

## Replacement Set

This set replaces an earlier handout  
with Ruby slides 260-267 followed  
by Prolog slides 1-35

# Prolog

CSC 372, Spring 2014  
The University of Arizona  
William H. Mitchell  
whm@cs

## 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 led by Alain Colmerauer.

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 Computer Systems" 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 zero (0) Prolog books on Safari!

*Prolog Programming in Depth*, by Covington, Nute, and Vellino

Available for free at <http://www.covingtoninnovations.com/books/PPID.pdf>. That PDF is scans of pages and is not searchable.

The copy at <http://cs.arizona.edu/classes/cs372/spring14/covington/ppid.pdf> has had a searchable text layer added.

The UA bookstore stocks this text:

*Programming in Prolog*, 5th edition, by Clocksin and Mellish

A PDF of Dr. Collberg's 372 slides on Prolog is here:

<http://cs.arizona.edu/classes/cs372/spring14/CollbergProlog.pdf>

There's no Prolog "home page" that I know of.

We'll be using SWI Prolog. More on it soon.

# 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.

We can say that **facts.pl** holds a Prolog *database* or *knowledgebase*—a collection of facts.

At hand:

```
% cat foods.pl  
food(apple).  
food(broccoli).  
...
```

**food**, **apple**, and **broccoli** are examples of *atoms*, which can be thought of as multi-character literals. Atoms should not be thought of as strings!

Here are two more atoms:

```
'bell pepper'  
'Whopper'
```

An atom can be written without single quotes if it starts with a lower-case letter and contains only letters, digits, and underscores.

Note the use of single quotes. (Double quotes mean something else!)

# Facts and queries, continued

On lectura, we can start SWI Prolog and load a file of facts like this:

```
% swipl  
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.4.1)  
...  
?- [foods]. % in cs.arizona.edu/classes/cs372/spring14/pl  
% foods compiled 0.00 sec, 9 clauses  
true.
```

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

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

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

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

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*.

Simple rule for now: use all-lowercase filenames with the suffix `.pl` (PL) for Prolog source files.



## Sidebar: Reconsulting with **make**

After a **.pl** file has been consulted (loaded), we can query **make.** to cause any modified files to be reconsulted (reloaded), after editing the file.

```
?- [foods].
```

```
% foods compiled 0.00 sec, 9 clauses
```

```
true.
```

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

```
false.
```

*[Edit **foods.pl** with in a different window, and add **food(pickle).**]*

```
?- make.
```

```
% /Users/whm/372/pl/foods compiled 0.00 sec, 2 clauses
```

```
true.
```

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

```
true.
```

How might the food information be represented in Haskell?

```
food "apple"      = True
food "broccoli"   = True
food "carrot"     = True
food "lettuce"    = True
food "rice"       = True
food _            = False
```

```
> food "apple"
True
```

Maybe a list would be better:

```
foods = ["apple", "broccoli", "carrot", "lettuce", "rice"]
```

```
> "pickle" `elem` foods
False
```

How might we represent it 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 that starts with a capital letter is a variable.

```
?- 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 find 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 by printing a period.

```
?- food(Edible).
```

```
Edible = apple .    % User hit <ENTER>; Prolog printed the period.
```

```
?-
```

## 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 = 'Big Mac'.

?-

Facts are searched in the order they appear in the file. Above, the user exhausts all the facts by repeatedly typing a semicolon.

**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 make an analog for the above behavior in Java, Haskell, or Ruby?

For two points of extra credit:

- (1) Create a file **foods.pl** and try the examples just shown.
- (2) Create a small database (a file of facts) about something other than food and demonstrate some queries with it using **swipl**. Minimum: 5 facts.
- (3) Copy/paste a transcript of your **swipl** session into a plain text file named **facts.txt**.
- (4) Turn in **facts.txt** via the **eca1** dropbox **before the start of the next lecture**.

Needless to say, feel free to read ahead in the slides and show experimentation with the following material, too.

Experiment with syntax, too. Where can whitespace appear? What can appear in a fact other than atoms like **apple**?

Look ahead a few slides for information about installing SWI Prolog on your machine.

## Yes and no vs. true. and false.

Unlike SWI Prolog, most Prolog implementations use "yes" and "no" to indicate whether an interactive query succeeds. Here's GNU Prolog:

```
% gprolog
GNU Prolog 1.4.4 (64 bits)
| ?- [foods].
compiling foods.pl for byte code...

| ?- food(apple).
yes

| ?- food(pickle).
no
```

Most Prolog texts, including Covington and Clocksin & Mellish use **yes/no**, too. Just read "yes" as **true.** and "no" as **false.**

Remember: we're using SWI Prolog; GNU Prolog is shown above just for contrast.

# "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?"

`food(apple).` is trivially proven because we've supplied a fact that says that apple is a food.

The query

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

produces **false**. because Prolog can't prove that pickle is a food based on the database (the facts) we've supplied. (We've given it no rules, either.)

# "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 = 'Whopper' ;
```

```
F = 'Big Mac'.
```

```
?-
```

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 a food, is trivial but Prolog is capable of finding proofs for an arbitrarily complicated body of knowledge expressed as facts and rules.



## "Can you prove it?", continued

`write` is one of many built-in *predicates*. It outputs a value.

```
?- write('Hello, world!').
```

```
Hello, world!
```

```
true.
```

Speculate: Why was "true." output, too?

Prolog is reporting that it's able to prove `write('Hello, world!')`!

A side-effect of "proving" `write(X)` is outputting the value of **X**!

# Getting and running SWI Prolog

# Getting and running SWI Prolog

**swi-prolog.org** is the home page for SWI Prolog.

On lectura, just run **swipl**.

Downloads for Windows and OS X:

**swi-prolog.org/download/stable**

For Windows, the non-64 bit version will be fine for our purposes:

**SWI-Prolog 6.6.4 for Windows XP/Vista/7/8**

Pick **Typical** as the **Install type**, **.pl** for file extension

For OS X there's only one choice:

**SWI-Prolog 6.6.4 for MacOSX 10.6 (Snow Leopard) and later...**

As the install page says, you'll need XQuartz 2.7.5 for the development tools. The handiest tool is perhaps the graphical tracer, launched with the **gtrace** predicate. (We'll see **gtrace** later.)

# Getting and running SWI Prolog, continued

On Windows, assuming you associated `.pl` files with SWI Prolog, running `foods.pl` on the command line or opening `foods.pl` in Explorer opens a window running SWI Prolog and consults the file, as if `[foods].` had been typed at the prompt.

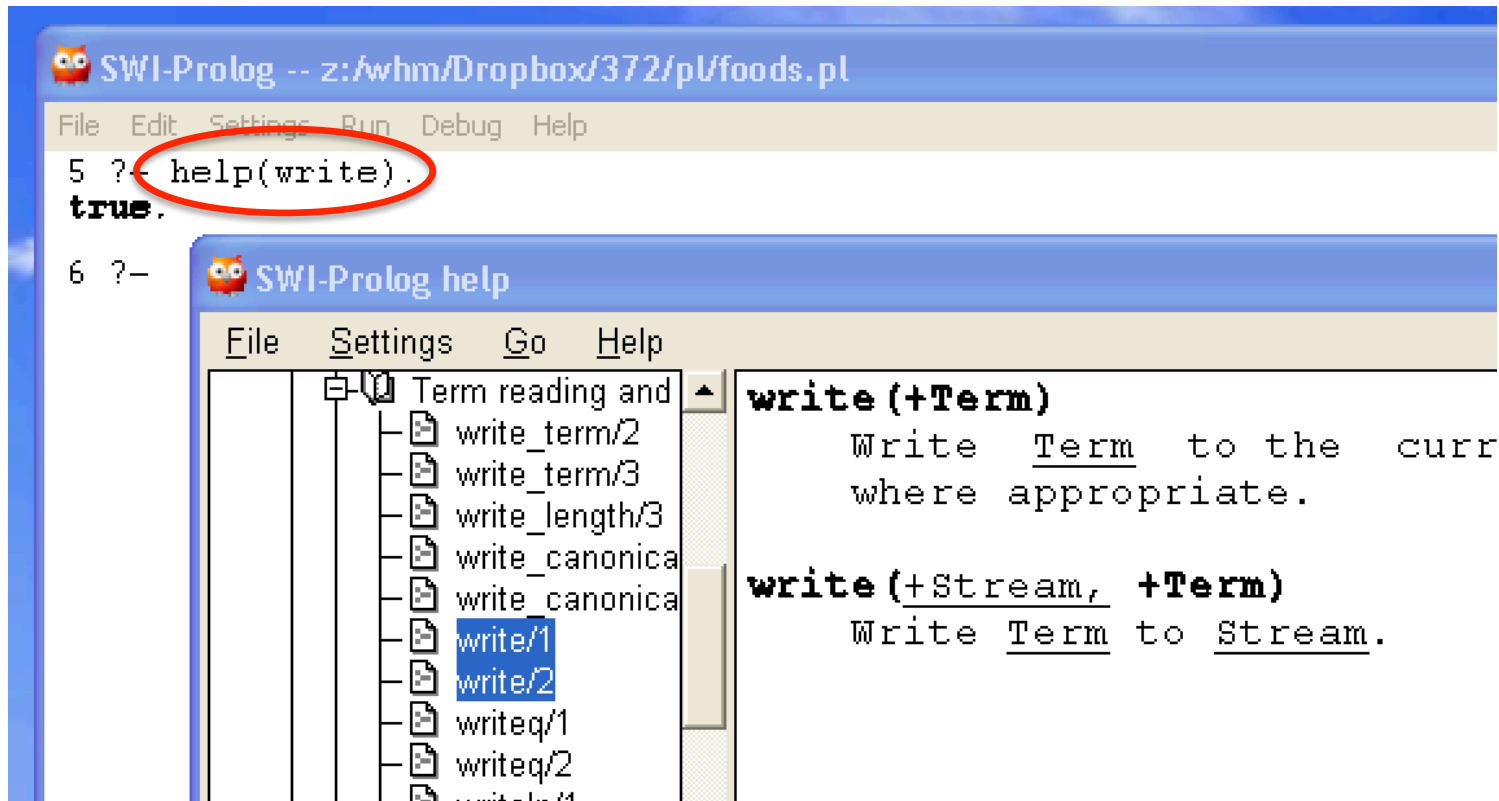


On Windows, a numbered query prompt is shown. ("1 ?-" above)

Remember: You can use `make.` to reconsult (reload) a file.

# Getting and running SWI Prolog, continued

To display help for a predicate, query `help(predicate-name)`. On Windows and OS X a window will open.



On lectura, help will be text based, but if you login to lectura from a Linux machine with "`ssh -X ...`", you'll get window-based help, too. (Ditto for OS X, assuming you're running XQuartz.)

# Getting and running SWI Prolog, continued

On all platforms a control-D or querying **halt.** exits SWI Prolog.

```
% swipl
```

```
...
```

```
?- halt.
```

```
%
```

A control-C while a query is executing will produce an **Action ... ?** prompt. Then typing **h** produces a textual menu:

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

```
X = apple ^C
```

```
Action (h for help) ? h
```

```
Options:
```

a:	abort	b:	break
c:	continue	e:	exit
g:	goals	t:	trace
h (?):	help		

Use **a** to return to the prompt; **e** exits to the shell.

