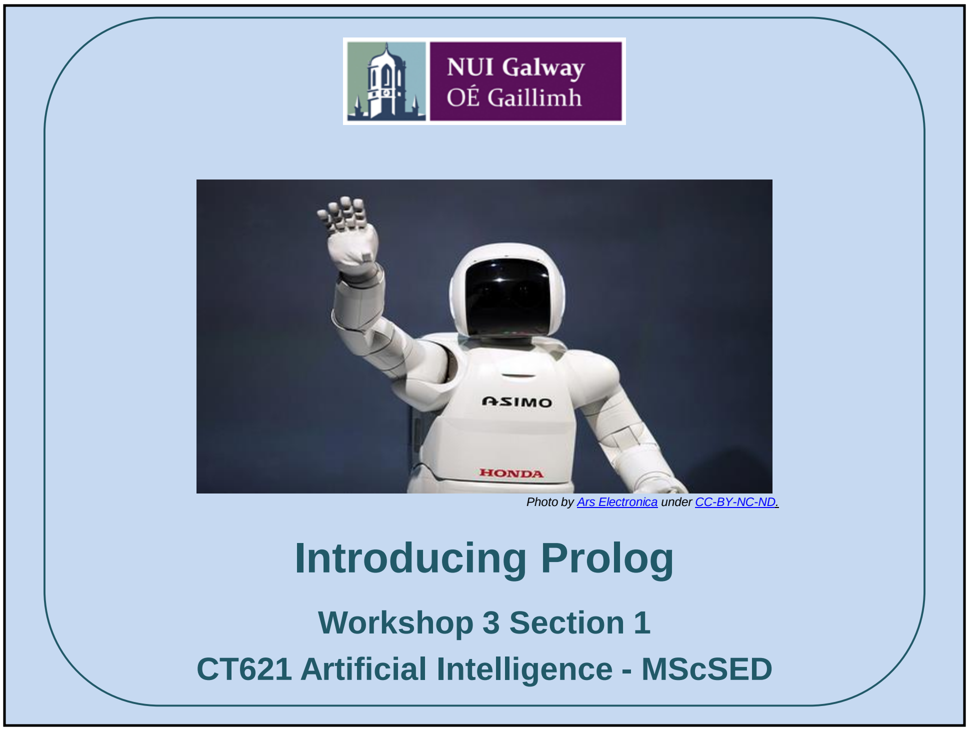


NUI Galway
OÉ Gaillimh

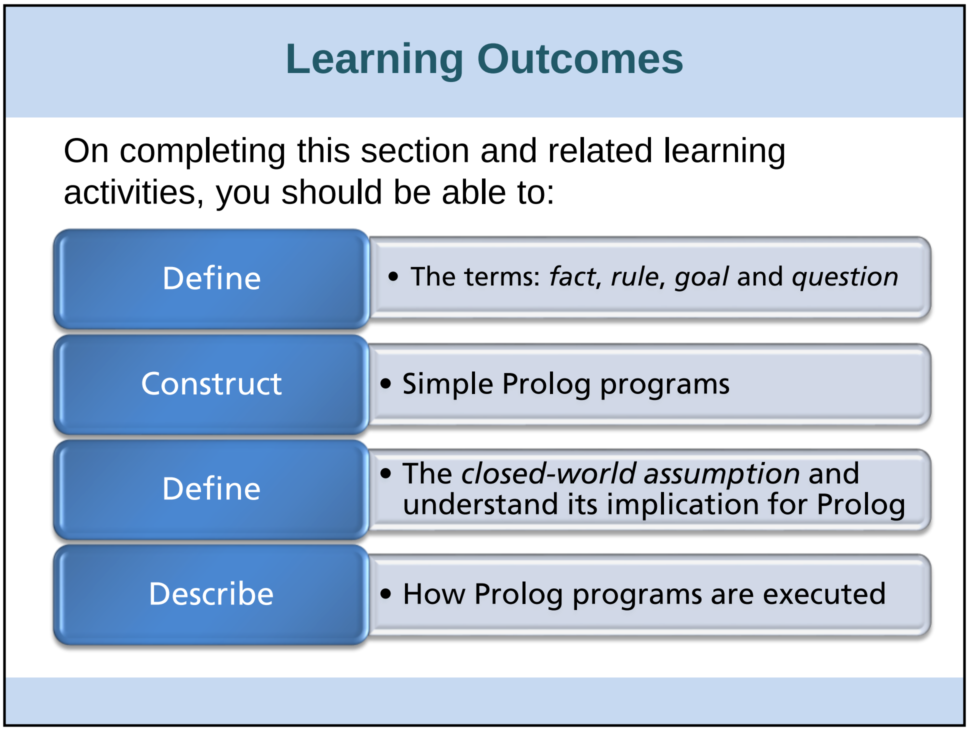
Photo by Ars Electronica under CC-BY-NC-ND.

Introducing Prolog

Workshop 3 Section 1

CT621 Artificial Intelligence - MScSED

Workshop 3 demonstrates how to write programs using Prolog, a programming language that was specifically designed for Artificial Intelligence. Prolog uses Predicate Calculus. We discussed Predicate Calculus in Workshop 1 for performing logical reasoning and inference.



Learning Outcomes

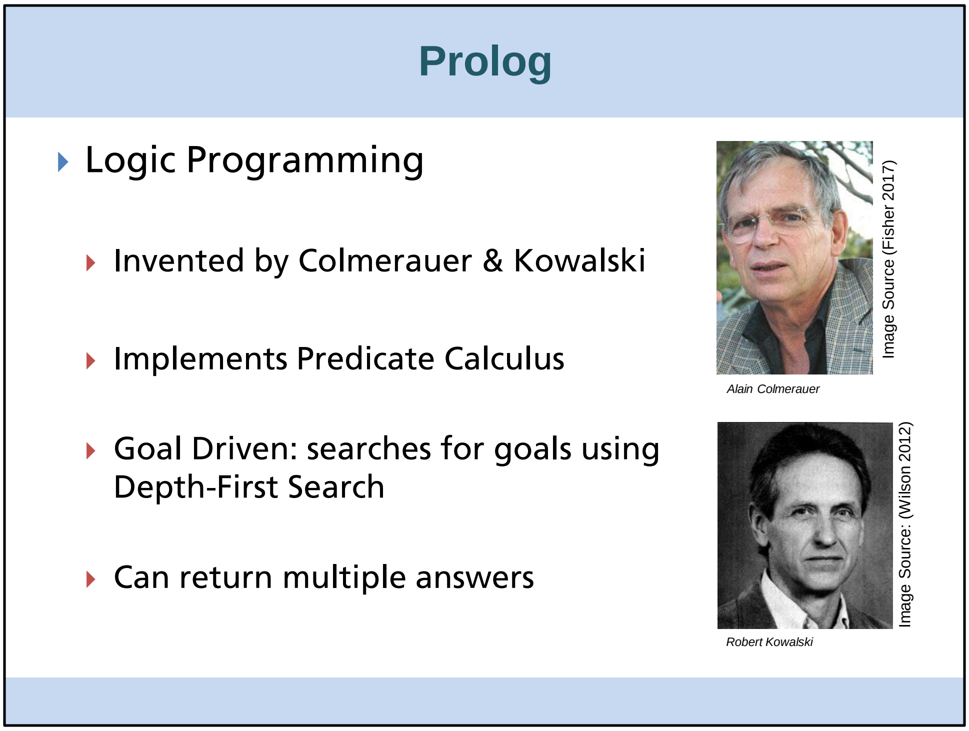
On completing this section and related learning activities, you should be able to:

Define
- The terms: *fact*, *rule*, *goal* and *question*

Construct
- Simple Prolog programs

Define
- The *closed-world assumption* and understand its implication for Prolog

Describe
- How Prolog programs are executed



Prolog

- Logic Programming
  - Invented by Colmerauer & Kowalski
  - Implements Predicate Calculus
  - Goal Driven: searches for goals using Depth-First Search
  - Can return multiple answers

Image Source (Fisher 2017)

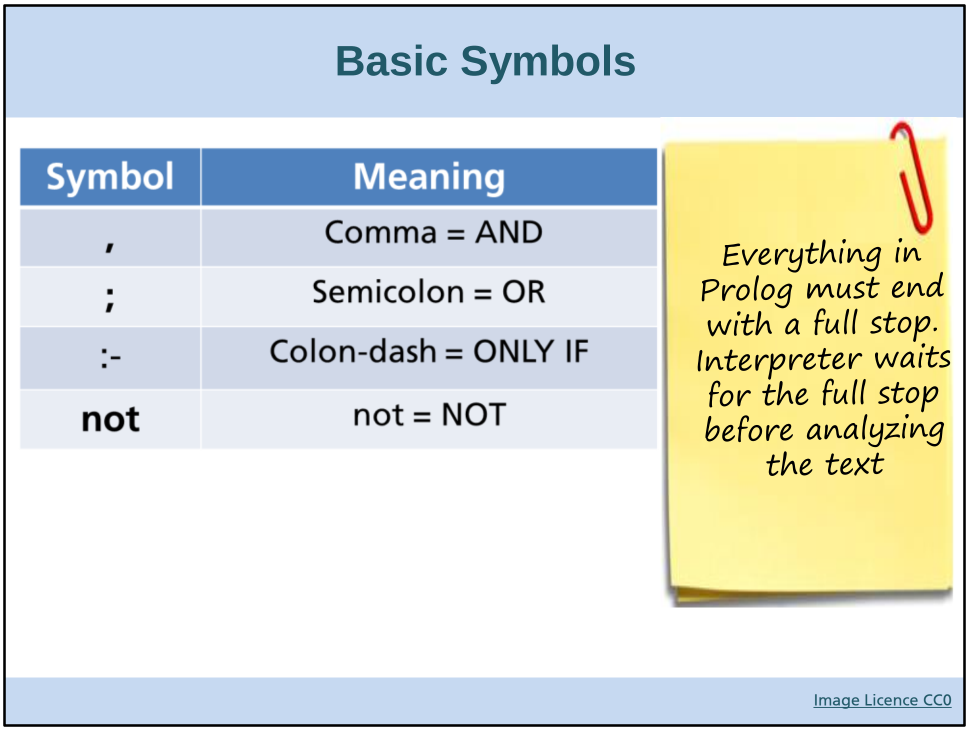
Alain Colmerauer

Image Source: (Wilson 2012)

Robert Kowalski

Prolog is a widely used AI language. It is particularly useful as it allows fast-prototyping of systems. This is due to the nature of the programming language, which removes the requirement from the programmer of having to tell the computer how to solve a problem. Instead, the programmer essentially describes the problem to be solved, and the Prolog environment takes care of the rest.

Prolog is essentially an implementation of Predicate Calculus (we met Predicate Calculus in Workshop 1 Section 2). It is goal-driven, which means that when running a Prolog program, we simply state what we want to try and deduce (the goal), and Prolog tries to see if it can satisfy our goal. It uses Depth-First Search, so if it needs to prove more than one thing, it works the first one out completely, before moving onto the rest. It is also capable of returning multiple answers, so we can ask questions and find all possible answers to them.



