# Prolog

## Fundamentals of Artificial Intelligence

Slides are taken from

William Mitchell (University of Arizona)

Ulle Endriss (Univ. Amsterdam)

Stuart C. Shapiro (SUNY, Buffalo)

# Basic Prolog

By

Ulle Endriss(University of Amsterdam)

# What is Prolog?

- Prolog (*pro*gramming in *log*ic) is a logic-based programming language: programs correspond to sets of logical formulas and the Prolog interpreter uses logical methods to resolve queries.
- Prolog is a *declarative* language: you specify *what* problem you want to solve rather than *how* to solve it.
- Prolog is very useful in *some* problem areas, such as artificial intelligence, natural language processing, databases, . . . , but pretty useless in others, such as for instance graphics or numerical algorithms.
- The objective of this first lecture is to introduce you to the most basic concepts of the Prolog programming language.

# Facts

A little Prolog program consisting of four *facts*:

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

# Queries

After compilation we can *query* the Prolog system:

```
?- bigger(donkey, dog).
Yes

?- bigger(monkey, elephant).
No
```

# A Problem

The following query does not succeed!

```
?- bigger(elephant, monkey).
No
```

The *predicate* `bigger/2` apparently is not quite what we want.

What we'd really like is the transitive closure of `bigger/2`. In other words: a predicate that succeeds whenever it is possible to go from the first animal to the second by iterating the previously defined facts.

# Rules

The following two *rules* define `is_bigger/2` as the transitive closure of `bigger/2` (via recursion):

```
is_bigger(X, Y) :- bigger(X, Y).

is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

↑ "if"

↑ "and"

# Now it works

```
?- is_bigger(elephant, monkey).
Yes
```

Even better, we can use the *variable* `X`:

```
?- is_bigger(X, donkey).
X = horse ;
X = elephant ;
No
```

Press `;` (semicolon) to find alternative solutions. `No` at the end indicates that there are no further solutions.

## Another Example

Are there any animals which are both smaller than a donkey and bigger than a monkey?

```
?- is_bigger(donkey, X), is_bigger(X, monkey).
No
```

# Terms

Prolog *terms* are either *numbers*, *atoms*, *variables*, or *compound terms*.

*Atoms* start with a lowercase letter or are enclosed in single quotes:

```
elephant, xYZ, a_123, 'Another pint please'
```

*Variables* start with a capital letter or the underscore:

```
X, Elephant, _G177, MyVariable, _
```

# Terms (cont.)

*Compound terms* have a *functor* (an atom) and a number of *arguments* (terms):

```
is_bigger(horse, X)
f(g(Alpha, _), 7)
'My Functor'(dog)
```

Atoms and numbers are called *atomic terms*.

Atoms and compound terms are called *predicates*.

Terms without variables are called *ground terms*.

# Facts and Rules

*Facts* are predicates followed by a dot. Facts are used to define something as being unconditionally true.

```
bigger(elephant, horse).
parent(john, mary).
```

*Rules* consist of a *head* and a *body* separated by `:-`. The head of a rule is true if all predicates in the body can be proved to be true.

```
grandfather(X, Y) :-
  father(X, Z),
  parent(Z, Y).
```

# Programs and Queries

*Programs:* Facts and rules are called *clauses*. A Prolog program is a list of clauses.

*Queries* are predicates (or sequences of predicates) followed by a dot. They are typed in at the Prolog prompt and cause the system to reply.

```
?- is_bigger(horse, X), is_bigger(X, dog).
X = donkey
Yes
```

# Built-in Predicates

- Compiling a program file:

```
?- consult('big-animals.pl').
Yes
```

- Writing terms on the screen:

```
?- write('Hello World!'), nl.
Hello World!
Yes
```

# Matching

Two terms *match* if they are either identical or if they can be made identical by substituting their variables with suitable ground terms.

We can explicitly ask Prolog whether two given terms match by using the equality-predicate `=` (written as an infix operator).