# Building blocks

We've seen that `apple`, `food`, and `'Big Mac'` are examples of *atoms*.

Typing an atom as a query doesn't do what we might expect, but we can output an atom with `write`.

```
?- 'just\ntesting'.
```

```
ERROR: toplevel: Undefined procedure: 'just\ntesting'/0
(DWIM could not correct goal)
```

```
?- write('just\ntesting').
```

```
just
```

```
testing
```

```
true.
```

Atoms composed of certain non-alphabetic characters do not require quotes:

```
?- write(#$&*+-. /:<=>?^~\).
```

```
#$&*+-. /:<=>?^~\
```

```
true.
```



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

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

```
true.
```

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

```
true.
```

```
?- atom(Apple).
```

```
false.
```

```
?- atom("apple").
```

```
false.
```

Alternate view: "Can you prove `apple` is an atom?"

Integer and floating point literals are *numbers*.

```
?- number(10).
```

```
true.
```

```
?- number(3.4).
```

```
true.
```

```
?- number(3.4e100).
```

```
true.
```

```
?- number('100').
```

```
false.
```

Numbers aren't atoms but they are "atomic".

```
?- atom(100).
```

```
false.
```

```
?- atomic(100). % Note: atomIC, not just atom.
```

```
true.
```

In Prolog, arithmetic doesn't work as you might expect:

```
?- 3 + 4.
```

```
ERROR: toplevel: Undefined procedure: (+)/2 (DWIM could  
not correct goal)
```

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

```
Y = 4+5.
```

```
?- write(3 + 4 * 5).
```

```
3+4*5
```

```
true.
```

We'll learn about arithmetic 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**, **odd**, and **number**.

It's common to refer to predicates using *predicate indicators* like **color/2**, **odd/1**, and **number/3**, where the number following the slash is the number of *terms*.

**number/3** above doesn't interfere with **number/1** (two slides back).

# 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')

**+/-**(3,4)        % *functor is symbolic atom*

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

The structure functors are **color**, **odd**, **number**, **+/-**, and **lunch**, respectively.

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

A structure can serve as a fact or a goal, depending on the context.

Here is a predicate **x** defined by three facts:

**x(just(testing,date(5,14,2014))).**

**x(10).**

**x(10,20).**

The first fact's term is a structure but the second fact's term is a number.  
That inconsistency is not considered to be an error.

**?- x(V).**

**V = just(testing, date(5, 14, 2014)) ;**

**V = 10.**

Further, is it **x/1** or **x/2**?

**?- x(A,B).**

**A = 10,**

**B = 20.**

# Predicate/goal mismatches, continued

At hand:

```
x(just(testing,date(5,14,2014))).  
x(10). x(A,B).
```

Here are some more queries:

```
?- x(abc).  
false.
```

```
?- x([1,2,3]). % A list...  
false.
```

```
?- x(a(b)).  
false.
```

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 "**false**" in each case because nothing it knows about aligns with anything it's being queried about.

## Predicate/goal mismatches, continued

At hand:

```
x(just(testing,date(5,14,2014))).  
x(10). x(A,B).
```

It's an error if there's no fact defined that has the same number of terms as the goal in a query. Alternatives are suggested.

```
?- x(little,green,apples).
```

```
ERROR: Undefined procedure: x/3
```

```
ERROR: However, there are definitions for:
```

```
ERROR: x/1
```

```
ERROR: x/2
```

What does the following tell us?

```
?- write(1,2).
```

```
ERROR: write/2: Domain error: `stream_or_alias' expected,  
found `1'
```



A new knowledgebase is to the right.

A query about green things:

?- color(Thing,green).

Thing = grass ;

Thing = broccoli ;

Thing = lettuce.

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

How can we state it in terms of "Can you prove...?"

*For what things can you prove their color is green?*

How could we query for each thing and its color?

?- color(Thing,Color).

Thing = sky,

Color = blue ;

Thing = dirt,

Color = brown ;

Thing = grass,

Color = green ;

Thing = broccoli,

Color = green ;

...

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

How can we state it in terms of "Can you prove...?"

*For what pairs of **Thing** and **Color** can you prove **color**?*

# Queries with multiple goals

A query can contain more than one goal.

Here's a query that directs Prolog to find a food that is green:

```
?- food(F), color(F,green).  
F = broccoli ; % (user entered ;)  
F = lettuce ;  
false.
```

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)**?"

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

# Queries with multiple goals, continued

Let's see if any foods are blue:

```
?- color(F,blue), food(F).  
false.
```

Note that the ordering of the goals was reversed. How might the order make a difference?

Goals are always executed from left to right.

What's the following query asking?

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

How about this one?

```
?- food(F), color(F,red), color(F,green).
```

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

## Sidebar: The meaning of a fact

Which of the following is meant by `color(apple,red)`?

All apples are red.

Some apples are red.

Some apples have a red area.

Some apples have a red area at some point in time.

A red apple has existed.

Facts (and rules) are abstractions that we create for the purpose(s) at hand.

An abstraction emphasizes the important and suppresses the irrelevant.

Don't get bogged down by trying to perfectly model the real world!

## Even more queries

Write these queries:

Who likes baseball?

?- likes(Who, baseball).

Who likes a food?

?- food(F), likes(Who,F).

Who likes green foods?

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

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

?- likes(mary,F), food(F),  
color(F, C), food(F2), color(F2,C),  
likes(Who,F2).

```
% fcl.pl
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).
```

## Even more queries, continued

Are any two foods the same color?

?- food(F1), food(F2), color(F1,C), color(F2,C).

F1 = F2, F2 = apple, *% an apple is the same color as an apple(!)*

C = red ;

F1 = F2, F2 = broccoli,

C = green ;

...

To avoid foods matching themselves we can specify "not equal" with  $\neq$  (symbolizing a struck-through  $=$ ).

?- food(F1), food(F2), F1  $\neq$  F2, color(F1,C), color(F2,C).

F1 = broccoli,

F2 = lettuce,

C = green ;

F1 = carrot,

F2 = C, C = orange ;

## Sidebar: Predicates in operator form

Recall that in Haskell, `3 + 4` can be written as `(+) 3 4`.

In Prolog, these two queries are equivalent:

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

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

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`"*.



`display` sheds a little light on the arithmetic oddities we saw earlier.

```
?- display(1 + 2).  
+(1,2)  
true.
```

```
?- display(1 + 2 * 3 - 5).  
-(+(1,*(2,3)),5)  
true.
```

Just FYI: The predicate `op/3` is used to create operators.

```
?- op(200,'xf',--). % precedence 200 postfix operator  
true.
```

```
?- display(x+y--).  
+(x,--(y))  
true.
```

Query `help(op)`.  
to learn more!

# 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.

What are orange foods?

```
?- thing(Name, orange, yes).  
Name = carrot ;  
Name = orange.
```

What things aren't foods?

```
?- thing(Name, _, no).  
Name = dirt ;  
Name = grass ;  
Name = sky.
```

```
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(orange, orange, yes).  
thing(rice, white, yes).  
thing(sky, blue, no).
```

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

## Alternate representation, continued

What is green that is not a food?

?- thing(N,green,no).

N = grass ;

false.

What color is lettuce?

?- thing(lettuce,C,\_).

C = green.

What foods are the same color as lettuce?

?- thing(lettuce,C,\_), thing(N,C,yes), N \== lettuce.

C = green,

N = broccoli ;

false.

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(orange, orange, yes).

thing(rice, white, yes).

thing(sky, blue, no).

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

# Unification

## == and \== are simple tests

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

The operators == and \== are simple tests for equality and inequality. They are roughly equivalent to Haskell's == and /=, and Ruby's == and !=.

```
?- abc == 'abc'.  
true.
```

```
?- 3 \== 5.  
true.
```

Just like comparing tuples and lists in Haskell, 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. (Recursive def'n.)

```
?- 3 + 4 == 4 + 3.  
false.
```

```
?- abc(3 + 4 * 5) == abc(+ (3, 4 * 5)).  
true.
```

The = operator, which we'll read as "unify" or "unify with", provides one way to do *unification*.

If a variable doesn't have a value it is said to be *uninstantiated*. At the start of a query all variables are uninstantiated.

If we unify an uninstantiated variable with a value, the variable is instantiated and unified with that value.

?- **A = 10, write(A).**

**10**

**A = 10.**

It can be read as "Unify **A** with 10 and write **A**."

That might look like assignment but **it's not assignment!**

Along with the output, "10", the value of **A** is shown.

At hand:

?-  $A = 10$ , write( $A$ ).

10

$A = 10$ .

An instantiated variable can be unified with a value only if the value equals whatever value the variable is already unified with.

?-  $A = 10$ , write( $A$ ),  $A = 20$ , write( $A$ ).

10

false.

The unification of the uninstantiated  $A$  with 10 succeeds, and write( $A$ ) succeeds, but unification of  $A$  with 20 fails because  $10 \neq 20$ .

The query fails because its third goal,  $A = 20$ , fails.

In essence the query is saying  $A$  must be 10 and  $A$  must be 20. Impossible!

The lifetime of a variable is the query in which it is instantiated.

```
?- A = 10, B = 20, write(A), write(' '), write(B).  
10, 20  
A = 10,  
B = 20.
```

If we use **A**, **B**, and (out of the blue) **C** in the next query, we find they are uninstantiated:

```
?- write(A), write(' '), write(B), write(' '), write(C).  
_G1571, _G1575, _G1579  
true.
```

Writing the value of an uninstantiated variable produces **\_G<NUMBER>**.

Some say *bound variable* and *free variable* for instantiated and not.



Consider the following:

?-  $A = B, C = 10, C = B, \text{write}(A).$

10

$A = B, B = C, C = 10.$

The code above...

Unifies  $A$  with  $B$  (but both are still uninstantiated).

Unifies  $C$  (uninstantiated) with 10.

Unifies  $B$  with  $C$ . Because  $A$  and  $B$  are already unified and instantiated with 10,  $A$ ,  $B$ , and  $C$  are 10.

How will an initial instantiation for  $A$  affect the query?

?-  $A = 3, A = B, C = 10, C = B, \text{write}(A).$

false.

## Unification, continued

With uninstantiated (free) variables, unification has a behavior when unifying with values that resembles conventional assignment.

With instantiated (bound) variables, unification has a behavior when unifying with values that resembles comparison.

Unification of uninstantiated variables seems like aliasing of some sort.

But don't think of unification as assignment, comparison and alias rolled into one. Think of unification as a distinct new concept!

Another way to think about things:

Unification is not a question or an action, it is a demand!

$\mathbf{X} = 3$  is a goal that demands that  $\mathbf{X}$  must be  $\mathbf{3}$ .

Yet another:

Unifications create constraints that Prolog upholds.

# Unification with structures

Variables can be unified with terms inside a structure.

?-  $x(A,B) = x(10,20)$ .

$A = 10$ ,

$B = 20$ .

?-  $f(X,Y,Z) = f(\text{just}, \text{testing}, f(a,b,c))$ .

$X = \text{just}$ ,

$Y = \text{testing}$ ,

$Z = f(a, b, c)$ .

?-  $f(X,20,Z) = f(10,Y,30)$ ,  $\text{New} = f(Z,Y,X)$ .