Basic Symbols

Symbol	Meaning
,	Comma = AND
;	Semicolon = OR
:-	Colon-dash = ONLY IF
not	not = NOT

Everything in Prolog must end with a full stop. Interpreter waits for the full stop before analyzing the text

Image Licence CC0

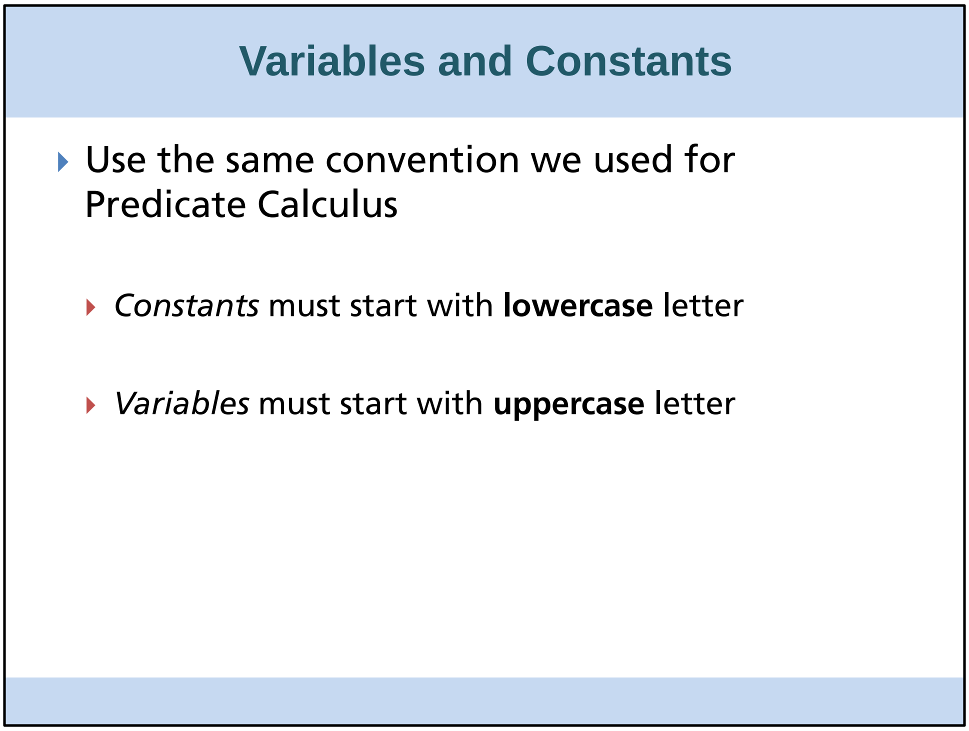
Before looking at any code, we'll cover the basic constructs used in Prolog.

There are only really four symbols used in Prolog:

"**,**" represents "and"
"**;**" represents "or"
"**:-**" represents "only if"
and "**not**" simply represents "not"

We'll see examples of these shortly.

The other main symbol used in Prolog is the full stop "**.**". ALL programming constructs in Prolog (rules, facts and questions) MUST end with a full stop. The Prolog interpreter relies on the presence of the full stop to detect the end of each programming construct. If you ask Prolog a question and it appears to do nothing, it is possible that you have missed the full stop. The interpreter won't start to answer your question until it encounters the full stop.

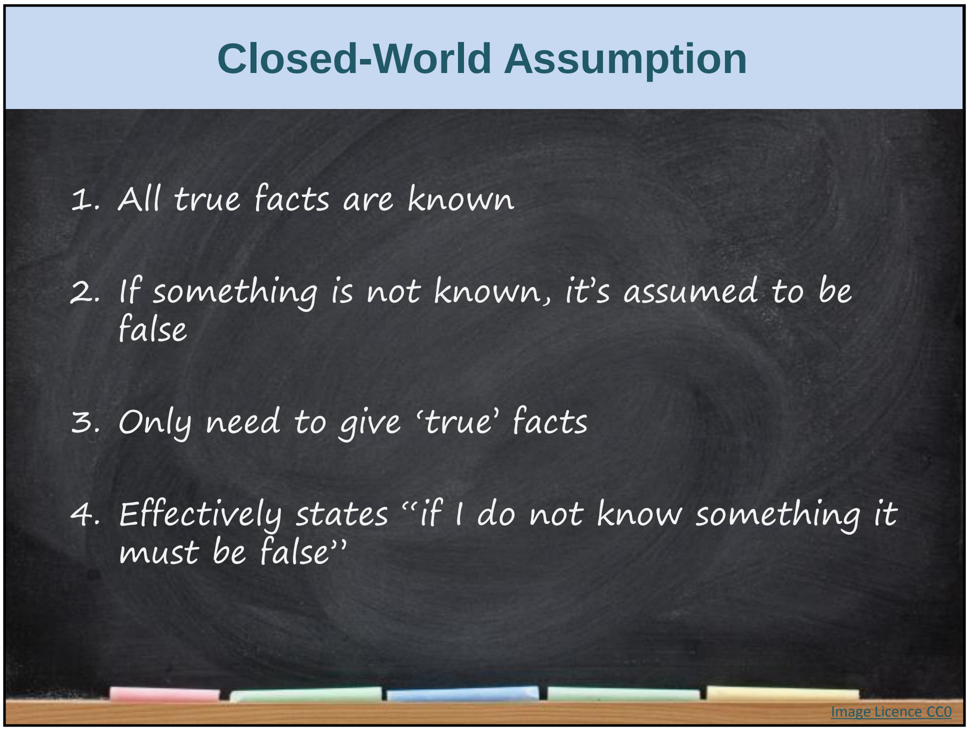


Variables and Constants

- Use the same convention we used for Predicate Calculus
  - *Constants* must start with **lowercase** letter
  - *Variables* must start with **uppercase** letter

There are no data types in Prolog, only *variables* and *constants*. Variables can be assigned ANY kind of value, as we'll see later in the workshop.

The same convention is used in Prolog as in Predicate Calculus, so variables MUST start with an uppercase letter and constants MUST start with a lowercase letter.

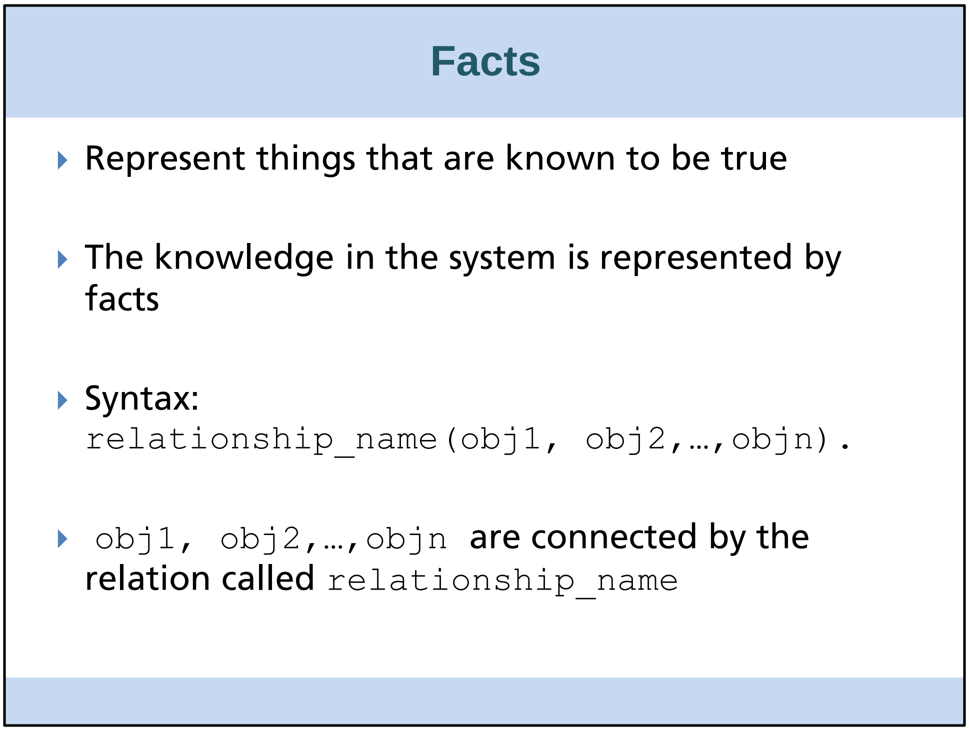


Closed-World Assumption

1. All true facts are known
2. If something is not known, it's assumed to be false
3. Only need to give 'true' facts
4. Effectively states "if I do not know something it must be false"

Image Licence CC0

One of the common problems with reasoning systems is deciding what knowledge to represent. Prolog makes the assumption that it knows everything that is true, therefore, anything that it doesn't know, or can't deduce, must be false. This has a big effect on how you formulate Prolog programs, as we only need to tell it what it needs to know. This is called the *Closed World Assumption*, as we have effectively defined the 'world' we are considering by the data we have given the program.



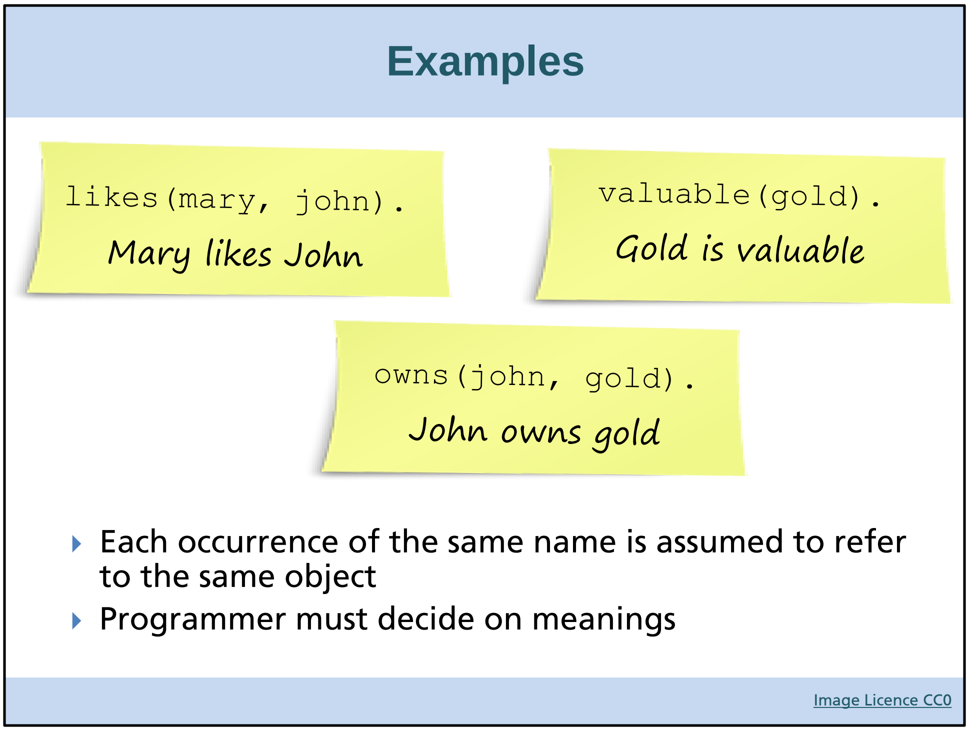
Facts

- Represent things that are known to be true
- The knowledge in the system is represented by facts
- Syntax:
  `relationship_name(obj1, obj2,…,objn).`
- `obj1, obj2,…,objn` are connected by the relation called `relationship_name`

The simplest programming construct in Prolog is the *fact*. Facts are used to state things that are known to be true, and essentially provide a way of creating a database of knowledge that the program can use.

