these predicates be described?

```
    rotate(Rect1, Rect2)

    square(Rectangle)

    equal_u(X, Y)
```

Here is the synopsis for `format/2`:

```
    format(+Format, +Arguments)
```

Speculate: What does `sformat/3` do?

```
    sformat(-String, +Format, +Arguments)
```

Amusement: What does the following query do?

```
    ?- between(1,100,X), format('Considering: ~d\n', X),
       between(50,52,X), format('Found ~d\n', X), fail.
```

Arithmetic

Just as we saw with \==, Prolog builds structures out of arithmetic expressions:

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

?- display(1 / 2 * ( 3 + 4)).
*(/(1, 2), +(3, 4))
Yes

?- display(300.0/X*(3+A*0.7**Y)).
*(/(300.0, _G373), +(3, *(_G382, **(0.7, _G380))))
```

Unlike \==, there are no predicates for the arithmetic operators. Example:

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

?- *(10,20).
ERROR: Undefined procedure: * /2
```

Question: Why would predicates be awkward for arithmetic expressions?

Arithmetic, continued

The predicate `is/2` evaluates a structure representing an arithmetic expression and unifies the result with a logical variable:

```
?- is(X, 3+4*5).

X = 23

Yes
```

`is/2` is usually used as an infix operator:

```
?- X is 3 + 4, Y is 7 * 5, Z is X / Y.

X = 7
Y = 35
Z = 0.2
```

All variables in the structure being evaluated by `is/2` must be instantiated:

```
?- A is 3 + X.
ERROR: is/2: Arguments are not sufficiently instantiated
```

Arithmetic, continued

It is not possible to directly specify an arithmetic value as an argument of most predicates:

```
?- write(3+4).
3+4
Yes

?- 3+4 == 7.
No

?- between(1, 5+5, 7).
ERROR: between/3: Type error: `integer' expected, found `5+5'
```

Arithmetic, continued

A full set of arithmetic operations is available. Here are some of them:

`-X`	negation
`X+Y`	addition
`X-Y`	subtraction
`X*Y`	multiplication
`X/Y`	division—produces float quotient
`X//Y`	integer division
`X mod Y`	integer remainder *(Watch out for* `X is Y % Z`*—that's a comment!)*
`integer(X)`	truncation to integer
`float(X)`	conversion to float
`sign(X)`	sign of `X`: -1, 0, or 1

```
?- X is 43243432422342123234 / 77777777777777777.
X = 555.987

?- X is 10 // 3.
X = 3

?- X is e ** sin(pi).
X = 1.0
```

Arithmetic, continued

Here are some predicates that employ arithmetic. Remember that we have to "return" values via instantiation.

```
add(X, Y, Sum) :- Sum is X + Y.

around(Prev,X,Next) :- Prev is X - 1, Next is X + 1.

area(rectangle(W,H), A) :- A is W * H.

area(circle(R), A) :- A is pi * R ** 2.

?- add(3,4,X).
X = 7

 ?- around(P, 7, N).
P = 6
N = 8

?- area(circle(3), A).
A = 28.2743

?- area(rectangle(2*3, 2+2), Area).
Area = 24
```

Comparisons

There are several numeric comparison operators:

`X =:= Y`	numeric equality	
`X =\= Y`	numeric inequality	
`X < Y`	numeric less than	
`X > Y`	numeric greater than	
`X =< Y`	numeric equal or less than	(NOTE the order, not <= !)
`X >= Y`	numeric greater than or equal	

Just like `is/2`, they evaluate their operands. Examples of usage:

```
?- 3 + 5 =:= 2*3+2.
Yes

?- X is 3 / 5, X > X*X.
X = 0.6

?- X is random(10), X > 5, writeln(X).
6
```

Note that the comparisons produce no value; they simply succeed or fail.

Comparisons, continued

Some predicates for grading:

```
grade(Score,Grade) :- Score >= 90, Grade = 'A'.
grade(Score,Grade) :- Score >= 80, Score < 90, Grade = 'B'.
grade(Score,Grade) :- Score >= 70, Score < 80, Grade = 'C'.
grade(Score,Grade) :- Score < 70, Grade = 'F'.

print_grade(Score) :- grade(Score,Grade),
    format('~w -> ~w\n', [Score, Grade]).
```

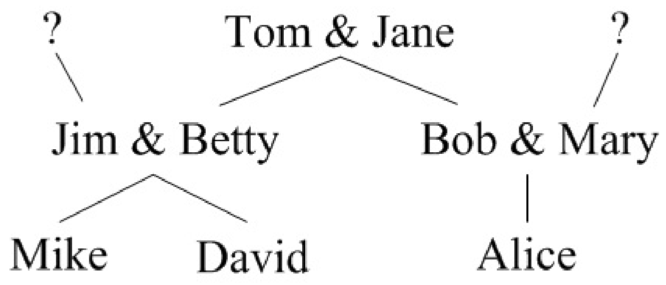
Usage:

```
?- grade(95,G).
G = 'A'
Yes

?- print_grade(80).
80 -> B
Yes

?- print_grade(55).
55 -> F
Yes
```

Another example of rules

Here is a set of facts describing parents and children:

```
male(tom).      parent(tom,betty).
male(jim).      parent(tom,bob).
male(bob).      parent(jane,betty).
male(mike).     parent(jane,bob).
male(david).    parent(jim,mike).
                parent(jim,david).
female(jane).   parent(betty,mike).
female(betty).  parent(betty,david).
female(mary).   parent(bob,alice).
female(alice).  parent(mary,alice).
```



`parent(P,C)` is read as "P is a parent of C".

Problem: Define rules for `father(F,C)` and `grandmother(GM,GC)`.

Another example, continued

```
    father(F,C) :- parent(F,C), male(F).
    mother(M,C) :- parent(M,C), female(M).

    grandmother(GM,GC) :- parent(P,GC), mother(GM,P).
```

Who is Bob's father?

For who is Tom the father?

What are all the father/child relationships?

What are all the father/daughter relationships?

What are the grandmother/grandchild relationships?

Problems: Define `sibling(A,B)`, such that "`A` is a sibling of `B`".
Using `sibling`, define `brother(B,A)` such that "`B` is `A`'s brother".

Another example, continued

```
sibling(A,B) :- father(F,A), mother(M,A),
               father(F,B), mother(M,B), A \== B.
```

Queries:

Is Mike a sibling of Alice?

What are the sibling relationships?

Who is somebody's brother?

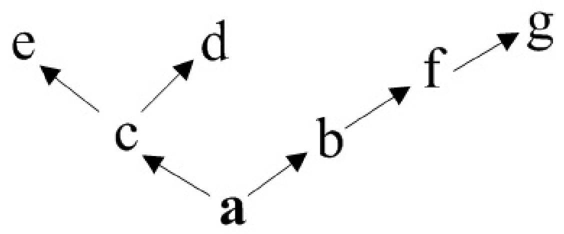
Is the following an equivalent definition of `sibling`?

```
sibling2(S1,S2) :- parent(P,S1), parent(P,S2), S1 \== S2.
```

Recursive predicates

Consider an abstract set of parent/child relationships:

```
parent(a,b).  parent(c,d).
parent(a,c).  parent(b,f).
parent(c,e).  parent(f,g).
```



If a predicate contains a goal that refers to itself the predicate is said to be recursive.

```
ancestor(A,X) :- parent(A, X).
ancestor(A,X) :- parent(P, X), ancestor(A,P).
```

"`A` is an ancestor of `X` if `A` is the parent of `X` or `P` is the parent of `X` and `A` is an ancestor of `P`."

```
?- ancestor(a,f).               % Is a an ancestor of f?
Yes

?- ancestor(c,b).               % Is c an ancestor of b?
No

?- ancestor(c,Descendant).     % What are the descendants of b?
Descendent = e ;
Descendent = d ;
No
```

Recursion, continued

A recursive rule can be used to perform an iterative computation.

Here is a predicate that prints the integers from 1 through N:

```
printN(0).
printN(N) :- N > 0, M is N - 1, printN(M), writeln(N).
```

Usage:

```
?- printN(3).
1
2
3

Yes
```

Note that we're asking if `printN(3)` can be proven. The side effect of Prolog proving it is that the numbers 1, 2, and 3 are printed.

Is `printN(0).` needed?

Which is better—the above or using `between/3`?

Recursion, continued

A predicate to calculate the sum of the integers from 1 to N:

```
sumN(0,0).
sumN(N,Sum) :-
    N > 0, M is N - 1, sumN(M, Temp), Sum is Temp + N.
```

Usage:

```
?- sumN(4,X).
X = 10

Yes
```

Note that this predicate can't be used to determine N for a given sum:

```
?- sumN(N, 10).
ERROR: >/2: Arguments are not sufficiently instantiated
```

Could we write `sumN` using `between/3`?

Sidebar: A common mistake with arithmetic

Here is the correct definition for `sumN`:

```
sumN(0,0).
sumN(N,Sum) :-
    N > 0, M is N - 1, sumN(M, Temp), Sum is Temp + N.
```

Here is a COMMON MISTAKE:

```
sumN(0,0).
sumN(N,Sum) :-
    N > 0, M is N - 1, sumN(M, Sum), Sum is Sum + N.
```

Unless `N` is zero, `Sum is Sum + N` fails every time!