$X = 10$ ,

$Z = 30$ ,

$Y = 20$ ,

$\text{New} = f(30, 20, 10)$ .

## Unification with structures, continued

The query `?- color(grass,C).` is a search for facts that can be unified with `color(grass,C).`

Here's a way to picture how Prolog considers the first fact, which is `color(sky, blue):`

`?- Query = color(grass, C), Fact = color(sky, blue), Fact=Query.  
false.`

The demands of the three unifications cannot be satisfied simultaneously.

The same is true for the second fact, `color(dirt, brown).`

The third fact produces a successful unification:

`?- Query = color(grass, C), Fact = color(grass, green),  
Fact=Query.`

`Query = Fact, Fact = color(grass, green),  
C = green.`

# Unification with structures, continued

Consider again this interaction:

?- **food(F).**

**F = apple ;**

**F = broccoli ;**

**F = carrot ;**

**F = lettuce ;**

...

Prolog is able to unify **food(apple)** with **food(F)**, and shows that **F** is unified with **apple**.

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

**food(broccoli)** is unified with **food(F)**, **F** is unified with **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.

# Query evaluation mechanics

# Understanding query execution with the *port model*

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 clauses which unify with the goal.
- exit:** Set a place marker at the clause 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.

# The port model, continued

Example:

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

?-



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



## The port model, continued

trace/0 activates "tracing" for a query.

?- trace, food(X).

Call: (7) food(\_G1571) ? creep

Exit: (7) food(apple) ? creep

X = apple ;

Redo: (7) food(\_G1571) ? creep

Exit: (7) food(broccoli) ? creep

X = broccoli ;

Redo: (7) food(\_G1571) ? creep

Exit: (7) food(carrot) ? creep



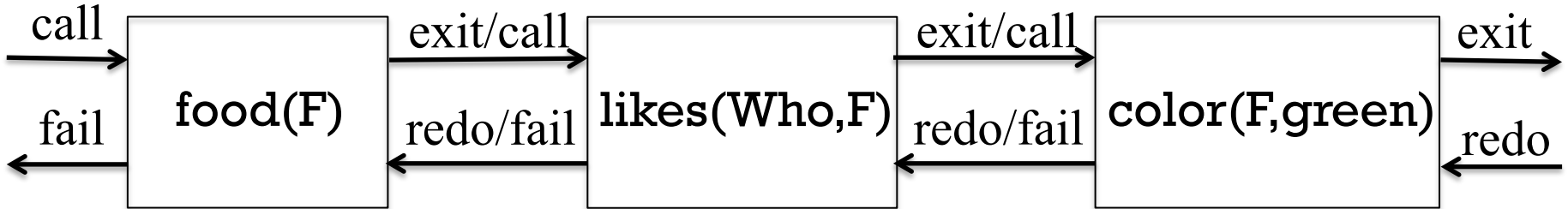
```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

Tracing shows the transitions through each port. The first transition is a call to the goal **food(X)**. The value shown, **\_G1571**, 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.)

# The port model, continued

Who likes green foods?

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



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

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

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

**Next: Trace it!**

We've seen that `write/1` always succeeds and as a side effect outputs the term it is called with. `writeln/1` is similar, but appends a newline.

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

```
apple pie
```

```
true.
```

```
?- writeln(apple), writeln(pie).
```

```
apple
```

```
pie
```

```
true.
```

`nl/0` outputs a newline:

```
?- nl, writeln(middle), nl.
```

```
middle
```

```
true.
```

## Producing output, continued

The predicate `format/2` is much like `printf` in Ruby, C, and others.

```
?- format('x = ~w\n', 101).
```

```
x = 101
```

```
true.
```

`~w` is one of many format specifiers. The "w" indicates to output the value using `write/1`. Use `help(format/2)` to see all the specifiers. (Don't forget the /2!)

If more than one value is to be output, the values must be in a list.

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

```
label = abc, value = 10, x = 3+4
```

```
true.
```

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.

First attempt to 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
```

```
F = carrot ;
```

```
...
```

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.

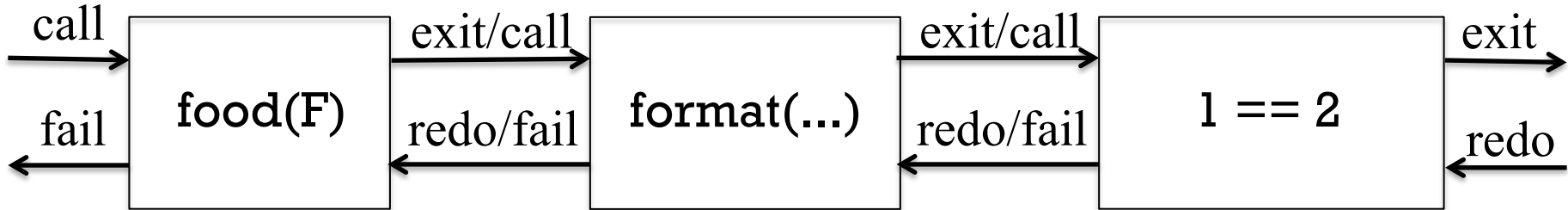
```
?- food(F), format('~w is a food\n', F), 1 == 2.
```

```
apple is a food
```

```
broccoli is a food
```

```
carrot is a food
```

...



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. `1 == 2` always 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.

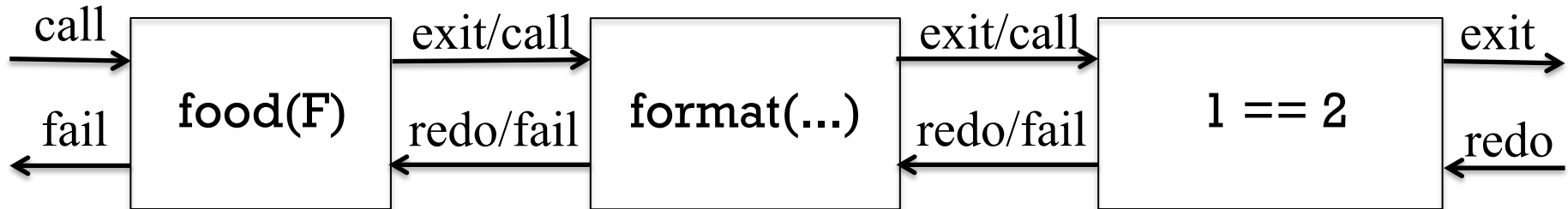
At hand:

?- food(F), format('~w is a food\n', F), 1 == 2.

apple is a food

broccoli is a food

...



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

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

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

At hand:

?- food(F), format('~w is a food\n', F), 1 == 2.

apple is a food

broccoli is a food

...

false.

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 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.

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



# The predicate **fail**

The predicate **fail/0** always fails. It's important to understand that an always-failing goal like `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
```

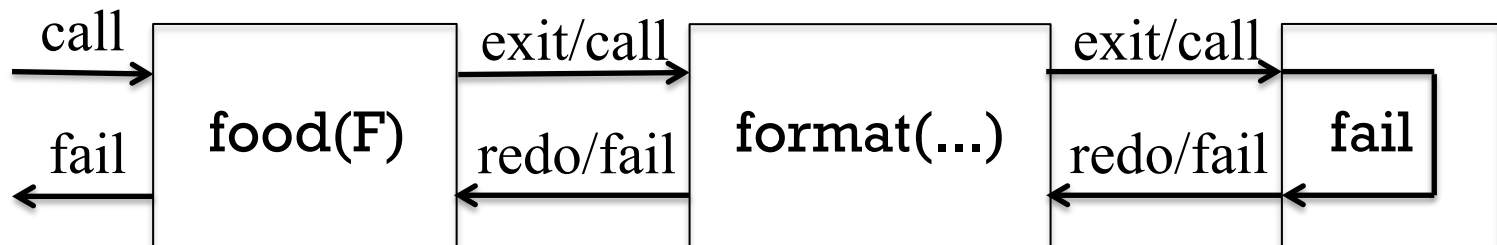
```
broccoli is a food
```

```
...
```

```
rice is a food
```

```
false.
```

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



The **built-in** predicate **between/3** can be used to instantiate a variable to a sequence of integer values:

```
?- between(1,3,X).
```

```
X = 1 ;
```

```
X = 2 ;
```

```
X = 3.
```

Problem: Print this sequence:

```
000
```

```
001
```

```
010
```

```
011
```

```
100
```

```
101
```

```
110
```

```
111
```

How about  
this one?

```
10101000
```

```
10101001
```

```
10101010
```

```
10101011
```

```
10111000
```

```
10111001
```

```
10111010
```

```
10111011
```

```
?- between(0,1,A),between(0,1,B),between(0,1,C),  
   format('~w~w~w\n', [A,B,C]), fail.
```

# Rules

## **showfoods**: a simple *rule*

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

**foods2.pl** starts with a rule and is followed by the food facts:

```
% cat foods2.pl
showfoods :- food(F), format('~w is a food\n', F), fail.

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

Even though **showfoods/0** uses **food/1**, it can either precede or follow the clauses for that predicate.

Remember that **listing/0** can be used to show the clauses that are currently loaded.

At hand:

```
% cat foods2.pl  
showfoods :- food(F), format('~w is a food\n', F), fail.
```

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

Usage:

```
% swipl -l foods2      % Another way to consult a file. (small L)
```

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

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

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

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

$p \text{ :- } q1, q2, q3.$

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 **true**. (If not, then **false**.)

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

At hand are the following rules:

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

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

We saw that we can print all the foods with this query:

```
?- showfoods.
```

```
apple is a food
```

```
broccoli is a food
```

```
carrot is a food
```

```
...
```

```
rice is a food
```

```
false.
```

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.

We send Prolog on a wild goose chase to get our work done!

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

orange is a food

rice is a food

false.

What's wrong?



At hand:

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

```
?- showfoods, showfoods, showfoods.
```

```
apple is a food
```

```
broccoli is a food
```

```
...
```

```
rice is a food
```

```
false.
```

```
[Just one listing of the foods]
```

Why does Prolog say **false.** after printing the foods?

The **showfoods** rule above always fails—we can't get past the **fail** at the end!

We get the output we want but because the first **showfoods** goal ultimately fails Prolog doesn't try the second two goals—it can't get past the first goal!

## 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, a fact, to the predicate **showfoods**. What will it be?

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

Result:

```
?- showfoods.  
apple is a food  
broccoli is a food  
...  
rice is a food  
true.    % Important: Now it says true., not false.
```

Prolog tried the two clauses for the predicate **showfoods** 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.

## Rules with arguments

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

```
food_color(Color) :- food(F), color(F,Color). % in foods2.pl
```

Usage:

```
?- food_color(green).  
true
```

To prove the goal `food_color(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. That unification causes **Color** to be instantiated to the atom **green**.

It then attempts to prove `food(F)`, and `color(F, green)` for some value of **F**.

The response **true** tells us that at least one green food exists, but that's all we know.

## Rules with arguments, continued

At hand:

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

The last slide didn't tell the whole truth. The cursor pauses right after **true**:

```
?- food_color(green).  
true _ (blink...blink...blink)
```

If we type semicolons we see this:

```
?- food_color(green).  
true ;  
true ;  
false.
```

It reveals that **food\_color(green)** is actually finding two green foods but we don't know what they are.

