Prolog

CSC 372, Spring 2018
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 Colmeraurer.

Originally intended as a tool for working with natural languages.

Achieved great popularity in Europe in the late 1970s.

Was picked by Japan in 1981 as a core technology for their "Fifth Generation Computer Systems" project.

Used in IBM's Watson for NLP (Natural Language Processing).

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

Prolog resources

There are no Prolog books on Safari.

Here are two Prolog books that I like:

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. This version of that PDF has had a searchable text layer added.

Programming in Prolog, 5th edition, by Clocksin and Mellish ("C&M")
A PDF is available via a the UA library:
(http://link.springer.com.ezproxy1.library.arizona.edu/book/10.1007%2F978-3-642-55481-0)

A PDF of Dr. Collberg's Prolog slides for 372 is here:
http://cs.arizona.edu/classes/cs372/spring18/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

Step one with Prolog

You'll eventually see lots of connections between elements of Prolog and other languages, especially Haskell, but for the moment...

$$\boxed{\text{Clear your mind!}}$$

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              (in spring18/prolog/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*.

Facts and queries, continued

At hand:

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

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

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 knowledgebase like this:

```
$ swipl foods.pl  ("swipple")
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
...
?-                    (?- is the swipl query prompt)
```

Once the knowledgebase is 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.

Prolog queries have one or more *goals*. The queries above have 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.

The meaning of **`food(apple).`** depends on where it appears:

- If **`food(apple).`** is typed at the interactive **`?-`** prompt, it is a query.

- If the line **`food(apple).`** is in **`foods.pl`**, it is a fact.

Loading a file that contains a knowledgebase is also known as *consulting* the file.

We'll see later that a knowledgebase can contain "rules", too. Facts and rules are the two types of *clauses* in Prolog.

Try it: What does the query **`?- listing(food).`** show?

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.

```
$ swipl foods.pl
Welcome to SWI-Prolog ...

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

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

```
?- make.
% /home/whm/372/foods compiled 0.00 sec, 2 clauses
true.

?- food(pickle).
true.

?- make.
true.
```

*(**foods.pl** hasn't changed since the last **make**)*

Sidebar: Consulting via query

An alternative to specifying a file on the command line is to consult using a query:

```
$ swipl
Welcome to SWI-Prolog ...

?- [foods].        (do not include the .pl suffix)
% foods compiled 0.00 sec, 8 clauses
true.
```

Consulting a file via a query is commonly shown in texts.

The end result of the two methods is the same.

Facts and queries, continued

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

Speculate: What's the following query asking?

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

Watch what happens when we type semicolons:

```
Edible = apple ;
Edible = broccoli ;
Edible = carrot ;
...
Edible = 'Big Mac'.
```

What's going on?

Facts and queries, continued

An alternative to specifying an atom, like **apple**, in a query is to specify a variable. An identifier that starts with a capital letter is a Prolog variable.

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

- The above query asks, "Tell me something that you know is a food."
- Prolog finds the first **food** fact, based on file order, 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 = 'Big Mac'.

?-
```

Facts are searched in the order they appear in **foods.pl**. Above, the user exhausts all the facts by typing semicolons. Prolog prints '**.**' after the last.

IMPORTANT: 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 those two in Java, Haskell, or Ruby?

Extra credit!

For three points of extra credit:

(1) Get a copy of **foods.pl** and try the examples previously shown.
http://www2.cs.arizona.edu/classes/cs372/spring18/prolog/foods.pl
/cs/www/classes/cs372/spring18/prolog/foods.pl (on lectura)

(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 **eca5.txt**.

(4) Before the next lecture starts, **turnin 372-eca5 eca5.txt**

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, or just use **swipl** on lectura.

Sidebar: **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 1.3.0
| ?- [foods].
compiling foods.pl for byte code...

| ?- food(apple).
yes

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

Most Prolog texts, including Covington and C&M 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 value of **F** can you prove that **F** is a food?
- If we are not satisfied with the value of **F** that's presented, a semicolon directs Prolog to search for another value of **F** for which **food(F)** 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**!

Speculate: What does Prolog think we're doing when we type **make.** ?

We're wanting to see if **make** can be proven!

A side effect of "proving" **make** is the knowledgebase is reconsulted (reloaded) if it's been modified.

Getting and running SWI Prolog

Getting SWI Prolog

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

Lectura:

Just run **swipl** as shown on slide 8+.
Lectura has version 7.2.3 but that's fine for us.

Windows:

Go to **http://swi-prolog.org/download/stable**

The 32-bit version will be fine for our purposes:

SWI-Prolog 7.6.4 for Microsoft Windows (32 bit)

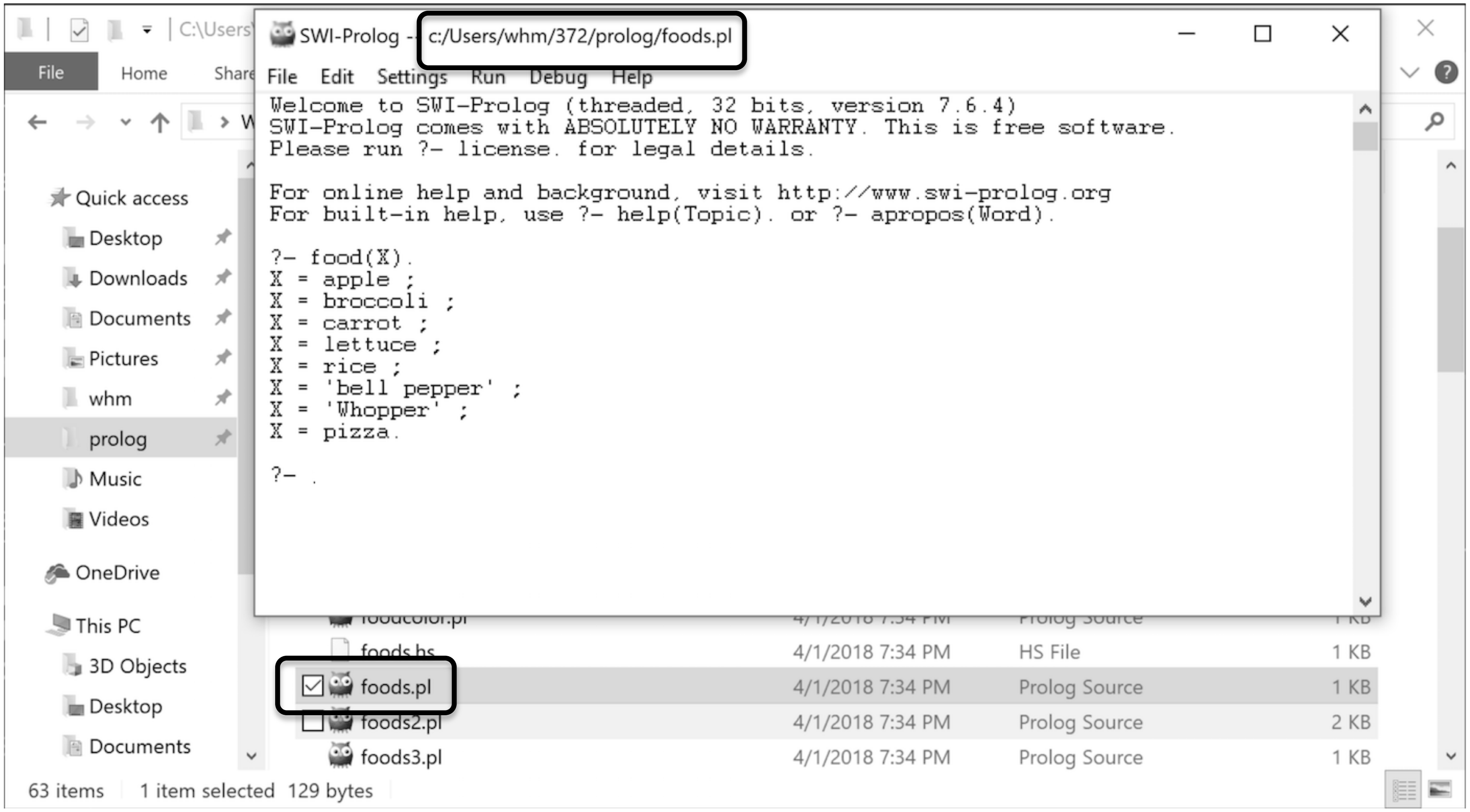
- Pick **Typical** as the **Install type**
- Use **.pl** for file extension (or **.pro**, to avoid a collision with Perl)

TODO:
if "[FATAL ERROR: Could not find system resources]"

```
   env -i HOME=$HOME swipl
```

Running SWI Prolog on Windows

Assuming you associated the **.pl** suffix with SWI Prolog, opening a **.pl** file with File Explorer causes SWI Prolog to consult the file.


SWI-Prolog -- c:/Users/whm/372/prolog/foods.pl
File Edit Settings Run Debug Help
Quick access
Desktop
Downloads
Documents
Pictures
whm
prolog
Music
Videos
OneDrive
This PC
3D Objects
Desktop
Documents
foods.hs 4/1/2018 7:34 PM HS File 1 KB
foods.pl 4/1/2018 7:34 PM Prolog Source 1 KB
foods2.pl 4/1/2018 7:34 PM Prolog Source 2 KB
foods3.pl 4/1/2018 7:34 PM Prolog Source 1 KB
63 items 1 item selected 129 bytes

```
Welcome to SWI-Prolog (threaded, 32 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- food(X).
X = apple ;
X = broccoli ;
X = carrot ;
X = lettuce ;
X = rice ;
X = 'bell pepper' ;
X = 'Whopper' ;
X = pizza.

?- .
```

After editing a file in another window, query **?- make.** to reconsult it.

Getting SWI Prolog for macOS

macOS:

Go to http://swi-prolog.org/download/stable

Get SWI-Prolog 7.6.4 for MacOSX 10.6 (Snow Leopard) and later on intel

You'll need XQuartz 2.7.11 for development tools that use graphics, the handiest of which is perhaps the graphical tracer, launched with the gtrace predicate. (We'll see gtrace later.)

- If you install XQuartz, set your firewall to block incoming connections for X11.bin.

SWI Prolog on macOS

This alias in my **~/.bashrc** lets me run **swipl** from Bash:

```
alias swipl='/Applications/SWI-Prolog.app/Contents/MacOS/swipl'
```

If you get an XQuartz error like the following,

```
$  swipl
?- help(write).
ERROR: /Applications/SWI-Prolog.app/Contents/swipl/xpce
/prolog/boot /pce_principal.pl:155:
    dlopen(/Applications/SWI-Prolog.app/Contents/swipl/lib
/x86_64darwin15.6.0/pl2xpce.dylib, 1): Library not loaded: /opt
/X11/lib/libfontconfig.1.dylib [...lots more...]
```

use the following alias instead:

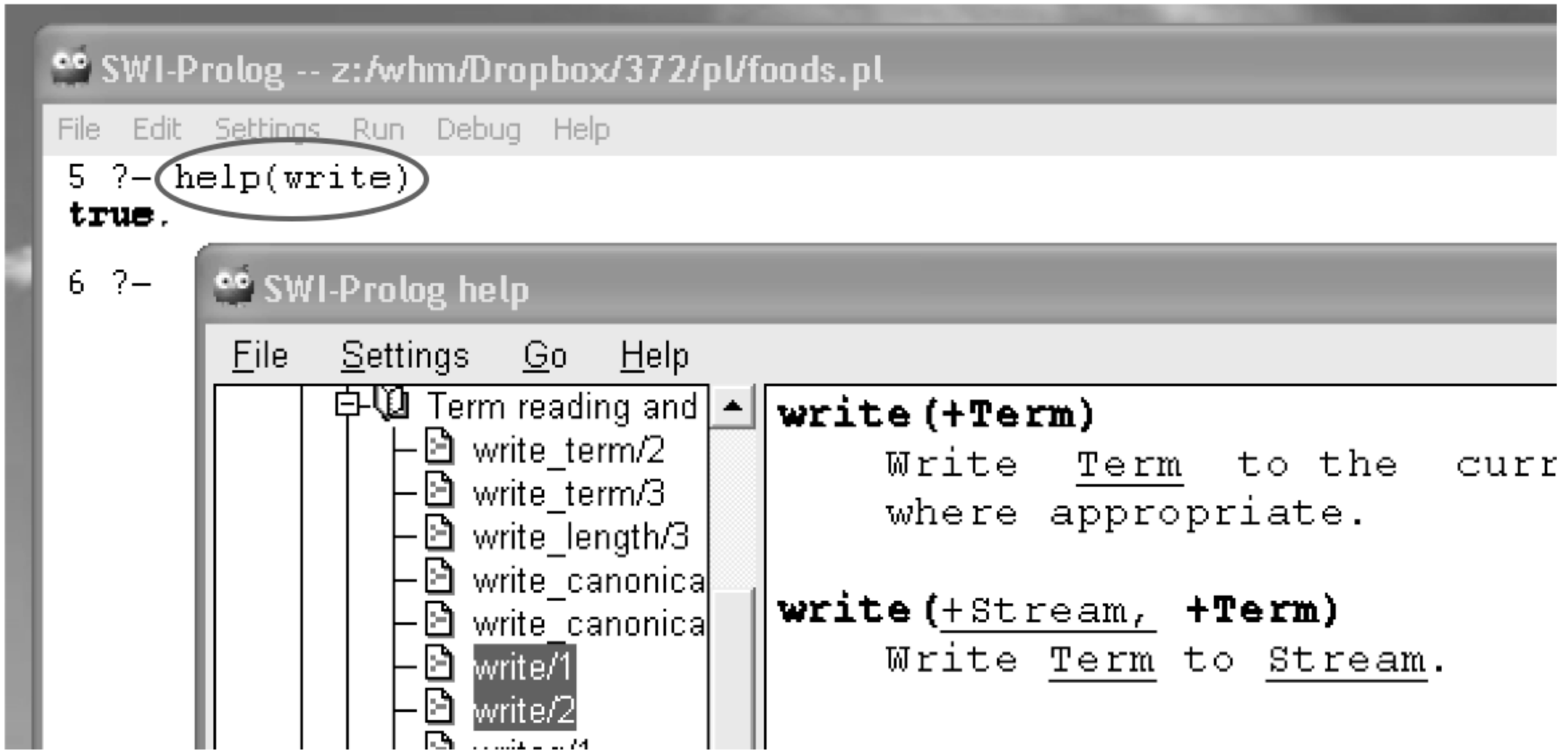
a space!

```
alias swipl="DISPLAY=   /Applications/SWI-Prolog.app/
Contents/MacOS/swipl"
```

(Sets the environment variable **DISPLAY** to an empty string for this invocation of **swipl**.)