```
?- born(mary, yorkshire) = born(mary, X).
X = yorkshire
Yes
```

The variable instantiations are reported in Prolog's answer.

# Matching (cont.)

```
?- f(a, g(X, Y)) = f(X, Z), Z = g(W, h(X)).
X = a
Y = h(a)
Z = g(a, h(a))
W = a
Yes

?- p(X, 2, 2) = p(1, Y, X).
No
```

# The Anonymous Variable

The variable _ (underscore) is called the *anonymous variable*. Every occurrence of _ represents a different variable (which is why instantiations are not being reported).

```
?- p(_, 2, 2) = p(1, Y, _).
Y = 2
Yes
```

## Answering Queries

Answering a query means proving that the goal represented by that query can be satisfied (according to the programs currently in memory).

*Recall:* Programs are lists of facts and rules. A fact declares something as being true. A rule states conditions for a statement being true.

# Answering Queries (cont.)

- If a goal matches with a *fact*, then it is satisfied.
- If a goal matches the *head of a rule*, then it is satisfied if the goal represented by the rule's body is satisfied.
- If a goal consists of several *subgoals* separated by commas, then it is satisfied if all its subgoals are satisfied.
- When trying to satisfy goals with built-in predicates like `write/1` Prolog also performs the associated action (e.g. writing on the screen).

# Example: Mortal Philosophers

Consider the following argument:

All men are mortal.

Socrates is a man.

---

Hence, Socrates is mortal.

It has two *premises* and a *conclusion*.

# Translating it into Prolog

The two premises can be expressed as a little Prolog program:

```
mortal(X) :- man(X).
man(socrates).
```

The conclusion can then be formulated as a query:

```
?- mortal(socrates).
Yes
```

# Goal Execution

(1) The query `mortal(socrates)` is made the initial goal.

(2) Prolog looks for the first matching fact or head of rule and finds `mortal(X)`. Variable instantiation: `X = socrates`.

(3) This variable instantiation is extended to the rule's body, i.e. `man(X)` becomes `man(socrates)`.

(4) New goal: `man(socrates)`.

(5) Success, because `man(socrates)` is a fact itself.

(6) Therefore, also the initial goal succeeds.

# Programming Style

It is extremely important that you write programs that are easily understood by others! Some guidelines:

- Use *comments* to explain what you are doing:

```
/* This is a long comment, stretching over several
lines, which explains in detail how I have implemented
the aunt/2 predicate ... */

aunt(X, Z) :-
  sister(X, Y),  % This is a short comment.
  parent(Y, Z).
```

# Programming Style (cont.)

- Separate clauses by one or more blank lines.
- Write only one predicate per line and use indentation:

```
blond(X) :-
  father(Father, X),
  blond(Father),
  mother(Mother, X),
  blond(Mother).
```

(Very short clauses may also be written in a single line.)

- Insert a space after every comma inside a compound term:

```
born(mary, yorkshire, '01/01/1980')
```

- Write short clauses with bodies consisting of only a few goals. If necessary, split into shorter sub-clauses.
- Choose meaningful names for your variables and atoms.

## Summary: Syntax

- All Prolog expression are made up from *terms* (numbers, atoms, variables, or compound terms).
- *Atoms* start with lowercase letters or are enclosed in single quotes; *variables* start with capital letters or the underscore.
- Prolog programs are lists of *facts* and *rules* (*clauses*).
- *Queries* are submitted to the system to initiate a computation.
- Some *built-in predicates* have special meaning.

# Summary: Answering Queries

- When answering a query, Prolog tries to prove that the corresponding goal is satisfiable (can be made true). This is done using the rules and facts given in a program.
- A goal is executed by *matching* it with the first possible fact or head of a rule. In the latter case the rule's body becomes the new goal.
- The variable instantiations made during matching are carried along throughout the computation and reported at the end.
- Only the *anonymous variable* _ can be instantiated differently whenever it occurs.

# More on Prolog

By

William Mitchell (University of Arizona)

# Facts and queries

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

Here is a small collection of Prolog *facts*:

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

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

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

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

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

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

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

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

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

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

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

Here's a fact:

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

Here's a query:

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

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

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

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

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

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

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

# Facts and queries, continued

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

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

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

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

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

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

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

?-
```