Remember that `is/2` unifies its left operand with the result of arithmetically evaluating it's right operand. Further remember that unification is neither assignment or comparison.

Recursion, continued

A common example of recursion is a factorial computation:

```
    factorial(0,1).

    factorial(N,F) :-
       N>0,
       N1 is N-1,
       factorial(N1,F1),
       F is N * F1.
```

The above example comes from
http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/2_2.html

Near the bottom the page has an excellent animation of the computation `factorial(3,X).` TRY IT!

Generating alternatives with recursion

Here is a predicate to test whether a number is odd:

```
odd(N) :- N mod 2 =:= 1.
```

Remember that `=:=` evaluates its operands.

An alternative:

```
odd(1).
odd(N) :- odd(M), N is M + 2.
```

How do the behavior of the two differ?

Generating alternatives, continued

For reference:

```
    odd(1).
    odd(N) :- odd(M), N is M + 2.
```

Usage:

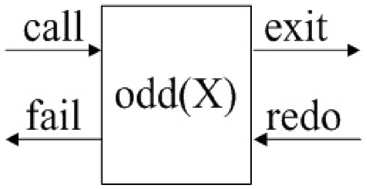
```
    ?- odd(5).
    Yes

    ?- odd(X).
    X = 1 ;
    X = 3 ;
    X = 5 ;
    ...
```

How does `odd(X).` work?

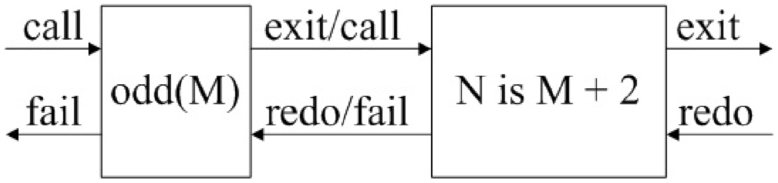
What does `odd(2).` do?

Generating alternatives, continued

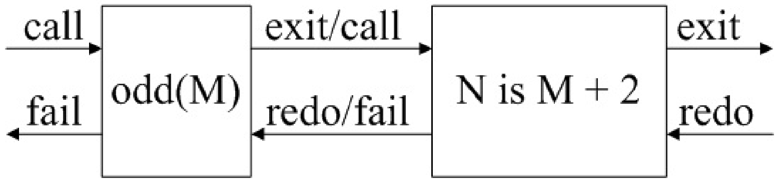
Query: `?- odd(X).`



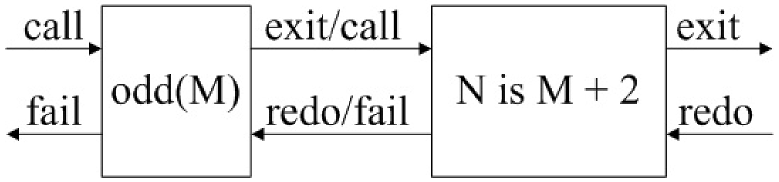
```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



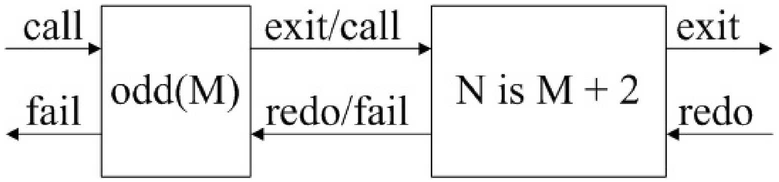
```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



Try: `?- gtrace, odd(X).` (Or `?- trace, odd(X).`)
`?- gtrace, odd(2).`

Generating alternatives, continued

For reference:

```
    odd(1).
    odd(N) :- odd(M), N is M + 2.
```

<u>With generative predicates like this one a key point to understand is that if an alternative is requested, another activation of the predicate is created.</u>

As a contrast, think about how execution differs with this set of clauses:

```
    odd(1).
    odd(3).
    odd(N) :- odd(M), N is M + 2.
```

Lists

Basics

Built-in list-related predicates

`member/2`

`append/3`

`findall/3`

Low-level list processing

Lists

A Prolog list can be literally specified by enclosing a comma-separated series of terms in square brackets:

```
[1, 2, 3]

[just, a, test, here]

[1, [one], 1.0, [a,[b,['c this']]]]
```

<u>Common mistake: Entering a list literal as a query is taken as a request to consult files</u>.

```
 ?- [abc, 123].
ERROR: source_sink `abc' does not exist
```

One way to "see" a list is to `write` it; another way is to unify it with a variable:

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

?- X = [a,1,b,2].
X = [a, 1, b, 2]
```

Lists, continued

Unifications can involve lists:

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

    ?- [X,Y] = [1,[2,[3,4]]].
    X = 1
    Y = [2, [3, 4]]

    ?- [X,Y] = [1].
    No

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

Note the similarity to ML patterns.

Lists, continued

What is produced by the following queries?

```
?- [A, B] = [X, A], X = a.

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

Write a predicate `empty(L)` that succeeds iff `L` is an empty list. BE SURE it succeeds only on lists and no other types.

Write a predicate `f(X)` that succeeds iff `X` is a list with one or three elements or `X` is an odd number.

Lists, continued

Answers:

```
?- [A, B] = [X, A], X = a.
A = a
B = a
X = a

?- [A,B,C] = [C,C,A].
A = _G296
B = _G296
C = _G296

empty([]).

f([_]).
f([_,_,_]).
f(N) :- number(N), N mod 2 =:= 1.
```

Built-in list-related predicates

Section A.1 in the SWI-Prolog manual (page 270 in the PDF) describes a number of predicates for list manipulation. Here are a few of them:

`length(?List, ?Len)` can be used to get the length of a list OR make a list of variables:

```
    ?- length([10,20,30],Len).
    Len = 3

    ?- length(L,3).
    L = [_G319, _G322, _G325]
```

Speculate: What is the result of `?- length(L,N).`?

`reverse(?List, ?Reversed)` unifies `List` with `Reversed`, a reverse of itself.

```
    ?- reverse([1,2,3], R).
    R = [3, 2, 1]

    ?- reverse([1,2,3], [1,2,3]).
    No
```

Speculate: What is the result of `?- reverse(X,Y).`?

List predicates, continued

`numlist(+Low, +High, -List)` unifies `List` with the integers from `Low` to `High`, inclusive:

```
?- numlist(5,10,L).
L = [5, 6, 7, 8, 9, 10]
```

`sumlist(+List, -Sum)` unifies `Sum` with the sum of the values in `List`, which must all be numbers or structures that can be evaluated with `is/2`.

```
?- numlist(1,5,L), sumlist(L,Sum).
L = [1, 2, 3, 4, 5]
Sum = 15

?- sumlist([1+2, 3*4, 5-6/7],X).
X = 19.1429
```

List predicates, continued

`atom_chars(?Atom, ?Charlist)` resembles `implode` and `explode` in ML:

```
?- atom_chars('abc',L).
L = [a, b, c]


?- atom_chars(A, [a, b, c]).
A = abc
```

List predicates, continued

`msort(+List, -Sorted)` unifies `List` with `Sorted`, a sorted copy of `List`:

```
?- msort([3,1,7], L).
L = [1, 3, 7]
```

If `List` is heterogeneous, elements are sorted in "standard order":

```
?- msort([xyz, 5, [1,2], abc, 1, 5, x(a)], Sorted).

Sorted = [1, 5, 5, abc, xyz, x(a), [1, 2]]
```

`sort/2` is like `msort/2` but also removes duplicates.

The `member` predicate

`member(?Elem, ?List)` succeeds when `Elem` can be unified with a member of `List`.

For one thing, `member` can be used to check for membership:

```
?- member(30, [10, twenty, 30]).
Yes
```

It can also be used to generate the members of a list:

```
?- member(X, [10, twenty, 30]).
X = 10 ;
X = twenty ;
X = 30 ;
No
```

Problem: Write a predicate `has_vowel(+Atom)` that succeeds iff `Atom` has a vowel.

```
?- has_vowel('ack').
Yes

?- has_vowel('pfft').
No
```

`member`, continued

Solution:

```
has_vowel(Atom) :-

    atom_chars(Atom,Chars),

    member(Char,Chars),

    member(Char,[a,e,i,o,u]).
```

How does it work?

The `append` predicate

The SWI manual states that `append(?List1, ?List2, ?List3)` succeeds when `List3` unifies with the concatenation of `List1`and `List2`.

```
    ?- append([1,2],[3,4,5],R).
    R = [1, 2, 3, 4, 5]
```

`append` is not limited to concatenation. Note what happens when we supply `List3`, but not `List1` and `List2`:

```
    ?- append(L1, L2, [1,2,3]).
    L1 = []
    L2 = [1, 2, 3] ;

    L1 = [1]
    L2 = [2, 3] ;

    L1 = [1, 2]
    L2 = [3] ;

    L1 = [1, 2, 3]
    L2 = [] ;
    No
```

append, continued

At hand:

```
?- append(L1, L2, [1,2,3]).
L1 = []
L2 = [1, 2, 3] ;

L1 = [1]
L2 = [2, 3] ;
...
```

Think of `append` as demanding a relationship between the three lists: `List3` must consist of the elements of `List1` followed by the elements of `List2`. If `List1` and `List2` are instantiated, `List3` must be their concatenation. If only `List3` is instantiated then `List1` and `List2` represent (in turn) all possible the ways to split `List3`.