Facts have a simple structure, the name of the fact, with a list of ‘arguments’ enclosed in brackets.

Note that the facts MUST end with a full stop.



Examples

```
likes(mary, john).
```

Mary likes John

```
valuable(gold).
```

Gold is valuable

```
owns(john, gold).
```

John owns gold

- Each occurrence of the same name is assumed to refer to the same object
- Programmer must decide on meanings

Image Licence CC0

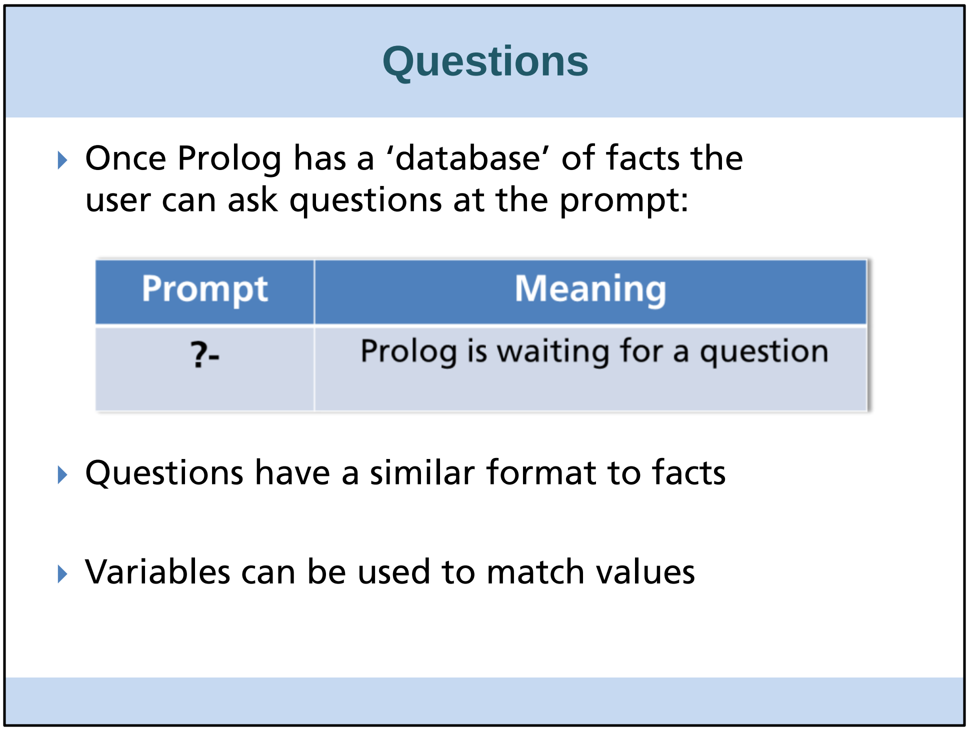
This slide shows some simple examples of facts.

They all start with the name of the fact (**likes, valuable or owns**) and relate the values in brackets in some way.

Notice that all of the fact names and the arguments start with a lowercase letter, so they are all constants.

There is no meaning to any of these constants, apart from the meaning that the programmer intended them to have.

Where a constant appears more than once in a program, it is assumed to refer to the same object (or entity).



Questions

- Once Prolog has a 'database' of facts the user can ask questions at the prompt:

Prompt	Meaning
?-	Prolog is waiting for a question

- Questions have a similar format to facts
- Variables can be used to match values

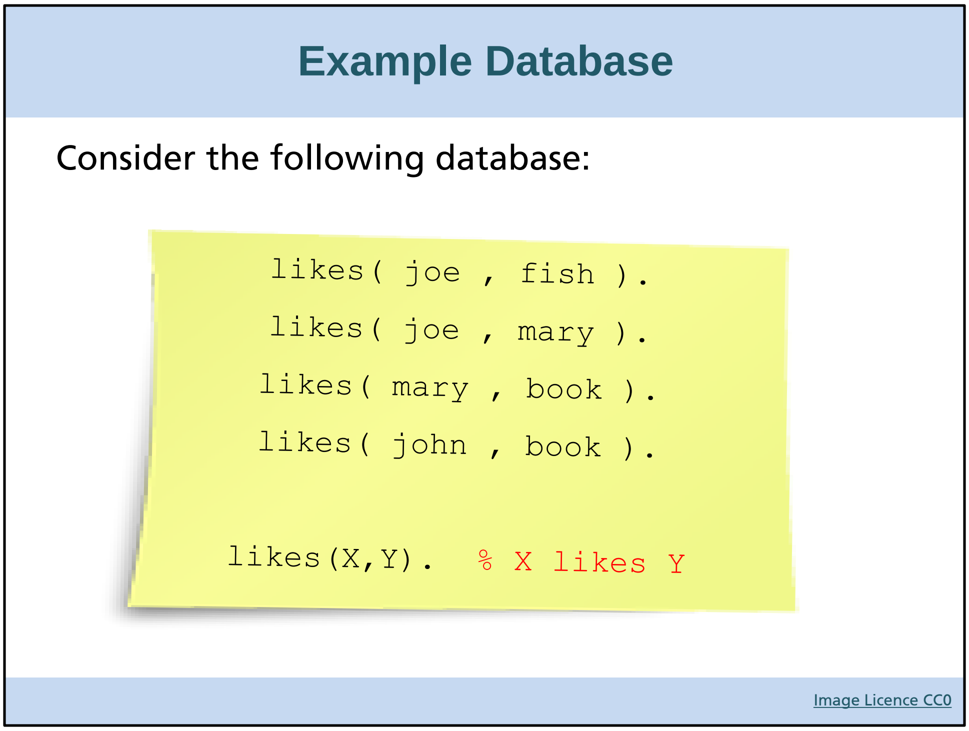
The interaction with a Prolog program is via *questions*, so these are the way that we tell the Prolog program what we want it to do.

When you are using Prolog and encounter the prompt:

```
?-
```

It tells you that Prolog is waiting for you to ask it a question.

The simplest form of a question is just the same as that of a fact, as we'll see on the next slide.



Example Database

Consider the following database:

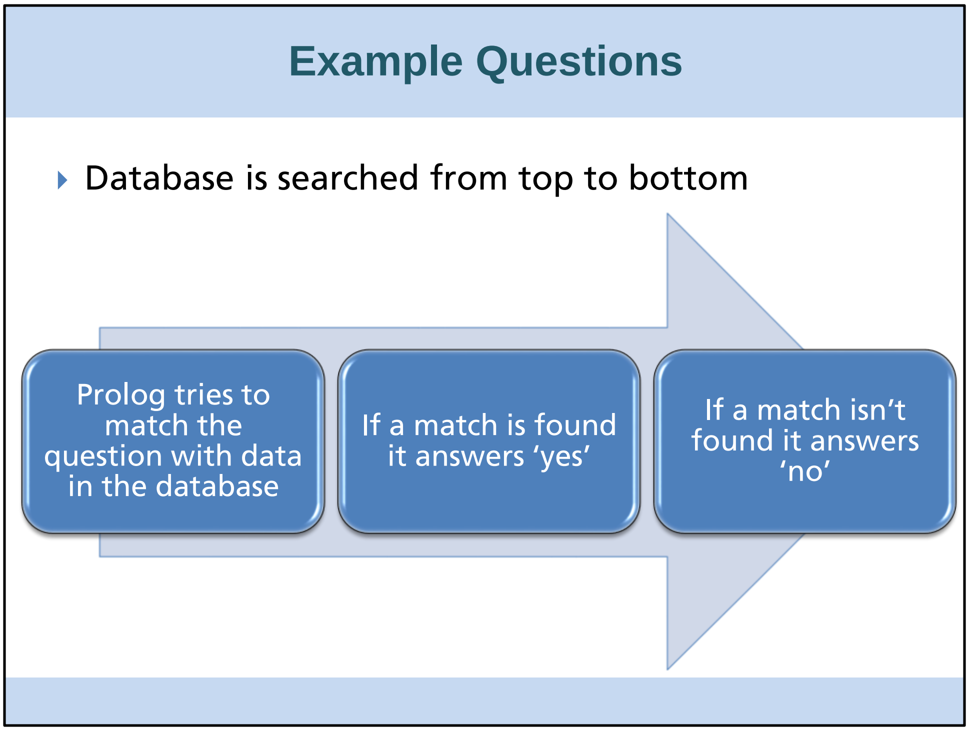
```
likes( joe , fish ).
likes( joe , mary ).
likes( mary , book ).
likes( john , book ).

likes(X,Y).   % X likes Y
```

Image Licence CC0

The examples on the next few slides are questions that are asked of the database on this slide.

This simple database, contains 4 facts, each of which represents the relation likes. The last line shows a convention for commenting facts, namely a statement of the meaning of a fact. So in this case, `likes(X,Y).` states that object `X` likes object `Y`.



Example Questions

- Database is searched from top to bottom

Prolog tries to match the question with data in the database

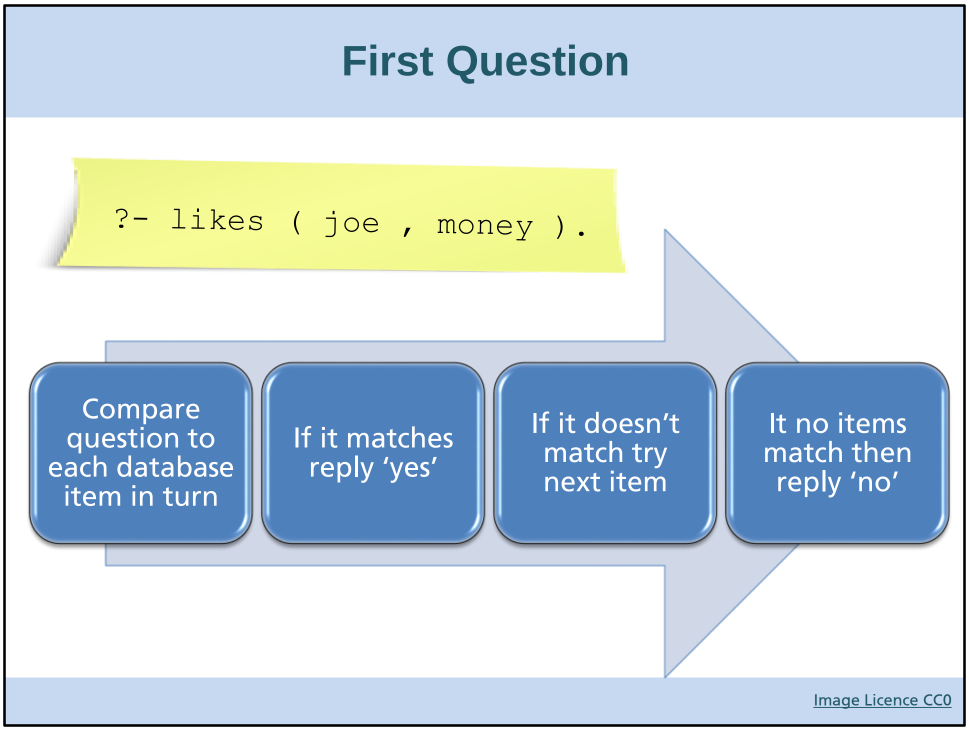
If a match is found it answers 'yes'

If a match isn't found it answers 'no'

When you ask Prolog a question, it tries to match your question with the facts in the database. It does this in a systematic way, by looking at the contents of the database, one at a time, starting with the first element in the database, and working its way down the database until it either finds a match, or reaches the end of the database.