# Facts and queries, continued

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

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

No
```

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

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

# "Can you prove it?"

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

The query

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

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

The query

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

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

# "Can you prove it?", continued

Consider again a query with a variable:

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

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

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

# Atoms

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

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

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

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

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

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

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

# Atoms, continued

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

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

?- atom(apple).
Yes

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

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

# Predicates, terms, and structures

Here are some more examples of facts:

```
color(sky, blue).

color(grass, green).

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

number(one, 1, 'English').

number(uno, 1, 'Spanish').

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

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

# Predicates, terms, and structures, continued

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

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

Here are examples of structures:

```
color(grass, green)

odd(1)

number(uno, 1, 'Spanish')

equal(V, V)

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

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

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

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

# More queries

A query that requests green things:

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

A query that requests each thing and its color:

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

Thing = dirt
Color = brown ;

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

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

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

# More queries, continued

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

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

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

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

Let's see if any foods are blue:

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

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

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

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

Goals are always executed from left to right.

# More queries, continued

Write these queries:

Who likes baseball?

Who likes a food?

Who likes green foods?

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

```
food(apple).
...

color(sky,blue).
...

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

Answers:

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

# More queries, continued

Are any two foods the same color?

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

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

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

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

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

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

# Alternative representations

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

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

What is a food?

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

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

# Alternate representation, continued

Practice:

What is green that is not a food?

What color is lettuce?

What foods are orange?

What foods are the same color as lettuce?

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

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

Answers:

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

# Unification

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

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

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

?- 3 \== 5.
Yes

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

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

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

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

# Unification, continued

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

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

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

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

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

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

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

# Unification, continued

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

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

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

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

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

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

# Unification, continued

We can think of the query

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

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

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

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

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

The third fact produces a successful unification:

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

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

# Unification, continued

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

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

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

# Unification, continued

Consider again this interaction:

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

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

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

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

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

# Unification, continued

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

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

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

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

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

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

Unification and variable instantiation are cornerstones of Prolog.

# Query execution

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


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

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

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

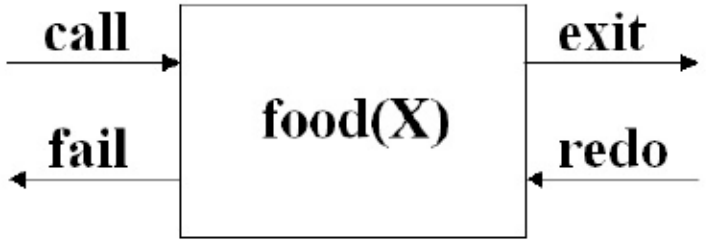
redo: Undo the updates to the variable table [that were made by this goal]. At the place marker, resume the search for a clause which unifies with the goal.

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

# Query execution, continued

Example:

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

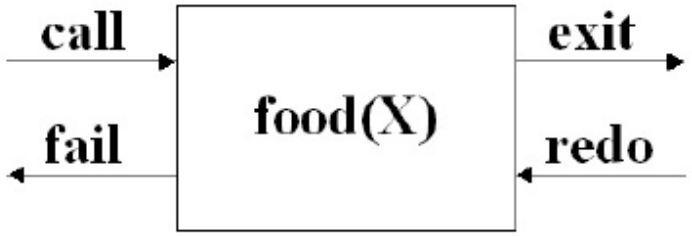



```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

# Query execution, continued

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

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



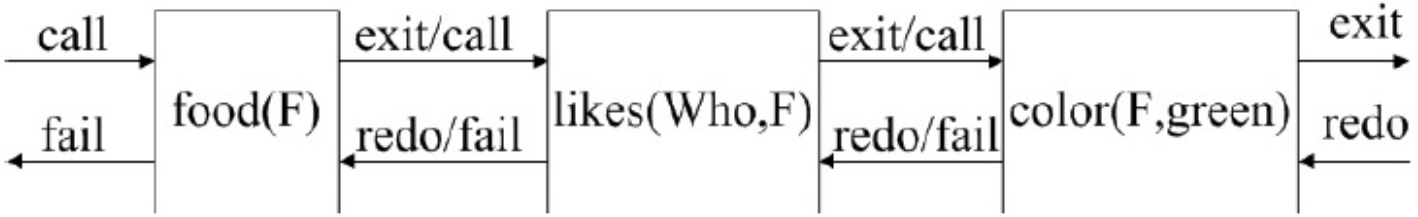