What else can we do with `append`?

`append`, continued

Here are some more computations that can be expressed with `append`:

```
    starts_with(L, Prefix) :- append(Prefix,_,L).

    take(L, N, Result) :- append(Result, _, L), length(Result,N).

    drop(L, N, Result) :-
                   append(Dropped, Result, L), length(Dropped, N).
```

What does the following query do?

```
    ?- append(A,B,X), append(X,C,[1,2,3]).
```

`append`, continued

Here is a predicate that generates successive N-long chunks of a list:

```
chunk(L,N,Chunk) :-
    length(Chunk,N),
    append(Chunk,_,L).

chunk(L,N,Chunk) :-
    length(Junk, N),
    append(Junk,Rest,L),
    chunk(Rest,N,Chunk).
```

Usage:

```
?- chunk([1,2,3,4,5],2,Chunk).

Chunk = [1, 2] ;

Chunk = [3, 4] ;

No
```

"Leftovers" are discarded.

The `findall` predicate

`findall` can be used to create a list of values that satisfy a goal. Here is a simple example:

```
?- findall(F, food(F), Foods).
F = _G350
Foods = [apple, broccoli, carrot, lettuce, rice]
```

`Foods` is in alphabetical order because the food facts happen to be in alphabetical order.

The first argument of `findall` is a *template*. It is not limited to being a single variable. It might be a structure. The second argument can be a series of goals joined with conjunction.

```
?- findall(F-C, (food(F),color(F,C)), FoodAndColors).
F = _G488
C = _G489
FoodAndColors = [apple-red, broccoli-green, carrot-orange,
    lettuce-green, rice-white]
```

Remember that `-/2` is a functor that can be placed between its terms. `findall(-(F,C),...` produces the same result as the above.

`findall`, continued

The following query creates a list structures representing foods and the lengths of their names.

```
?- findall(Len-F, (food(F),atom_length(F,Len)), LFs).
Len = _G497
F = _G498
LFs = [5-apple, 8-broccoli, 6-carrot, 7-lettuce, 4-rice]
```

Two predicates that are related to `findall` are `bagof/3` and `setof/3` but they differ more than may be grasped at first glance. Section 6.3 in the text has a discussion of them; if they're needed in an assignment we'll discuss them more.

`keysort(+List, -Sorted)` sorts a list of structures with the functor `'-'` based on their first term:

```
 ?- findall(Len-F, (food(F),atom_length(F,Len)), LFs),
    keysort(LFs,Sorted), writeln(Sorted), fail.
[4-rice, 5-apple, 6-carrot, 7-lettuce, 8-broccoli]
No
```

Low-level list processing

A list can be specified in terms of a head and a tail.

The list `[1,2,3]` can be specified as this:

```
[1 | [2, 3]]
```

More generally, a list can be described as a sequence of initial elements and a tail.

The list `[1,2,3,4]` can be specified in any of these ways:

```
[1 | [2,3,4]]

[1,2 | [3,4]]

[1,2,3 | [4]]

[1,2,3,4 | []]
```

Low-level list processing, continued

Consider this unification:

```
?- [H|T] = [1,2,3,4].
H = 1
T = [2, 3, 4]
```

What instantiations are produced by these unifications?

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


?- [X, Y | T] = [1, 2].


?- [1, 2 | [3,4]] = [H | T].


?- A = [1], B = [A|A].
```

How does list construction and unification in Prolog compare to ML?

Writing list predicates

A rule to produce the head of a list:

```
head(L, H) :- L = [H|_].
```

The head of L is H if L unifies with a list whose head is H.

Usage:

```
?- head([1,2,3],H).
H = 1

?- head([], X).
No

?- L = [X,X,b,c], head(L, a).
L = [a, a, b, c]
X = a
```

Problem: Define `head/2` more concisely.

Writing list predicates, continued

Here is one way to implement the standard `member/2` predicate:

```
member(X,L) :- L = [X|_].
member(X,L) :- L = [_|T], member(X, T).
```

Usage:

```
?- member(1, [2,1,4,5]).
Yes

?- member(a, [2,1,4,5]).
No
```

We've seen that `member` can be used to generate the elements in a list:

```
?- member(X, [a,b,c]).
X = a ;
X = b ;
...
```

How does that generation work?

Problem: Define `member/2` more concisely.

Writing list predicates, continued

A more concise definition of `member/2`:

```
member(X,[X|_]).
```
X is a member of the list having X as its head

```
member(X,[_|T]) :- member(X,T).
```
X is a member of the list having T as its tail if X is a member of T

Problem: Define a predicate `last(L,X)` that describes the relationship between a list L and its last element, X.

```
?- last([a,b,c], X).
X = c
Yes

?- last([], X).
No

?- last(L,last), head(L,first), length(L,2).
L = [first, last]
Yes
```

Writing list predicates, continued

`last` is a built-in predicate but here is one way to write it:

```
lastx([X],X).
lastx([_|T],X) :- last(T,X).
```

Problem: Write a predicate `len/2` that behaves like the built-in `length/2`

```
?- len([], N).
N = 0

?- len([a,b,c,d], N).
N = 4

?- len([a,b,c,d], 5).
No

?- len(L,1).
L = [_G295]
```

Writing list predicates, continued

Here is `len`:

```
len([], 0).
len([_|T],Len) :- len(T,TLen), Len is TLen + 1.
```

Problem: Define a predicate `allsame(L)` that describes lists in which all elements have the same value.

```
?- allsame([a,a,a]).
Yes

?- allsame([a,b,a]).
No

?- allsame([A,B,C]), B = 1.
A = 1
B = 1
C = 1

?- length(L,5), allsame(L), L = [x|_]. % Note change from handout!
L = [x, x, x, x, x]
```

Writing list predicates, continued

Here's another way to test `allsame`:

```
    ?- allsame(L).

    L = [_G305] ;

    L = [_G305, _G305] ;

    L = [_G305, _G305, _G305] ;

    L = [_G305, _G305, _G305, _G305] ;

    L = [_G305, _G305, _G305, _G305, _G305] ;
```

Could we test `member` in a similar way? How about `append`?

Writing list predicates, continued

Problem: Define a predicate `listeq(L1,L2)` that describes the relationship between two lists `L1` and `L2` that hold the same sequence of values.

Problem: Define a predicate `p/1` that behaves like this:

```
?- p(X).
X = [] ;
X = [_G272] ;
X = [_G272, _G272] ;
X = [_G272, _G272, _G278] ;
X = [_G272, _G272, _G278, _G278] ;
X = [_G272, _G272, _G278, _G278, _G284] ;
```

The `append` predicate

Recall the description of the built-in append predicate:

```
    append(?List1, ?List2, ?List3)
```
Succeeds when `List3` unifies with the concatenation of `List1` and `List2`.

The usual definition of `append`:

```
    append([], X, X).
    append([X|L1], L2, [X|L3]) :-  append(L1, L2, L3).
```

How does it work?

Try tracing it. To avoid getting the built-in version, define the above as `my_append` instead of `append` (three places). Then try these:

```
    ?- gtrace, my_append([1,2,3,4],[a,b,c,d],X).

    ?- gtrace, my_append([a,b,c,d,e,f,g],[],X).
```

Lists are structures

In fact, lists are structures:

```
 ?- display([1,2,3]).
.(1, .(2, .(3, [])))
```

Essentially, `./2` is the "cons" operation in Prolog.

By default, lists are shown using the `[...]` notation:

```
?- X = .(a, .(b,[])).
X = [a, b]
```

We can write `member/2` like this:

```
member(X, .(X,_)).
member(X, .(_,T)) :- member(X,T).
```

Speculate: What the following produce?

```
?- X = .(1,1).
```

Some odds and ends

Reversing failure with \+

Cut (!)

The "cut-fail" idiom

Green cuts and red cuts

Reversing failure with \+

The query \+*goal* succeeds if *goal* fails.

```
?- food(computer).
No

?- \+food(computer).
Yes
```

\+ is sometimes read as "can't prove" or "fail if".

Example: *What foods are not green?*

```
?- food(F), \+color(F, green).
F = apple ;
F = carrot ;
F = rice ;
No
```

Problem: Write a predicate `inedible(X)` that succeeds if something is not a food.

Speculate: What will the query `?- inedible(X).` do?

Reversing failure, continued

`inedible/1` and some examples of use:

```
?- listing(inedible).

inedible(A) :- \+food(A).

?- inedible(computer).
Yes

?- inedible(banana).
Yes

?- inedible(X).
No
```

Any surprises?

Cut (!)

Backtracking can be limited with the "cut" operator, represented by an exclamation mark.

A cut always succeeds when evaluated, but inhibits backtracking.