If a match is found it displays the word "yes" and then the prompt. If a match isn't found, it displays the word "no" and then the prompt.

It is important that you understand how Prolog tries to answer questions, as the order that the facts (and other constructs) are entered can have an effect on the way the program behaves.



First Question

```
?- likes ( joe , money ).
```

Compare question to each database item in turn

If it matches reply 'yes'

If it doesn't match try next item

It no items match then reply 'no'

The first question we'll look at is:
?- likes (joe , money).

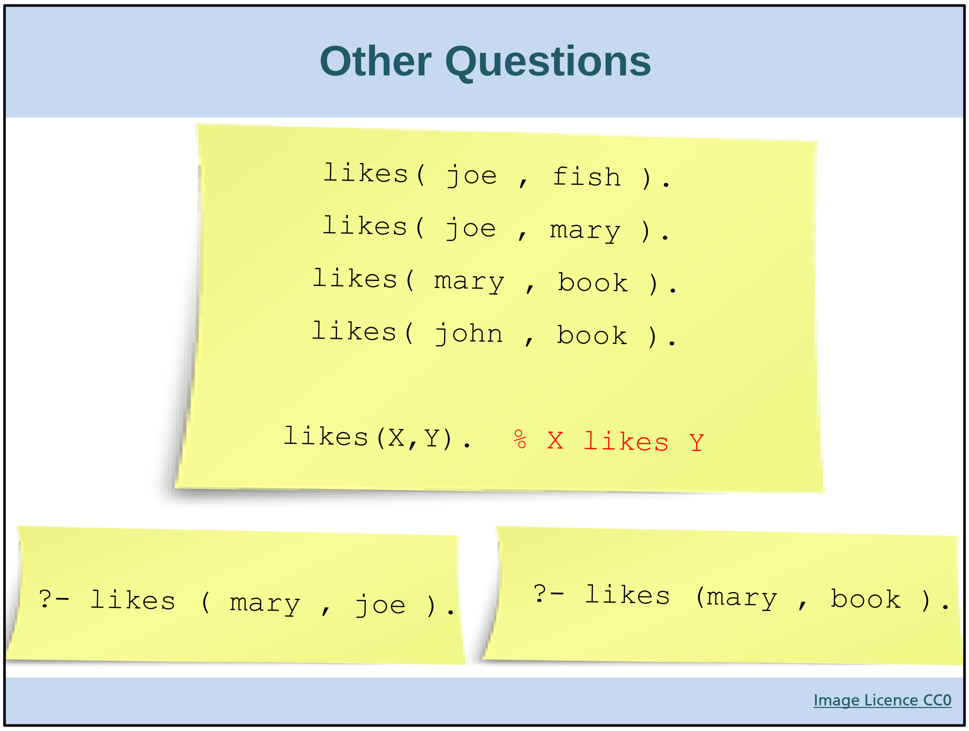
The **?**- is Prolog's prompt, so what we are asking is "does joe like money?". Note the full stop at the end of the question.

To attempt to answer this question, Prolog compares it to each of the facts in the database in turn. If it finds a match it replies "yes", otherwise it moves onto the next fact. If the end of the database is reached without finding a match, the "no" is returned.

For this question, Prolog achieves a partial matching for each fact in the database, as they are all **likes** facts. The first two facts also match the first argument, as the first arguments are the same. In these two cases, the second argument is also examined, but in neither case does it match. As the fact **likes(joe,money**). is not in the database, Prolog returns the answer no.

Video: Programming In Prolog Part 1 - Facts, Rules and Queries

Take a look at this short demo outlining Programming In Prolog Part 1 - Facts, Rules and Queries.
YouTube link: https://www.youtube.com/watch?v=gJOZZvYijqk



Other Questions

```
likes( joe , fish ).
likes( joe , mary ).
likes( mary , book ).
likes( john , book ).

likes(X,Y).  % X likes Y
```

```
?- likes ( mary , joe ).
```

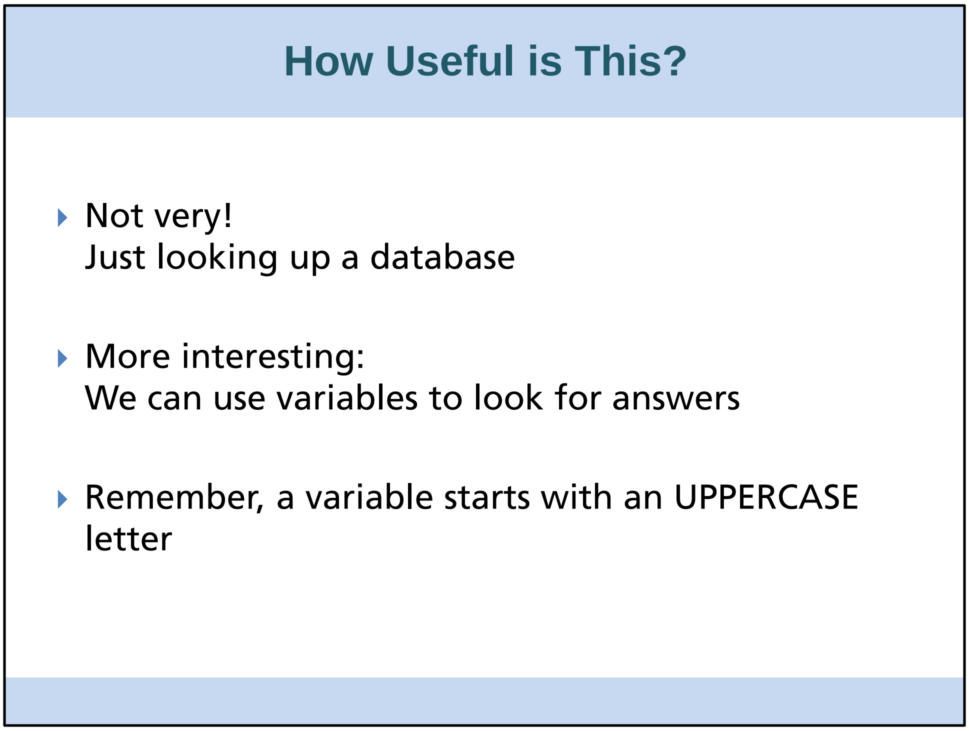
```
?- likes (mary , book ).
```

Image Licence CC0

Using the same database as previously, what are the answers to these questions?

For the first question, the third element partially matches, but not completely, so the answer is “no”.

For the second question, the third element matches exactly, so the answer is “yes”.



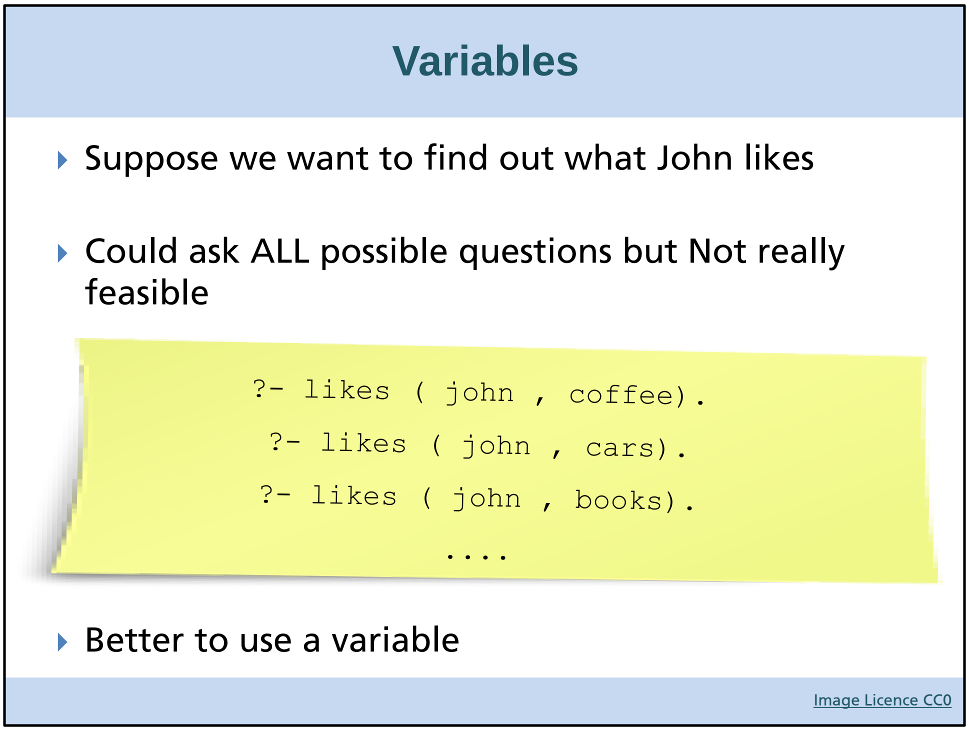
How Useful is This?

- Not very!
  Just looking up a database
- More interesting:
  We can use variables to look for answers
- Remember, a variable starts with an UPPERCASE letter

So far we've seen how we can define some facts and ask simple questions. This is not very useful, as all we are doing is checking to see whether a particular fact is in the database or not.

By using variables, we'll be able to ask more interesting questions.

Remember, the difference between a variable and a constant is that a variable starts with an uppercase letter.