```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).
```

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

# Query execution, continued

Query: Who likes green foods?

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




Facts:

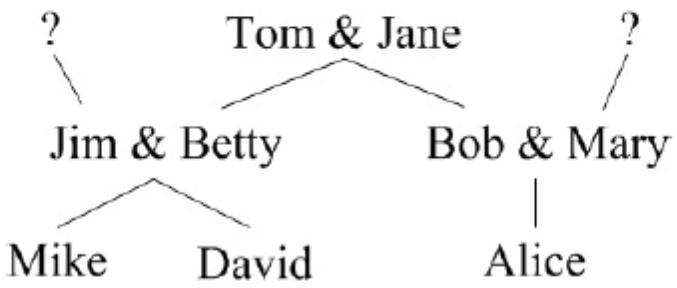
```
food(apple).
food(broccoli).
food(carrot).
food(lettuce).
food(rice).

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

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

Try tracing it!

# Another example of rules

Here is a set of facts describing parents and children:

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




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

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

# Another example, continued

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

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

Who is Bob's father?

For who is Tom the father?

What are all the father/child relationships?

What are all the father/daughter relationships?

What are the grandmother/grandchild relationships?

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

# Another example, continued

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

Queries:

Is Mike a sibling of Alice?

What are the sibling relationships?

Who is somebody's brother?

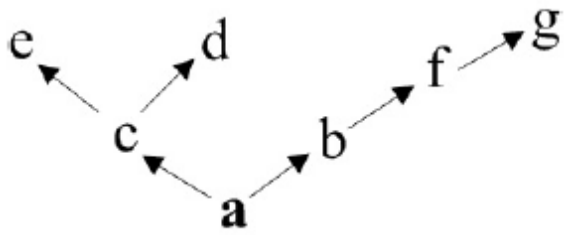
Is the following an equivalent definition of `sibling`?

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

# Recursive predicates

Consider an abstract set of parent/child relationships:

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




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

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

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

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

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

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

# Logic and Prolog

by

Stuart C. Shapiro (SUNY, Buffalo)

# Horn Clauses

A Horn Clause is a clause with at most one positive literal.

Either $\{\neg Q_1(\overline{x}), \ldots, \neg Q_n(\overline{x})\}$ (negative Horn clause)

or $\{C(\overline{x})\}$ (fact or positive or definite Horn clause)

or $\{\neg A_1(\overline{x}), \ldots, \neg A_n(\overline{x}), C(\overline{x})\}$ (positive or definite Horn clause)

which is the same as

$A_1(\overline{x}) \wedge \cdots \wedge A_n(\overline{x}) \Rightarrow C(\overline{x})$

where $A_i(\overline{x})$, $C(\overline{x})$, and $Q(\overline{x})$ are atoms.

# SLD Resolution

Selected literals, Linear pattern, over Definite clauses

SLD derivation of clause $c$ from set of clauses $S$ is

$c_1, \ldots, c_n = c$

s.t. $c_1 \in S$

and $c_{i+1}$ is resolvent of $c_i$ and a clause in $S$. [B&L, p. 87]

If $S$ is a set of Horn clauses,

then there is a resolution derivation of $\{\}$ from $S$

iff there is an SLD derivation of $\{\}$ from $S$.