Getting help for predicates

To get help for a predicate, query **help(*predicate-name*).** On Windows you'll see:



```
5 ?- help(write)
true.

6 ?-
```

```
write(+Term)
    Write Term to the curr
    where appropriate.

write(+Stream, +Term)
    Write Term to Stream.
```

OS X will be similar, assuming you've got XQuartz installed. If not, or you're using the **swipl** alias with "**DISPLAY= ...**", help will be text-based.

Help will be text based on lectura, but if you login to lectura from a Linux machine in the CS labs with "**ssh -X ...**", you'll get window-based help there, too.

We'll later learn the meanings of **+**, **-**, **?** et al. in predicate documentation.

Getting out of SWI Prolog

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 an "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       s:          C-backtrace
t:          trace       p:           Show PID
h (?):      help
```

Use a to return to Prolog's query prompt; e exits to Bash.

Building blocks

Atoms

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!

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

But we can output an atom with **write**.

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

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

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

Atoms, continued

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

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

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

?- atom(Ant).
false.

?- atom(atom).
true.
```

How can we read **atom(apple)** with a "Can you prove it?" mindset?

"Can you prove **apple** is an atom?"

Numbers

Integer and floating point literals are *numbers*.

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

?- number(3.4).
true.

?- number('100').
false.

?- integer(1e2).
false.
```

Speculate: Are numbers atoms?

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

Numbers, continued

Some arithmetic in Prolog:

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

?- y = 4 + 5.
false.

?- 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: (imagine these lines are in a file)

```
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***.

"The collection of clauses for a given predicate is called a *procedure*."—C&M

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 collide with the built-in predicate ***number/1*** we saw earlier.

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') % 's not needed around number and uno

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

What are the structure functors?
color, **odd**, **number**, and **lunch**, respectively.

What are **sandwich(ham)** and **drink(coke)**?
Terms of the **lunch** structure that are structures themselves.

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

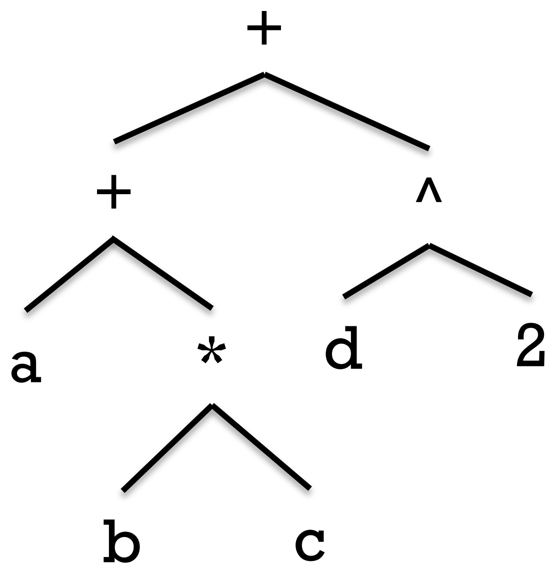
Structures with symbolic functors

Structures can have symbolic functors:

```
+(3,4)
+(3,*(4,5))
\/(x,y)
```

When Prolog encounters an expression with operators, it builds a structure. `display/1` can be used to examine such structures.

```
?- display(a+b*c+d^2).
+(+(a,*(b,c)),^(d,2))
true.
```



Some predicates evaluate structures but most do not, and simply treat the structure as a value.

Sidebar: **op/3**

Query **help(op)** to see the predefined operators and precedences. It shows this:

```
_______________________________________________________________________
| 1200 |xfx  |-->, :-                                                  |
| 1200 | fx  |:-, ?-                                                   |
| 1100 |xfy  |;, |                                                     |
| 1050 |xfy  |->, *->                                                  |
| 1000 |xfy  |,                                                        |
|  990 |xfx  |:=                                                       |
|  900 | fy  |\+                                                       |
|  700 |xfx  |<, =, =.., =@=, \=@=, =:=, =<, ==,                       |
|      |     |=\=, >, >=, @<, @=<, @>, @>=, \=, \==,                   |
|      |     |as, is, >:<, :<                                          |
|  600 |xfy  |:                                                        |
|  500 | yfx |+, -, /\, \/, xor                                        |
|  500 | fx  |?                                                        |
|  400 | yfx |*, /, //, div, rdiv, <<, >>, mod, rem                    |
|  200 |xfx  |**                                                       |
|  200 |xfy  |^                                                        |
|  200 | fy  |+, -, \                                                  |
|  100 | yfx |.                                                        |
|____1_|_fx__|$________________________________________________________|
```

- Left column is precedence; `1200` is lowest.
- `xfy, yfx, fx, fy` etc. are specifications of associativity and infix/prefix/postfix forms.

Challenge: Using only **display**, figure out the difference between **xfx**, **xfy**, and **yfx**. Mail your findings to **372s18**.

op/3, continued

Operators can be created with **op/3**.

```
?- op(150,'xf',--).          % precedence 150 postfix operator
true.

?- op(200, xfy, @).
true.
```

The **f** in **xf** and **xfy** (above) specify where the functor can appear wrt. the operands.

```
?- display(x @ y @ zz--).
@(x,@(y,--(zz)))
true.
```

Operator predicates

Most operators are not predicates—they can't be a goal in a query.

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

But a few operators are predicates. Two are `\==` and `==`. Examples:

```
?- \==(this,that).          % prefix form
true.

?- 3 == 3.                  % infix form
true.

?- 3 == 2+1.
false.
```

Reason: The number `3` is not equal to the structure `2+1`.

How can we characterize the value produced by `==` and `\==`?

They don't produce a value! They simply succeed or fail.

Is **swipl** a REPL?

In conventional languages there are expressions.

A conventional REPL evaluates expressions and prints the value produced.

At **swipl**'s query prompt we can see if one or more goals can be proven.

In the process of trying to prove all the goals, side effects like output may occur and variables may be instantiated but the only result of evaluating goals is success or failure.

So is **swipl** a REPL?

More queries

More queries

Here's a new knowledgebase.

```
$ cat foodcolor.pl
...food facts not shown...
color(sky, blue).
color(dirt, brown).
color(grass, green).
color(broccoli, green).
color(lettuce, green).
color(apple, red).
color(carrot, orange).
color(rice, white).
```

A query about green things:

```
?- color(Thing, green).
Thing = grass ;
Thing = broccoli ;
Thing = lettuce.
```

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

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

More queries

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(Thing,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 ;
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.

How could we state with ~"can you prove"?
"Is there an **F** for which you can prove both **food(F)** and **color(F, green)**?

```
$ cat 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?

What if 100 food facts but 1000 color facts?

<u>Goals are always tried from left to right.</u>

What's the following query asking?

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

Is there a food whose name is its color?

How about this one?

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

Is there a food that is red and 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).
```

```
$ cat fcl.pl
food(apple).
...more food facts...

color(sky, blue).
...more color facts...

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 ;
...
```

How can we avoid those self-matches?

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

F1 = carrot,
F2 = C, C = orange ;
...
```

Alternative representations

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

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

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.
```

The underscore designates an anonymous variable:

- Any value matches
- We don't want to have the value associated with a variable
- No value is displayed

Alternate representation, continued

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

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.
```

Is **thing/3** better or worse than **food/1** and **color/2** combo?

- If you've had 460, how would **thing/3** be described?

  It can be said that **thing/3** is a *denormalized* representation.

Predicate/goal mismatches

Consider this knowledgebase:

```
x(just(testing,date(5,14,2014))).
x(10).
x(10,20).
```

What will the following queries produce?

```
?- x(V).
V = just(testing, date(5, 14, 2014)) ;
V = 10.

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

Having facts with a mix of types and arities is not a problem.

Predicate/goal mismatches, continued

Our knowledgebase as a one-liner:

```
    x(just(testing,date(5,14,2014))). x(10). x(10,20).
```

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. <u>There's no indication that those queries are fundamentally mismatched with respect to the terms in the facts.</u>

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

Predicate/goal mismatches, continued

Our knowledgebase:

```
x(just(testing,date(5,14,2014))). x(10). x(10, 20).
```

Speculate: What will the following produce?

```
?- 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(a,b).
ERROR: stream `a' does not exist
```

We've seen `write/1` but there must also be a `write/2`.

Unification

== and \== are tests

Before talking about unification let's note that == and \== are tests.

- They are roughly equivalent to Haskell's == and /=, and Ruby's == and !=.

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

?- 3 \== 1 + 2.
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.
```

Unification

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

<u>If a variable doesn't have a value it is said to be *uninstantiated*</u>. At the start of a query <u>all</u> 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 **<u>it is not assignment</u>**!

Alternate reading: "Can you unify `A` with `10` and prove `write(A)`?"

Unification, continued

At hand:

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

An <u>instantiated</u> 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 == 20` fails.

The query fails because its third goal, the unification `A = 20`, fails.

In essence the query is saying `A` must be 10 <u>and</u> `A` must be 20. Impossible!

Unification, continued

The lifetime (scope) 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>**.

- **swipl** version difference: latest version produces **_<NUMBER>**

Unification, continued

Consider the following:

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

The code above...

- Unifies `A` and `B` (but both are still uninstantiated).
- Unifies `C` and `10`, instantiating `C`.
- Unifies `C` and `B`.
    - Because `A` and `B` are already unified this also instantiates `A` and `B` to `10`.

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

```
?- A = 3, A = B, C = 10, C = B, write(A).
false.
```

Try it: See if swapping the operands of `=` makes a difference.

Unification, continued

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

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

- Unification of uninstantiated variables seems like aliasing of some sort.

However, do <u>**not**</u> think of unification as assignment, comparison and aliasing rolled into one. <u>Think of unification as a distinct new concept</u>!

Another way to think about unification:

<u>Unification is not a question or an action, it is a demand</u>!

`X = 3` is a goal that demands that `X` must be `3`. If not, the goal fails.

Yet another:

Unifications create constraints that Prolog upholds.

Unification with structures

Unification works with structures, too.

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

?- f(X, Y, Z) = f(just, testing,  f(a,b,c+d)).
X = just,
Y = testing,
Z = f(a, b, c+d).

?- f(X, Y, f(P1,P2,P3)) = f(just, testing,  f(a,b,c+d)).
X = just,
Y = testing,
P1 = a,
P2 = b,
P3 = c+d.
```

TODO: See notes

Unification with structures, continued

```
?- pair(A, A) = pair(3,5).
false.

?- pair(A, A) = pair(3,3).
A = 3.

?- lets(r,a,d,a,r) = lets(C1,C2,C3,C2,C1).
C1 = r,
C2 = a,
C3 = d.

?- f(X,20,Z) = f(10,Y,30),  New = f(Z,Y,X).
X = 10,
Z = 30,
Y = 20,
New = f(30, 20, 10).
```

Unification with structures, continued

Consider again this interaction:

```
?- food(F).
F = apple ;
F = broccoli ;
...
```

The query **food(F)** causes Prolog to search for facts that unify with **food(F)**.

Prolog is able to unify **food(apple)** with **food(F)**. It then 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.

Alternate terminology

Instead of saying a variable is uninstantiated, we can say that it is a *free variable* or an *unbound variable.*

Similarly, we can say "`X` is *bound*" instead of "`X` is instantiated".

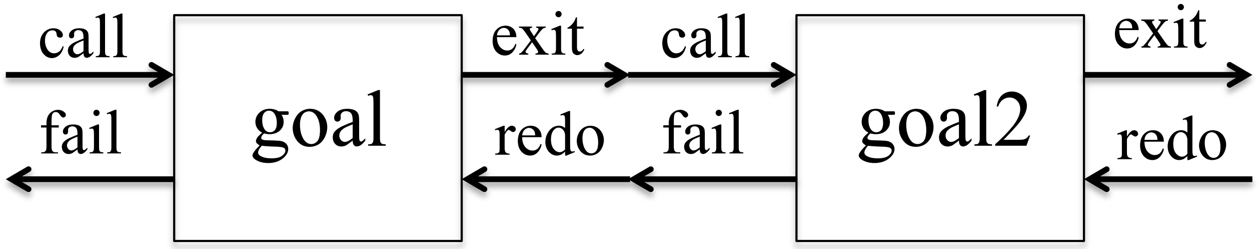
A term can be characterized as being *ground* if it contains no uninstantiated (free) variables.

```
?- ground(coffee).
true.

?- ground(coffee(Beans)).
false.

?- A=10, B=7, ground(A+B+C*3).
false.
```

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.