Variables

- Suppose we want to find out what John likes
- Could ask ALL possible questions but Not really feasible

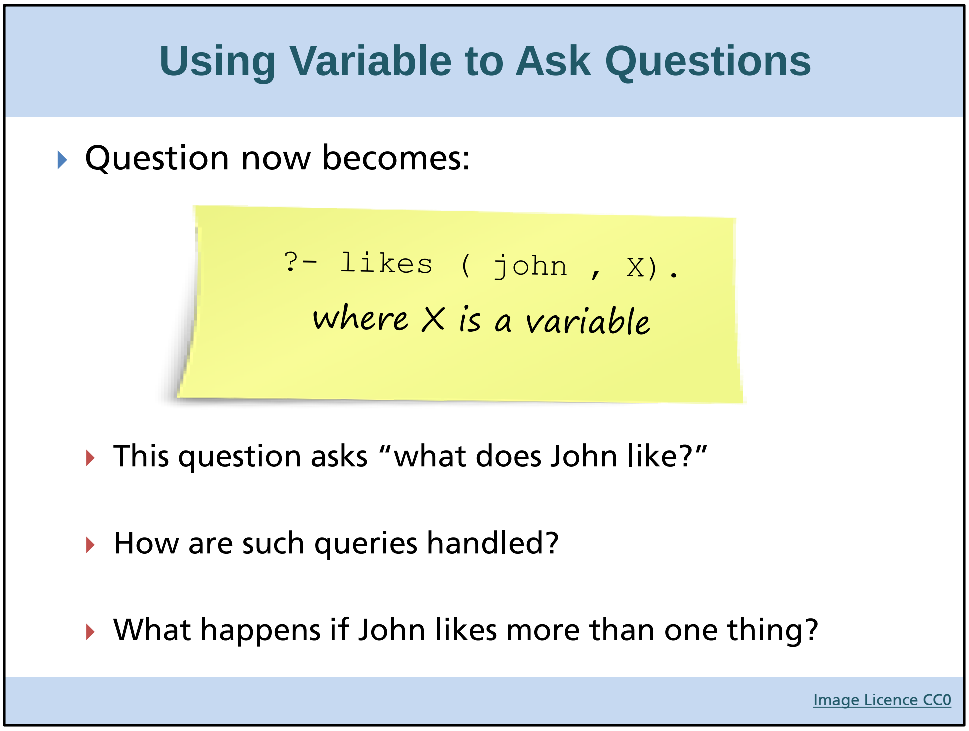
```
?- likes ( john , coffee).
?- likes ( john , cars).
?- likes ( john , books).
....
```

- Better to use a variable

Image Licence CC0

If we wanted to ask a more general question than those we have seen so far, such as “What does John like?”, we could simply ask repeated questions, with different values for the second argument, which isn’t really very practical.

It is far simpler to use a variable.



Using Variable to Ask Questions

- Question now becomes:

```
?- likes ( john , X).
```

where X is a variable

- This question asks "what does John like?"
- How are such queries handled?
- What happens if John likes more than one thing?

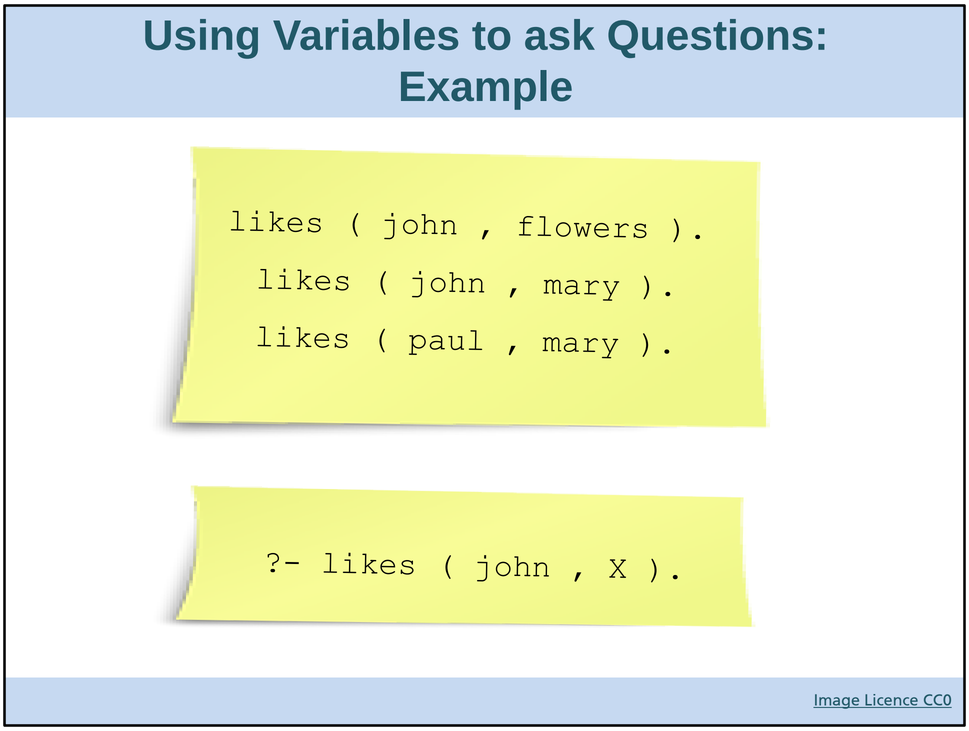
Image Licence CC0

By using a variable, our question becomes:

?- likes (john , X) .

The only difference between this question and those that we have seen before is that the second argument is a variable. So unlike previous questions, we are asking something general, rather than something specific. In this case the question is "What does John like?".

We'll see how this question is handled over the next few slides.



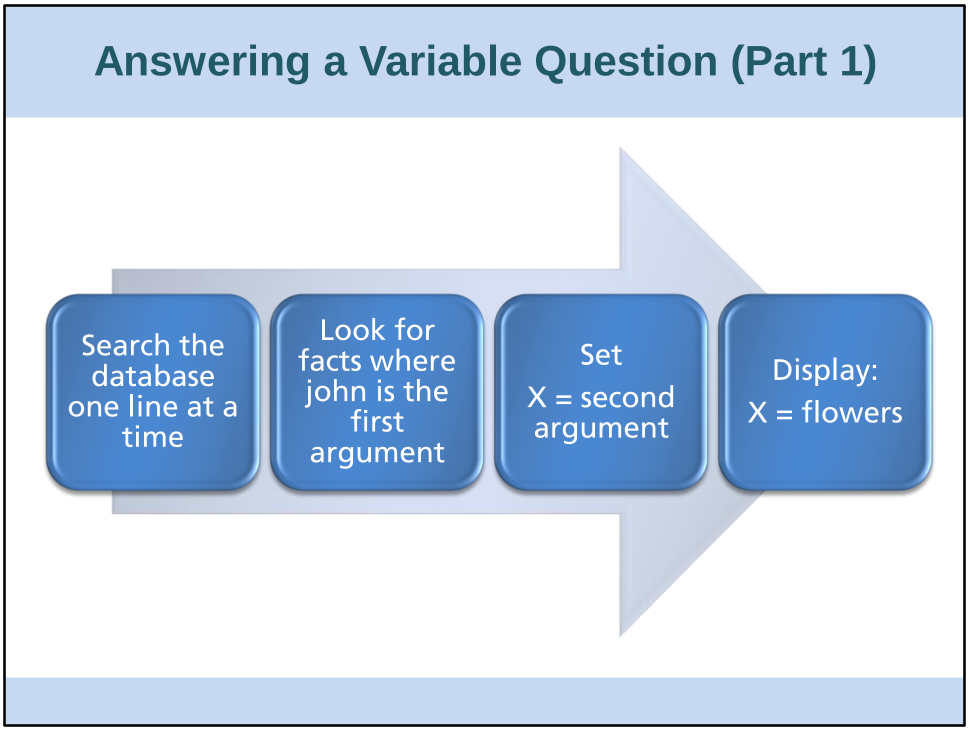
Using Variables to ask Questions: Example

```
likes ( john , flowers ).
likes ( john , mary ).
likes ( paul , mary ).
```

```
?- likes ( john , X ).
```

Image Licence CC0

So given this simple database, and the question (?- likes (john , X).), how does Prolog try to answer this question? It is handled in essentially the same way as the previous questions.



Answering a Variable Question (Part 1)

Search the database one line at a time

Look for facts where john is the first argument

Set X = second argument

Display: X = flowers

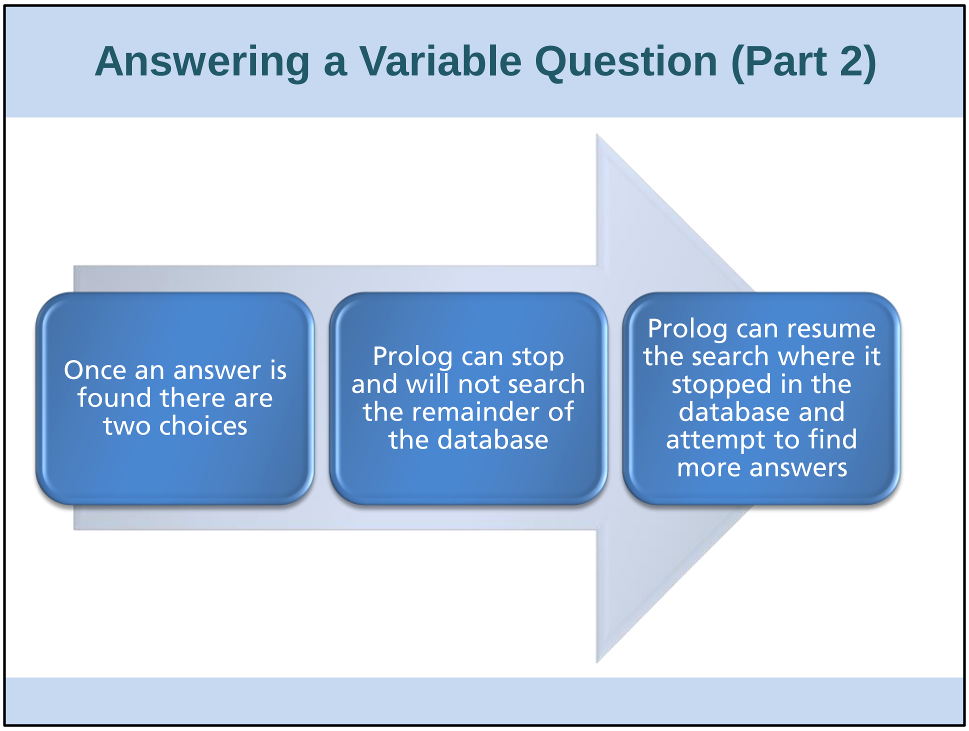
Essentially the same approach that we have already seen is applied. The database is searched from top to bottom, until either a match is found, or the end of the database is reached.

In this case, the first fact has the same name as the question (`likes`), so that matches. The first arguments are then compared, in this case they are both `john` so again we have a match. Prolog now looks at the last argument, in the question we have a variable (`X`), whilst in the fact we have a value (`flowers`). For these to match, we need to assign the variable `X` a value, so it is given the value `flowers`, to match the value in the database. This assigning of a value to a variable is called *instantiation*.

Once this has happened, Prolog has matched all of the values in the question with an element in the database, so it has answered the question. To tell the user the answer, it displays the following:

```
X = flowers
```

to show the value it has found for the variable `X`. Remember in our previous questions the answer was simply “yes”.
At this point the prompt does not appear, as Prolog is waiting for us to tell it what to do next.