# SLDSolve

```
procedure SLDSolve(KB,query) returns true or false {
   /* KB = {rule_1, ..., rule_n}
    * rule_i = {h_i, ¬b_i1, ..., ¬b_ik_i}
    * query = {¬q_1, ..., ¬q_m}    */
   if (m = 0) return true;
   for i := 1 to n {
      if((μ := Unify(q_1, h_i)) ≠ FAIL
         and SLDSolve(KB, {¬b_i1μ, ..., ¬b_ik_iμ, ¬q_2μ, ..., ¬q_mμ})) {
        return true;
        }
      }
      return false;
}
```

Where $h_i$, $b_{ij}$, and $q_i$ are atomic formulae.

See B&L, p. 92

# Example Prolog Interaction

```
<timberlake:~/.xemacs:1:35> sicstus
SICStus 4.0.5 (x86_64-linux-glibc2.3): Thu Feb 12 09:48:30 CET 2009
Licensed to SP4cse.buffalo.edu
| ?- consult(user).
% consulting user...
| driver(X) :- drives(X,_).
| passenger(Y) :- drives(_,Y).
| drives(betty,tom).
|
% consulted user in module user, 0 msec 1200 bytes
yes
| ?- driver(X), passenger(Y).
X = betty,
Y = tom ?
yes
| ?- halt.
```

# Prolog Program with two answers

```
% From Rich & Knight, 2nd Edition (1991) p. 192.

likesToEat(X,Y) :- cat(X), fish(Y).
cat(X) :- calico(X).
fish(X) :- tuna(X).

tuna(charlie).
tuna(herb).
calico(puss).
```

# Listing the Fish Program

```
| ?- listing.
calico(puss).

cat(A) :-
        calico(A).

fish(A) :-
        tuna(A).

likesToEat(A, B) :-
        cat(A),
        fish(B).

tuna(charlie).
tuna(herb).

yes
```

Note: `consult(File)` loads the `File` in interpreted mode, whereas `[File]` loads the `File` in compiled mode. `listing` is only possible in interpreted mode.

# Running the Fish Program

```
<timberlake:CSE563:1:39> sicstus
SICStus 4.0.5 (x86_64-linux-glibc2.3): Thu Feb 12 09:48:30 CET 2009
Licensed to SP4cse.buffalo.edu
| ?- ['fish.prolog'].
% compiling /projects/shapiro/CSE563/fish.prolog...
% compiled /projects/shapiro/CSE563/fish.prolog in module user, 0 mse
yes

| ?- likesToEat(puss,X).
X = charlie ? ;
X = herb ? ;
no

| ?- halt.
<timberlake:CSE563:1:40>
```

# Tracing the Fish Program

```
| ?- ['fish.prolog'].
% consulting /projects/shapiro/CSE563/fish.prolog...
% consulted /projects/shapiro/CSE563/fish.prolog in module user, 0
yes

| ?- trace.
% The debugger will first creep -- showing everything (trace)
yes
% trace
```

# Tracing First Answer

```
| ?- likesToEat(puss,X).
        1      1 Call: likesToEat(puss,_442) ?
        2      2 Call: cat(puss) ?
        3      3 Call: calico(puss) ?
        3      3 Exit: calico(puss) ?
        2      2 Exit: cat(puss) ?
        4      2 Call: fish(_442) ?
        5      3 Call: tuna(_442) ?
?       5      3 Exit: tuna(charlie) ?
?       4      2 Exit: fish(charlie) ?
?       1      1 Exit: likesToEat(puss,charlie) ?
X = charlie ? ;
```

# Tracing the Second Answer

```
X = charlie ? ;
        1      1 Redo: likesToEat(puss,charlie) ?
        4      2 Redo: fish(charlie) ?
        5      3 Redo: tuna(charlie) ?
        5      3 Exit: tuna(herb) ?
        4      2 Exit: fish(herb) ?
        1      1 Exit: likesToEat(puss,herb) ?
X = herb ? ;
no
% trace
| ?- notrace.
% The debugger is switched off
yes
```