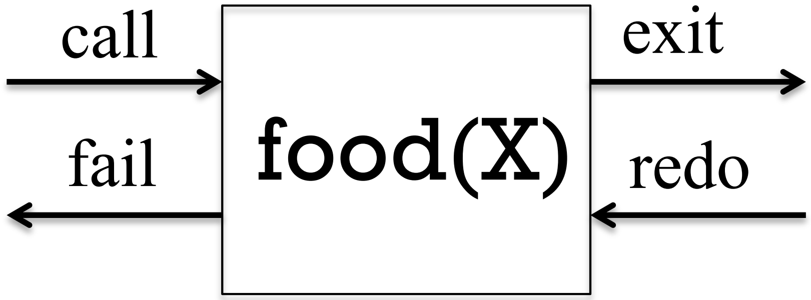
redo: Undo the updates to the variable table [that were made by this goal]. At the place marker, resume the search for a clause which unifies with the goal.

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.

?-
```

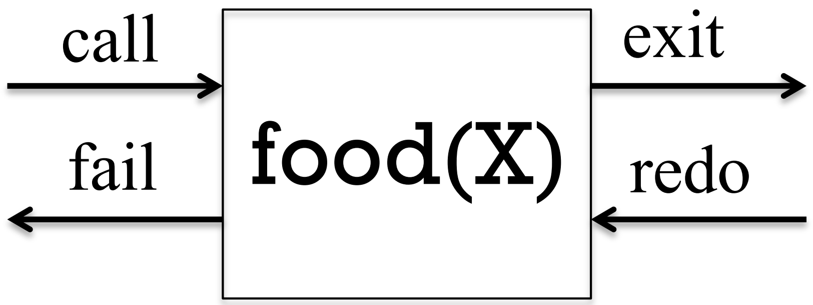


```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

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
```



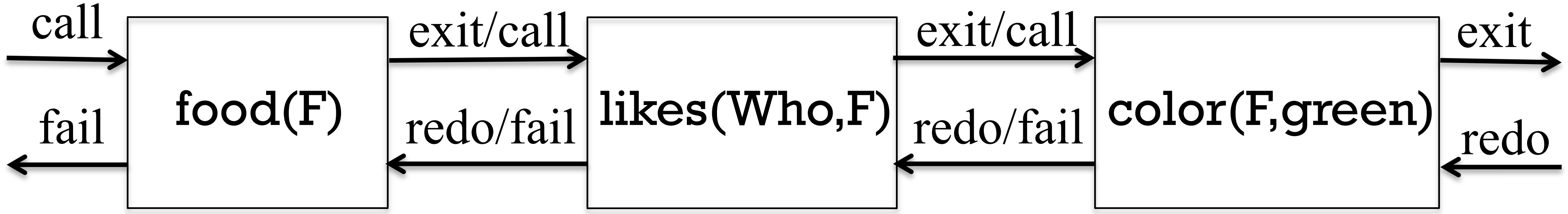
```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

<u>Tracing shows the transitions through each port.</u> 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).
```

<u>**Next: Trace it!**</u> (Use `?- nodebug.` when done.)

Producing output

We've seen that **write/1** always succeeds and, as a side effect, outputs the term it is called with.

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

writeln/1 is similar, but appends a newline.

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

nl/0 outputs a newline. (Note the blank lines before and after **middl**e.)

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

middle

true.
```

Producing output, continued

The predicate **format/2** is conceptually 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.

Producing output, continued

A 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 ;
...
```

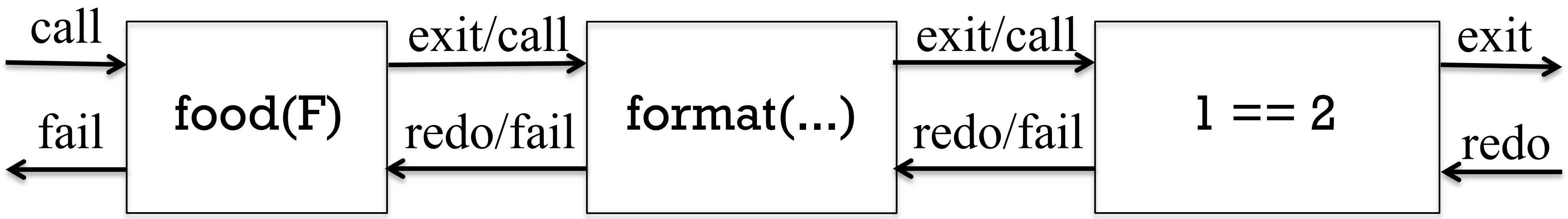
Ick—we have to type semicolons to cycle through them!

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
...
```

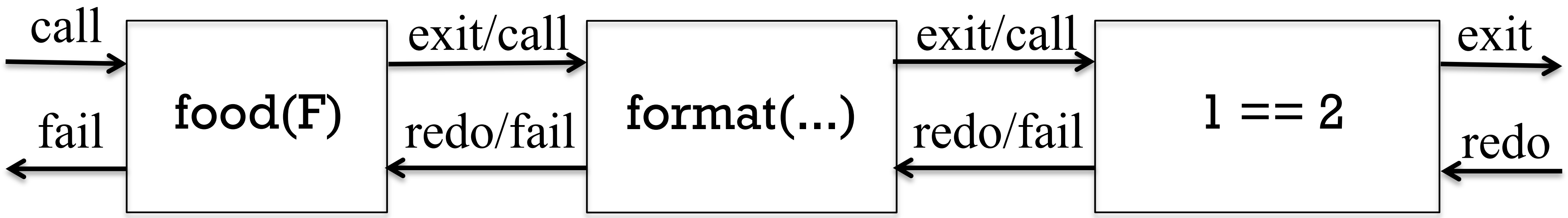


This query is a loop! **food(F)** unifies with the first **food** fact and instantiates **F** to its term, the atom **apple**. Then **format** is called, printing a string with the value of **F** interpolated. **1 == 2** always fails. Control then moves left, into the redo port of **format**. **format** 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.

Backtracking

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."

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

Backtracking, continued

At hand:

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

Predicates respond to "redo" in various ways.

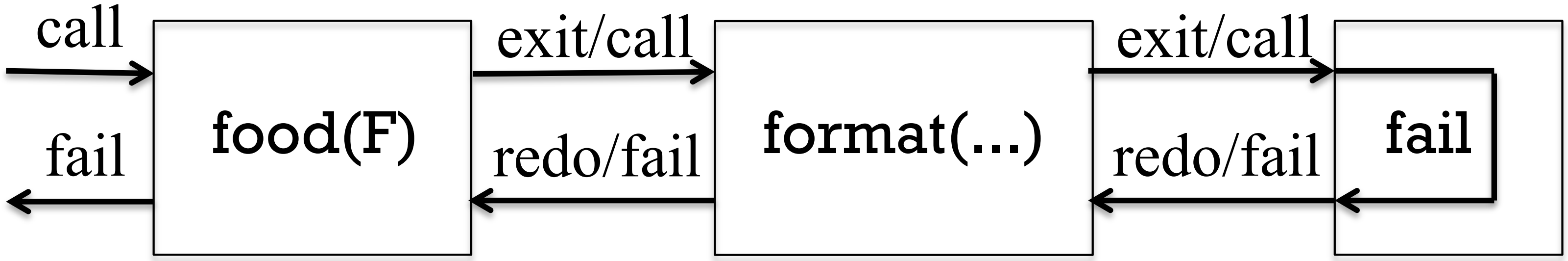
- "redo" for **food(F)** simply uninstantiates (unbinds) **F** and searches for another **food** clause to unify with and instantiate **F** again. If there is one, the goal exits (control goes to the right). If not, it fails (control goes to the left).
- For **format('~w is a food\n', F)** "redo" causes the goal to fail, but the output isn't somehow retracted. (!)
- We'll see other kinds of responses, too, but "redo" **always** causes any previous unifications to be undone.

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:



Sidebar: **between**

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
```

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

How about this one?

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

Rules

Rule basics

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

Here's a rule:

```
increasing(A,B,C) :- A < B, B < C.
```

If all the goals in a rule are true, then the rule is true.

Usage:

```
?- increasing(2,5,10).
true.

?- increasing(5,10,3).
false.
```

Rule anatomy:

$$\underset{\text{head}}{\underset{\uparrow}{\texttt{increasing(A,B,C)}}} \quad \overset{\text{neck}}{\overset{\downarrow}{\texttt{:-}}} \quad \underset{\text{body}}{\underset{\uparrow}{\texttt{A < B, B < C.}}}$$

Rule basics, continued

A rule:

```
    increasing(A,B,C) :- A < B, B < C.
```

A query that uses the rule:

```
    ?- increasing(2,5,10).
```

As part of query processing, the terms in the head of the rule are unified with terms in the query.

- Looks like parameter passing in other languages.
- But the underlying mechanism is unification, with all that implies.

The scope of the variables `A`, `B`, and `C` is only this rule.

Rule basics, continued

Problem: Write a rule that tells Prolog how to prove that three values are in decreasing order.

Usage:

```
?- decreasing(10,3,2).
true.

?- decreasing(10,3,12).
false.
```

Solution:

```
decreasing(A,B,C) :- increasing(C,B,A).
```

For reference:

```
increasing(A,B,C) :- A < B, B < C.
```

increasing/3 and **decreasing/3** are in **spring18/prolog/rules2.pl**

Sidebar: Horn Clauses

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

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

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

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

SKIP

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.

Rule basics, continued

increasing/3 and **decreasing/3** are predicates with one clause.

A predicate can have many clauses. Prolog will try each clause in turn.

Problem: Write a predicate **ordered(A,B,C)** that's true iff its three terms are in either increasing or decreasing order.

Examples:

```
?- ordered(1,2,3).
true .

?- ordered(7,3,5).
false.
```

Solution:

```
ordered(A,B,C) :- increasing(A,B,C).
ordered(A,B,C) :- decreasing(A,B,C).
```

Rule basics, continued

Again:

A predicate can have many clauses. Prolog will try each clause in turn.

Recall `food/1`:

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

For reference:

```
ordered(A,B,C) :- increasing(A,B,C).
ordered(A,B,C) :- decreasing(A,B,C).
```

The same mechanism makes...

`?- food(F).` cycle through the `food/1` facts

`?- ordered(3,1,5)` try both `increasing(3,1,5)` and `decreasing(3,1,5).`

Example: finding increasing sequences

Here's a collection of values expressed as facts:

```
v(10). v(7). v(12). v(3).
```

Problem: Write a query that will print all combinations of those values that are in increasing order:

```
?- v(A), v(B), v(C), increasing(A,B,C),
   format('~w ~w ~w~n', [A,B,C]), fail.
7 10 12
3 10 12
3 7 10
3 7 12
false.
```

How does it work?

Problem: Write the above in Java, Haskell, or Ruby.

Example, continued

Problem: Package the preceding query as a predicate **all_incr/0**:

```
?- all_incr.
Increasing:
7 10 12
3 10 12
3 7 10
3 7 12
true.
```

Solution: (almost)

```
all_incr :-
    writeln('Increasing:'),
    v(A), v(B), v(C),
    increasing(A,B,C),
    format('~w ~w ~w~n', [A,B,C]),
    fail.
```

Execution:

```
?- all_incr.
Increasing:
7 10 12
3 10 12
3 7 10
3 7 12
false.
```

What's wrong, and why?

Example, continued

In **rules2.pl**:

```
    all_incr :- writeln('Increasing:'), v(A), v(B), v(C),
        increasing(A,B,C), format('~w ~w ~w~n', [A,B,C]), fail.
```

Execution:

```
  ?- all_incr.
  ...
  false.
```

all_incr's rule cannot be proven but in the process of trying prove it, the desired lines are printed.

How can we get both the output we want and a query that succeeds?

Solution:

```
    all_incr :- writeln('Increasing:'), ...the rest..., fail.
    all_incr.
```

Example, continued

At hand:

```
    all_incr :- writeln('Increasing:'), ...the rest..., fail.
    all_incr.
```

Remember: Each clause of a predicate is tried in turn.

What happens:

- We do the query **?- all_incr.**
- The first clause of **all_incr** ultimately fails but output is produced as a side-effect.
- The second clause of **all_incr**, a fact, is trivially proven.

Problem

Write a predicate `p(A, R, B)` that tests whether the relationship `R` holds between `A` and `B`.

Usage:

```
?- p(3, lt, 4).     % The atom lt represents "less than"
true .

?- p(5,gt,3).
true.

?- p(2,eq,1).
false.
```

Solution:

```
p(A, lt, B) :- A < B.
p(A, eq, B) :- A == B.
p(A, gt, B) :- A > B.
```

Problem, continued

At hand:

```
p(A, lt, B) :- A < B.
p(A, eq, B) :- A == B.
p(A, gt, B) :- A > B.
```

What else can we do with `p/3` besides seeing if a particular relationship holds?

```
?- p(3,R,4).
R = lt .

?- p(3,X,2).
X = gt.
```

Instantiation as "return"

At hand: (in **rules2.pl**)

```
    p(A, lt, B) :- A < B.
    p(A, eq, B) :- A == B.
    p(A, gt, B) :- A > B.
```

Prolog predicates either succeed or fail. There's no notion of returning a value.

Instead, predicates that need to produce a result use instantiation, and underlying it, unification.

Given

```
    ?- p(4,R,3).
```

Prolog responds with

```
    R = gt.
```

meaning "I can prove `p(4,R,3)` if `R` is `gt`."

We can then use `R` in a later goal in the same query:

```
     ?- p(4,R,3), ..., q(R), ...
```

Instantiation as "return", continued

Recall **between/3**:

```
?- between(1,3,X).
X = 1 ;
X = 2 ;
X = 3.
```

A new rule:

```
nl_when(N, When) :- N == When, nl.
nl_when(_,_).
```

What does the following query do?