Answering a Variable Question (Part 2)

Once an answer is found there are two choices

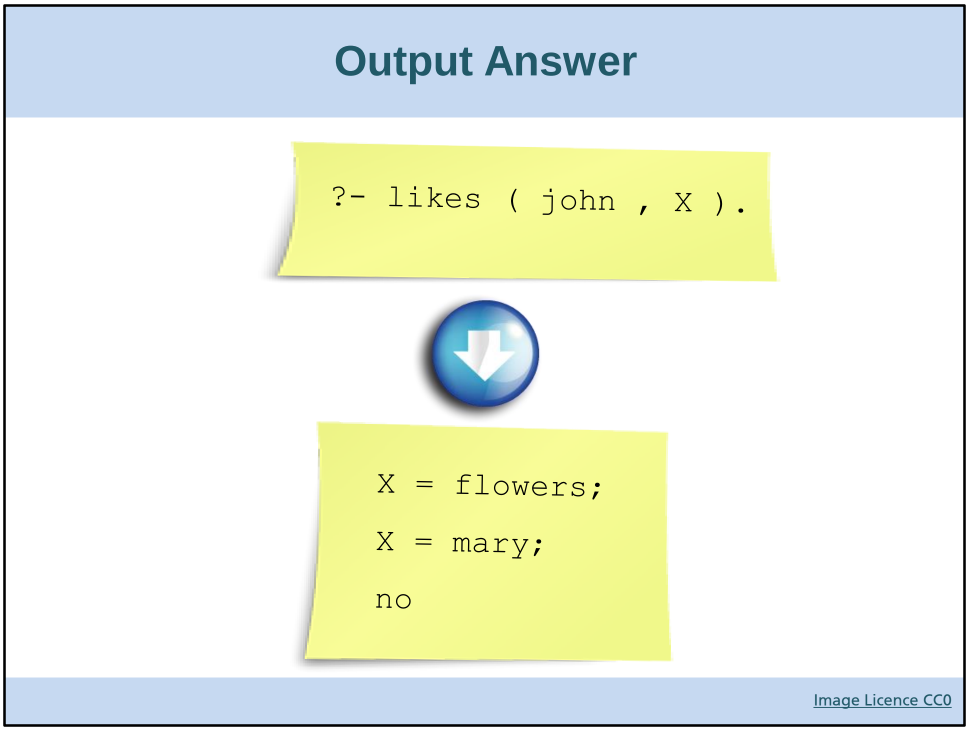
Prolog can stop and will not search the remainder of the database

Prolog can resume the search where it stopped in the database and attempt to find more answers

Once Prolog has found an answer to our question, we have two choices; we can either look for more answers or stop.

To stop, we just press the return key, at which point the prompt reappears and Prolog is ready for our next question.
If we want to look for more answers, we type a semi-colon ";" and then return.

Prolog now resumes its search from where it stopped to try and find other answers. If it finds another answer it displays it as before, otherwise it returns "no". Note that we could keep searching for as many answers as there are in the database.



Output Answer

```
?- likes ( john , X ).
```

```
X = flowers;
X = mary;
no
```

Image Licence CC0

This slide shows the answers we get to the question:

```
?- likes ( john , X ) .
```

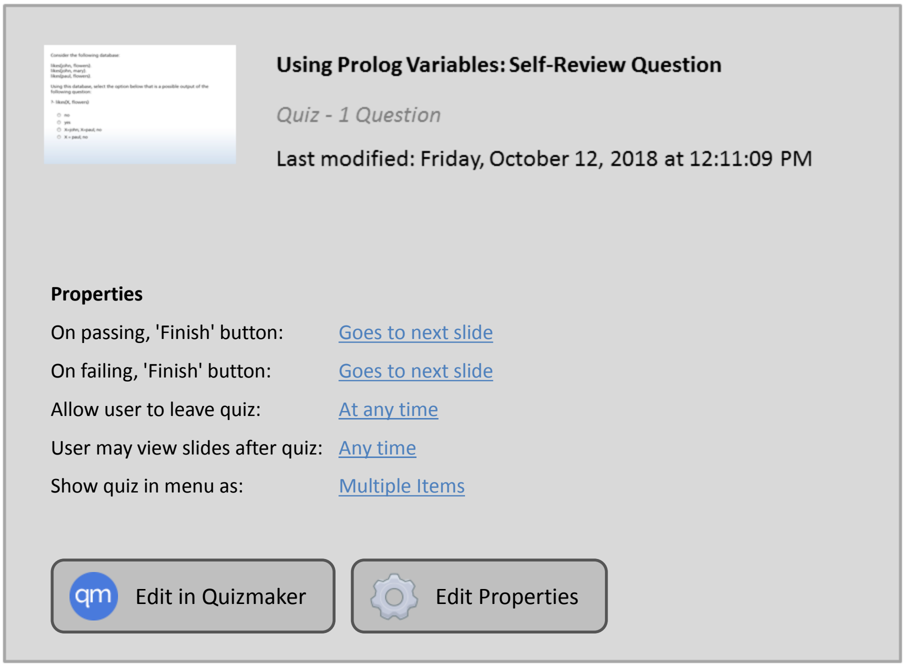
It initially returns:

```
X = flowers
```

On us typing a “;” and then return, it then resumes it search and then returns:

```
X = mary
```

Once again we type “;” and return, to look for more answers. This time there are no more answers, so it returns ”no”.



Consider the following database:

```
likes(john, flowers).
likes(john, mary).
likes(paul, flowers).
```

Using this database, select the option below that is a possible output of the following question:

```
?- likes(X, flowers)
```

- ○ no
- ○ yes
- ○ X=john; X=paul; no
- ○ X = paul; no

Using Prolog Variables: Self-Review Question

Quiz - 1 Question

Last modified: Friday, October 12, 2018 at 12:11:09 PM

Properties

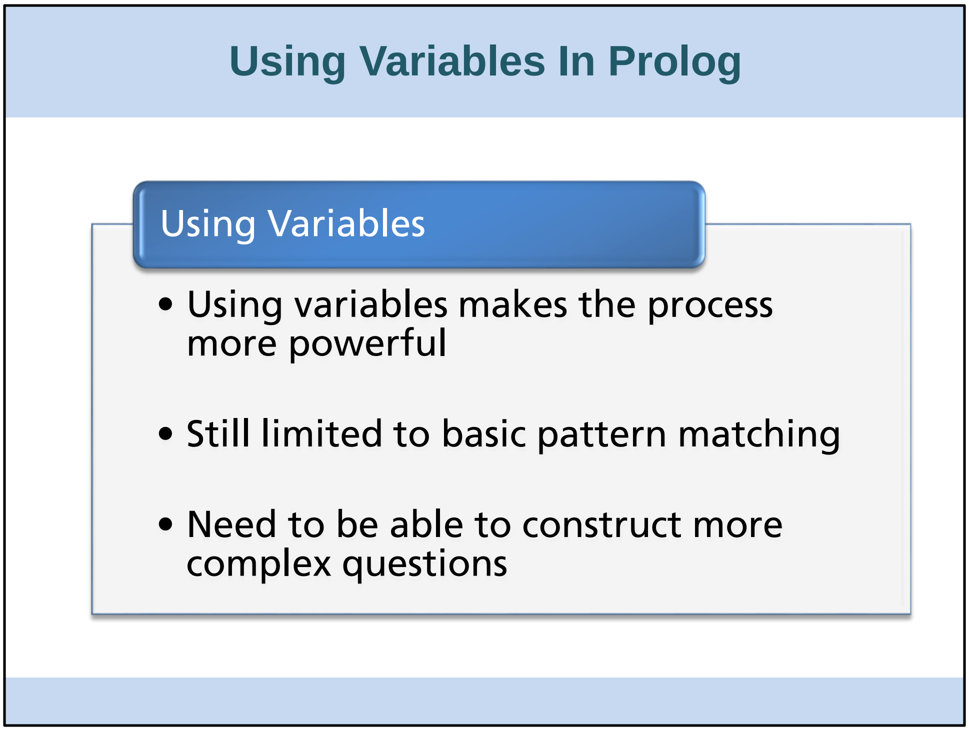
On passing, 'Finish' button: Goes to next slide

On failing, 'Finish' button: Goes to next slide

Allow user to leave quiz: At any time

User may view slides after quiz: Any time

Show quiz in menu as: Multiple Items

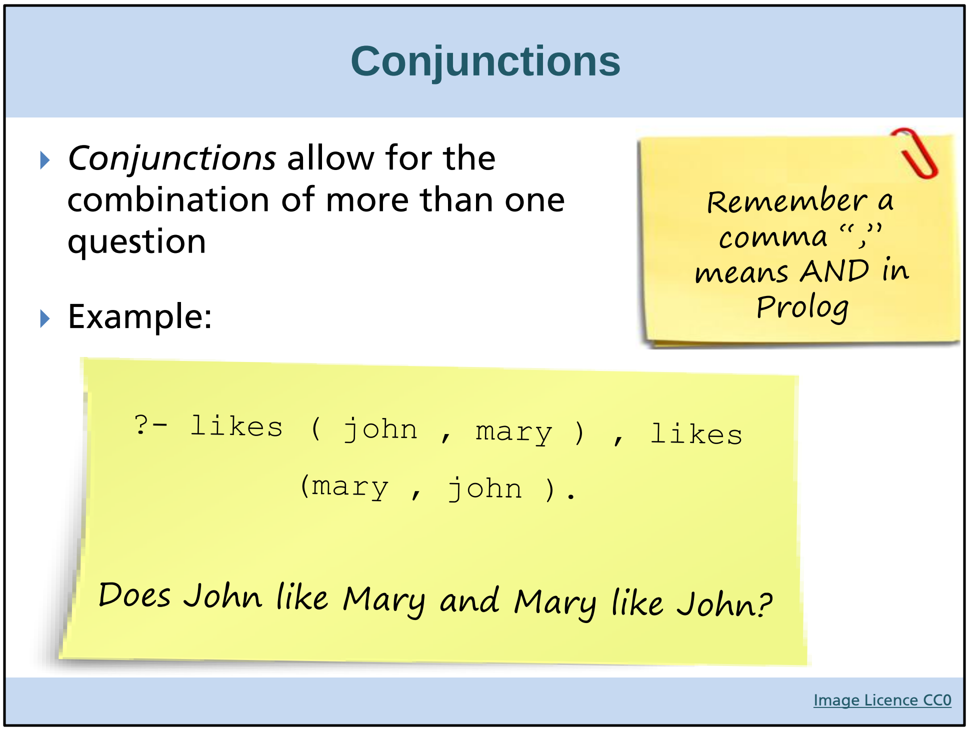


Using Variables In Prolog

Using Variables

- Using variables makes the process more powerful
- Still limited to basic pattern matching
- Need to be able to construct more complex questions

We now are able to ask more complex questions, but they are still fairly limited. We are still just matching our questions to a fact in the database. It would be useful to be able to ask more complex questions.