# Backtracking Example

Program:

```
bird(tweety).
bird(oscar).
bird(X) :- feathered(X).
feathered(maggie).
large(oscar).
ostrich(X) :- bird(X), large(X).
```

Run (No backtracking needed):

```
| ?- ostrich(oscar).
        1      1 Call: ostrich(oscar) ?
        2      2 Call: bird(oscar) ?
?       2      2 Exit: bird(oscar) ?
        3      2 Call: large(oscar) ?
        3      2 Exit: large(oscar) ?
?       1      1 Exit: ostrich(oscar) ?
yes
```

# Backtracking Used

```
| ?- ostrich(X).
        1      1 Call: ostrich(_368) ?
        2      2 Call: bird(_368) ?
?       2      2 Exit: bird(tweety) ?
        3      2 Call: large(tweety) ?
        3      2 Fail: large(tweety) ?
        2      2 Redo: bird(tweety) ?
?       2      2 Exit: bird(oscar) ?
        4      2 Call: large(oscar) ?
        4      2 Exit: large(oscar) ?
?       1      1 Exit: ostrich(oscar) ?
X = oscar ?
yes
```

# Backtracking: Effect of Query

/projects/shapiro/CSE563/Examples/Prolog/backtrack.prolog:

```
supervisorOf(X,Y) :- managerOf(X,Z), departmentOf(Y,Z).
managerOf(jones,accountingDepartment).
managerOf(smith,itDepartment).
departmentOf(kelly,accountingDepartment).
departmentOf(brown,itDepartment).
```

Backtracking not needed:

```
| ?- supervisorOf(smith,X).
        1      1 Call: supervisorOf(smith,_380) ?
        2      2 Call: managerOf(smith,_772) ?
        2      2 Exit: managerOf(smith,itDepartment) ?
        3      2 Call: departmentOf(_380,itDepartment) ?
        3      2 Exit: departmentOf(brown,itDepartment) ?
        1      1 Exit: supervisorOf(smith,brown) ?
X = brown ?
yes
```

# Backtracking Example, part 2

```
supervisorOf(X,Y) :- managerOf(X,Z), departmentOf(Y,Z).
managerOf(jones,accountingDepartment).
managerOf(smith,itDepartment).
departmentOf(kelly,accountingDepartment).
departmentOf(brown,itDepartment).

| ?- supervisorOf(X,brown).
        1      1 Call: supervisorOf(_368,brown) ?
        2      2 Call: managerOf(_368,_772) ?
?       2      2 Exit: managerOf(jones,accountingDepartment) ?
        3      2 Call: departmentOf(brown,accountingDepartment) ?
        3      2 Fail: departmentOf(brown,accountingDepartment) ?
        2      2 Redo: managerOf(jones,accountingDepartment) ?
        2      2 Exit: managerOf(smith,itDepartment) ?
        4      2 Call: departmentOf(brown,itDepartment) ?
        4      2 Exit: departmentOf(brown,itDepartment) ?
        1      1 Exit: supervisorOf(smith,brown) ?
X = smith ?
yes
```

# Negation by Failure & The Closed World Assumption

```
| ?- [user].
% consulting user...
| manager(jones, itSection).
| manager(smith, accountingSection).
|
% consulted user in module user, 0 msec 416 bytes
yes
| ?- manager(smith, itSection).
no
| ?- manager(kelly, accountingSection).
no
```

Negation by failure: "no" means didn't succeed.

CWA: If it's not in the KB, it's not true.

# Cut: Preventing Backtracking

## KB Without Cut

```
| ?- consult(user).
% consulting user...
| bird(oscar).
| bird(tweety).
| bird(X) :- feathered(X).
| feathered(maggie).
| large(oscar).
| ostrich(X) :- bird(X), large(X).
|
% consulted user in module user, 0 msec 1120 bytes
yes
```

# No Backtracking Needed

```
| ?- trace.
% The debugger will first creep -- showing everything (trace)
yes
% trace
| ?- ostrich(oscar).
        1      1 Call: ostrich(oscar) ?
        2      2 Call: bird(oscar) ?
?       2      2 Exit: bird(oscar) ?
        3      2 Call: large(oscar) ?
        3      2 Exit: large(oscar) ?
?       1      1 Exit: ostrich(oscar) ?
yes
% trace
```