```
?- between(1,3,X), between(1,3,Y), p(X,R,Y),
   format('~w ~w ~w, ',[X,R,Y]), nl_when(Y,3), fail.
1 eq 1, 1 lt 2, 1 lt 3,
2 gt 1, 2 eq 2, 2 lt 3,
3 gt 1, 3 gt 2, 3 eq 3,
false.
```

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'.

?- char_type('A', T).
T = alnum ;
T = alpha ;
...
T = upper(a) ;
...
T = xdigit(10).
```

Replacements: 92+

Instantiation as "return", continued

Some predicates will fill in uninstantiated terms.

```
?- term_to_atom(date(10,1,1891), A).
A = 'date(10,1,1891)'.

?- term_to_atom(date(M,D,Y), 'date(10,1,1891)').
M = 10,
D = 1,
Y = 1891.
```

Joseph Astier, 372 Fall 1996, said,

"A Prolog predicate is like a DC motor: If you apply electricity to the motor, the rotor turns.   If you turn the rotor, it generates electricity."

(Original Thought!)

Instantiation as "return", continued

In Prolog, what is **ten-four**?

A two-term structure with the functor '**-**'. Its terms are the atoms **ten** and **four**. (**display(ten-four)** shows **-(ten,four)**.)

Consider this predicate:

```
swap_struct(X-Y, R) :- R = Y-X.
```

Usage:

```
?- swap_struct(ten-four, X).
X = four-ten.

?- swap_struct(X, 10-20).
X = 20-10.
```

A little more...

```
?- swap_struct(a-b*c,R).
R = b*c-a.

?- swap_struct(a-b+c,R).
false.
```

Can **swap_struct** be simplified?

```
swap_struct(X-Y, Y-X).
```

Instantiation as "return", continued

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

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

Solution:

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

Instantiation as "return", continued

Problem: Write a predicate with these four behaviors:

```
?- 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.
```

The fourth:

```
?- describe_food(apple-red).
true.

?- describe_food(apple-blue).
false.
```

Solution:

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

Nested goals

What is the output of the following query?

```
?- writeln(food(F)), fail.
food(_6100)
false.
```

What's happening?

We're calling **writeln** with a one-term structure, with functor **food**, whose term is the uninstantiated variable **F**.

Unlike expressions in conventional languages, Prolog goals don't nest.

Sidebar: Describing predicates

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

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

- If an argument has a plus prefix, like **+Low** and **+High**, it means that the argument is an input to the predicate 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 Int2= Int1+1 and Int1>=0. At least one of the arguments must be instantiated to a natural number. ...

What are two ways **succ/2** can be used?

```
?- succ(10,N).
N = 11.

?- 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".
```

Arithmetic

Why are there no arithmetic predicates?

We've seen that there are predicates for comparisons but not for arithmetic operations:

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

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

Why is this the case?

- Queries succeed or fail.
- The result of a comparison can be viewed as success or failure but there's simply no place for the result of `3 + 4` to appear. (There's no "outlet" for it.)

is/2

The predicate **is(?Value, +Expr)** evaluates **Expr**, a structure representing an arithmetic expression, and unifies the result with **Value**.

```
?- is(X, 3+4*5).
X = 23.
```

The atom **is** has been defined to be an operator using **op/3**.

```
?- X is 3 + 4, Y is 7 * 5, Z is X / Y.
X = 7,
Y = 35,
Z = 0.2.
```

All variables in a structure being evaluated by **is/2** must be instantiated.

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

(The query **?- ground(3+X).** fails—the term **3+X** has free variables.)

Arithmetic, continued

is/2 supports a number of arithmetic operations. Here are some of them:

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

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

```
?- X is 7777777777777777777777*3333333333333333333333.
X = 25925925925925925925925664074074074074074074074 1.
```

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

```
?- X is e ** sin(pi).
X = 1.0000000000000002.
```

What are e *and* pi*? Is* sin *a Prolog"function"?*

Arithmetic, continued

Problem: Write a predicate **around/3** that works like this:

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

Solution:

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

We can use **around** to test, too, but the second term must be "ground".

```
?- around(1,2,3).
true.

?- around(1,X,3).
ERROR: is/2: Arguments are not sufficiently instantiated
```

Arithmetic, continued

Here are some predicates to compute the area of shapes. Note the use of unification to "label" the structure's term(s).

```
area(rectangle(W,H), A) :- A is W * H.
area(circle(R), A) :- A is pi * R ** 2.
```

Usage:

```
?- area(circle(3), A).
A = 28.274333882308138.

?- area(rectangle(5,7), A).
A = 35.
```

Problem: A "figure 8" is two circles touching at a point. Write another clause for `area/2` so that `area(figure8(3,4),A)` will work.

```
area(figure8(R1,R2), A) :-
    area(circle(R1),A1), area(circle(R2),A2), A is A1 + A2.
```

Arithmetic, continued

Do the following queries work?

```
?- area(rect(pi*3,sqrt(e/2)),R).
R = 10.98761340505857.

?- A=5, B=A, C is A+B, area(rect(A+B,area(figure8(B,C))),R).
ERROR: Arithmetic: `figure8/2' is not a function
```

Comparisons

There are several numeric comparison operators.

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

Just like **is/2**, these operators 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.

The "singleton" warning(!)

The "singleton" warning

Here's a predicate **add(+X,+Y, ?Sum)**:

```
$ cat add.pl
add(X, Y, Sum) :- S is X + Y.
```

Bug: **Sum** is used in the head but **S** is used in the body!

Observe what happens when we load it:

```
$ swipl add.pl
Warning: /cs/www/classes/cs372/spring18/prolog/add.pl:1:
    Singleton variables: [Sum,S]
...
```

What is Prolog telling us with that warning?

The variables **Sum** and **S** appear only once in the rule on line 1.

Fact: <u>If a variable appears only once in a rule, its value is never used</u>.

<u>**A singleton warning may indicate a misspelled or misnamed variable.**</u>

Pay attention to singleton warnings!

Singleton warnings are easy to overlook!

Note that singleton warnings appear **before** "Welcome to SWI-Prolog"!

```
$ swipl print_stars.pl          (first version)
Warning: /cs/www/classes/cs372/spring18/prolog/print_stars.pl:1:
        Singleton variables: [X]
Warning: /cs/www/classes/cs372/spring18/prolog/print_stars.pl:2:
        Singleton variables: [N]
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
software, and you are welcome to redistribute it under certain
conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?-
```

All errors found when consulting a file appear **BEFORE** "Welcome to SWI...".

Recursive predicates

Recursive predicates

Predicates can be recursive.

Here is a recursive 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 .
```

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

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

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

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? How about **N > 0**?

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

Sidebar: A common mistake with recursion

Here's a correct definition for **factorial**:

```
factorial(0, 1).
factorial(N, F) :- N > 0,
                   M is N - 1,
                   factorial(M, FM),
                   F is N * FM.
```

Here is a **common mistake**:

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

Usage:

```
?- factorial(4,F).
false.
```

What's the mistake?

Under what circumstances does **F is N * F** succeed?

Generating alternatives with recursion

Here's 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 does the behavior of the two differ?

Generating alternatives, continued

For reference:

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

Usage:

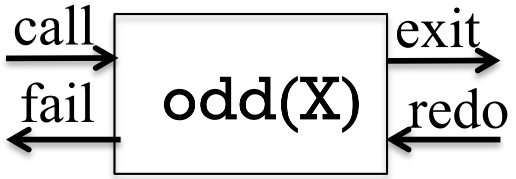
```
    ?- odd(5).
    true .

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

What does ***odd(2)*** do?

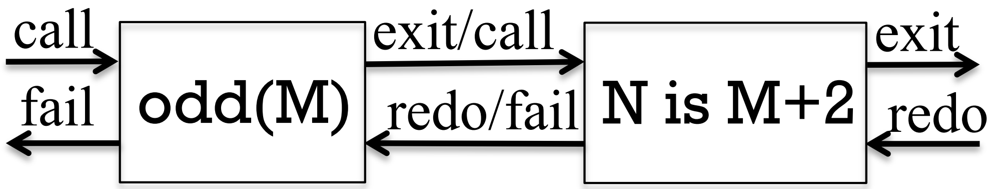
How does ***odd(X)*** work?

Generating alternatives, cont.

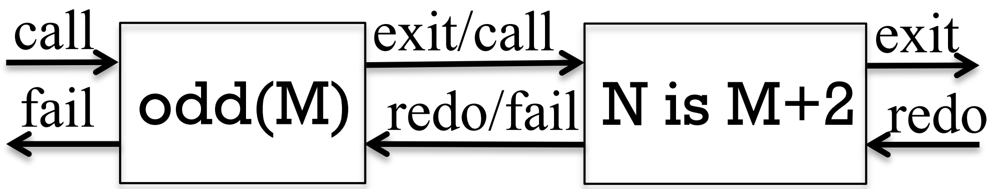
Query: ?- odd(X).



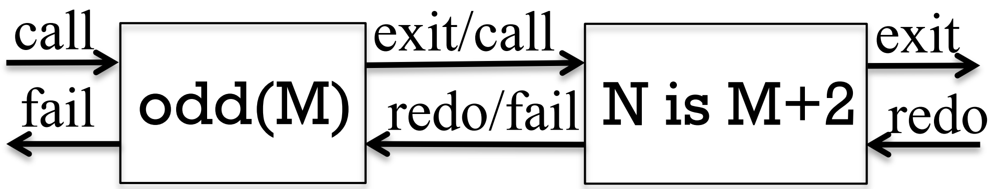
```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



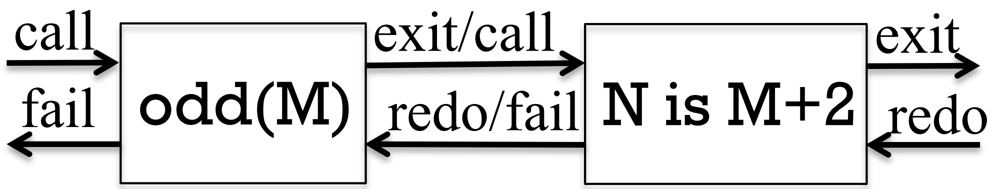
```
odd(1).
odd(N) :- odd(M), N is M + 2.
```



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



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



Generating alternatives, continued

For reference:

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

The key point with generative predicates:

<u>If an alternative is requested, another activation of the predicate is created</u>.

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

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

GOTO slide 129 (a8 Supplement)

Lists

List basics

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']]]]
```

What's the output of the following?

```
?- write([1, 2, odd(3), 4+5, atom(6)]).
[1,2,odd(3),4+5,atom(6)]
true.
```

What's the result of the following?

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

A list literal as a query is taken as a request to consult files!

Unification with lists

Here are some unifications with lists:

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

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

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

We'll later see a head-and-tail syntax for lists.

Unification with lists, continued

Problem: Write a predicate **`empty(X)`** that succeeds iff **`X`** is an empty list. If called with something other than a non-empty list, it fails.

Examples:

```
?- empty([]).
true.

?- empty([3,4,5]).
false.
```

Solution:

```
empty([]).
```

What will the following do?

```
?- empty(10).
false.
?- empty(L).
L = [].
```

Unification with lists, continued

Write a predicate `as123(X)` that succeeds iff `X` is a list with one, two, or three identical elements. Example:

```
?- as123([a]), as123([b,b]), write(ok), as123([1,2,3]),
     write('oops').
ok
false.

?- as123(L).
L = [_G2456] ;
L = [_G2456, _G2456] ;
L = [_G2456, _G2456, _G2456].
```

Solution:

```
as123([_]).
as123([X,X]).
as123([X,X,X]).
```

Built-in list-related predicates

SWI Prolog has a number of built-in predicates that operate on lists. One is **nth0**:

nth0(?Index, ?List, ?Elem)

True when Elem is the Index'th element of List. Counting starts at 0.

What's the question and answer for each of the following?

```
?- nth0(2, [a,b,a,d,c], X).
X = a.
```

What is the third element of [a,b,a,d,c]?

```
?- nth0(0, [a,b,a,d,c], b).
false.
```

Is b the first element of [a,b,a,d,c]?

```
?- nth0(N, [a,b,a,d,c], a).
N = 0 ;
N = 2 ;
false.
```

Where does a occur in [a,b,a,d,c]?

```
?- nth0(N, [a,b,a,d,c], X).
N = 0,
X = a ;
N = 1,
X = b ;
...
```

What are the positions and values for all?

NOTE: **nth0** makes for a good example here, but use indexing judiciously! There are often better alternatives!

Built-ins for lists, continued

Recall:

```
as123([_]).
as123([X,X]).
as123([X,X,X]).
```

Problem: Using `as123` and `nth0`, write a predicate with this behavior:

```
?- gen3(test, L).
L = [test] ;
L = [test, test] ;
L = [test, test, test].
```

Solution:

```
gen3(X,L) :- as123(L), nth0(0, L, X).
```

Does the order of the goals matter?

More:

```
?- gen3(test, [test]).
true .

?- gen3(test, [a,b]).
false.
```

Built-ins for lists, continued

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

Get the length of a list:

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

Anything else?

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 ...
```

Built-ins for lists, continued

What do you think **reverse(?List, ?Reversed)** does?

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] ;
```

Built-ins for lists, continued

What might **numlist(+Low, +High, -List)** do?

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

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

Problem: Write **rev_numlist(+High, +Low, -List)**

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

Solution:

```
rev_numlist(High,Low,List) :-
    numlist(Low,High,List0), reverse(List0,List).
```

Built-ins for lists, continued

sumlist(+List, -Sum) unifies **Sum** with the sum of the values in **List**.