Conjunctions

- *Conjunctions* allow for the combination of more than one question
- Example:

Remember a comma "," means AND in Prolog

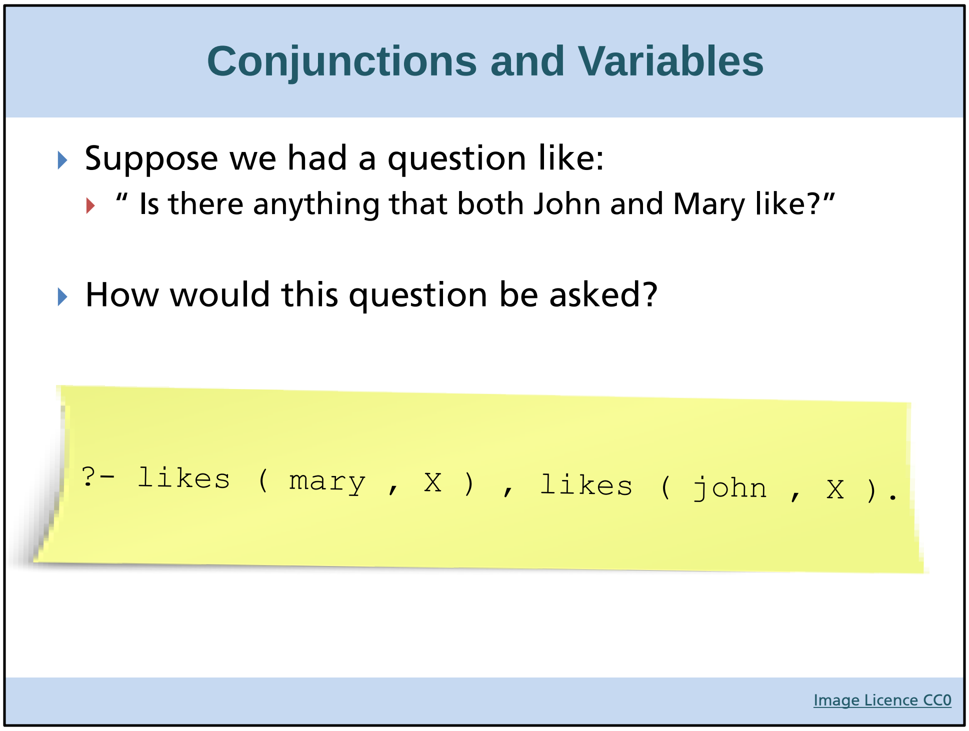
```
?- likes ( john , mary ) , likes
        (mary , john ).
```

Does John like Mary and Mary like John?

Image Licence CC0

The simplest way to ask more complex questions is to use conjunctions. A conjunction is a mathematical term for the logical **AND** operator. So we can ask two questions in one, and look for an answer that satisfies both.
Recall that the comma symbol , is used to represent ”and”. So to ask two questions we simply separate them by a comma.
So the question shown in this slides asks “Does John like Mary AND does Mary like John?”

Such questions are handled in the same way as we’ve seen, except that the second question will only be examined if an answer can be found to the first question. Both of them need to be true for the conjunction to be true, in the same way as logical expressions are evaluated in ‘C’.



Conjunctions and Variables

- Suppose we had a question like:
  - " Is there anything that both John and Mary like?"
- How would this question be asked?

```
?- likes ( mary , X ) , likes ( john , X ).
```

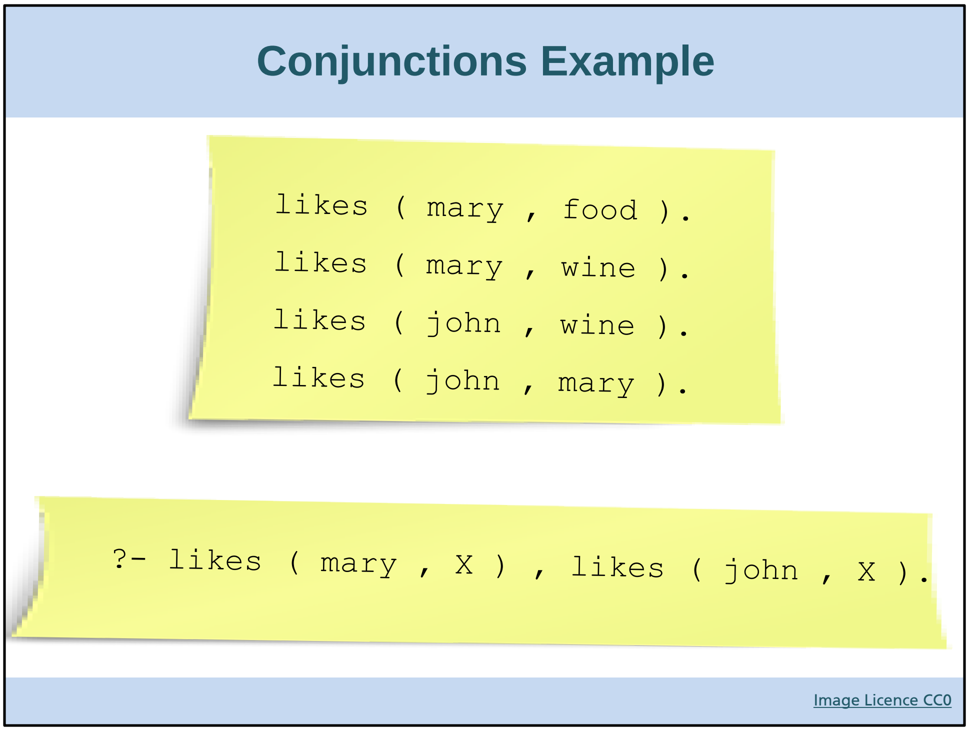
Image Licence CC0

We can extend our conjoined questions to include variables, just like we did with other questions.

We saw before that **likes (mary , X).** asks the question “What does Mary like?”.

By conjoining two similar questions, such as: **?- likes (mary , X) , likes (john , X).**
our question is now “What does Mary like AND what does John like?”.

But as we’ve used the same variable in both questions, we need to find something that they both like, so the question is actually “What do BOTH Mary AND John like?”



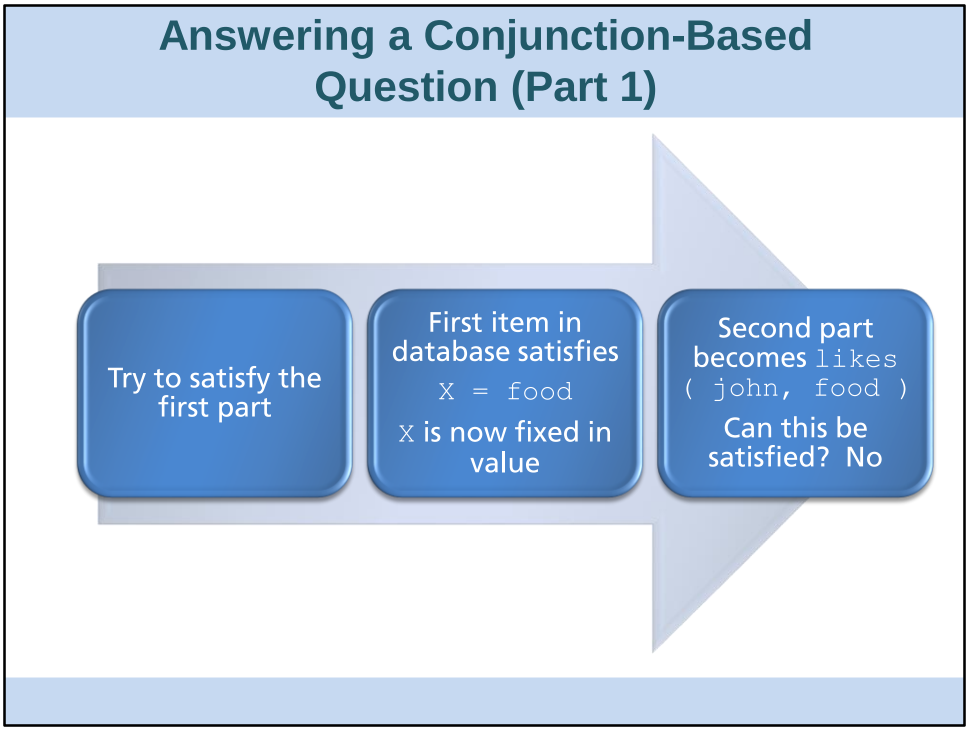
Conjunctions Example

```
likes ( mary , food ).
likes ( mary , wine ).
likes ( john , wine ).
likes ( john , mary ).
```

```
?- likes ( mary , X ) , likes ( john , X ).
```

This slide has a simple database that will be used to show how conjoined questions are answered.

The question we are going to ask is also shown on the slide, and it is the one we met in the last slide, namely “What do Mary and John both like?”.



Answering a Conjunction-Based Question (Part 1)

Try to satisfy the first part

First item in database satisfies `X = food` `X` is now fixed in value

Second part becomes `likes ( john, food )` Can this be satisfied? No

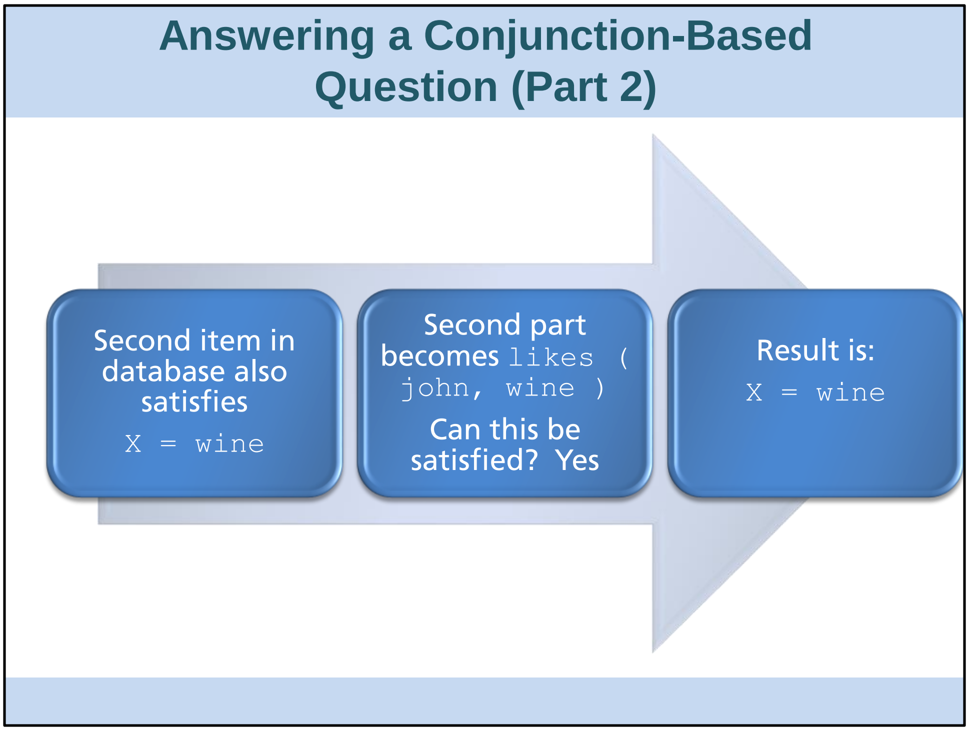
Prolog tries to satisfy the questions individually, so it will try to satisfy the first question and only if this succeeds will it try to satisfy the second question.

So the first question, `likes ( mary , X )` is considered first. This question is dealt with in the way we've already seen. Prolog searches down through the database to find a fact that matches the fact name and the first argument. If it does it instantiates the variable to the second argument. In this case the first fact in our database matches our first question and so X is instantiated to '`food`'.

Prolog now tries to answer the second question. As X has now been instantiated, the second question is now:

```
likes ( john , food )
```

So Prolog searches the database for a match, which it cannot find, so the second question fails.
What Prolog does now is to try and find another answer to the first question.