A failure:

```
?- food_color(blue).  
false.
```

## Rules with arguments, continued

At hand:

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

Does `food_color` let us do anything other than asking if there is a food with a particular color?

*We can ask for all the colors of foods.*

```
?- food_color(C).
```

```
C = red ;
```

```
C = green ;
```

```
C = orange ;
```

```
C = green ;
```

```
C = white.
```

We get `green` twice because there are two green foods. We'll later see ways to deal with that.

# Rules with arguments, continued

At hand:

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

```
?- food_color(C).
```

```
C = red ;
```

```
C = green ;
```



These are unified

A very important rule:

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 will instantiate 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 ultimately comes up with **green**.

At hand:

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

Prolog has no analog for "return x"! In Prolog there is no way to say something like this,

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

or this,

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

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

```
?- food_color(C), writeln(C), fail.
```

```
red
```

```
green
```

```
...
```

## Instantiation as "return", continued

Some examples of instantiation as "return" with built-in predicates:

```
?- atom_length(testing, Len).
```

```
Len = 7.
```

```
?- upcase_atom(testing, Caps).
```

```
Caps = 'TESTING'.
```

```
?- term_to_atom(date(4,3,2013), A).
```

```
A = 'date(4,3,2013)'.
```

```
?- term_to_atom(T, 'date(4,3,2013)').
```

```
T = date(4, 3, 2013).
```

```
?- term_to_atom(date(M,D,Y), 'date(4,3,2013)').
```

```
M = 4,
```

```
D = 3,
```

```
Y = 2013.
```



## Instantiation as "return", continued

Problem: Using `term_to_atom` write a predicate with this behavior:

```
?- swap('ten-four', R).  
R = 'four-ten'.
```

First cut:

```
swap(A,Result) :-  
    term_to_atom(T,A),  
    First-Second = T, Swapped = Second-First,  
    term_to_atom(Swapped, Result).
```

Better:

```
swap2(A, Result) :-  
    term_to_atom(First-Second,A),  
    term_to_atom(Second-First, Result).
```

## Instantiation as "return", continued

Problem: Write a predicate with this behavior:

```
?- describe_food(apple-X).
```

```
X = red.
```

```
?- describe_food(X-green).
```

```
X = broccoli ;
```

```
X = lettuce ;
```

```
false.
```

```
?- describe_food(X).
```

```
X = apple-red ;
```

```
X = broccoli-green ;
```

```
...
```

```
X = orange-orange ;
```

```
X = rice-white.
```

Solution:

```
describe_food(Food-Color) :- food(Food), color(Food,Color).
```

## Sidebar: Describing predicates

Recall `between(1,10,X)`. Here's what `help(between)` shows:

```
between(+Low, +High, ?Value)
```

Low and High are integers, High  $\geq$  Low. If Value is an integer, Low  $\leq$  Value  $\leq$  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 and must be instantiated. A question mark indicates that the argument can be input or output, and thus may or may not be instantiated.

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

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

```
X = 1 ;
```

```
...
```

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

```
true.
```

Note: This is a documentation convention;  
do not use the + and ? symbols in code!

# Describing predicates, continued

Another:

`term_to_atom(?Term, ?Atom)`

True if Atom describes a term that unifies with Term. When Atom is instantiated, Atom is converted and then unified with Term. ...

Here is a successor predicate:

`succ(?Int1, ?Int2)`

True if  $\text{Int2} = \text{Int1} + 1$  and  $\text{Int1} \geq 0$ . At least one of the arguments must be instantiated to a natural number. ...

`?- succ(10,N).`

`N = 11.`

There's no `pred` (predecessor) predicate. Why?

`?- succ(N,10).`

`N = 9.`

## Describing predicates, continued

Here is the synopsis for `format/2`:

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

Speculate: What does `sformat/3` do?

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

The minus in `-String` indicates that the term should be an uninstantiated variable.

```
?- sformat(S, 'x = ~w', 1).
```

```
S = "x = 1".
```

```
?- sformat("x = 1", 'x = ~w', 1).
```

```
false.
```

(Yes, we don't know what "... " is yet!)

# Arithmetic

We've seen that Prolog builds structures out of expressions with operators.

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

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

```
?- display(300.0/X*(3+A*0.7**Y)).  
*(/(300.0,_G204),+(3,*(_G212,**(0.7,_G210))))
```

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

```
?- \==(3,4).  
true.
```

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

Question: Why are there no predicates for arithmetic operators?

$X == Y$  works fine as a goal but what we would do with the result of  $3 + 4$ ?

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.
```

`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
```



It is not possible to directly specify an arithmetic expression as an argument of most predicates.

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

```
3+4
```

```
true.
```

```
?- 3+4 == 7.
```

```
false.
```

```
?- 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:

<code>-X</code>	negation
<code>X+Y</code>	addition
<code>X-Y</code>	subtraction
<code>X*Y</code>	multiplication
<code>X/Y</code>	division—produces float quotient
<code>X//Y</code>	integer division
<code>X rem Y</code>	integer remainder
<code>integer(X)</code>	truncation to integer
<code>float(X)</code>	conversion to float
<code>sign(X)</code>	sign of <code>X</code> : -1, 0, or 1

```
?- X is 7777777777777777777777777777*33333333333333333333333333333333.  
X = 259259259259259259259259259256640740740740740740740741.
```

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

```
?- X is e ** sin(pi).  
X = 1.00000000000000002.
```

`help(rem)` is a quick way to open up the documentation section with the arithmetic operations.

# Arithmetic, continued

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

```
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.
```

```
length(point(X1,Y1), point(X2,Y2), Length) :-  
    Length is sqrt((X1-X2)**2+(Y1-Y2)**2). % note structure as sqrt arg!
```

```
?- around(P ,7, N).
```

```
P = 6,
```

```
N = 8.
```

```
?- area(circle(3),A).
```

```
A = 28.274333882308138.
```

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

```
Area = 24.
```

```
?- length(point(3,0),point(0,4),Len).
```

```
Len = 5.0.
```

There are several numeric comparison operators.

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

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

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

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

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

```
?- X is random(10), X > 5.  
X = 9.
```

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

Two 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'
```

```
?- print_grade(80).
```

```
80 -> B
```

```
?- print_grade(55).
```

```
55 -> F
```

# More with rules

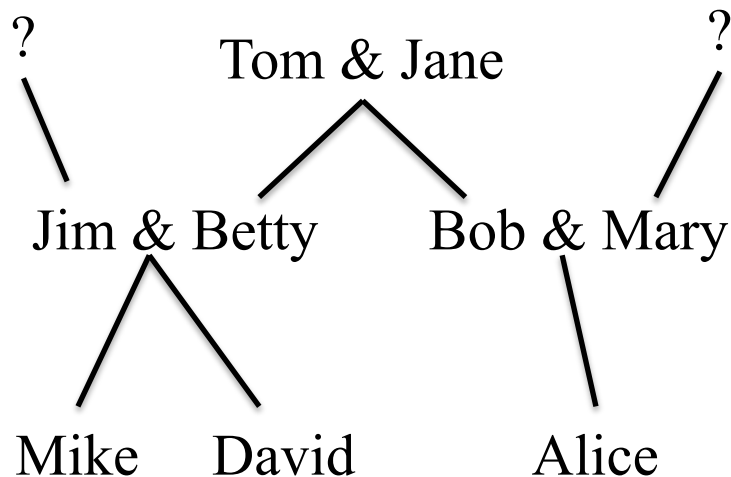
# Parents and children

Here is a set of facts for parents and children:

```
male(tom).  
male(jim).  
male(bob).  
male(mike).  
male(david).
```

```
female(jane).  
female(betty).  
female(mary).  
female(alice).
```

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



Define a rule for `father(F,C)`.

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

```
?- father(F,betty).
```

```
F = tom ;
```

```
false.
```

```
?- father(F,C).
```

```
F = tom,
```

```
C = betty ;
```

```
F = tom,
```

```
C = bob ;
```

```
...
```

```
false.
```

```
?- father(F,_).
```

```
F = tom ;
```

```
F = tom ;
```

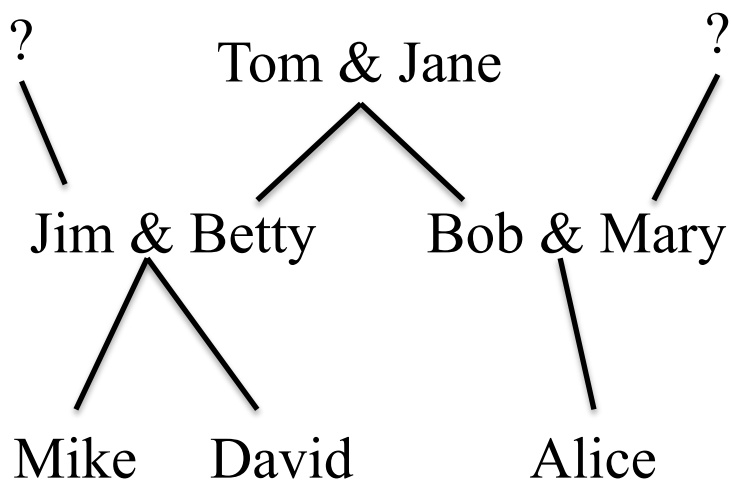
# Parents and children, continued

Here is a set of facts for parents and children:

```
male(tom).
male(jim).
male(bob).
male(mike).
male(david).
```

```
female(jane).
female(betty).
female(mary).
female(alice).
```

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



```
Define grandmother(GM,C).
grandmother(GM,C) :-
    female(GM), parent(GM, P),
    parent(P, C).
```

?- grandmother(GM,C).

GM = jane,

C = mike ;

GM = jane,

C = david ;

GM = jane,

C = alice ;

false.

Or, we could have defined **mother(M,C)** and written **grandmother** using **mother**.



## Parents and children, continued

For who is Tom the father?

?- father(tom,C).

C = betty ;

C = bob.

What are all the father/daughter relationships?

?- father(F,D), female(D).

F = tom,

D = betty ;

F = bob,

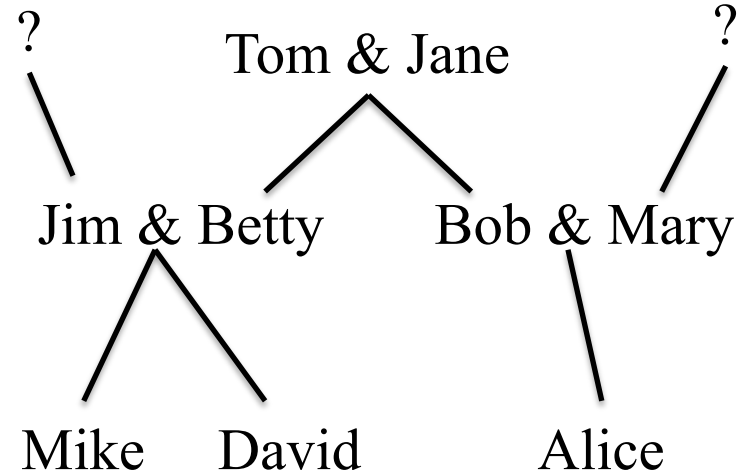
D = alice ;

false.