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

Will the following work?

```
?- sumlist([1+2, 3*4, 5-6/7], Sum).
Sum = 19.142857142857142.

?- X = 5, sumlist([X+X, X-X, X*X, X/X],R).
X = 5,
R = 36.
```

Is it good that **sumlist** handles arithmetic structures, too?

a8 supplement
(Follows slide 117)

A cons list structure

In Haskell we can create a list like this:

```
    10 : 20 : 30 : []
```

Here's an analog with a nested Prolog structure, indented for clarity:

```
    cell(10,
        cell(20,
            cell(30, empty))))
```

- The first term of each (cons) cell structure is the head
- The second term is the tail.
- The atom **empty** represents an empty list.

A cons list structure, continued

At hand:

```
cell(10, cell(20, cell(30, empty))))     % like [10,20,30]
empty                                    % like []
```

Problem: Write **sum_cl(+CL, -Sum)** that instantiates **Sum** to the sum of the elements in **CL**.

```
?- sum_cl(empty, Sum).
Sum = 0.

?- sum_cl(cell(7,cell(5,empty)), Sum).
Sum = 12.
```

Solution:

```
sum_cl(empty, 0).
sum_cl(cell(Head,Tail), Sum) :-
    sum_cl(Tail,TailSum), Sum is Head + TailSum.
```

Let's go symbolic

help(op) shows that there's a colon (**:**) operator that's right associative (**xfy**) and with precedence 600.

Let's use that colon operator to build a structure and see what it looks like:

```
?- display(10:20:30:empty).
:(10,:(20,:(30,empty)))
true.
```

How does the structure above differ from the following?

```
cell(10,cell(20,cell(30,empty)))
```

The only difference is the functor, '**:**' vs. '**cell**'!

Going symbolic, continued

At hand:

```
?- display(10:20:30:empty).
:(10,:(20,:(30,empty)))
true.
```

Recall:

```
sum_cl(empty, 0).
sum_cl(cell(Head,Tail), Sum) :-
    sum_cl(Tail,TailSum), Sum is Head + TailSum.
```

Problem:

Convert **sum_cl** to work with structures like **10:20:30:empty**.

```
?- sum_cl(10:20:30:empty,N).
N = 60.

?- sum_cl(empty,N).
N = 0.
```

Going symbolic, continued

With '**cell**' as a functor:

```
sum_cl(empty, 0).
sum_cl(cell(Head,Tail), Sum) :-
    sum_cl(Tail,TailSum), Sum is Head + TailSum.
```

With '**:**' as a functor:

```
sum_cl(empty, 0).
sum_cl(Head:Tail, Sum) :-
    sum_cl(Tail,TailSum), Sum is Head + TailSum.
```

Usage (again):

```
% swipl conslist.pl
?- sum_cl(5:3:7:2:empty,X).
X = 17.
```

member_cl

Problem:

Write a predicate **member_cl(+Value, +ConsList)** that tests whether **Value** appears in **ConsList**.

```
?- member_cl(12, 3:1:7:12:empty).
true .

?- member_cl(xyz, just:a:test:of:this:empty).
false.
```

Solution:

```
member_cl(Value, Value : _).
    % Value is a member of the list if it is the head.
member_cl(Value, _ : Tail) :- member_cl(Value, Tail).
    % Value is a member of the list if it appears in the tail.
```

member_cl, continued

What will the following do?

```
?- member_cl(V, a:10:x(1):empty).
V = a ;
V = 10 ;
V = x(1) ;
false.
```

Problem: Fill in the blank to make the following work as shown.

```
?- A = a:b:empty, B = 10:20:empty,
   member_cl(VA, A), member_cl(VB, B), writeln(VA-VB),fail.
a-10
a-20
b-10
b-20
false.
```

"cut"

Motivation

Problem: Implement a `max/3` predicate:

```
?- max(10,20,M).
M = 20.

?- max(7,3,M).
M = 7 .
```

One solution: (`max1.pl`)

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

Question: Why is there a space here?

Motivation, continued

Here's another version of **max/3**. Does it work?

```
    max(X, Y, X) :- X >= Y.                                  % max2.pl
    max(_, Y, Y).    % If X isn't bigger, it must be Y!
```

Usage:

```
    ?- max(3,7,M).
    M = 7.

    ?- max(7,3,M).
    M = 7 ;
    M = 3.
```

"cut"

Our broken **max**:

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

We can make it work with a "cut", a predicate whose name is an exclamation mark:

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

The cut above says,

"If you get to me, don't try any further clauses of **max**.

Important: if we don't reach a cut, it has no effect.

Cut (!/0) is a *control predicate*, like fail/0. It affects the flow of control.

"cut", continued

Here's a predicate that attempts to classify a value:

```
what(X,atom) :- atom(X).                                % cut2.pl
what(X,integer) :- integer(X).
what(X,float) :- float(X).
what(_,wat).
```

How does it behave?

```
?- what([7],W).
W = wat.

?- what(34,W).
W = integer ;
W = wat.
```

How can we fix it?

"cut", continued

Solution:

```
what(X,atom) :- atom(X), !.
what(X,integer) :- integer(X), !.
what(X,float) :- float(X), !.
what(_,wat).
```

There's lots more to know about "cut", and lots of ways to make a mess with it, but our usage of "cut" will be simple:

Use a cut when you want to say "if this rule succeeds, don't come back and consider any further clauses of this predicate."

Other ways to think of it:

"This is my final answer."

"Burn the bridge!"

GOTO slide 118 (Lists)

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's summation formula and have some fun with lists!

<u>This is CLE 9, worth three points.</u>

- Put NetIDs of tablemates in **cle9.txt**
- Submit with **turnin 372-cle9 cle9.txt**

Built-ins for lists, continued

Speculate: What does **atom_chars(?Atom, ?Charlist)** do?

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

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

Problem: write **rev_atom/2**. Hint: Write it as a test, the latter case.

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

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

Solution:

```
rev_atom(A,RA) :-
    atom_chars(A,AL), reverse(AL,RL), atom_chars(RA,RL).
```

Built-ins for lists, continued

Problem: write `eqlen(+A1,+A2)`, to test whether two atoms are the same length.

```
?- eqlen(test,this).
true.

?- eqlen(test,it).
false.
```

Solution:

```
eqlen(A1,A2) :- atom_chars(A1,C1), length(C1,Len),
   atom_chars(A2,C2), length(C2,Len).
```

(Note unification with `Len` vs. `Len1 == Len2`.)

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, 3.4, x(a)], Sorted).
Sorted = [1, 3.4, 5, abc, xyz, x(a), [1, 2]].
```

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

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

What does the following do?

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

member, continued

Problem: Write **uchars(+A,-C)** that instantiates **C** to each of the unique characters in the atom **A** in ascending order.

```
?- uchars(peep, C).
C = e ;
C = p.
```

Solution:

```
uchars(A,C) :- atom_chars(A,L), sort(L,S), member(C,S).
```

Problem, using **uchars**, print the capital letters in '**a Test Here**'.

```
?- uchars('a Test Here',C), char_type(C,upper(_)), writeln(C), fail.
H
T
false.
```

Recall:

```
?- char_type('A', T).
...
T = upper(a) ;
...
```

member, continued

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]).
```

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].
```

Speculate: What else can we do with **append**?

append, continued

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 this:

"For what values of `A` and `B` is their concatenation `[1,2,3]`?

`append`, continued

Think of `append(L1,L2,L3)` as demanding a relationship between the three lists:

> `L3` must consist of the elements of `L1` followed by the elements of `L2`.

If `L1` and `L2` are instantiated, `L3` must be their concatenation.

If only `L3` is instantiated then `L1` and `L2` represent (in turn) all the possible ways to divide `L3`.

What are the other possibilities?

Important:

> We can do a lot of list processing by establishing constraints with `append` (and other predicates) and asking Prolog to find cases when those constraints are true.

`append` is the Swiss Army Knife of list processing in Prolog!

append, continued

Problem: Using **append**, write **starts_with(?List, ?Prefix)** that expresses the relationship that **List** starts with **Prefix**.

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

?- starts_with([1,2,3], L).
L = [] ;
L = [1] ;
L = [1, 2] ;
L = [1, 2, 3] ;
false.
```

Solution:

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

What will the following do?

```
?- starts_with(Start, [1,2,3]).
Start = [1, 2, 3|_G1182].
```

append, continued

Problem: Write **ends_with**.

```
?- ends_with([a,b,c],[d,e]).
false.

?- ends_with([a,b,c],[b,c]).
true ;
false.
```

Solution:

```
ends_with(List, Suffix) :- append(_, Suffix, List).
```

append, continued

Haskell meets Prolog:

```
?- 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] ;
...
```

Solution:

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

Practice with **append**: **sumsegs**

Write **sumsegs(+List, +N, -Sums)**, where **Sums** is a list with the sum of the first **N** elements of **List**, then the sum of the next **N**, and so forth.

```
?- sumsegs([1, 2, 3, 4, 5, 6],2,R).
R = [3, 7, 11] ;
false.

?- sumsegs([1,2,3,4,5,6],4,R).
R = [10] ;
false.

?- sumsegs([1,2,3,4,5,6],7,R).
R = [] ;
false.
```

How can we approach it?

sumsegs, continued

For reference:

```
?- sumsegs([1, 2, 3, 4, 5, 6], 2, R).
R = [3, 7, 11] ;
```

Solution:

```
% If fewer than N elements remain, produce an empty list.
sumsegs(List, N, []) :- length(List,Len), Len < N.

sumsegs(List, N, Sums) :-
    % Get the first N elements into Seg and compute their sum.
    length(Seg, N), append(Seg, Rest, List), sumlist(Seg, Sum),

    % Compute the sums for the rest of the list.
    sumsegs(Rest, N, RestOfSums),

    % Specify the result by forming a list whose first element is the
    % sum of the first segment followed by the sums for the rest of the list.
    append([Sum], RestOfSums, Sums).
```

Key technique!

```
?- sumsegs([3, 1, 5, 7, 4], 2, R).

  sumsegs(List, N, Sums) :-

    length(Seg, N), append(Seg, Rest, List), sumlist(Seg, Sum),

      sumsegs(Rest, N, RestOfSums),

        sumsegs(List, N, Sums) :-

          length(Seg, N), append(Seg, Rest, List), sumlist(Seg, Sum),

          sumsegs(Rest, N, RestOfSums),

              sumsegs(List, N, []) :- length(List,Len), Len < N.

          append([Sum], RestOfSums, Sums).

     append([Sum], RestOfSums, Sums).

R = [4, 12] .
```

A video for 156-158 is on the web.

Replacements: 159+

sumsegs, continued

Let's activate tracing on **sumsegs**:

```
$ swipl slides.pl
?- trace(sumsegs).
%          sumsegs/3: [call,redo,exit,fail]
true.

[debug]  ?- sumsegs([3,1,5,7,4],2,R).
 T Call: (8) sumsegs([3, 1, 5, 7, 4], 2, _2544)
 T Redo: (8) sumsegs([3, 1, 5, 7, 4], 2, _2544)
 T Call: (9) sumsegs([5, 7, 4], 2, _2890)
 T Redo: (9) sumsegs([5, 7, 4], 2, _2890)
 T Call: (10) sumsegs([4], 2, _2914)
 T Exit: (10) sumsegs([4], 2, [])
 T Exit: (9) sumsegs([5, 7, 4], 2, [12])
 T Exit: (8) sumsegs([3, 1, 5, 7, 4], 2, [4, 12])
R = [4, 12] .
```

Generation with **append**

Here's 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.
```

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

Here's **chunk** again. How does it work?

```
    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).
```

Consider the call **chunk([a,b,c,d,e,f], 2, Chunk)**:

The first clause produces the first **N** elements of **L**. (**Chunk = [a,b]**)

The second clause first uses **length** and **append** to form a list **Rest** that is **L** minus the first **N** elements (**Rest = [c,d,e,f]**).

The second clause then calls **chunk([c,d,e,f], 2, Chunk)**, creating another activation of **chunk**.

Its first clause will produce the first **N** elements of **[c,d,e,f]**.
Its second clause will end up calling **chunk([e,f], 2, Chunk)** creating a third activation of **chunk**.

Important: Note the similarity to **odd** on slide 115.

gensums (practice with generation)

Recall **sumsegs**:

```
?- sumsegs([1, 2, 3, 4, 5, 6],2,R).
R = [3, 7, 11] ;
false.
```

Problem: Instead of producing a list, generate the sums:

```
?- gensums([1,2,3,4,5,6,7], 2, R).
R = 3 ;
R = 7 ;
R = 11 ;
false.
```

Two solutions, one with **chunk** and one without:

```
gensums(List, N, Sum) :-
    chunk(List, N, Seg), sumlist(Seg, Sum).

gensums2(List, N, Sum) :-
    sumsegs(List, N, Sums), member(Sum, Sums).
```

"Getting it" with **append**

A problem from a past semester:

Write **splits(+List,-Split)**. It unifies **Split** with each "split" of **List** in turn.

Example:

```
?- splits([3,1,5,7], S).
S = [3]/[1, 5, 7] ;
S = [3, 1]/[5, 7] ;
S = [3, 1, 5]/[7] ;
false.
```

"The concept encountered in **splits.pl** is simple in hindsight, but represents something pivotal to even vaguely understanding Prolog. There was a moment several minutes ago when it finally struck me that **append** is *not* a function, but some ephemeral statement of fact with several combinations of conditions that satisfy it."

—Bailey Swartz, Spring '15

findall/3

The **findall** predicate

Here are some examples with a new predicate, **findall**:

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

?- findall(pos(N,X), nth0(N, [a,b,a,d,c], X), Posns).
Posns = [pos(0, a), pos(1, b), pos(2, a), pos(3, d), pos(4, c)].

?- findall(X, (between(1,100,X), X rem 13 =:= 0), Nums).
Nums = [13, 26, 39, 52, 65, 78, 91].
```

In your own words, what does **findall** do?

findall, continued

For reference:

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

SWI's documentation: (with a minor edit)

findall(+Template, :Goal, -List)

> Create a list of the instantiations Template gets successively on backtracking over Goal and unify the result with List. 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.

findall, continued

For reference:

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

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?