# Unwanted Backtracking

```
| ?- ostrich(tweety).
        1      1 Call: ostrich(tweety) ?
        2      2 Call: bird(tweety) ?
?       2      2 Exit: bird(tweety) ?
        3      2 Call: large(tweety) ?
        3      2 Fail: large(tweety) ?
        2      2 Redo: bird(tweety) ?
        4      3 Call: feathered(tweety) ?
        4      3 Fail: feathered(tweety) ?
        2      2 Fail: bird(tweety) ?
        1      1 Fail: ostrich(tweety) ?

no
```

No need to try to solve `bird(tweety)` another way.

# KB With Cut

```
| ?- consult(user).
% consulting user...
| bird(oscar).
| bird(tweety).
| bird(X) :- feathered(X).
| feathered(maggie).
| large(oscar).
| ostrich(X) :- bird(X), !, large(X).
|
% consulted user in module user, 0 msec -40 bytes
yes
% trace
```

# No Extra Backtracking

```
| ?- ostrich(tweety).
        1      1 Call: ostrich(tweety) ?
        2      2 Call: bird(tweety) ?
?       2      2 Exit: bird(tweety) ?
        3      2 Call: large(tweety) ?
        3      2 Fail: large(tweety) ?
        1      1 Fail: ostrich(tweety) ?
no
% trace
```

# `fail`: Forcing Failure

If something is a canary, it is not a penguin.

```
| ?- consult(user).
% consulting user...
| penguin(X) :- canary(X), !, fail.
| canary(tweety).
|
% consulted user in module user, 0 msec 416 bytes
yes
% trace
| ?- penguin(tweety).
        1      1 Call: penguin(tweety) ?
        2      2 Call: canary(tweety) ?
        2      2 Exit: canary(tweety) ?
        1      1 Fail: penguin(tweety) ?
no
% trace
```

# Cut Fails the Head Instance: Program

```
penguin(X) :- canary(X), !, fail.
penguin(X) :- bird(X), swims(X).

canary(tweety).
bird(willy).
swims(willy).
```

# Cut Fails the Head Instance: Run

```
| ?- penguin(willy).
        1      1 Call: penguin(willy) ?
        2      2 Call: canary(willy) ?
        2      2 Fail: canary(willy) ?
        3      2 Call: bird(willy) ?
        3      2 Exit: bird(willy) ?
        4      2 Call: swims(willy) ?
        4      2 Exit: swims(willy) ?
        1      1 Exit: penguin(willy) ?
yes
% trace
| ?- penguin(tweety).
        1      1 Call: penguin(tweety) ?
        2      2 Call: canary(tweety) ?
        2      2 Exit: canary(tweety) ?
        1      1 Fail: penguin(tweety) ?
no
```

# Cut Fails Head Alternatives

```
| ?- penguin(X).
        1      1 Call: penguin(_368) ?
        2      2 Call: canary(_368) ?
        2      2 Exit: canary(tweety) ?
        1      1 Fail: penguin(_368) ?
no
```

Moral:

Use cut when seeing if a ground atom is satisfied (T/F question), but not when generating satisfying instances (wh questions).

# Bad Rule Order

```
penguin(X) :- bird(X), swims(X).
penguin(X) :- canary(X), !, fail.
bird(X) :- canary(X).
canary(tweety).

% trace
| ?- penguin(tweety).
        1      1 Call: penguin(tweety) ?
        2      2 Call: bird(tweety) ?
        3      3 Call: canary(tweety) ?
        3      3 Exit: canary(tweety) ?
        2      2 Exit: bird(tweety) ?
        4      2 Call: swims(tweety) ?
        4      2 Fail: swims(tweety) ?
        5      2 Call: canary(tweety) ?
        5      2 Exit: canary(tweety) ?
        1      1 Fail: penguin(tweety) ?
no
```