Who is the father of Jim?

?- father(F,jim).

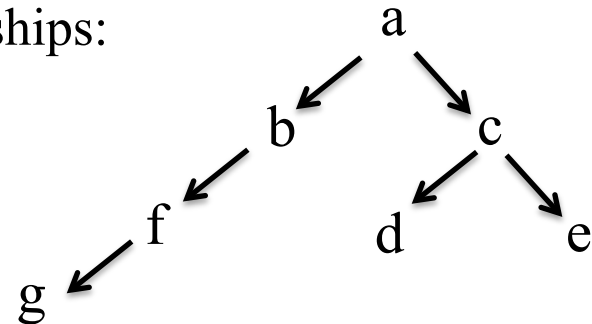
false.



# 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).
```



Here is a recursive predicate for the relationship that **X** is an ancestor of **A**.

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

In English:

"**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**."

Usage:

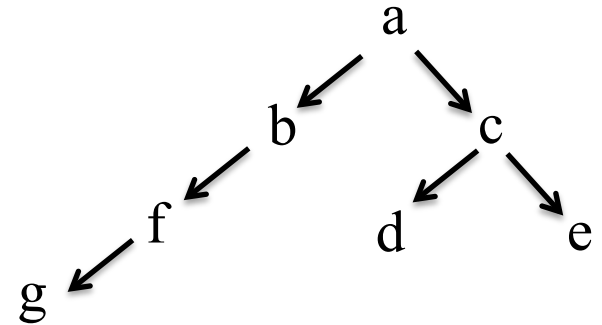
```
?- ancestor(a,f).           % Is a an ancestor of f?  
true
```

```
?- ancestor(c,b).           % Is c an ancestor of b?  
false.
```

# Recursive predicates

At hand:

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



More examples:

```
?- ancestor(c,Descendant). % What are the descendants of c?  
Descendant = e ;  
Descendant = d ;  
false.
```

What's the following query asking?

```
?- ancestor(A, e), ancestor(A,g).  
A = a ;  
false.
```

# Iteration with recursion

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  
true .
```

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`?

A predicate to sum the integers from 0 to N: (ignoring Gauss...)

```
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 .
```

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`?

## Sidebar: A common mistake with arithmetic

Here's 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 its right operand. Further remember that unification is neither assignment or comparison.

Here's the common example of recursion—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](http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/2_2.html)

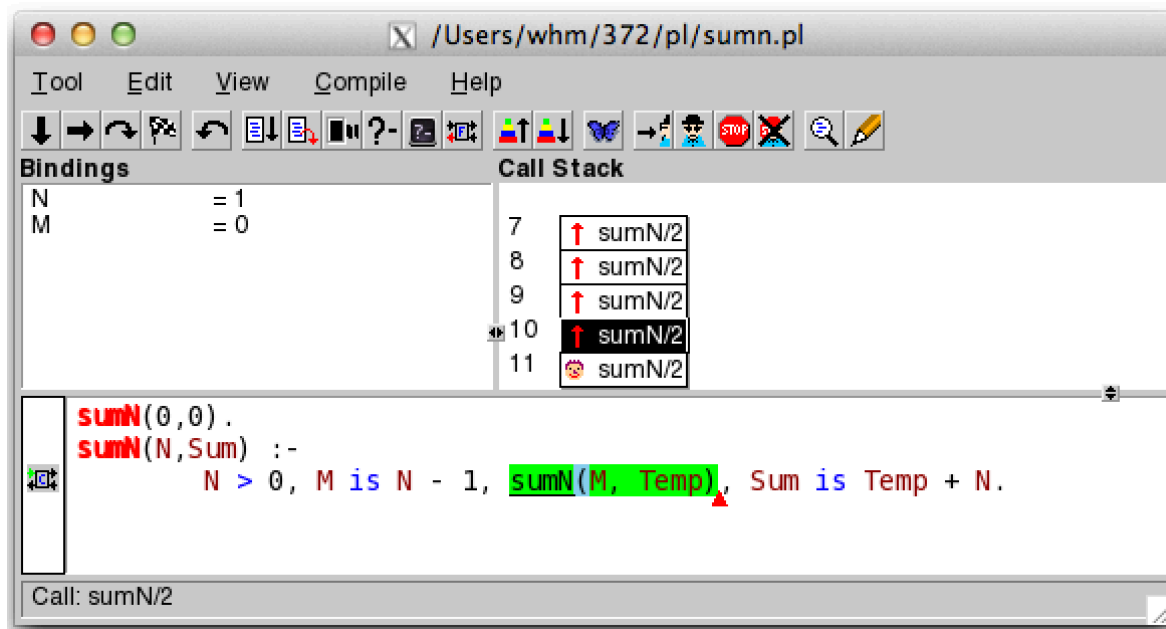
Near the bottom the page is an excellent animation of the computation `factorial(3,X)`. Try it if you don't mind dealing with a Java applet.

# Sidebar: graphical tracing with gtrace

**gtrace** is the graphical counterpart of **trace**. Start it like this:

?- gtrace, sumN(4,Sum).

% The graphical front-end will be used for subsequent tracing



Type space to through step goals one at a time. Click on call stack elements to show bindings in that call. The **ancestor** predicate makes a good demo, too.

**gtrace** should work immediately on Windows and Macs. On a Linux machine in the labs use "**ssh -X ...**" to login to `lectura`, and it should work there, too.



# Generating alternatives with recursion

Here is a predicate that tests whether a number is odd:

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

Note that `N mod 2` works because `=:=` 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:

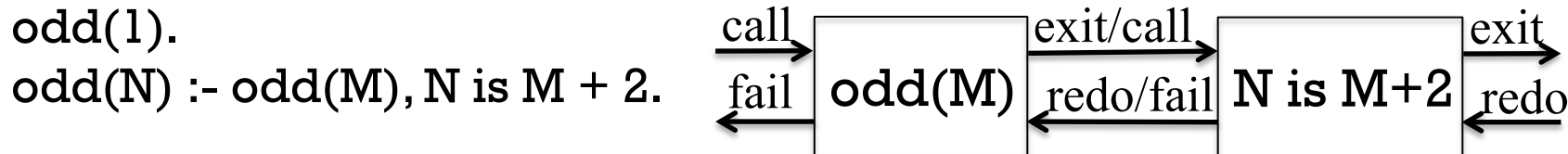
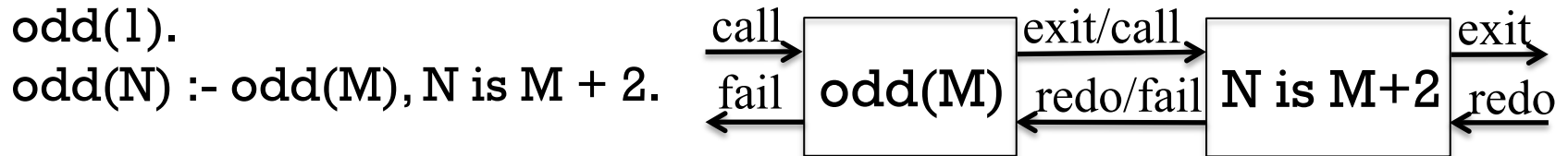
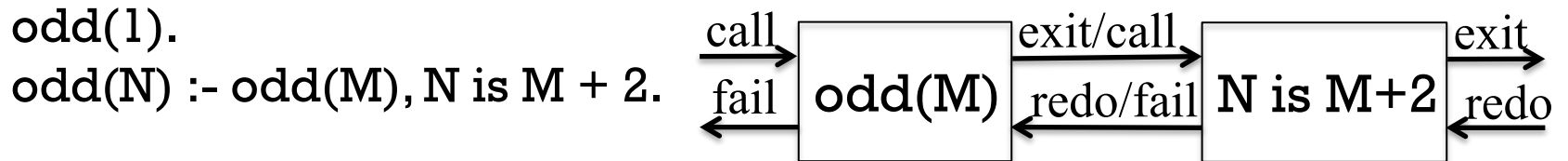
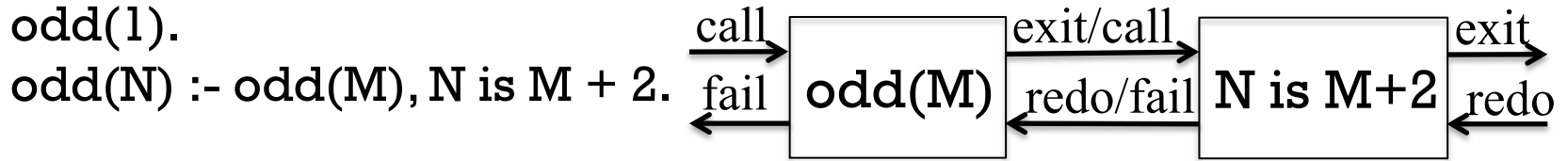
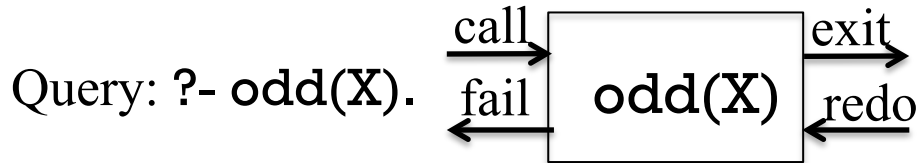
```
?- odd(5).  
true .
```

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

What does `odd(2)` do?

How does `odd(X)` work?

# Generating alternatives, cont.



Corrections:

odd(M)

"fail" on lower left port

# Generating alternatives, continued

For reference:

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

The key point with generative predicates:

**If an alternative is requested, another activation of the predicate is created.**

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

```
odd(1).  
odd(3).  
odd(5).  
odd(N) :- odd(M), N is M + 2.
```

Try `gtrace` with both the two-clause version at the top and the four-clause version just above.

# 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']]]]
```

Common mistake: Entering a list literal as a query is taken as a request to consult files!

```
?- [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]
```

Here are some unifications with 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].

false.

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

Z = [1, [2, 3], 1],

X = 1,

Y = [2, 3].

Note the similarity to Haskell patterns.

What is produced by the following queries?

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

A = B, B = X, X = a.

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

A = B, B = C.

Write a predicate **empty(L)** that succeeds iff **L** is an empty list. Be sure it succeeds only on lists and no other types!

**empty([]).**

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

**f([\_]).**

**f([\_,\_,\_]).**

**f(N) :- number(N), N mod 2 =:= 1.**



# Built-in list-related predicates

SWI Prolog has a number of built-in predicates that operate on lists. One of them is `length/2`:

What do you think `length(?List, ?Len)` does?

Get the length of a list:

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

```
Len = 3
```

And?

Make a list of uninstantiated variables:

```
?- length(L,3).
```

```
L = [_G907, _G910, _G913].
```

Speculate—what will `length(L,N)` do?

```
?- length(L,N).
```

```
L = [],
```

```
N = 0 ;
```

```
L = [_G919],
```

```
N = 1 ;
```

```
L = [_G919, _G922],
```

```
N = 2 ...
```

What does `reverse(?List, ?Reversed)` do?

Unifies a list with a reversed copy of itself.

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

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

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

```
false.
```

Write `palindrome(L)`.