```
?- food(F), writeln(F), fail.
apple
broccoli
carrot
lettuce
rice

No

?- food(F), writeln(F), !, fail.
apple

No
```

One way to picture a cut is like a one-way gate: control can go through it from left to right, but not from right to left.

Cut, continued

Consider these facts:

```
f(' f1 ').   g(' g1 ').
f(' f2 ').   g(' g2 ').
f(' f3 ').   g(' g3 ').
```

Queries and cuts:

```
?- f(F), write(F), g(G), write(G), fail.
 f1  g1  g2  g3  f2  g1  g2  g3  f3  g1  g2  g3

?- f(F), write(F), !, g(G), write(G), fail.
 f1  g1  g2  g3

?- f(F), write(F), g(G), write(G), !, fail.
 f1  g1
```

What's the output of the following query?

```
?- !, f(F), !, write(F), !, g(G), !, write(G), !, fail.
```

Another analogy: A cut is like a door that locks behind you.

Cut, continued

In fact, cuts are rarely used at the query prompt. The real use of cuts is in rules. Here's an artificial example:

```
ce1(one,1) :- true.             % Equivalent to ce1(one,1).
ce1(two,2) :- true, !.
ce1(_, 'something') :- true.
```

Usage:

```
?- ce1(one,X).
X = 1 ;
X = something ;
No

?- ce1(two, X).
X = 2 ;
No
```

<u>Note that the third clause is not tried for `ce1(two,X)`. That's due to the cut.</u>

<u>**In a rule, a cut still acts as a one-way gate in the rule itself but it also prevents consideration of subsequent clauses for the current call of that predicate.**</u> It's "do-or-die" (succeed or fail) with the rule at hand.

Cut, continued

Control never backtracks through a cut but backtracking can occur between goals on each side of a cut.

Consider this abstract rule,

```
    x :- a, b, c, !, d, e, f, !, g, h, i.
```

and a query:

```
    ?- x, y.
```

Control may circulate between `a`, `b`, and `c` but once `c` is proven (and the cut passed through), `a`, `b`, and `c` will be not be considered again during a particular call of `x`. Similarly, once `f` succeeds, we are further committed.

However, if `x` succeeds and `y` fails, control will backtrack into `g`, `h`, and `i` if they contain unexplored alternatives.

Cut, continued

Below is a faulty "improvement" for the `grade/2`, in the box at right.

```
grade(Score,'A') :-
    Score >= 90.
grade(Score,'B') :-
    Score >= 80, Score < 90.
grade(Score,'C') :-
    Score >= 70, Score < 80.
grade(Score,'F') :-
    Score < 70.
```

```
grade2(Score, 'A') :- Score >= 90.
grade2(Score, 'B') :- Score >= 80.
grade2(Score, 'C') :- Score >= 70.
grade2(_, 'F').
```

Usage:

```
?- grade2(85,G).
G = 'B' ;
G = 'C' ;
G = 'F' ;

No
```

Cut, continued

Here is a procedure based on `grade2`:

```
grade2(Score, 'A') :- Score >= 90.
grade2(Score, 'B') :- Score >= 80.
grade2(Score, 'C') :- Score >= 70.
grade2(_, 'F').
```

```
do_grades(Students) :-
    member(Who-Avg, Students),
    grade2(Avg,Grade),
    format('~w: ~w~n',
    [Who, Grade]), fail.
```

Usage:

```
?- do_grades([bob-87, mary-92]).
bob: B
bob: C
bob: F
mary: A
mary: B
mary: C
mary: F
```

What's the problem? How can we fix it?

Cut, continued

Recall:

<u>In a rule, a cut still acts as a one-way gate in the rule itself but it also prevents consideration of subsequent clauses for the current call of that predicate.</u> It's "do-or-die" (succeed or fail) with the rule at hand.

Solution:

```
grade3(Score, 'A') :- Score >= 90, !.
grade3(Score, 'B') :- Score >= 80, !.
grade3(Score, 'C') :- Score >= 70, !.
grade3(_, 'F').
```

Usage:

```
?- grade3(85,G).
G = 'B' ;
No
```

```
do_grades(Students) :-
    member(Who-Avg, Students),
    grade3(Avg,Grade),
    format('~w: ~w~n', [Who, Grade]), fail.
```

```
?- do_grades([bob-87, mary-92]).
bob: B
mary: A
```

Problem: Write a predicate `agf(-F)` that finds at most one green food.

Cut, continued

Here is an example from *Clause and Effect* by Clocksin:

```
max(X,Y,X) :- X >= Y, !.
max(_,Y,Y).
```

What's the other way to write it?

Also borrowed from *Clause and Effect*, here's a variant of `member/2`:

```
xmember(X, [X|_]) :- !.
xmember(X, [_|T]) :- xmember(X,T).
```

How does its behavior differ from the standard version? (below)

```
member(X, [X|_]).
member(X, [_|T]) :- member(X,T).
```

There's a built-in predicate, `memberchk/2`, that has the same behavior as `xmember`. When might it be appropriate to use it instead of `member/2`?

Cut, continued

`memberchk` is useful to avoid exploring alternatives with values that are known to be worthless. Here is an artificial example that represents a computation that selects several occurrences of the same value (3) from a list and tries each.

```
?- X = 3, member(X,[1,3,5,3,3]), writeln('Trying X...'), X > 3.
Trying X...
Trying X...
Trying X...

No
```

In fact, if the first 3 doesn't work then's no reason to hope that a subsequent 3 will work, assuming no side-effects. Observe the behavior with `memberchk`:

```
 ?- X = 3, memberchk(X,[1,3,5,3,3]), writeln('Trying X...'),
    X > 3.
Trying X...

No
```

The "cut-fail" idiom

Predicates naturally fail when a desired condition is absent but sometimes we want a predicate to fail when a particular condition is present.

Here is a recursive predicate that succeeds iff all numbers in a list are positive:

```
    allpos([X]) :- X > 0.
    allpos([X|T]) :- X > 0, allpos(T).
```

Another way to write it is with a "cut-fail":

```
    allpos(L) :- member(X, L), X =< 0, !, fail.
    allpos(_).
```

Remember that a cut effectively eliminates all subsequent clauses for the active predicate. If a non-positive value is found the cut eliminates `allpos(_).` and then the rule fails.

```
    ?- allpos([3,-1,5]).
    No
```

Another way to think about cut-fail: "My final answer is No!"

"cut-fail", continued

The SWI documentation for `is_list` includes its implementation:

```
    is_list(X) :- var(X), !, fail.
    is_list([]).
    is_list([_|T]) :-  is_list(T).
```

A cut-fail is used in the first rule to say that if `X` is a free variable (i.e., it is uninstantiated), look no further and fail.

Problem: Add a clause for `food/1` that "turns off" all the foods:

```
    ?- food(F).

    No
```

Green cuts and red cuts

A cut is said to be a "green cut" if it simply makes a predicate more efficient. By definition, adding or removing a green cut does not effect the set of results for any call of a predicate.

A "red cut" affects the set of results produced by a predicate.

Are there any examples of green cuts in the preceding examples?

Database manipulation

`assert` and `retract`

A simple command interpreter

Word tally

Unstacking blocks

`assert` and `retract`

A Prolog program is a database of facts and rules.

The database can be changed dynamically by adding facts with `assert/1` and deleting facts with `retract/1`.

A predicate to establish that certain things are food:

```
    makefoods :-
        assert(food(apple)),
        assert(food(broccoli)),
        assert(food(carrot)),
        assert(food(lettuce)),
        assert(food(rice)).
```

`assert` and `retract`, continued

Evaluating `makefoods` adds facts to the database:

```
?- food(X).
ERROR: Undefined procedure: food/1

?- makefoods.
Yes

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

A food can be "removed" with `retract`:

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

?- food(carrot).
No
```

`assert` and `retract`, continued

In a predicate like `makefoods`, use `retractall` to start with a clean slate:

```
makefoods :-
    retractall(food(_)),
    assert(food(apple)),
    assert(food(broccoli)),
    assert(food(carrot)),
    assert(food(lettuce)),
    assert(food(rice)).
```

Without `retractall`, each call to `makefoods` creates an identical, duplicated set of foods.

Here's the behavior of the first version of `makefoods`:

```
?- makefoods, makefoods, makefoods.
Yes

?- findall(F, food(F), Foods).
Foods = [apple, broccoli, carrot, lettuce, rice, apple,
broccoli, carrot, lettuce|...]
```

`assert` and `retract`, continued

Asserts and retracts are NOT undone with backtracking.

```
?- assert(f(1)), assert(f(2)), fail.
No

?- f(X).
X = 1 ;
X = 2 ;
No

?- retract(f(1)), fail.
No

?- f(X).
X = 2 ;
No
```

There is no notion of directly changing a fact. Instead, a fact is changed by retracting it and then asserting it with different terms.

`assert` and `retract`, continued

A rule to remove foods of a given color (assuming the `color/2` facts are present):

```
rmfood(C) :- food(F), color(F,C),
        retract(food(F)),
         write('Removed '), write(F), nl, fail.
```

Usage:

```
?- rmfood(green).
Removed broccoli
Removed lettuce
No

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

The color facts are not affected—`color(broccoli, green)` and `color(lettuce,green)` still exist.