# Good Rule Order

```
penguin(X) :- canary(X), !, fail.
penguin(X) :- bird(X), swims(X).
bird(X) :- canary(X).
canary(tweety).

% trace
| ?- penguin(tweety).
        1      1 Call: penguin(tweety) ?
        2      2 Call: canary(tweety) ?
        2      2 Exit: canary(tweety) ?
        1      1 Fail: penguin(tweety) ?
no
```

# SICSTUS Allows "or" In Body.

```
bird(willy).
swims(willy).
canary(tweety).
penguin(X) :-
        canary(X), !, fail;
        bird(X), swims(X).
bird(X) :- canary(X).

| ?- ['twoRuleCutOr.prolog'].
% compiling /projects/shapiro/CSE563/twoRuleCutOr.prolog...
* clauses for user:bird/1 are not together
* Approximate lines: 8-10, file: '/projects/shapiro/CSE563/twoRuleCutOr.prolog'
% compiled /projects/shapiro/CSE563/twoRuleCutOr.prolog in module user, 0 msec 928 bytes
yes
| ?- penguin(willy).
yes
| ?- penguin(tweety).
no
```

# not: "Negated" Antecedents

A bird that is not a canary is a penguin.

```
| penguin(X) :- bird(X), !, \+canary(X).
| bird(opus).
| canary(tweety).
% compiled user in module user, 0 msec 512 bytes

| ?- penguin(opus).
        1      1 Call: penguin(opus) ?
        2      2 Call: bird(opus) ?
        2      2 Exit: bird(opus) ?
        3      2 Call: canary(opus) ?
        3      2 Fail: canary(opus) ?
        1      1 Exit: penguin(opus) ?
yes
```

\+ is SICStus Prolog's version of **not**.

It is negation by failure, not logical negation.

# Can Use Functions

```
driver(X) :- drives(X,_).
drives(mother(X),X) :- schoolchild(X).
schoolchild(betty).
schoolchild(tom).

| ?- driver(X).
X = mother(betty) ? ;
X = mother(tom) ? ;
no
```

# Infinitely Growing Terms

```
driver(X) :- drives(X,_).
drives(mother(X),X) :- commuter(X).
commuter(betty).
commuter(tom).
commuter(mother(X)) :- commuter(X).

| ?- driver(X).
X = mother(betty) ? ;
X = mother(tom) ? ;
X = mother(mother(betty)) ? ;
X = mother(mother(tom)) ? ;
X = mother(mother(mother(betty))) ? ;
X = mother(mother(mother(tom))) ?
yes
```

# Prolog Does Not Do the Occurs Check

```
<pollux:CSE563:2:31> sicstus
...
| ?- [user].
% consulting user...
| mother(motherOf(X), X).
|
% consulted user in module user, 0 msec 248 bytes
yes
| ?- mother(Y, Y).
Y = motherOf(motherOf(motherOf(motherOf(motherOf(motherOf(motherOf(
              motherOf(motherOf(motherOf(motherOf(motherOf(...)))))))))) ?
yes
| ?-
```

# "=" and "is"

```
| ?- p(X, b, f(c,Y)) = p(a, U, f(V,U)).
U = b,
V = c,
X = a,
Y = b ?
yes
| ?- X is 2*(3+6).
X = 18 ?
yes
| ?- X = 2*(3+6).
X = 2*(3+6) ?
yes
| ?- X is 2*(3+6), Y is X/3.
X = 18,
Y = 6.0 ?
yes
| ?- Y is X/3, X is 2*(3+6).
! Instantiation error in argument 2 of is/2
! goal:  _76 is _73/3
```

# Avoid Left Recursive Rules

To define `ancestor` as the transitive closure of `parent`.

The base case: `ancestor(X,Y) :- parent(X,Y).`

Three possible recursive cases:

1. `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).`
2. `ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).`
3. `ancestor(X,Y) :- ancestor(X,Z), ancestor(Z,Y).`

Versions (2) and (3) will cause infinite loops.