Important:

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

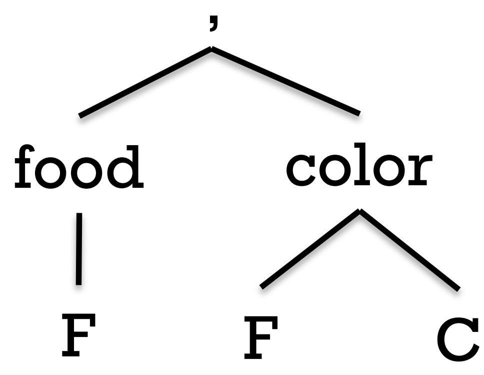
findall, continued

Here's a case where **: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.
```



The conjunction is a two-term structure whose functor is a comma.

findall, continued

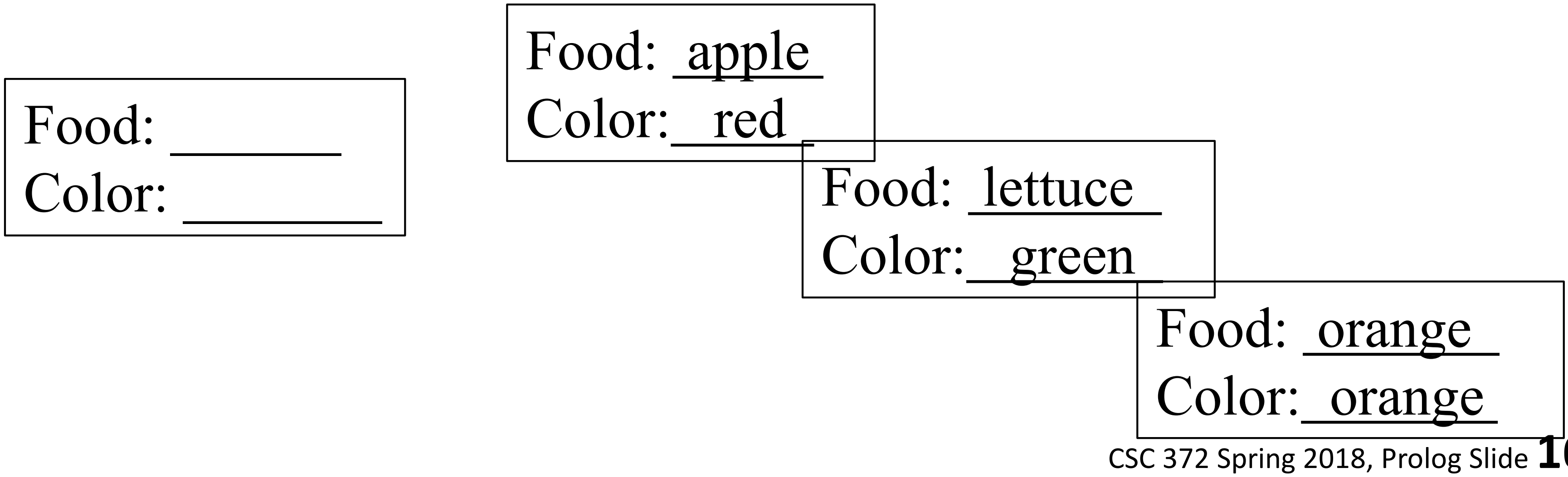
Original Thought from Noah Sleiman, Spring '14 :

"An easy way to think of it when using the uninstantiated first term (to find the elements of interest) is this:

findall(What I call it, How I got it, Where I put it)"

Another view:

- Think of the template (the first argument) as a paper form with some number of blanks to fill in.
- Each time the goal produces a result, we fill out a copy of that form and put it on the list.
- A list of the filled-out forms is the result of **findall**.



member vs. findall

member and findall are somewhat inverses of each other.

If we want to generate values from a list, we can use member:

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

If we have a query that generates values, we can make a list with findall:

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

Practice with **findall**

Problem: Write a predicate **sumlists** that produces a list of the sums of integer lists.

```
?- sumlists([[1,2], [10,20,30], []],Sums).
Sums = [3, 60, 0].
```

Recall sumlist:

```
?- sumlist([1,2,3],Sum).
Sum = 6.
```

Solution:

```
sumlists(Lists, Sums) :-
  findall(Sum, (member(List,Lists),sumlist(List,Sum)), Sums).
```

Note that **findall**'s goal is a conjunction of two goals.

Practice with **findall**, continued

Problem: Write a variant of **sumlists** that requires sums to meet a minimum:

```
?- minsums([[10,20,30],[1,2,3],[50]], 25, Sums).
Sums = [sum([10, 20, 30], 60), sum([50], 50)].

?- minsums([[10,20,30],[1,2,3],[50]], 250, Sums).
Sums = [].
```

Note that the result is a list of structures holding both the list and its sum.

Solution:

```
minsums(Lists, Min, Sums) :-
    findall(
       sum(List,Sum),
       (member(List,Lists),sumlist(List,Sum),Sum>=Min),
       Sums).
```

A scoping issue with **findall**

What's happening in the following query?

```
?- X=a, findall(X-Y, member(Y, [a,b,c]), Values), write(X-Y).
a-_G1095
X = a,
Values = [a-a, a-b, a-c].
```

The scope of variables created during a **findall** query is limited to that query.

Above, **X** is bound prior to the **findall** and can be used in it.

The **Y** inside the **findall** is <u>unrelated</u> to the **Y** in **write(X-Y)**.

Low-level list processing

Heads and tails

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:[]
```

General form: $[E_1, E_2, \ldots, E_n \mid Tail]$

Unifications with lists

What instantiations are produced by these unifications?

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

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

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

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

Simple list predicates

Here's a rule that describes the relationship between a list and 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.
```

Can we make better use of unification and define **head/2** more concisely?
```
head([H|_], H).
```
The head of a list whose head is **H** *is* **H***.*

Prolog vs. Haskell

Note the contrast between Haskell and Prolog:

Haskell:

`head` is a function that produces the first element of a list.

Prolog:

`head` is a predicate that describes the relationship between a value and the first element of a list.

In Prolog, **`head`** can:

- Produce the first element of a list.
- See if the first element of a list is a given value.
- Produce a list that will unify with any list whose head is a given value.

Implementing member

Recall the built-in member/2:

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

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

?- member(X, [2,1,4,5]).
X = 2 ;
X = 1 ;
X = 4 ;
X = 5.
```

member, continued

Problem: Implement the built-in **member(?Elem, ?List)** predicate with two clauses, a fact and a rule. Think of them this way:

X is a member of the list having X as its head.

```
member(X,[X|_]).
```

X is a member of the list having T as its tail iff X is a member of T.

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

Exercise: Following the example of slide 116 or 158, trace through how **member** generates elements from a list, like this:

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

Implementing **last**

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.
```

Solution:

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

What does the following produce?

```
?- last(L,last), head(L,first), length(L,3).
L = [first, _G1736, last] ;
```
...crickets...

allsame

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

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

?- 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]).
```

What's a simple way to test **allsame**?

```
?- allsame(L).
L = [_G1635] ;
L = [_G1635, _G1635] ;
...
```

adjacent

Write a predicate **adjacent(?A, ?B, ?L)** that expresses the relationship that **A** and **B** are adjacent in the list **L**. (But, in the order **A, B**.)

```
?- adjacent(3, 4, [1,2,3,4,5]).
true ;
false.

?- adjacent(a, X, [a,b,a,a,c,a]).
X = b ;
X = a ;
X = c ;
false.

?- adjacent(A,B,[1,2,3,4]).
A = 1, B = 2 ;
A = 2, B = 3 ;
A = 3, B = 4 ;
false.
```

```
?- adjacent(A,B,L).
L = [A, B|_G28] ;
L = [_G30, A, B|_G28] ;
L = [_G30, _G36, A, B|_G28] ;
```

Solution: (hint—use **append**!)

```
    adjacent(A,B,L) :- append(_, [A,B|_], L).
```

sf_gen

Write a predicate **sf_gen** that generates elements from a list in this order:
Second, first, fourth, third, sixth, fifth, ...

Usage:

```
?- sf_gen([a,b,c,d,e],X).
X = b ;
X = a ;
X = d ;
X = c ;
false.   % doesn't produce e because it would break the pattern.
```

sf_gen, continued

```
?- sf_gen([a,b,c,d,e],X).
X = b ;
X = a ;
X = d ;
X = c ;
```

Solution:

```
    % Produce the second element.
    sf_gen([_,X|_], X).

    % Produce the first element, if at least two.
    sf_gen([X,_|_], X).

    % Get rid of the first two elements and start all over.
    sf_gen([_,_|T], X) :- sf_gen(T,X).
        sf_gen([a, b|[c,d,e]], X) :- sf_gen([c,d,e], X).
```

A reified example of the third clause

Implementing **numlist**

Problem: Implement a slight variant of the built-in **numlist** predicate.

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

?- numlist(5,1,L).  % the built-in numlist fails for this case
L = [].
```

Solution, v1:

```
numlist(Low, High, []) :- Low > High, !.

numlist(Low, High, Result) :-
    Next is Low + 1,
    numlist(Next, High, Rest),
    Result = [Low|Rest].
```

```
numlist(1,4, Result) :-
    Next is 1 + 1,
    numlist(2, 4, Rest),
        Rest gets bound to [2,3,4]
    Result = [1|[2,3,4]].
```

What happens if we remove the cut?

numlist, continued

Solution, v1:

```
numlist(Low, High, []) :- Low > High, !.

numlist(Low, High, Result) :-
    Next is Low + 1,
    numlist(Next, High, Rest),
    Result = [Low|Rest].
```

How can we make better use of unification?

```
numlist(Low, High, []) :- Low > High, !.

numlist(Low, High, [Low|Rest]) :-
    Next is Low + 1,
    numlist(Next, High, Rest).
```

Implementing **length**

Problem: Write a predicate that behaves like the built-in **length/2**.

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

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

?- length(L,1).
L = [_G901] .

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

Solution:

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

Exercise: Work through how the fourth example generates.

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?

Note the similarity to **++** in Haskell:

```
(++) [] rhs = rhs
(++) (x:xs) rhs = x : (xs ++ rhs)
```

But, Haskell's **++** only lets us concatenate lists. Prolog's **append** expresses a relationship between three lists.

Implementing **append**, continued

At hand:

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

?- trace, append([1,2,3],[a,b,c],X).
   Call: (8) append([1, 2, 3], [a, b, c], _G971) ? creep
   Call: (9) append([2, 3], [a, b, c], _G1097) ? creep
   Call: (10) append([3], [a, b, c], _G1100) ? creep
   Call: (11) append([], [a, b, c], _G1103) ? creep
   Exit: (11) append([], [a, b, c], [a, b, c]) ? creep
   Exit: (10) append([3], [a, b, c], [3, a, b, c]) ? creep
   Exit: (9) append([2, 3], [a, b, c], [2, 3, a, b, c]) ? creep
   Exit: (8) append([1, 2, 3], [a, b, c], [1, 2, 3, a, b, c]) ? creep
X = [1, 2, 3, a, b, c].
```

Note that all of the **Exit:** lines in the trace above show an **append** relationship that's true.

delete

Problem: Implement the built-in predicate **delete**.

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

Solution:

```
delete([], _, []).
delete([X|T], X, R) :- delete(T, X, R), !.
delete([E|T], X, [E|R]) :- delete(T, X, R).
```

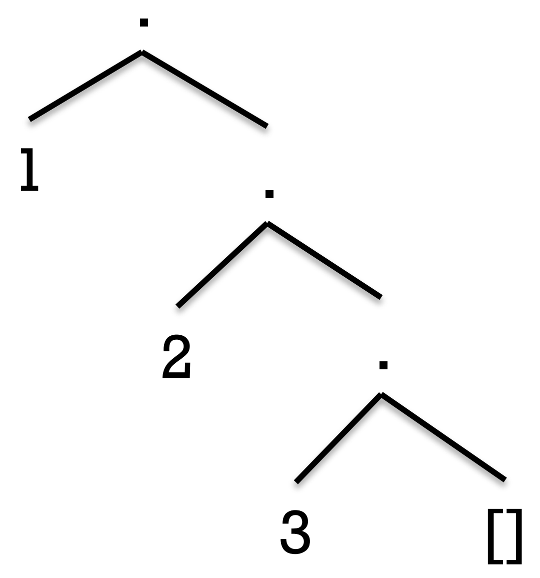
How could we write it without a cut?

```
delete([], _, []).
delete([X|T], X, R) :- delete(T, X, R).
delete([E|T], X, [E|R]) :- E \== X, delete(T, X, R).
```

Lists are structures

Traditionally, Prolog lists are structures. Here's GNU Prolog:

```
| ?- 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,.(.(3,[]),[]))).
X = [a,b,[3]]
```

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].
```

A Lisp programmer would call this a "dotted-pair".

In the SWI Prolog docs, *5.1 Lists are special*, talks about SWI's handling of lists.

http://www.swi-prolog.org/pldoc/man?section=ext-lists

"univ"

`=../2`, read as "univ", expresses a relationship between structures and lists:

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

?- f(a,g(c,d),e(f))  =..  L.
L = [f, a, g(c, d), e(f)].

?- 1*2+3/4  =..  L.
L = [+, 1*2, 3/4].

?- S  =..  [writeln,hello], call(S).   % call is a higher-order predicate
hello
S = writeln(hello).
```

"univ", continued

Problem: Create a predicate **functor** that produces the functors in a binary tree.

```
?- functor(1 * 2 + 3 / 4, F).
F = (+) ;
F = (*) ;
F = (/) ;
false.
```

Solution:

```
functor(S,F) :- S =.. [F|T], T \== [].
functor(S,F) :- S =.. [_,Left,_], functor(Left,F).
functor(S,F) :- S =.. [_,_,Right], functor(Right,F).
```

Which is the better name for this predicate, **functor** or **functor<u>s</u>**?