A simple command interpreter

Imagine a simple interpreter to manipulate an integer value:

```
?- run.
? print.
0
? add(20).
? sub(7).
? print.
13
? set(40).
? print.
40
? exit.

Yes
```

Note that the commands themselves are Prolog terms.

A simple command interpreter, continued

Here is a loop that simply reads and prints terms:

```
run0 :- repeat, write('? '), read(X), format('Read ~w~n', X),
        X = exit.
```

Usage:

```
?- run0.
? a.
Read a
? ab(c,d,e).
Read ab(c, d, e)
? exit.
Read exit

Yes
```

`repeat/0` always succeeds. If `repeat` is backtracked into, it simply sends control back to the right. (Think of its `redo` port being wired to its `exit` port.)

The predicate `read(-X)` reads a Prolog term and unifies it with `X`.

Problem: Explain the operation of the loop.

A simple command interpreter, continued

Partial implementation:

```
init :-
    retractall(value(_)),
    assert(value(0)).

do(set(V)) :-
    retract(value(_)),
    assert(value(V)).

do(print) :- value(V), write(V), nl.

do(exit).

run :-
    init,
    repeat, write('? '), read(X), do(X), X = exit.
```

```
?- run.
? print.
0
? add(20).
? sub(7).
? print.
13
? set(40).
? print.
40
? exit.

Yes
```

How can `add(N)` and `sub(N)` be implemented?

Challenge: Add `mul` and `div` without having any repetitious code. Hint: Think about building a structure to evaluate with `is/2`.

Word tally

Manipulation of facts is very efficient in most Prolog implementations. We can use facts like we might use a Java map or a Ruby hash.

Imagine a word tallying program in Prolog:

```
?- tally.
|: to be or
|: not to be ought not
|: to be the question
|: (Empty line ends the input)

-- Results --
be          3
not         2
or          1
ought       1
question    1
the         1
to          3

Yes
```

Word tally, continued

Implementation:

```
    readline(Line) :-
        current_input(In), read_line_to_codes(In, Codes),
        atom_chars(Line, Codes).

    tally :- retractall(word(_,_)),
        repeat, readline(Line), do_line(Line), Line == '',
        show_counts.

    do_line('').
    do_line(Line) :-
            concat_atom(Words, ' ', Line), % splits Line on blanks
            member(Word, Words),
            count(Word), fail.
    do_line(_).

    count(Word) :-
        word(Word,Count0), retract(word(Word,_)),
        Count is Count0+1, assert(word(Word,Count)), !.

    count(Word) :- assert(word(Word,1)).
```

Word tally, continued

`show_counts` shows the results.

```
show_counts :-
    writeln('\n-- Results --'),
    findall(W-C, word(W,C), Pairs),
    keysort(Pairs, Sorted),
    member(W-C, Sorted),
    format('~w~t~12|~w~n', [W,C]),
    fail.
show_counts.
```

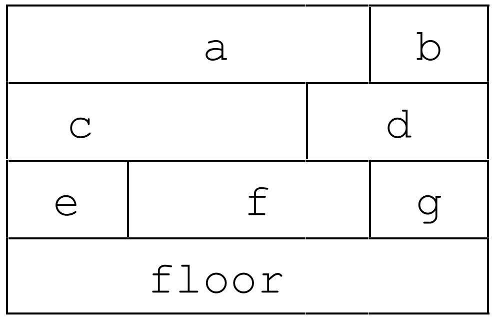
Problem: Modify the above so that words are ordered by count.

Is `keysort` really needed in the version shown?

Challenge: Create `show_counts(+Which)` where `Which` is `w` or `c` to cause ordering by words or counts.

Example: Unstacking blocks

Consider a stack of blocks, each of which is uniquely labeled with a letter:



This arrangement could be represented with these facts:

```
on(a,c).  on(c,e).  on(e,floor).
on(a,d).  on(c,f).  on(f,floor).
on(b,d).  on(d,f).  on(g,floor).
          on(d,g).
```

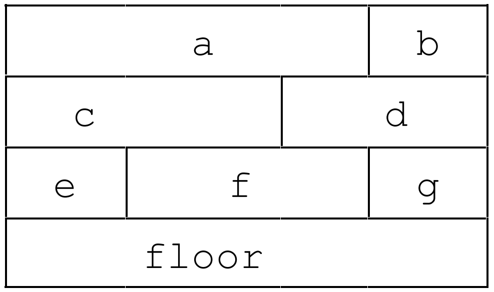
Problem: Define a predicate `clean` that will print a sequence of blocks to remove from the floor such that no block is removed until nothing is on it.

A suitable sequence of removals for the above diagram is: `a`, `c`, `e`, `b`, `d`, `f`, `g`. Another is `a`, `b`, `c`, `d`, `e`, `f`, `g`.

Unstacking blocks, continued

One solution: (`blocks.pl`)

```
removable(B) :- \+on(_,B).

remove(B) :-
    removable(B),
    retractall(on(B,_)),
    write('Remove '), write(B), nl.

remove(B) :-
  on(Above,B), remove(Above), remove(B).

clean :- on(B,floor), remove(B), clean.
clean.
```



```
on(a,c).        on(c,e).
on(a,d).        on(c,f).
on(b,d).        on(d,f).
on(e,floor).    on(d,g).
on(f,floor).
on(g,floor).
```

Important: If we want to be able to `assert` and `retract` facts of a predicate defined in a file, we must use the `dynamic` directive:

```
% cat blocks.pl
:-dynamic(on/2).
on(a,c). on(a,d).
...
```

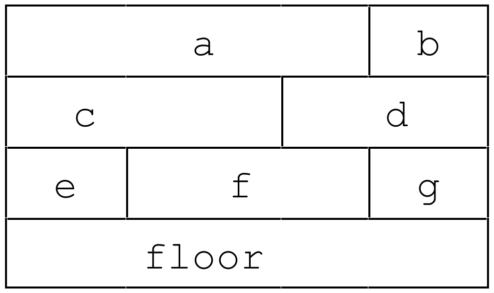
Unstacking blocks, continued

A more concise solution: (`blocks2.pl`)

```
clean :-
    on(A,_), \+on(_,A),
    write('Remove '), write(A), nl,
    retractall(on(A,_)), clean.
clean.
```

Output:

```
?- clean.
Remove a
Remove b
Remove c
Remove d
Remove e
Remove f
Remove g

Yes
```



```
on(a,c).       on(c,e).
on(a,d).       on(c,f).
on(b,d).       on(d,f).
on(e,floor).   on(d,g).
on(f,floor).
on(g,floor).
```

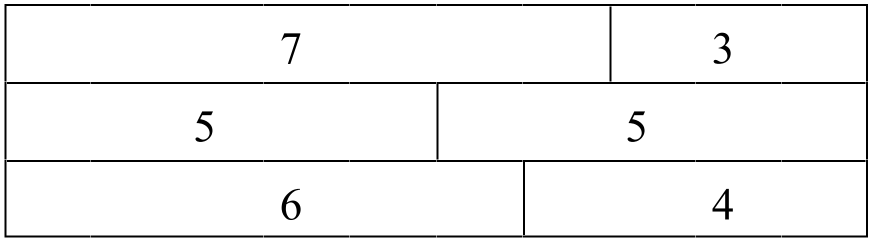
Puzzle solving

Brick laying

The Zebra Puzzle

Example: brick laying

Consider six bricks of lengths 7, 5, 6, 4, 3, and 5. One way they can be laid into three rows of length 10 is like this:



Problem: Write a predicate `laybricks/3` that produces a suitable sequence of bricks for three rows of a given length:

```
?- laybricks([7,5,6,4,3,5], 10, Rows).
Rows = [[7, 3], [5, 5], [6, 4]]

?- laybricks([7,5,6,4,3,5], 12, Rows).
No
```

In broad terms, how can we approach this problem?

Brick laying, continued

Here is a helper predicate `getone(X, List, Remaining)` that produces in `Remaining` a copy of `List` with `X` removed:

```
getone(X, [X|T], T).
getone(X, [H|T], [H|N]) :- getone(X, T, N).
```

Usage:

```
?- getone(X,[a,b,a,d],R).
X = a
R = [b, a, d] ;

X = b
R = [a, a, d] ;

X = a
R = [a, b, d] ;

X = d
R = [a, b, a] ;
```

What is the result of `?- getone(a, [a,b,a], R).`?

Brick laying, continued

`layrow` produces a sequence of bricks for a row of a given length:

```
layrow(Bricks, 0, Bricks, []).

layrow(Bricks, RowLen, Left, [Brick|MoreBricksForRow]) :-
   getone(Brick, Bricks, Left0),
   RemLen is RowLen - Brick, RemLen >= 0,
   layrow(Left0, RemLen, Left, MoreBricksForRow).
```

Usage:

```
?- layrow([3,2,7,4], 7, BricksLeft, Row).
BricksLeft = [2, 7]
Row = [3, 4] ;

BricksLeft = [3, 2, 4]
Row = [7] ;

BricksLeft = [2, 7]
Row = [4, 3] ;
No
```

What were the intermediate steps to arrive at the first row cited?