```
palindrome(L) :- reverse(L,L).
```

Speculate—what's the result of `reverse(X,Y)`.?

```
?- reverse(X,Y).
```

```
X = Y, Y = [] ;
```

```
X = Y, Y = [_G913] ;
```

```
X = [_G913, _G916],
```

```
Y = [_G916, _G913] ;
```

```
X = [_G913, _G922, _G916],
```

```
Y = [_G916, _G922, _G913] ;
```

How about `numlist(+Low, +High, -List)`?

?- `numlist(5,10,L)`.

`L = [5, 6, 7, 8, 9, 10]`.

How can we make `[7, 6, ..., 1]`?

?- `numlist(1,7,L), reverse(L,R)`.

`L = [1, 2, 3, 4, 5, 6, 7]`,

`R = [7, 6, 5, 4, 3, 2, 1]`.

`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 = 19.142857142857142`.

# REPLACEMENTS!

"X" out 116-119 in the previous set

## Sidebar: Developing a list-based predicate goal-by-goal

Write a predicate `sumGreater(+Target, -N, -Sum)` that finds the smallest `N` for which the sum of `1..N` is greater than `Target`.

```
?- sumGreater(50,N,Sum).
```

```
N = 10,
```

```
Sum = 55 .
```

```
?- sumGreater(1000000,N,Sum).
```

```
N = 1414,
```

```
Sum = 1000405 .
```

Let's ignore Gauss and have some fun with lists!

Step one: Have a goal that instantiates **N** to 1, 2, ...

?- **between(1, inf, N).**

**N = 1 ;**

**N = 2 ;**

**N = 3 ;**

**...**

Step two: instantiate **L** to lists [1], [1,2], ...

?- **between(1, inf, N), numlist(1,N,L).**

**N = 1,**

**L = [1] ;**

**N = 2,**

**L = [1, 2] ;**

**N = 3,**

**L = [1, 2, 3] ;**

**...**

Step three: Compute sum of 1..N.

?- `between(1, inf, N), numlist(1,N,L), sumlist(L,Sum).`

`N = Sum, Sum = 1,`

`L = [1] ;`

`N = 2,`

`L = [1, 2],`

`Sum = 3 ;`

...

Step four: Test sum against target value.

?- `between(1, inf, N), numlist(1,N,L), sumlist(L,Sum), Sum > 20.`

`N = 6,`

`L = [1, 2, 3, 4, 5, 6],`

`Sum = 21 .`

Note the incremental process followed, adding goals one-by-one and being sure the results for each step are what we expect.

Step four, for reference:

```
?- between(1, inf, N), numlist(1,N,L), sumlist(L,Sum), Sum > 20.
```

```
N = 6,
```

```
L = [1, 2, 3, 4, 5, 6],
```

```
Sum = 21 .
```

Step five: Package as a predicate.

```
% cat sg.pl
```

```
sumGreater(Target,N,Sum) :-
```

```
    between(1,inf,N), numlist(1,N,L), sumlist(L,Sum), Sum > Target.
```

```
% pl -l sg
```

```
...
```

```
?- sumGreater(1000,N,Sum).
```

```
N = 45,
```

```
Sum = 1035 ;
```

```
N = 46,
```

```
Sum = 1081 ;
```

Is it good or bad that it produces alternatives?



## Built-ins for lists, continued

Here's `atom_chars(?Atom, ?Charlist)`:

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

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

Note that `atom_chars` is the analog of both **explode** and **implode** in ML.

Problem: write `rev_atom/2`.

```
?- rev_atom(testing,R).  
R = gnitset.
```

```
?- rev_atom(testing,gnitset).  
true.
```

```
rev_atom(A,RA) :-          % warning: a rule shown in the middle of queries!  
    atom_chars(A,AL), reverse(AL,RL), atom_chars(RA,RL).
```

```
?- rev_atom(X,gnitset).  
ERROR: atom_chars/2: Arguments are not sufficiently instantiated
```

How should `rev_atom`'s arguments be described with `+`, `?`, and `-`?

## Built-ins for lists, continued

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

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

```
L = [1, 3, 7].
```

```
?- atom_chars(prolog, L), msort(L,S), atom_chars(A,S).
```

```
L = [p, r, o, l, o, g],
```

```
S = [g, l, o, o, p, r],
```

```
A = gloopr.
```

If the 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.

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

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

# The `member` predicate

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

`member` can be used to check for membership:

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

`member` can be used to generate the members of a list:

```
?- member(X, [10, twenty, 30]).  
X = 10 ;  
X = twenty ;  
X = 30.
```

Problem: Print the numbers from 100 through 1.

```
?- numlist(1,100,L), reverse(L,R), member(E,R), writeln(E), fail.  
100  
99  
...
```

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

```
?- has_vowel(ack).  
true
```

```
?- has_vowel(pfft).  
false.
```

Solution:

```
has_vowel(Atom) :-  
    atom_chars(Atom,Chars),  
    member(Char,Chars),  
    member(Char,[a,e,i,o,u]).
```

Explain it!

# The `append` predicate

Here's how the documentation describes `append/3`:

```
?- help(append/3).
```

```
append(?List1, ?List2, ?List1AndList2)
```

`List1AndList2` is the concatenation of `List1` and `List2`

Usage:

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

```
R = [1, 2, 3, 4, 5].
```

```
?- numlist(1,4,L1), reverse(L1,L2), append(L1,L2,R).
```

```
L1 = [1, 2, 3, 4],
```

```
L2 = [4, 3, 2, 1],
```

```
R = [1, 2, 3, 4, 4, 3, 2, 1].
```

What else can we do with `append`?

What will the following do?

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

`A = [],`

`B = [1, 2, 3] ;`

`A = [1],`

`B = [2, 3] ;`

`A = [1, 2],`

`B = [3] ;`

`A = [1, 2, 3],`

`B = [] ;`

`false.`

The query can be thought of as asking, "For what values of **A** and **B** is their concatenation `[1,2,3]`?"

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 divide **List3**.

Let's write some more predicates using `append`.

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

Usage:

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

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

```
L = [] ;
```

```
L = [1] ;
```

```
L = [1, 2] ;
```

```
L = [1, 2, 3] ;
```

```
L = [1, 2, 3, 4] ;
```

```
false.
```

Haskell meets Prolog:

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

```
?- take([1,2,3,4,5], 3, L).  
L = [1, 2, 3].
```

```
?- take([1,2,3,4,5], N, L).  
N = 0,  
L = [] ;  
N = 1,  
L = [1] ;  
N = 2,  
L = [1, 2] ;  
...
```

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



# 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,L).
```

```
L = [1, 2] ;
```

```
L = [3, 4] ;
```

```
false.
```

```
?- numlist(1,100,L), chunk(L,5,C), sumlist(C,Sum), between(300,350,Sum).
```

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9 | ...],
```

```
C = [61, 62, 63, 64, 65],
```

```
Sum = 315 ;
```

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9 | ...],
```

```
C = [66, 67, 68, 69, 70],
```

```
Sum = 340 ;
```

```
false.
```

# The **findall** predicate

**findall** can be used to create a list of values that satisfy a goal. A simple example:

```
?- findall(F, food(F), Foods).
```

```
Foods = [apple, broccoli, carrot, lettuce, orange, rice].
```

SWI's documentation:

```
findall(+Template, :Goal, -Bag)
```

Create a list of the instantiations **Template** gets successively on backtracking over **Goal** and unify the result with **Bag**. Succeeds with an empty list if **Goal** has no solutions.

**Template** is not limited to being a single variable. It might be a structure.

The second argument can be a single goal, or several goals joined with conjunction.

The third argument is instantiated to a list of terms whose structure is determined by the template. Above, each term is just an atom.

For reference:

**findall(+Template, :Goal, -Bag)** (*The colon in :Goal means "meta-argument"*)

?- **findall(F, food(F), Foods).**

**Foods = [apple, broccoli, carrot, lettuce, orange, rice].**

Examples to show the relationship of the template and the resulting list:

?- **findall(x, food(F), Foods).**

**Foods = [x, x, x, x, x, x].**

?- **findall(x(F), food(F), Foods).**

**Foods = [x(apple), x(broccoli), x(carrot), x(lettuce), x(orange), x(rice)].**

?- **findall(1-F, food(F), Foods).**

**Foods = [1-apple, 1-broccoli, 1-carrot, 1-lettuce, 1-orange, 1-rice].**

What does **findall** remind you of?

**findall** is said to be a *higher-order predicate*. It's a predicate that takes a predicate, **food(F)** in this case.

Here's a case where the **:Goal** is a conjunction of two goals.

```
?- findall(F-C, (food(F),color(F,C)), FoodsAndColors).
```

```
FoodsAndColors = [apple-red, broccoli-green, carrot-orange,  
lettuce-green, orange-orange, rice-white].
```

**display** sheds some light on that conjunction:

```
?- display((food(F),color(F,C))).
```

```
,(food(_G835),color(_G835,_G838))
```

```
true.
```

It's a two-term structure whose functor is ',' (just a comma).

For reference:

```
findall(+Template, :Goal, -Bag)
```

Describe the following computation.

```
?- numlist(1,9,L),
```

```
  findall(
```

```
    sum(Pfx,Sum),
```

```
    (append(Pfx,_,L), sumlist(Pfx,Sum), Sum<10),
```

```
    Sums).
```

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9],
```

```
Sums = [sum([], 0), sum([1], 1), sum([1, 2], 3), sum([1, 2, 3], 6)].
```

Find all prefixes of the list [1, 2, 3, 4, 5, 6, 7, 8, 9] whose sum is less than 10. Instantiate **Sums** to a list of **sum** structures whose terms are the list and the sum of its elements.

What you need for  
a7 ends here!

# Low-level list processing

The list `[1,2,3]` can be specified in terms of a head and a tail, like this:

`[1 | [2, 3]]`

More generally, a list can be specified 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 | []]`

Haskell equivalents:

`1:[2,3,4]`

`1:2:[3,4]`

`1:2:3:[4]`

`1:2:3:4:[]`

Consider this unification:

$$\begin{aligned} ?- [H|T] &= [1,2,3,4]. \\ H &= 1, \\ T &= [2, 3, 4]. \end{aligned}$$

What instantiations are produced by these unifications?

$$\begin{aligned} ?- [X,Y | T] &= [1, 2, 3]. \\ X &= 1, \\ Y &= 2, \\ T &= [3]. \end{aligned}$$

$$\begin{aligned} ?- [X,Y | T] &= [1, 2]. \\ X &= 1, \\ Y &= 2, \\ T &= []. \end{aligned}$$

$$\begin{aligned} ?- [1, 2 | [3,4]] &= [H | T]. \\ H &= 1, \\ T &= [2, 3, 4]. \end{aligned}$$

$$\begin{aligned} ?- A = [1], B &= [A|A]. \\ A &= [1], \\ B &= [[1], 1]. \end{aligned}$$



# Simple list predicates

A rule that describes the relationship between a list and its head:

`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([2],H).`  
`H = 2.`

`?- head([],H).`  
`false.`

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

Problem: Define `head/2` more concisely.

`head([H | _], H).`

# Implementing **member**

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]).  
true ;  
false.
```

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

How does generation of list elements with **member** work?

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

For reference:

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

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

Problem: Define `member` more concisely.