Sidebar: "univ"

C&M 5e p.130 says,

The predicate "=.." (pronounced "univ" for historical reasons)...

I asked about "univ" on the prolog channel on irc.freenode.net:

```
x77686d: C&M says that =.. is called "univ" for historical reasons.
         Anybody know the story behind that?
dmiles: for a long time we could only used named operators
dmiles: why it was called univ instead of t2l .. i dont know
dmiles: oh unify vector
dmiles: erl v in prolog means array/vector
```

The first edition of C&M (1981) has that same line...

"Can't prove"

"can't prove"

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

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

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

We'll read \+ as "can't prove".

Of course, Prolog only knows what we tell it...

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

"can't prove", continued

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?

Yes!

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

Describe the behavior of **inedible/1**:

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

inedible(X) succeeds if something is <u>not known</u> to be a food.

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

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

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

"can't prove", continued

What's the following query asking?

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

What are things with known colors that aren't food?

Let's try reversing the goals:

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

Why do the results differ?

Variables are never instantiated by a "can't prove" goal.

Try `?- \+color(Thing, purple).`

"can't prove" with "cut-fail"

Here's an implementation of `\+` using the higher-order predicate **call/1** and a "cut-fail":

```
cant_prove(G) :- call(G), !, fail.   % 'My final answer is "no".'
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?

Yes, it uses an argument as a goal.

Pit-crossing Puzzle

The problem

Consider the problem of crossing over a series of pits using wooden planks as bridges.

Here's a case with two pits:

```
-----+        +---+   +------
     |        |   |   |
     |        |   |   |
     +--------+   +---+
     5       12   15 18
```

Pits are represented with **pit/2** facts, with a starting position and a width:

```
pit(5,7).          % Think of the pit as [5,12).
pit(15,3).
```

There may be any number of **pit** facts. Pits never overlap. Pits always have some ground between them.

The problem, continued

Here's a crossing of distance 20 with the sequence of planks [3, 10, 10]:

```
===          ==========

   ==========

----+      +--+  +-------
    |      |  |  | ^
    |      |  |  | 20
    +------+  +--+
    5     12  15 18
```

Planks are drawn with vertical offsets to show their widths.

- Planks must be placed so that both ends rest on solid ground, rather than having an end over a pit.
- Planks must extend continuously from zero (the starting point) to, or through, a specified length.

The problem, continued

The sequence **[9, 11]** is an <u>invalid</u> crossing:

```
=========
         ===========
----+      +--+  +----
    |      |  |  | ^
    |      |  |  | 20
    +------+  +--+
    5     12  15 18
```

It's invalid because the two planks meet over a pit, at distance 9.

- A joint at distance **D** is considered to be over a pit if
  start-of-pit <= D < end-of-pit
- Over-pit distances for the above pits are 5-11 and 15-17.
- Valid joint starting positions include 4, 12, 13, 14, 18, 19, and more.

The problem, continued

For reference, with two pits: **pit(5,7)** and **pit(15,3)**:

```
-----+       +---+   +------
     |       |   |   |
     |       |   |   |
     +-------+   +---+
     5      12  15  18
```

Problem: Write **cross(+Distance, +Planks, -Solution)**.

```
?- cross(20, [10,10,3], S).
S = [3, 10, 10] .

?- cross(20, [9,11], S).
false.

?- cross(20, [1,2,4,5,5,9], S).
S = [4, 9, 1, 5, 2] .
```

Distance **D** is over a pit if
pit-start <= D < pit-end

layplanks

At hand:

```
?- cross(20, [1,2,4,5,5,9], S).
S = [4, 9, 1, 5, 2] .
```

Let's start with a helper predicate:

layplanks(+Goal, +Supply, +Current, -Solution)

- It succeeds if we can reach from the **Current** distance to the **Goal** with the given **Supply** of planks.
- **Solution** is instantiated to a suitable sequence of planks.

layplanks will be recursive. What's the base case? (English first!)

If we're at or past the goal distance, it takes no planks to reach the goal distance.

```
layplanks(Goal, _, Current, []) :- Current >= Goal.
```

```
?- layplanks(10, [3,1,5], 12, S).
S = [] .
```

layplanks, continued

What should happen with a **layplanks** call like the following?

```
?- layplanks(20, [8,10,3], 0, S).
```

Pick a plank and see if we can add it to the solution.

- If so, then solve from the new distance with the remaining planks
- If not, pick a different plank.
- If no plank works, fail.

```
----+      +--+  +----
    |      |  |  |
    +------+  +--+
    5     12  15 18
```

What if we pick **8**?

We're over a pit!

What if we pick **3**?

We're not over a pit, so we lay down the plank and see if we can go from **3** to **20** with the remaining planks.

```
?- layplanks(20, [8,10], 3, S).
```

[this line removed!]

layplanks, continued

For reference:

Pick a plank and see if we can add it to the solution.

- If so, then solve from the new state with the remaining planks
- If not, pick a different plank.
- If no plank works, fail.

```
-----+       +--+  +-----
     |       |  |  |
     +-------+  +--+
     5      12  15 18
```

Current state:

?- layplanks(20, [8,10], 3, S).

What if we pick 8?

We're over a pit! (3+8 == 11)

What if we pick 10?

?- layplanks(20, [8], 13, S).

Picks 8 and does

?- layplanks(20, [], 21, S).

S = [] .

The full sequence via a spy point

We can see the sequence of calls and returns with a *spy point*:

```
?- spy(layplanks).
% Spy point on layplanks/4
true.

[debug]  ?- layplanks(20, [3,8,10], 0, S).
 * Call: (8) layplanks(20, [3, 8, 10], 0, _2534) ? leap
 * Call: (9) layplanks(20, [8, 10], 3, _2764) ? leap
 * Call: (10) layplanks(20, [8], 13, _2776) ? leap
 * Call: (11) layplanks(20, [], 21, _2794) ? leap
 * Exit: (11) layplanks(20, [], 21, []) ? leap
 * Exit: (10) layplanks(20, [8], 13, [8]) ? leap
 * Exit: (9) layplanks(20, [8, 10], 3, [10, 8]) ? leap
 * Exit: (8) layplanks(20, [3, 8, 10], 0, [3, 10, 8]) ? leap
S = [3, 10, 8] .
```

Note that once the recursion hits the base case, the solution is built tail-first as the recursive calls to **layplanks** return.

Writing **layplanks**

layplanks needs to pick a plank and know which planks are left.

We'll use the built-in **select** for that:

```
select(?Elem, ?List1, ?List2)
   Is true when List1, with Elem removed, results in List2.
```

Example:

```
?- select(Plank, [10,8,3], Remaining).
Plank = 10,
Remaining = [8, 3] ;
Plank = 8,
Remaining = [10, 3] ;
Plank = 3,
Remaining = [10, 8] ;
false.
```

An implementation of **select**:

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

Example: layplanks(20, [10,8,3], 0, S).

Recall our base case:

```
    layplanks(Goal,_,Current,[]) :- Current >= Goal.
```

Now we're ready to write the recursive case:

```
    layplanks(Distance, Planks, Current, [Plank|MorePlanks]) :-
        % Pick a plank.
        select(Plank, Planks, Remaining),

        % See how far it extends.
        NewEnd is Current + Plank,

        % Be sure we're not over a pit.
        \+over_pit(NewEnd),  % todo!

        % Solve it from here with the remaining planks.
        layplanks(Distance, Remaining, NewEnd, MorePlanks).
```

Loose ends

Problem: Write **over_pit**.

```
over_pit(N) :-
   pit(Start,Width),
   End is Start + Width,
   N >= Start, N < End.
```

We still need to use **layplanks(+Goal, +Supply, +Current, -Solution)** to write **cross**:

```
?- cross(20, [1,2,4,5,5,9], S).
S = [4, 9, 1, 5, 2] .
```

```
====         =     ==
    ========= =====
----+      +--+  +----
    |      |  |  |
    +------+  +--+
    5     12  15 18
```

```
cross(Goal, Planks, Solution) :-
     layplanks(Goal, Planks, 0, Solution).
```

Experiment with this! It's in **cross.pl**.

Backtracking makes this work!

Key point:

A failure when attempting to place the very last plank may cause backtracking across predicate calls all the way back through the choice of the first plank!

Here's the general pattern for problems involving finding a valid sequence of parts, steps, movements, etc.:

- Pick one of the things to add to the solution
- If it can be added, compute the new state.
  - If it can't be added, backtrack and pick a different thing, or fail.
- Solve it from the new state with the remaining things

Note that **cross** isn't very smart. It didn't even check to see if we had enough planks to go the full distance, irrespective of the pits.

Prolog's automatic uninstantiation of variables when a goal's redo port is entered makes backtracking easy!

Pit-crossing variations

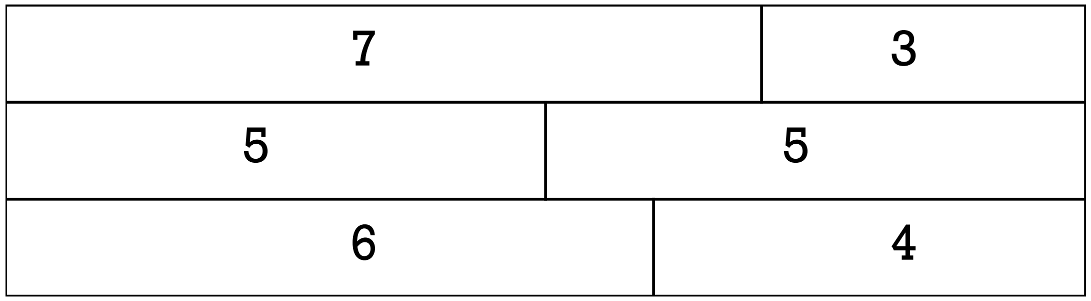
Variations:

- What if some pits had fire but we had some steel "planks", too?
- What if we could cut planks?
- What if we could cut planks but had only a limited amount of gas for our chainsaw?
- What if the space to cross were two-dimensional?
- What are some more variations?

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:



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?

Helper layrow

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

Three rows of bricks

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 = [].
```

What's the purpose of ***LeftAfter3 = []***?

Usage:

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

How can we generalize it to N rows?

N rows of bricks

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).
```

N rows of bricks, continued

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?

Modify the base case:

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

?- laybricks([4,3,2,1], 2, 3, Rows).
Rows = [[3], [2, 1]] .
```

We'll call this variant **laybricks2**.

Testing

Some facts for testing:

```
laybricks(+Bricks, +NRows, +RowLen, ?Rows)
```

```
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]] .
```

Testing, continued

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

Zebra Puzzle, continued

We can solve this problem by representing all the information with 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 three facts can be represented with this goal:

```
Houses = [house(norwegian, _, _, _, _),     % First house
          _,                                % Second house
          house(_, _, _, milk, _),          % Middle house
          _, _]                             % 4th and 5th houses
```

Instances of **house** structures represent knowledge about a house.

house structures have five terms: nationality, pet, smoking preference (remember, it was 1962!), beverage of choice and house color.

Anonymous variables are used to represent "don't-knows".

Zebra Puzzle, continued

Remember: **house(Nation, Pet, Smoke, Drink, Color)**

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.*?

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 ;
...
```

Problem: Write a goal to express the green-to-right-of-ivory fact.

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

Zebra Puzzle, continued

We have these "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?

How can we use **left_right(L, R, List)** to write **next_to(X, Y, List)**?

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

Zebra Puzzle, continued

These "next to" facts are 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.

Remember: **house(Nation, Pet, Smoke, Drink, Color)**

How can we express the facts above as goals?

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

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

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

Zebra Puzzle, continued

Remember: **house(Nation, Pet, Smoke, Drink, Color)**

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)
```

Zebra Puzzle, continued

A rule that comprises all the goals:

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

  % The questions of interest:
  member(house(Zebra_Owner, zebra, _, _, _), Houses),
  member(house(Water_Drinker, _, _, water, _), Houses).
```

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.

?- statistics.
% Started at Wed Apr 25 00:53:50 2018
% 0.100 seconds cpu time for 467,242 inferences
```

...more.. (try it!)

Credits

Credits:

The code presented was adapted from code by Ng Pheng Siong in `sandbox.rulemaker.net/ngps/119`

Siong apparently adapted it from prior work by Bill Clementson in Allegro Prolog.

Typing in Prolog

Static or dynamic?

Recall that with a statically typed language, the type of every variable and expression can be determined by static analysis of code.

Is Prolog statically typed or dynamically typed? Or is it something else?

Wikipedia says, "Prolog is an untyped language." (4/25/2018)
    Does Prolog not have types?

BCPL is sometimes described as an untyped language where all values are word-sized objects.

Imagine a language where everything is a string. Is it untyped?

"A programming language is untyped if it allows [you] to apply any operation on any data, and all datatypes are considered as sequences of bits of various lengths."—**http://progopedia.com/typing/untyped**

The books say...