Brick laying, continued

Now we can write `laybricks`, which is hardwired for three rows:

```
laybricks(Bricks, RowLen, [Row1, Row2, Row3]) :-
   layrow(Bricks, RowLen, LeftAfter1, Row1),
   layrow(LeftAfter1, RowLen, LeftAfter2, Row2),
   layrow(LeftAfter2, RowLen, LeftAfter3, Row3),
   LeftAfter3 = [].
```

Usage:

```
?- laybricks([2,1,3,1,2], 3, Rows).
Rows = [[2, 1], [3], [1, 2]] ;

Rows = [[2, 1], [3], [2, 1]] ;

Rows = [[2, 1], [1, 2], [3]] ;

...many more...
```

What is the effect of `LeftAfter3 = []`?

Brick laying, continued

A more general predicate, to lay `Nrows` rows of length `RowLen`:

```
laybricks2([], 0, _, []).

laybricks2(Bricks, Nrows, RowLen, [Row|Rows]) :-
        layrow(Bricks, RowLen, BricksLeft, Row),
        RowsLeft is Nrows - 1,
        laybricks2(BricksLeft, RowsLeft, RowLen, Rows).
```

Usage:

```
?- laybricks2([5,1,6,2,3,4,3], 3, 8, Rows).
Rows = [[5, 3], [1, 4, 3], [6, 2]]

?- laybricks2([5,1,6,2,3,4,3], 8, 3, Rows).
No

?- laybricks2([5,1,6,2,3,4,3], 2, 12, Rows).
Rows = [[5, 1, 6], [2, 3, 4, 3]]

?- laybricks2([5,1,6,2,3,4,3], 4, 6, Rows).
Rows = [[5, 1], [6], [2, 4], [3, 3]]
```

As is, `laybricks2` requires that all bricks be used. How can we remove that requirement?

Brick laying continued

Some facts for testing:

```
b(3, [8,5,1,4,6,6,2,3,4,3,3,6,3,8,6,4]).   % 6x12
b(4, [8,5,1,4,6,6,2,3,4,3,3,6,3,8,6,4,1]). % 6x12 with extra 1
```

Usage:

```
?- b(3,Bricks), laybricks2(Bricks,6,12,Rows).
Bricks = [8, 5, 1, 4, 6, 6, 2, 3, 4|...]
Rows = [[8, 1, 3], [5, 4, 3], [6, 6], [2, 4, 3, 3], [6, 6], [8,
4]]
Yes
```

Is the `getone` predicate really necessary? Or could we just use `permute/2` and try all combinations of bricks?

Brick laying, continued

Let's try a set of bricks that can't be laid into six rows of twelve:

```
?- b(4,Bricks), laybricks2(Bricks, 6, 12, Rows).
...[the sound of a combinatorial explosion]...
^C
Action (h for help) ? a
% Execution Aborted

?- statistics.
8.05 seconds cpu time for 25,594,610 inferences
2,839 atoms, 1,854 functors, 1,451 predicates, 23 modules,
35,904 VM-codes
```

The speed of a Prolog implementation is sometimes quoted in LIPS—logical inferences per second.

The "Zebra Puzzle"

The Wikipedia entry for "Zebra Puzzle" presents a puzzle said to have been first published in the magazine *Life International* on December 17, 1962. Here are the facts:

> There are five houses.
> The Englishman lives in the red house.
> The Spaniard owns the dog.
> Coffee is drunk in the green house.
> The Ukrainian drinks tea.
> The green house is immediately to the right of the ivory house.
> The Old Gold smoker owns snails.
> Kools are smoked in the yellow house.
> Milk is drunk in the middle house.
> The Norwegian lives in the first house.
> The man who smokes Chesterfields lives in the house next to the man with the fox.
> Kools are smoked in the house next to the house where the horse is kept.
> The Lucky Strike smoker drinks orange juice.
> The Japanese smokes Parliaments.
> The Norwegian lives next to the blue house.

The article asked readers, "Who drinks water? Who owns the zebra?"

The Zebra Puzzle, continued

We can solve this problem creating a set of goals and asking Prolog to find the condition under which all the goals are true.

A good starting point is these two facts:

> The Norwegian lives in the first house.
> Milk is drunk in the middle house.

That information can be represented as this goal:

```
Houses = [house(norwegian, _, _, _, _),   % First house
            _,
            house(_, _, _, milk, _),        % Middle house
            _, _]
```

Note that there are five variables: nationality, pet, smoking preference (remember, it was 1962!), beverage of choice and house color. We'll use instances of `house` structures to represent knowledge about a house. Anonymous variables are used to represent "don't-knows".

The Zebra Puzzle, continued

Some facts can be represented with goals that specify structures as members of `Houses` with unknown position:

The Englishman lives in the red house.

```
member(house(englishman, _, _, _, red), Houses)
```

The Spaniard owns the dog.

```
member(house(spaniard, dog, _, _, _), Houses)
```

Coffee is drunk in the green house.

```
member(house(_, _, _, coffee, green), Houses)
```

How can we represent *The green house is immediately to the right of the ivory house.*?

The Zebra Puzzle, continued

At hand:

The green house is immediately to the right of the ivory house.

Here's a predicate that expresses left/right positioning:

```
left_right(L, R, [L, R | _]).
left_right(L, R, [_ | Rest]) :- left_right(L, R, Rest).
```

Testing:

```
?- left_right(Left,Right, [1,2,3,4]).
Left = 1
Right = 2 ;

Left = 2
Right = 3 ;
...
```

Goal:
```
left_right(house(_, _, _, _, ivory),
           house(_, _, _, _, green), Houses)
```

The Zebra Puzzle, continued

Remaining facts:

The man who smokes Chesterfields lives in the house next to the man with the fox.

Kools are smoked in the house next to the house where the horse is kept.

The Norwegian lives next to the blue house.

How can we represent these?

The Zebra Puzzle, continued

> The man who smokes Chesterfields lives in the house next to the man with the fox.
> Kools are smoked in the house next to the house where the horse is kept.
> The Norwegian lives next to the blue house.

We can say that two houses are next to each other if one is immediately left or right of the other:

```
next_to(X, Y, List) :- left_right(X, Y, List).
next_to(X, Y, List) :- left_right(Y, X, List).
```

More goals:

```
next_to(house(_, _, chesterfield, _, _),
        house(_, fox, _, _, _), Houses)

next_to(house(_, _, kool, _, _),
        house(_, horse, _, _, _), Houses)

next_to(house(norwegian, _, _, _, _),
        house(_, _, _, _, blue), Houses)
```

The Zebra Puzzle, continued

A rule that comprises all the goals developed thus far:

```
zebra(Houses, Zebra_Owner, Water_Drinker) :-
  Houses = [house(norwegian, _, _, _, _), _,
            house(_, _, _, milk, _), _, _],
  member(house(englishman, _, _, _, red), Houses),
  member(house(spaniard, dog, _, _, _), Houses),
  member(house(_, _, _, coffee, green), Houses),
  member(house(ukrainian, _, _, tea, _), Houses),
  left_right(house(_,_,_,_,ivory), house(_,_,_,_,green), Houses),
  member(house(_, snails, old_gold, _, _), Houses),
  member(house(_, _, kool, _, yellow), Houses),
  next_to(house(_,_,chesterfield,_,_),house(_, fox,_,_,_), Houses),
  next_to(house(_,_,kool,_,_), house(_, horse, _, _, _), Houses),
  member(house(_, _, lucky_strike, orange_juice, _), Houses),
  member(house(japanese, _, parliment, _, _), Houses),
  next_to(house(norwegian,_,_,_,_), house(_,_,_,_, blue), Houses),
  member(house(Zebra_Owner, zebra, _, _, _), Houses),
  member(house(Water_Drinker, _, _, water, _), Houses).
```

Note that the last two goals ask the questions of interest.

The Zebra Puzzle, continued

The moment of truth:

```
?- zebra(_, Zebra_Owner, Water_Drinker).
Zebra_Owner = japanese
Water_Drinker = norwegian ;
No
```

The whole neighborhood:

```
?- zebra(Houses,_,_), member(H,Houses), writeln(H), fail.
house(norwegian, fox, kool, water, yellow)
house(ukrainian, horse, chesterfield, tea, blue)
house(englishman, snails, old_gold, milk, red)
house(spaniard, dog, lucky_strike, orange_juice, ivory)
house(japanese, zebra, parliment, coffee, green)

No
```

Credit: The code above was adapted from http://sandbox.rulemaker.net/ngps/119, by Ng Pheng Siong, who in turn apparently adapted it from work by Bill Clementson in Allegro Prolog.

Parsing and grammars

A very simple grammar

A very simple parser

Grammar rule notation

Goals in grammar rules

Parameters in non-terminals

Accumulating an integer

A list recognizer

"Real" compilation

Credit: The first part of this section borrows heavily from chapter 12 in Covington, et al.