`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.
```

```
?- last([],X).
```

```
false.
```

```
?- last(L,last), head(L,first), length(L,2).
```

```
L = [first, last] .
```

**last** is a built-in predicate but here's how we'd write it.

```
last([X],X).
```

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

# Implementing length

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(L,1).  
L = [_G901] .
```

```
?- len(L,N).  
L = [],  
N = 0 ;  
L = [_G913],  
N = 1 ;  
L = [_G913, _G916],  
N = 2 ;  
...
```

```
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]).
```

```
true
```

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

```
false.
```

```
?- L = [A,B,C], allsame(L), B = 7, write(L).
```

```
[7,7,7]
```

```
L = [7, 7, 7],
```

```
A = B, B = C, C = 7 .
```

```
?- length(L,5), allsame(L), head(L,x).
```

```
L = [x, x, x, x, x] .
```

Solution:

```
allsame([_]).
```

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

Here's another way to test it:

```
?- allsame(L).
```

```
L = [_G1635] ;
```

```
L = [_G1635, _G1635] ;
```

```
L = [_G1635, _G1635, _G1635] ;
```

```
...
```

# Implementing `append`

Recall the description of the built-in `append` predicate:

```
?- help(append/3).
```

```
append(?List1, ?List2, ?List1AndList2)
```

List1AndList2 is 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 `myapp` instead of `append`. Then try these:

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

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

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).
```

What does the following produce?

```
?- X='.'(3,4).  
X = [3|4].      % Lisp programmers would call this a "dotted-pair".
```



"Can't prove"

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

```
?- food(computer).  
false.
```

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

An incomplete set of facts can produce oddities.

```
?- \+food(cake).  
true.
```

`\+` 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 = orange ;
```

```
F = rice ;
```

```
F = 'Big Mac'.
```

If there's no color fact for a food, will the query above list that food?

How can we see if there are any foods don't have a **color** fact?

```
?- food(F), \+color(F,_).
```

```
F = 'Big Mac'.
```

## "can't prove", continued

The predicate `inedible(X)` succeeds if something is not a food.

```
inedible(X) :- \+food(X).
```

```
?- inedible(rock).  
true.
```

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

```
?- inedible(X).  
false.
```

What's this query asking?

```
?- color(X,_), \+food(X).  
X = sky ;  
X = dirt ;  
X = grass ;  
false.
```

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
```

```
...
```

```
false.
```

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

```
apple
```

```
false.
```

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

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

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

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

## Cut, continued

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. It's "do-or-die" (succeed or fail) with the rule at hand.

Here's an example from *Clause and Effect* by Clocksin.

```
drink(milk) :- true.      % Equivalent to drink(milk).
drink(beer) :- true, !.
drink(gin)  :- true.
```

```
?- drink(gin).
true.
```

```
?- drink(D), writeln(D), fail.
milk
beer
false.
```

Because of the cut in `drink(beer)`'s body, further alternatives are not considered.



At hand:

```
drink(milk) :- true.  
drink(beer) :- true, !.  
drink(gin)  :- true.
```

Queries:

```
?- drink(X), drink(Y).  
X = Y, Y = milk ;  
X = milk,  
Y = beer ;  
X = beer,  
Y = milk ;  
X = Y, Y = beer.
```

```
?- drink(X), !, drink(Y).  
X = Y, Y = milk ;  
X = milk,  
Y = beer.
```

Remember:

A cut in a rule prevents subsequent clauses from being considered in that call.

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

Consider this abstract rule,

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

$x$  :-  $j.$

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 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.

Experiment with [www/pl/cut1.pl](http://www.pl/cut1.pl), where `www` is

<http://www.cs.arizona.edu/classes/cs372/spring14>

Below is a faulty "improvement" for `grade` (original version in the box).

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

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

Usage:

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

Can we fix `grade2` with a cut?

Recall:

A cut in a rule prevents consideration of subsequent clauses for the current call of that predicate. It's "do-or-die" (succeed or fail) with the rule at hand.

A version of `grade` with cuts:

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

Usage:

```
?- grade3(85, G).
G = 'B'.
```

```
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
```

Here's one way to write `max`:

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

```
max(X, Y, Y) :- X < Y.
```

Usage:

```
?- max(10,3,Max).
```

```
Max = 10 ;
```

```
false.
```

Can we shorten it with a cut?

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

```
max(_, Y, Y).
```

(Adapted from *Clause and Effect*.)

Also from *Clause and Effect*, here's a variant of **member**:

```
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**?

## 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.

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

## "cut-fail", continued

average\_taxpayer(X) :-  
foreigner(X), !, fail.

A person is not an average taxpayer if they are a foreigner.

average\_taxpayer(X) :-  
spouse(X, Spouse),  
gross\_income(Spouse, SpouseIncome),  
SpouseIncome > 3000, !, fail.

A person is not an average taxpayer if they've got a spouse and the spouse makes over 3000.

average\_taxpayer(X) :-  
gross\_income(X, Inc),  
Inc > 2000, Inc =< 20\_000.

A person is an average taxpayer if their income is between 2000 and 20,000.

gross\_income(X, GrossIncome) :-  
receives\_pension(X, GrossIncome),  
GrossIncome < 5000, !, fail.

A person is not considered to have a gross income if they receive a pension of less than 5000.

gross\_income(X, GrossIncome) :-  
gross\_salary(X, GrossSalary),  
investment\_income(X, InvestmentIncome),  
GrossIncome is GrossSalary + InvestmentIncome.

investment\_income(X, InvestmentIncome) :- ...

A cut says, "If you get this far, you've picked the right rule for this goal." – C&M



Here's how we could implement `\+` (can't prove) using the higher-order predicate `call/1` and a cut-fail:

```
cant_prove(G) :- call(G), !, fail.  
cant_prove(_).
```

Usage:

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

```
?- cant_prove(food(computer)).  
true.
```

```
?- cant_prove(color(_,purple)).  
true.
```

Is `cant_prove` a higher-order predicate?

## A rainbow of 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.

Which of the preceding examples of cuts are red cuts and which are green cuts?

There are also blue and "grue" cuts.

# Database (knowledgebase) manipulation

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 foods:

**makefoods :-**

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

Evaluating **makefoods** adds facts to the database:

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

```
ERROR: toplevel: Undefined procedure: food/1
```

```
?- makefoods.
```

```
true.
```

```
?- findall(F,food(F),L).
```

```
L = [apple, broccoli, carrot, lettuce, rice].
```

A fact can be "removed" with **retract**:

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

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

**retractall** removes all matching facts.

```
?- retractall(food(_)).  
true.
```

```
?- food(X).  
false.
```

## assert and retract, continued

If we query `makefoods` multiple times, it makes multiple sets of food facts.

```
?- makefoods.
```

```
true.
```

```
?- makefoods.
```

```
true.
```

```
?- findall(F,food(F),Foods).
```

```
Foods = [apple, broccoli, carrot, lettuce, rice, apple, broccoli, carrot,  
lettuce | ...].
```

That behavior might be useful in some cases but if not, start the rule with a `retractall` to get a clean slate:

```
makefoods :-
```

```
    retractall(food(_)),
```

```
    assert(food(apple)),
```

```
    assert(food(broccoli)), assert(food(carrot)),
```

```
    assert(food(lettuce)), assert(food(rice)).
```

## assert and retract, continued

Asserts and retracts are not undone with backtracking.

```
?- assert(f(1)), assert(f(2)), fail.  
false.
```

```
?- f(X).  
X = 1 ;  
X = 2.
```

```
?- retract(f(1)), fail.  
false.
```

```
?- f(X).      % A redo of retract(f(1)) did not restore f(1).  
X = 2.
```

There is no ability to directly change 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
```

```
false.
```

```
?- findall(F, food(F), L).
```

```
L = [apple, carrot, rice].
```

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.  
true.
```

Note that the commands themselves are Prolog terms.

Code is in [www/pl/interp.pl](http://www.pl/interp.pl)

# Simple interpreter, continued

A loop that reads and prints terms:

```
run0 :- prompt(_, '> '),  
        repeat, read(T), format('Read ~w~n', T), T = exit, !.
```

Interaction:

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

How does the loop work?

**prompt/2** sets the prompt that's printed when **read/1** is called.

**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**.

# Simple interpreter, continued

Partial implementation:

```
init :-  
    retractall(value(_)),  
    assert(value(0)).  
  
do(set(V)) :-  
    retract(value(_)),  
    assert(value(V)).  
  
do(print) :- value(V), writeln(V).  
  
do(exit).  
  
run :-  
    init, prompt(_, '> '),  
    repeat, read(T), do(T), T = exit, !.
```

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

How can `add(N)` and `sub(N)` be implemented? (No repetitious code, please!)

**add and subtract:**

```
do(add(X)) :-  
  value(V0),  
  V is V0 + X,  
  do(set(V)).
```

Is this a nested call to **set(V)**?!

```
do(sub(X0)) :-  
  X is -X0,  
  do(add(X)).
```

Tangent: Could **sub** be shortened to the following?

```
do(sub(X)) :- do(add(-X)).
```

Try `add(3+4*5)`, too.

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  
true.
```

# Input handling for tally

`read_line_to_codes` produces a list of ASCII character codes for a line of input.

```
?- read_line_to_codes(user_input,Codes).  
|: ab CD 12  
Codes = [97, 98, 32, 67, 68, 32, 49, 50].
```

```
?- read_line_to_codes(user_input,Codes).  
|: (hit ENTER)  
Codes = [].
```

`atom_codes` can be used to form an atom from a list of codes.

```
?- atom_codes(Atom, [97, 98, 10, 49, 50]).  
Atom = 'ab\n12'.
```

`readline` reads a line and produces an atom.

```
readline(Line) :-  
    read_line_to_codes(user_input, Codes),  
    atom_codes(Line, Codes).
```

```
?- readline(Line).  
|: a test of this  
Line = 'a test of this'.
```

# Counting words

We'll use `word(Word, Count)` facts to maintain counts. We'll write a `count(Word)` predicate to create and update `word/2` facts.

```
?- retractall(word(_, _)).  
true.
```

```
?- count(test).  
true.
```

```
?- word(W, C).  
W = test,  
C = 1.
```

```
?- count(this), count(test), count(now).  
true.
```

```
?- findall(W-C, word(W, C), L).  
L = [this-1, test-2, now-1].
```

For reference:

```
?- retractall(word(_,_)).
```

```
?- count(test), count(this), count(test), count(now).
```

```
?- findall(W-C, word(W,C), L).
```

```
L = [this-1, test-2, now-1].
```

How can we implement `count`?

```
count(Word) :-
```

```
    word(Word,Count0),
```

```
    retract(word(Word,_)),
```

```
    Count is Count0+1,
```

```
    assert(word(Word,Count)), !.
```

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



**tally** clears the counts then loops, reading a line and then processing it.

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

How does **tally** terminate?

**do\_line** breaks up a line into words and calls **count** on each word.

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

## Showing the counts

`keysort/2` sorts a list of **A-B** structures on the value of the **A** terms.

```
?- keysort([zoo-3, apple-1, noon-4],L).  
L = [apple-1, noon-4, zoo-3].
```

With `keysort` in hand we're ready to write `show_counts`.