There are only two clear references to data types in C&M:

p. 28: "The functor names the general kind of structure, and corresponds to a **datatype** in an ordinary programming language."

p. 122, under "Classifying Terms": If we wish to define predicates which will be used with a wide variety of argument **types**, it is useful to be able to distinguish in the definition what should be done for each possible **type**."

Covington has several references to types, including these:

p. 93: "Terms of this form are called STRUCTURES. The functor is always an atom, but the arguments can be terms of any **type** whatever."

p. 130: "If **`number_codes`** is given a string that doesn't make a valid number, or if either of its arguments is of the wrong **type**, it raises a runtime error condition."

Another voice:

ISO Prolog's exception handling mechanism has a **`type_error(Type,Term)`** structure.

swipl says...

Let's see if any predicates concern types.

```
?- apropos(type).
...
integer/1        Type check for integer
rational/1       Type check for a rational number
number/1         Type check for integer or float
atom/1           Type check for an atom
blob/2           Type check for a blob
string/1         Type check for string
```

Can we produce a type error?

```
?- atom_length(a(1), Len).
ERROR: atom_length/2: Type error: `text' expected, found `a(1)'
```

Could we find the above error with static analysis?

Bottom line: I'm comfortable saying that Prolog has types.

Back to "statically typed or dynamically typed?"

Again:

In a statically typed language, the type of every variable and expression can be determined by static analysis of code.

Can we construct a Prolog program where a value's type cannot be determined by looking at the code?

Here's such a program:

```
  f('one').  f(a(1)).

  prog :- f(X), random(2) > 0,
      atom_length(X, Len), writeln(Len).
```

```
?- prog.
3
true .

?- prog.
false.

?- prog.
ERROR: atom_length:
Type error: ...
```

The type of X depends on a random number and thus varies from run to run.

Therefore, Prolog is dynamically typed! Right?

Odds and ends

Collberg's *Architecture Discovery Tool*

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

The basic idea:

Use the C compiler of the target system to compile a large number of small but carefully crafted programs and then examine the machine code produced for each program to make inferences about the architecture.

End result:

A machine description that in turn can be used to generate a code generator for the architecture.

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

Paper:

cs.arizona.edu/~collberg/content/research/papers/collberg02automatic.pdf

The Prolog 1000

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

AALPS

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

ACACIA

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

ASIGNA

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

The Prolog 1000, continued

Coronary Network Reconstruction

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

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

DAMOCLES

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

The Prolog 1000, continued

DUST-EXPERT

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

EUREX

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

GUNGA CLERK

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

MISTRAL

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

The full list is in `prolog/Prolog1000.txt`. Several are over 100K lines of code.

Lots of Prologs

For a Fall 2006 honors section assignment, Maxim Shokhirev was given the task of finding as many Prolog implementations as possible in one hour. His results:

1. DOS-PROLOG
http://www.lpa.co.uk/dos.htm
2. Open Prolog
http://www.cs.tcd.ie/open-prolog/
3. Ciao Prolog
http://www.clip.dia.fi.upm.es/Software/Ciao
4. GNU Prolog
http://pauillac.inria.fr/~diaz/gnu-prolog/
5. Visual Prolog (PDC Prolog and Turbo Prolog)
http://www.visual-prolog.com/
6. SWI-Prolog
http://www.swi-prolog.org/
7. tuProlog
http://tuprolog.alice.unibo.it/
8. HiLog
ftp://ftp.cs.sunysb.edu/pub/TechReports/kifer/hilog.pdf
9. ?Prolog
http://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/
10. F-logic
http://www.cs.umbc.edu/771/papers/flogic.pdf
11. OW Prolog
http://www.geocities.com/owprologow/
12. FLORA-2
http://flora.sourceforge.net/
13. Logtalk
http://www.logtalk.org/
14. WIN Prolog
http://www.lpa.co.uk/
15. YAP Prolog
http://www.ncc.up.pt/~vsc/Yap
16. AI::Prolog
http://search.cpan.org/~ovid/AI-Prolog-0.734/lib/AI/Prolog.pm
17. SICStus Prolog
http://www.sics.se/sicstus/
18. ECLiPSe Prolog
http://eclipse.crosscoreop.com/
19. Amzi! Prolog
http://www.amzi.com/
20. B-Prolog
http://www.probp.com/
21. MINERVA
http://www.ifcomputer.co.jp/MINERVA/
22. Trinc Prolog
http://www.trinc-prolog.com/

And 50 more!

Ruby meets Prolog

http://www.artima.com/forums/flat.jsp?forum=123&thread=182574 describes a "tiny Prolog in Ruby".

Here is **member**:

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

Here's the common family example:

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

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

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

In conclusion...

If we had a whole semester...

- Knowledgebase manipulation (slides 249-268 in **prolog.pdf**)
- Parsing with definite clause grammars (slides 269-285 in **prolog.pdf**)
- More with...
  - Puzzle solving
  - Higher-order predicates
- Expert systems
- Natural language processing
- Constraint programming

  http://www.swi-prolog.org/pldoc/man?section=clpfd
- Look at Prolog implementation with the Warren Abstract Machine.

Continued study:

More in Covington and Clocksin & Mellish.

The Craft of Prolog by O'Keefe

The Art of Prolog by Sterling and Shapiro

Database (knowledgebase) manipulation

assert and **retract**

A Prolog program is a database of facts and rules.

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

A predicate to establish that certain things are foods:

```
makefoods :-                                    % foods3.pl
   assert(food(apple)),
   assert(food(broccoli)),  assert(food(carrot)),
   assert(food(lettuce)),  assert(food(rice)).
```

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

```
?- food(F).      ("positive-control" test—be sure no foods already!)
ERROR: toplevel: Undefined procedure: food/1

?- makefoods.
true.

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

assert and **retract**, continued

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|...].
```

Let's start **makefoods** with a **retractall** to get a clean slate every time.

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

assert and **retract**, continued

Important: 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).
X = 2.
```

A redo of **retract(f(1))** *did not restore* **f(1)**.

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 calculator

Here's a very simple calculator: (calc.pl)

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

Note that the commands themselves are Prolog terms.

Simple calculator, continued

A loop that reads and prints terms:

```
calc0 :- prompt(_, '> '),
        repeat, read(T), format('Read ~w~n', T), T = exit, !.
```

Interaction:

```
?- calc0.
> 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 calculator, 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).

calc :-
    init, prompt(_, '> '),
    repeat, read(T), do(T), T = exit, !.
```

```
?- calc.
> 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!)

Simple calculator, continued

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)).
```

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

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

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

Exercise: Add **double** and **halve** commands.

Word tally

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

Let's use **word(Word, Count)** facts to maintain counts.

Let's write a **count(Word)** predicate to create and update **word/2** facts.

Example of operation:

```
?- 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].
```

listing displays the clauses for a predicate:

```
?- listing(word).
:- dynamic word/2.

word(this, 1).
word(test, 2).
word(now, 1).

true.
```

count implementation

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].
```

Problem: Implement the predicate **count**.

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

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

Top-level and a helper

tally clears the counts and then loops, reading lines and processing each.

```
tally :-
   retractall(word(_,_)),
   repeat,
      readline(Line),
      do_line(Line),
      Line == '', !,           % note that '' is an empty atom
      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) :-
      atomic_list_concat(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**, to produce the output at right.

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

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

Full source is in **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 = {}**

How could we maintain multiple tallies simultaneously?

We could add an id of some sort as another term for **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)
```

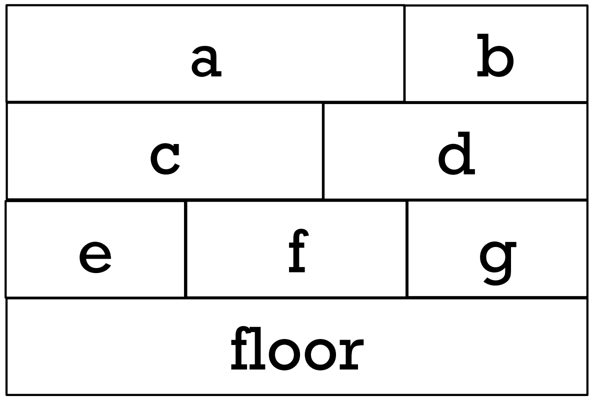
and create facts like these:

```
word(quotes, testing, 3)
word(body, testing, 10)
```

Analogy: Imagine a Ruby constant **HASH** that is **the** instance of **Hash**.

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.

What's a suitable sequence of removals for the above diagram?

`a, c, e, b, d, f, g`

Another: `a, b, c, d, e, f, g`.

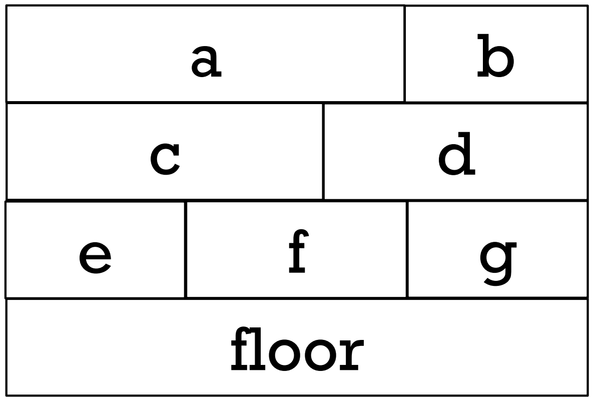
Unstacking blocks, continued

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 in be in Java or Ruby?

Can we tighten it up?



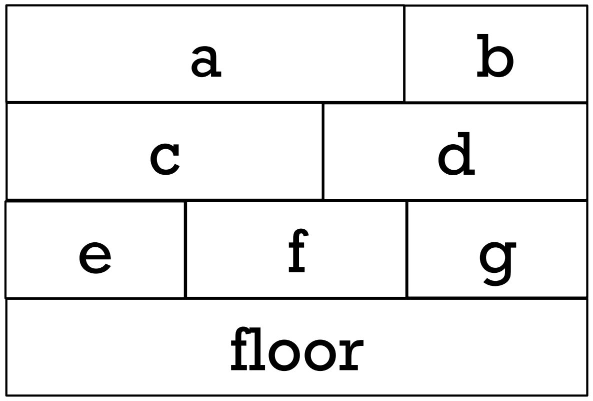
```
on(a,c). on(a,d). on(b,d). ...
```

```
?- 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.

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. It has four *productions*.

```
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 appear in the strings of the language. Generation terminates with them.

Sentence, **Article**, **Noun** and **Verb** are *non-terminal symbols*—they can produce something more.

Sentence is the *start symbol*. We can generate sentences by starting with it and replacing non-terminals with terminals and non-terminals until only terminals remain.

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]).
```

```
Sentence => Article Noun Verb
Article  => the | a
Noun     => dog | cat | girl | boy
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] ;
...
```

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. <u>The key idea: **article**, **noun**, and **verb** each consume an expected word and produce the remaining words.</u>

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].
```

This is Prolog source code, too!

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

At hand: (a *definite clause grammar*)

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

?- 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]).

?- sentence([a,dog,talked,to,me], Leftover).
Leftover = [to, me] .

?- sentence([a,bird,talked,to,me], Leftover).
false.
```

Remember that **sentence**, **article**, **verb**, and **noun** are non-terminals. **dog**, **cat**, **ran**, **talked**, are terminals, represented as atoms in singleton lists.

Grammar rule notation, continued

Below we've added a second term to the call to **sentence**, and mixed in a regular rule for **verb** along with the grammar rule.

```
 s(String) :-                                                    % parser1a.pl
      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).

?- s('a boy computed').
true .

?- s('a boy computed pi').
false.
```

Goals in grammar rules

We can insert ordinary goals into grammar rules by enclosing the goal(s) in curly braces.

Here is a chatty parser that recognizes the language described by the regular expression **a***:

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

string --> as.

as --> [a], {writeln('got an a')}, as.
as --> [], {writeln('empty match')}.
```

Usage:

```
?- parse(aaa).
got an a
got an a
got an a
empty match
true .
```

```
?- parse(aab).
got an a
got an a
empty match
empty match
empty match
false.
```

What if the **as** clauses are swapped?

```
?- parse(aaa).
empty match
got an a
empty match
got an a
empty match
got an a
empty match
true.
```

Parameters in non-terminals

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

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

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

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

Usage:

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

?- parse(aaab, N).
false.
```

Parameters in non-terminals, continued

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

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

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

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

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

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

?- parse(aabbbbcccccc, L).
L = [2, 4, 6] .

?- parse(aabbc, L).
false.
```

Can this language be described with a regular expression?

Parameters in non-terminals, continued

How could we handle ***a***^{X}***b***^{Y}***c***^{Z} where X <= Y <= Z?

```
    ?- parse(abbbccc, L).
    L = [1, 3, 3] .

    ?- parse(ccccc, L).
    L = [0, 0, 5] .

    ?- parse(aaabbc, L).
    false.

parse(S,L) :- atom_chars(S,Chars), string(L, Chars, []). % parser7b.pl

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

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

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

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

Accumulating an integer

Problem: Write a parser that recognizes a string of digits and creates an integer from them:

```
?- parse('4341', N).
N = 4341 .

?- parse('1x3', N).
false.
```

Solution:

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

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

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

digit('0'). digit('1'). digit('2'). ...
```

How do the **digits(...)** rules work?

A list recognizer

Consider a parser that recognizes lists consisting of positive integers and lists:

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

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

?- parse('[ 1, 2 , 3 ]').  % Whitespace!  How could we handle it?
false.
```

Implementation: (**list.pl**)

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

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

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

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

"Real" compilation

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

Input text like this,

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

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

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

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

We could use a pair of Prolog grammars to parse source code:

- The first one would parse character-by-character and generate a token stream like the list above. (A *scanner*.)
- The second grammar would parse that token stream.