A very simple grammar

Here is a grammar for a very simple language:

```
Sentence => Article Noun Verb

Article     => the | a

Noun        => dog | cat | girl | boy

Verb        => ran | talked | slept
```

Here are some sentences in the language:

```
the dog ran

a boy slept

the cat talked
```

A very simple parser

Here is a simple parser for the grammar: (`parser0.pl`)

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).

article([the| Left], Left). article([a| Left],  Left).
noun([Noun| Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

Usage:

```
?- sentence([the,dog,ran]).
Yes

?- sentence([the,dog,boy]).
No

?- sentence(S).  % Generates all valid sentences
S = [the, dog, ran] ;
S = [the, dog, talked]
...
```

A very simple parser, continued

For reference:

```
    sentence(Words) :-
        article(Words, Left0), noun(Left0, Left1), verb(Left1, []).

    article([the|Left], Left). article([a| Left],  Left).
    noun([Noun|Left], Left) :- member(Noun, [dog,cat,girl,boy]).
    verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

Note that the heads for `article`, `noun`, and `verb` all have the same form.

Let's look at a clause for `article` and a unification:

```
    article([the|Left], Left).

    ?- article([the,dog,ran], Remaining).
    Remaining = [dog, ran]
```

If `Words` begins with `the` or `a`, then `article(Words, Remaining)` succeeds and unifies `Remaining` with the rest of the list. **The key idea: `article`, `noun`, and `verb` each consume an expected word and produce the remaining words.**

A very simple parser, continued

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).
```

A query sheds light on how `sentence` operates:

```
?- article(Words, Left0), noun(Left0, Left1),
    verb(Left1, Left2), Left2 = [].

Words = [the, dog, ran]
Left0 = [dog, ran]
Left1 = [ran]
Left2 = []

?- sentence([the,dog,ran]).
Yes
```

Each goal consumes one word. The remainder is then the input for the next goal.

Why is `verb`'s result, `Left2`, unified with the empty list?

A very simple parser, continued

For convenient use, let's add a predicate `s/1` that uses `concat_atom` to split up a string based on spaces:

```
s(String) :-
    concat_atom(Words, ' ', String), sentence(Words).

sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1),
    verb(Left1, []).
```

Usage:

```
?- s('the dog ran').
Yes

?- s('ran the dog').
No
```

Grammar rule notation

Prolog's *grammar rule notation* provides a convenient way to express these stylized rules. Instead of this,

```
sentence(Words) :-
    article(Words, Left0), noun(Left0, Left1), verb(Left1, []).
article([the| Left], Left).
article([a| Left],  Left).
noun([Noun| Left], Left) :- member(Noun, [dog,cat,girl,boy]).
verb([Verb|Left], Left) :- member(Verb, [ran,talked,slept]).
```

we can take advantage of grammar rule notation and say this,

```
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].
```

Note that the literals are specified as singleton lists.

The semicolon is an "or". Alternative: `noun --> [dog]. noun --> [cat]. ...`

Grammar rule notation, continued

```
% cat parser1.pl
sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].
verb --> [ran]; [talked]; [slept].
```

`listing` can be used to see the clauses generated by Prolog:

```
 ?- listing(sentence).
sentence(A, D) :-  article(A, B), noun(B, C), verb(C, D).

?- listing(article).
article(A, B) :-
        (   'C'(A, a, B)
        ;   'C'(A, the, B)
        ).

?- listing('C').
'C'([A|B], A, B).
```

Note that the predicates generated for `sentence`, `article` and others have an arity of 2.

Grammar rule notation, continued

```
% cat parser1a.pl
s(String) :-
    concat_atom(Words, ' ', String), sentence(Words,[]).

sentence --> article, noun, verb.
article --> [a]; [the].
noun --> [dog]; [cat]; [girl]; [boy].

verb --> [ran]; [talked]; [slept].
verb([Verb|Left], Left) :- verb0(Verb).

verb0(jumped). verb0(ate). verb0(computed).
```

Points to note:

`sentence`, `article`, `verb`, and `noun` are known as *non-terminals*. `dog`, `cat`, `ran`, talked, etc. are known as *terminals*.

The call to `sentence` in `s` now has two terms to match the generated rule.

`verb` has both a grammar rule and an ordinary rule. The latter makes use of `verb0` facts.

Goals in grammar rules

We can insert ordinary goals into grammar rules by enclosing the goal(s) in curly braces. Here is a chatty parser that recognizes the language described by the regular expression `a*`:

```
parse(S) :- atom_chars(S,Chars), string(Chars,[]). % parser6.pl

string --> as.

as --> [a], {writeln('Got an a')}, as.
as --> [], {writeln('out of a\'s')}.
```

Usage:

```
?- parse('aaa').
Got an a
Got an a
Got an a
out of a's
Yes
```

What would the behavior be if the order of the `as` clauses was reversed?

Parameters in non-terminals

We can add parameters to the non-terminals in grammar rules. The following grammar recognizes `a*` and produces the length, too.

```
parse(S, Count) :-                                     % parser6a.pl
    atom_chars(S, Chars), string(Count, Chars, []).

string(N) --> as(N).

as(N) --> [a], as(M), {N is M + 1}.
as(0) --> [].
```

Usage:

```
?- parse('aaa',N).
N = 3

?- parse('aaab',N).
No
```

Parameters in non-terminals, continued

Here is a grammar that recognizes `a`^{N}`b`^{2N}`c`^{3N}: (`parser7a.pl`)

```
parse(S,L) :- atom_chars(S,Chars), string(L, Chars, []).

string([N,NN,NNN])
    --> as(N), {NN is 2*N}, bs(NN), {NNN is 3*N}, cs(NNN).

as(N) --> [a], as(M), {N is M+1}. as(0) --> [].

bs(N) --> [b], bs(M), {N is M+1}. bs(0) --> [].

cs(N) --> [c], cs(M), {N is M+1}. cs(0) --> [].

?- parse('aabbbbcccccc', L).
L = [2, 4, 6]
?- parse('abbc', L).
No
```

Can this language be described with a regular expression?

Problem: Recognize `a`^{X}`b`^{Y}`c`^{Z} where `X <= Y <= Z`.

Parameters in non-terminals, continued

A parser for $a^X b^Y c^Z$ where `X <= Y <= Z`: (`parser7b.pl`)

```
parse(S,L) :- atom_chars(S,Chars), string(L, Chars, []).

string([X,Y,Z]) --> as(X), bs(Y), {X =< Y}, cs(Z), {Y =< Z}.

as(N) --> [a], as(M), {N is M+1}. as(0) --> [].
bs(N) --> [b], bs(M), {N is M+1}. bs(0) --> [].
cs(N) --> [c], cs(M), {N is M+1}. cs(0) --> [].
```

Usage:

```
?- parse('abbbcccccc', L).
L = [1, 3, 6]

?- parse('aabbc', L).
No

?- parse('c', L).
L = [0, 0, 1]
```

Accumulating an integer

Here is a parser that recognizes a string of digits and creates an integer from them:

```
parse(S,N) :- % parser8.pl
    atom_chars(S, Chars), intval(N,Chars,[]), integer(N).

intval(N) --> digits(Digits), { atom_number(Digits,N) }.

digits(Digit) --> [Digit], {digit(Digit)}.
digits(Digits) -->
    [Digit], {digit(Digit)},
    digits(More), { concat_atom([Digit,More],Digits)}.

digit('0'). digit('1'). ... digit('9').
```

Usage:

```
?- parse('4312', N).
N = 4312
```

A list recognizer

Here is a parser that recognizes lists consisting of integers and lists: (`list.pl`)

```
parse(S) :- atom_chars(S, Chars), list(Chars, []).

list --> ['['], values, [']'].
list --> ['['], [']'].

values --> value.
values --> value, [','], values.

value --> digits(_).
value --> list.
% assume digits from previous slide

?- parse('[1,20,[30,[[40]],6,7],[]]').
Yes

?- parse('[1,20,,30,[[40]],6,7],[]]').
No

?- parse('[1, 2, 3]').  % Whitespace!  How could we handle it?
No
```

"Real" compilation

These parsing examples fall short of what's done in a compiler. The first phase of compilation is typically to break the input into "tokens". Tokens are things like identifiers, individual parentheses, string literals, etc.

The input

```
[ 1, [30+400], 'abc']
```

might be represented as a stream of tokens with this Prolog list:

```
[lbrack, integer(1), comma, lbrack, integer(30), plus,
integer(400), rbrack, comma, string(abc), rbrack]
```

The second phase of compilation is to parse the stream of tokens and generate code (traditional compilation) or execute it immediately (interpretation).

We could use a pair of Prolog grammars to parse source code. The first one would parse character-by-character and generate a token stream like the list above. The second grammar would parse that token stream.

Odds and Ends

Collberg's ADT

The Prolog 1000

Lots of Prologs

Ruby meets Prolog

Collberg's *Architecture Discovery Tool*

In the mid-1990s Christian Collberg developed a system that is able to discover the instruction set, registers, addressing modes and more for a machine given only a C compiler for that machine.

The basic idea is to use the C compiler of the target system to compile a large number of small but carefully crafted programs and then examine the machine code produced for each program to make inferences about the architecture. The end result is a machine description that in turn can be used to generate a code generator for the architecture.

The system is written in Prolog. What do you suppose makes Prolog well-suited for this task?

See http://www.cs.arizona.edu/~collberg/Research/AutomaticRetargeting/index.html for more details.