```
show_counts :-  
    writeln('\n-- Results --'),  
    findall(W-C, word(W,C), Pairs),  
    keysort(Pairs, Sorted),  
    member(W-C, Sorted),  
    format('~w~t~l2 | ~w~n', [W,C]), fail.  
show_counts.
```

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

Full source is in [www/pl/tally.pl](http://www.pl/tally.pl)

## Facts vs. Java maps, Ruby hashes, etc.

What's a key difference between using Prolog facts and maps/hashes/etc. to maintain word counts?

A hash or map can be passed around as a value, but Prolog facts are fundamentally objects with global scope. The collection of **word/2** facts can be likened to a Ruby global, like **\$words = {}**

If we wanted to maintain multiple tallies simultaneously we could add an id of some sort to **word** facts.

Example: We might tally word counts for quotations in a document separately from word counts for body content. Calls to **count** might look like this,

```
count(Type, Word)
```

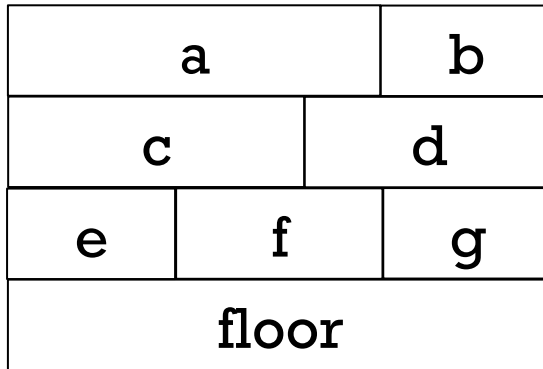
and create facts like these:

```
word(quotes, testing, 3)
```

```
word(body, testing, 10)
```

## 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

Here's one solution: (**blocks.pl**)

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

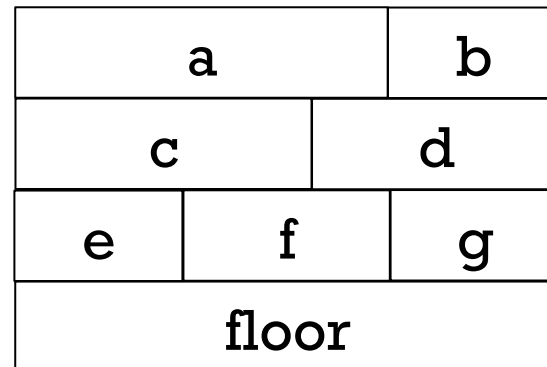
```
remove(B) :-  
    removable(B),  
    retractall(on(B,_)),  
    format('Remove ~w\n', B).
```

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

```
clean :- on(B,floor), remove(B), clean, !.  
clean :- \+on(_,_).
```

How long would it be in Java or Ruby?

Can we tighten it up?



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

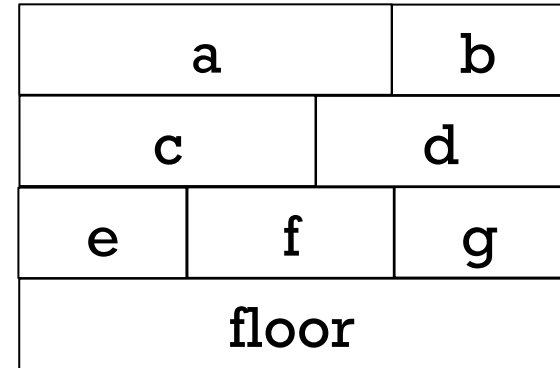
```
?- clean.  
Remove a  
Remove c  
Remove e  
Remove b  
Remove d  
Remove f  
Remove g  
true.
```

# Unstacking blocks, continued

A more concise solution:

```
clean :-  
    on(Block,_), \+on(_,Block),  
    format('Remove ~w\n', Block),  
    retractall(on(Block,_)), clean, !.
```

```
clean :- \+on(_,_).
```



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

Output:

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

Previous sequence:

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

Find a block that's on something and that has nothing on it, and remove it.

Recurse, continuing as long as there's a block that's on something.

# Brick laying puzzle

# 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:

7	3
5	5
6	4

Problem: Write a predicate **laybricks** 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]] ;

Rows = [[7, 3], [5, 5], [4, 6]] ;

Rows = [[7, 3], [6, 4], [5, 5]] .

?- laybricks([7,5,6,4,3,5],12,Rows).

false.

In broad terms, how can we approach this problem?



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] ;
```

```
false.
```

```
?- getone(a, [a,b,a],R).
```

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

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

```
false.
```

```
?- getone(a, [a,b,c,a],R).
```

```
R = [b, c, a] ;
```

```
R = [a, b, c] ;
```

```
false.
```

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

```
?- 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] ;
```

```
false.
```

Implementation:

```
layrow(Bricks, 0, Bricks, []).    % A row of length zero consists of no  
                                  % bricks and doesn't touch the supply.
```

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

Let's write `lay3rows`, which is hardwired for three rows:

```
lay3rows(Bricks, RowLen, [Row1,Row2,Row3]) :-  
    layrow(Bricks,      RowLen,  LeftAfter1, Row1),  
    layrow(LeftAfter1, RowLen,  LeftAfter2, Row2),  
    layrow(LeftAfter2, RowLen,  LeftAfter3, Row3),  
    LeftAfter3 = [].
```

Usage:

```
?- lay3rows([2,1,3,1,2], 3, Rows).  
Rows = [[2, 1], [3], [1, 2]] ;  
...  
Rows = [[2, 1], [1, 2], [3]] ;  
...
```

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

How can we generalize it to N rows?

`laybricks(+Bricks, +NRows, +RowLen, ?Rows)` works like this:

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

`Rows = [[5, 3], [1, 4, 3], [6, 2]]` .

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

`false`.

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

`Rows = [[5, 1, 6], [2, 3, 4, 3]]` .

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

`Rows = [[5, 1], [6], [2, 4], [3, 3]]` .

Implementation:

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

`laybricks(Bricks, Nrows, RowLen, [Row | Rows]) :-`

`layrow(Bricks, RowLen, BricksLeft, Row),`

`RowsLeft is Nrows - 1,`

`laybricks(BricksLeft, RowsLeft, RowLen, Rows)`.

At hand:

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

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

`laybricks` requires that all bricks be used. How can we remove that requirement?

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

*...second rule the same, but with a call to `laybricks2`...*

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

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

Some facts for testing:

```
b(1, [7,5,6,4,3,5]).
```

```
b(2, [5,1,6,2,3,4,3]).
```

```
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
```

We can query `b(N, Bricks)` to set `Bricks` to a particular list.

```
?- b(1,Bricks), laybricks(Bricks, 2, 10, Rows).
false.
```

```
?- b(1,Bricks), laybricks2(Bricks, 2, 10, Rows). % laybricks2
Bricks = [7, 5, 6, 4, 3, 5],
Rows = [[7, 3], [5, 5]] .
```

```
?- b(3,Bricks), laybricks(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]] .
```

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

```
?- b(4,Bricks), laybricks(Bricks,6,12,Rows).
```

```
...[the sound of a combinatorial explosion]...
```

```
^CAction (h for help) ? abort
```

```
% Execution Aborted
```

```
?- statistics.
```

```
8.240 seconds cpu time for 74,996,337 inferences
```

```
...
```

```
true.
```

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

2006 numbers, for contrast:

```
?- statistics.
```

```
8.05 seconds cpu time for 25,594,610 inferences
```

# The "Zebra Puzzle"



# 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. 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 three facts:

- There are five houses.
- The Norwegian lives in the first house.
- Milk is drunk in the middle house.

Those facts can be represented as this goal:

```
Houses = [house(norwegian, _, _, _, _),           % First house
           _,                                     % Second house
           house(_, _, _, milk, _),               % Middle house
           _, _]                                  % 4th and 5th houses
```

house structures have five terms: nationality, pet, smoking preference (remember, it was 1962!), beverage of choice and house color. Instances of **house** structures 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 the list **Houses**, but 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

Some "next to" 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?

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).
```

# The "Zebra Puzzle", continued

At hand:

- 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.

The facts above expressed as 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 few more simple **house & member** goals complete the encoding:

- The Ukrainian drinks tea.  
`member(house(ukrainian, _, _, tea, _), Houses)`
- The Old Gold smoker owns snails.  
`member(house(_, snails, old_gold, _, _), Houses)`
- Kools are smoked in the yellow house.  
`member(house(_, _, kool, _, yellow), Houses)`
- The Lucky Strike smoker drinks orange juice.  
`member(house(_, _, lucky_strike, orange_juice, _),  
Houses)`
- The Japanese smokes Parliaments.  
`member(house(japanese, _, parliment, _, _), Houses)`

# The "Zebra Puzzle", continued

A rule that comprises all the goals:

```
zebra(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 ;  
false.
```

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)  
false.
```

Credit: The code above was adapted from [sandbox.rulemaker.net/ngps/119](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

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

# 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**

**the, dog, cat, etc.** are *terminal symbols*—they can't be expanded into anything else. Generation terminates with them.

**Sentence, Article, Noun and Verb** are *non-terminal symbols*—they can be expanded into something more.

# A very simple parser

Here is a simple parser for the grammar, expressed as clauses: (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]).  
true .
```

```
?- sentence([the,dog,boy]).  
false.
```

```
?- sentence(S).      % Generates all valid sentences
```

```
S = [the, dog, ran] ;
```

```
S = [the, dog, talked] ;
```

```
S = [the, dog, slept] ;
```

```
...
```

Sentence => Article Noun Verb
Article => the   a
Noun => dog   cat   girl   boy
Verb => ran   talked   slept

# A very simple parser, continued

For reference:

**sentence(Words) :-**

**article(Words, Left1), noun(Left1, Left2), verb(Left2, []).**

**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, Left1), noun(Left1, Left2), verb(Left2, []).`

A query sheds light on how `sentence` operates:

```
?- article(Words, Left1), noun(Left1, Left2),  
  verb(Left2, Left3), Left3 = [].
```

```
Words = [the, dog, ran],
```

```
Left1 = [dog, ran],
```

```
Left2 = [ran],
```

```
Left3 = [] .
```

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

```
true .
```

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

Why is `verb`'s result, `Left3`, unified with the empty list?

## A very simple parser, continued

Here's a convenience predicate that splits up a string and calls **sentence**.

**s(String) :-**

**concat\_atom(Words, ' ', String), sentence(Words).**

**sentence(Words) :-**

**article(Words, Left1), noun(Left1, Left2), verb(Left2, []).**

Usage:

?- s('the dog ran').

true .

?- s('ran the dog').

false.

# 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 (terminals) 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 for that grammar.

```
?- [parser1].
```

```
...
```

```
?- listing(sentence).
```

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

```
?- listing(article).
```

```
article(A, B) :-
```

```
    ( A=[a | B]
```

```
    ; A=[the | B]
```

```
    ).
```

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

# Grammar rule notation, continued

A slight variation: (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 non-terminals. **dog**, **cat**, **ran**, **talked**, are terminals, represented with singleton lists.
- The call to **sentence** in **s** now has two terms, to match the rule generated for **sentence**.
- **verb** has both a grammar rule and an ordinary rule, demonstrating they can be mixed. The ordinary rule makes use of **verb0** facts.