The Prolog 1000

The Prolog 1000 is a compilation of applications written in Prolog and related languages. Here is a sampling of the entries:

AALPS

The Automated Air Load Planning System provides a flexible spatial representation and knowledge base techniques to reduce the time taken for planning by an expert from weeks to two hours. It incorporates the expertise of loadmasters with extensive cargo and aircraft data.

ACACIA

A knowledge-based framework for the on-line dynamic synthesis of emergency operating procedures in a nuclear power plant.

ASIGNA

Resource-allocation problems occur frequently in chemical plans. Different processes often share pieces of equipment such as reactors and filters. The program ASIGNA allocates equipment to some given set of processes. (2,000 lines)

The Prolog 1000, continued

Coronary Network Reconstruction

The program reconstructs a three-dimensional image of coronary networks from two simultaneous X-Ray projections. The procedures in the reconstruction-labelling process deal with the correction of distortion, the detection of center-lines and boundaries, the derivation of 2-D branch segments whose extremities are branching, crossing or end points and the 3-D reconstruction and display.

All algorithmic components of the reconstruction were written in the C language, whereas the model and resolution processes were represented by predicates and production rules in Prolog. The user interface, which includes a main panel with associated control items, was developed using Carmen, the Prolog by BIM user interface generator.

DAMOCLES

A prototype expert system that supports the damage control officer aboard Standard frigates in maintaining the operational availability of the vessel by safeguarding it and its crew from the effects of weapons, collisions, extreme weather conditions and other calamities. (> 68,000 lines)

The Prolog 1000, continued

DUST-EXPERT

Expert system to aid in design of explosion relief vents in environments where flammable dust may exist. (> 10,000 lines)

EUREX

An expert system that supports the decision procedures about importing and exporting sugar products. It is based on about 100 pages of European regulations and it is designed in order to help the administrative staff of the Belgian Ministry of Economic Affairs in filling in forms and performing other related operations. (>38,000 lines)

GUNGA CLERK

Substantive legal knowledge-based advisory system in New York State Criminal Law, advising on sentencing, pleas, lesser included offenses and elements.

MISTRAL

An expert system for evaluating, explaining and filtering alarms generated by automatic monitoring systems of dams. (1,500 lines)

The full list is in `fall06/pl/Prolog1000.txt`. Several are over 100,000 lines of code.

Lots of Prologs

For an honors section assignment Maxim Shokhirev was given the task of finding as many Prolog implementations as possible in one hour. His results:

1. DOS-PROLOG
http://www.lpa.co.uk/dos.htm
2. Open Prolog
http://www.cs.tcd.ie/open-prolog/
3. Ciao Prolog
http://www.clip.dia.fi.upm.es/Software/Ciao
4. GNU Prolog
http://pauillac.inria.fr/~diaz/gnu-prolog/
5. Visual Prolog (PDC Prolog and Turbo Prolog)
http://www.visual-prolog.com/
6. SWI-Prolog
http://www.swi-prolog.org/
7. tuProlog
http://tuprolog.alice.unibo.it/
8. HiLog
ftp://ftp.cs.sunysb.edu/pub/TechReports/kifer/hilog.pdf
9. ?Prolog
http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/
10. F-logic
http://www.cs.umbc.edu/771/papers/flogic.pdf
11. OW Prolog
http://www.geocities.com/owprologow/
12. FLORA-2
http://flora.sourceforge.net/
13. Logtalk
http://www.logtalk.org/

DataLog implementations:
----
14. bddbddb
http://bddbddb.sourceforge.net/
15. ConceptBase
http://www-i5.informatik.rwth-aachen.de/CBdoc/
16. DES
http://www.fdi.ucm.es/profesor/fernan/DES/
17. DLV
http://www.dlvsystem.com/
18. XSB
http://xsb.sourceforge.net/

19. WIN Prolog
http://www.lpa.co.uk/
20. YAP Prolog
http://www.ncc.up.pt/~vsc/Yap
21. AI::Prolog
http://search.cpan.org/~ovid/AI-Prolog-0.734/lib/AI/Prolog.pm
22. SICStus Prolog
http://www.sics.se/sicstus/

23. ECLiPSe Prolog
http://eclipse.crosscoreop.com/
24. Amzi! Prolog
http://www.amzi.com/
25. B-Prolog
http://www.probp.com/
26. MINERVA
http://www.ifcomputer.co.jp/MINERVA/
27. Trinc Prolog
http://www.trinc-prolog.com/
28. Strawberry Prolog
http://www.dobrev.com/
29. hProlog
http://www.cs.kuleuven.ac.be/~bmd/hProlog/
30. ilProlog
http://www.pharmadm.com/dmax.asp
31. CxProlog
http://ctp.di.fct.unl.pt/~amd/cxprolog/
32. NanoProlog
http://ctp.di.fct.unl.pt/~amd/cxprolog/
33. BinProlog
http://www.binnetcorp.com/BinProlog/
34. Quintus Prolog
http://www.sics.se/quintus/
35. Allegro Prolog
http://www.franz.com/products/prolog/
36. ALS Prolog
http://www.als.com/
37. W-Prolog
http://ktiml.mff.cuni.cz/~bartak/prolog/testing.html
38. Aquarius Prolog
listserv@acal-server.usc.edu
39. EZY Prolog
http://www.ezy-software.com/ezyprolog/EZY_Prolog_-_born_
to_be_easy.htm
40. Poplog
http://www.cs.bham.ac.uk/research/poplog/freepoplog.html
41. OpenPoplog
http://openpoplog.sourceforge.net/
42. REBOL Prolog
http://www.rebol.org/cgi-bin/cgiwrap/rebol/view-script.r?script
=prolog.r
43. Arity Prolog
pgweiss@netcom.com
44. CHIP
http://www.cosytec.com/
45. IF/Prolog
http://www.ifcomputer.com/IFProlog/
46. LPA Prolog
http://www.lpa.co.uk/
47. cu-Prolog
ftp://ftp.icot.or.jp/pub/cuprolog/
48. Tricia
ftp://ftp.csd.uu.se/pub/Tricia/
49. Visual Prolog Personal Edition
http://www.visual-prolog.com/vip/vipinfo/freeware_version.ht
m
50. JLog
http://jlogic.sourceforge.net/
51. JIProlog
http://www.ugosweb.com/jiprolog
52. CKI-Prolog
http://www.students.cs.uu.nl/~smotterl/prolog/index.htm
53. JavaLog

http://www.exa.unicen.edu.ar/~azunino/javalog.html
54. jProlog
http://www.cs.kuleuven.ac.be/~bmd/PrologInJava/
55. tiny_prolog.rb (Ruby implementation)
http://swik.net/Ruby/Inside+Ruby/A+Basic+Prolog+Implementation+in+Ruby/ofun
56. WAM (Warren Abstract Machine)
http://64.233.187.104/search?q=cache:fgeYFaPXclIJ:www.clip.dia.fi.upm.es/~logalg/slides/PS/8_wam.ps+Prolog+Implementation+list&hl=en&gl=us&ct=clnk&cd=49
57. Allegro Prolog
http://www.franz.com/support/tech_corner/prolog-071504.lhtml
58. NU-Prolog
http://www.cs.mu.oz.au/~lee/src/nuprolog/
59. Mimola Software System (MSS)
citeseer.ist.psu.edu/22901.html
60. LogTalk
http://www.logtalk.org/
61. Palm Toy Language
http://www.geocities.com/willowfung/
62. Qu-Prolog
http://www.itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html
63. K-Prolog
http://prolog.isac.co.jp/doc/en/
64. ProFIT
http://www.coli.uni-sb.de/~erbach/formal/profit/profit.html
65. Arity/Prolog
http://www.arity.com/www.pl/products/ap.htm
66. Brain Aid Prlog
http://www.comnets.rwth-aachen.de/~ost/private.html
67. Reform Prolog
http://user.it.uu.se/~thomasl/reform.html
68. CMU Free/Shareware Prolog
http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/impl/prolog/0.html
69. Babol Prolog
http://zamek.gda.pl/~budyn/
70. INRIA wamcc
ftp://ftp.inria.fr/INRIA/Projects/contraintes/wamcc/
71. PyProlog
http://sourceforge.net/projects/pyprolog/
72. Artiware Prolog
www.artiware.com
73. DGKS Prolog
www.geocities.com/SiliconValley/Campus/7816/

Ruby meets Prolog

http://eigenclass.org/hiki.rb?tiny+prolog+in+ruby describes a "tiny Prolog in Ruby".

From that page, here is `member`:

```
    member[cons(:X,:Y), :X] <<= :CUT
    member[cons(:Z,:L), :X] <<= member[:L,:X]
```

Here's the standard family example:

```
    sibling[:X,:Y] <<= [parent[:Z,:X], parent[:Z,:Y], noteq[:X,:Y]]
    parent[:X,:Y] <<= father[:X,:Y]
    parent[:X,:Y] <<= mother[:X,:Y]

    # facts: rules with "no preconditions"
    father["matz", "Ruby"].fact
    mother["Trude", "Sally"].fact
    ...

    query sibling[:X, "Sally"]
    # >> 1 sibling["Erica", "Sally"]
```