Answering a Conjunction-Based Question (Part 2)

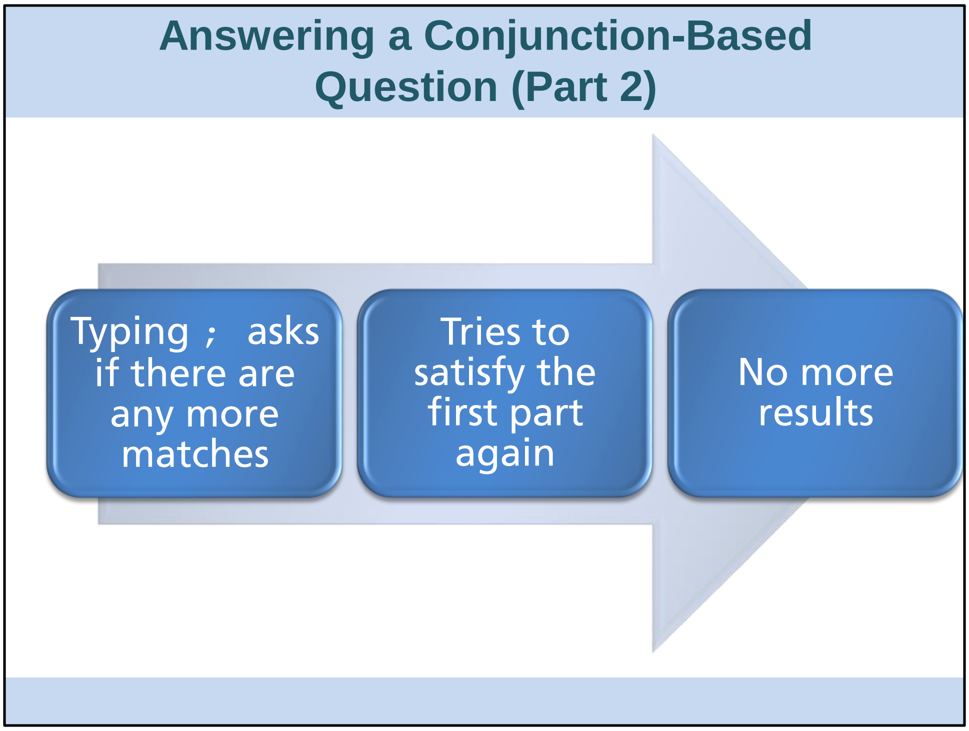
Second item in database also satisfies `X = wine`

Second part becomes `likes ( john, wine )` Can this be satisfied? Yes

Result is: `X = wine`

On searching further down through the database, Prolog now finds that the second fact in the database also matches the first question and so X is now instantiated to ‘`wine`’.

Prolog now tries to answer the second question, which is now `likes ( john , wine )` so Prolog searches the database for this fact and finds it. The second question has now been answered, and so our conjoined question has also been satisfied. Prolog responds in the usual way by displaying the value of the variable that satisfies both questions, namely `X = wine.`



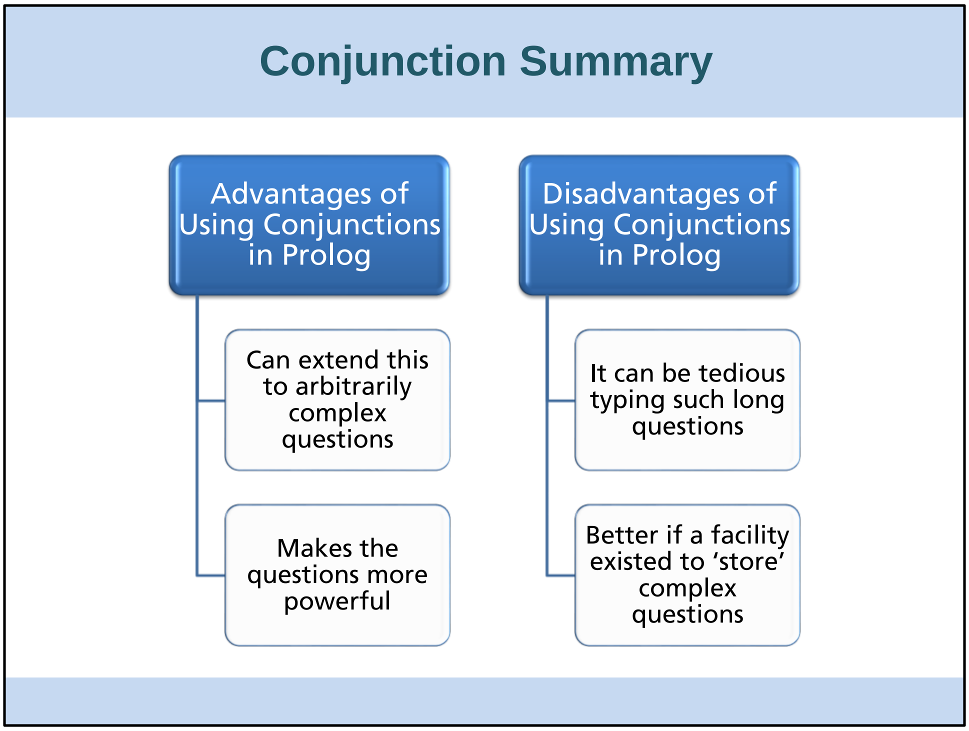
Answering a Conjunction-Based Question (Part 2)

Typing ; asks if there are any more matches

Tries to satisfy the first part again

No more results

As before, we can now ask for more answers by typing ";" Prolog will now try to find other answers to our question but can't so the answer "`no`" is returned.



Conjunction Summary

Advantages of Using Conjunctions in Prolog

- Can extend this to arbitrarily complex questions
- Makes the questions more powerful

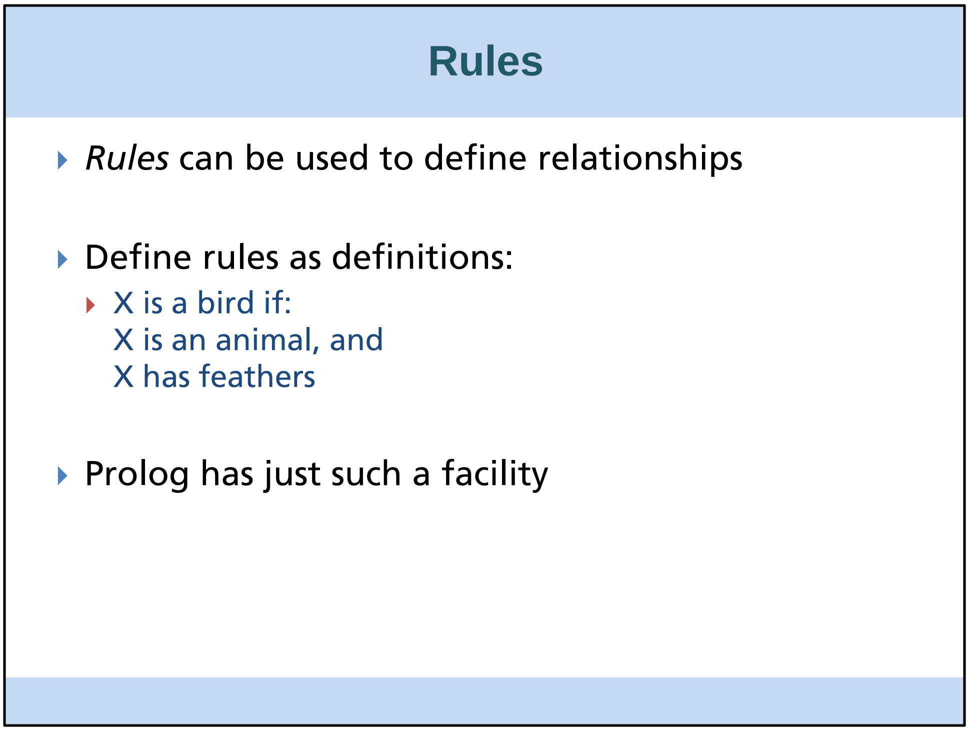
Disadvantages of Using Conjunctions in Prolog

- It can be tedious typing such long questions
- Better if a facility existed to 'store' complex questions

We can now write fairly complex questions as we can extend this idea of conjoining questions to make as complicated a query as we need.

The drawback with this approach, is that it can be tedious having to type long questions, especially if we want to ask the same (or similar) question again.

It would therefore be very useful if we could record such questions and ask these stored questions. Prolog has just such a facility, the rule.



Rules

- *Rules* can be used to define relationships
- Define rules as definitions:
  - X is a bird if:
    X is an animal, and
    X has feathers
- Prolog has just such a facility

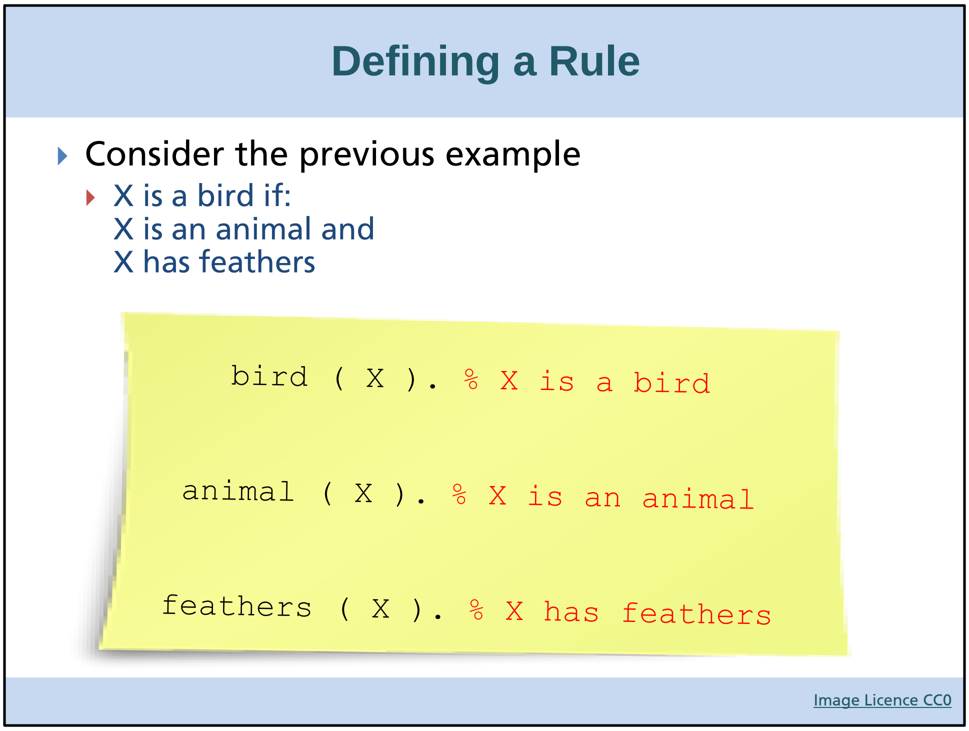
Rules allow us to define relationships between objects, so that we do not need to define everything about an object explicitly.

For example we could define a relationship that birds are animals with feathers, or more formally:

X is a bird if:
    X is an animal, and
    X has feathers

So some entity X is a bird if it is an animal and it has feathers.

Fortunately, Prolog has just such a facility and the rest of this workshop will examine it.



Defining a Rule

- Consider the previous example
  - X is a bird if:
    X is an animal and
    X has feathers

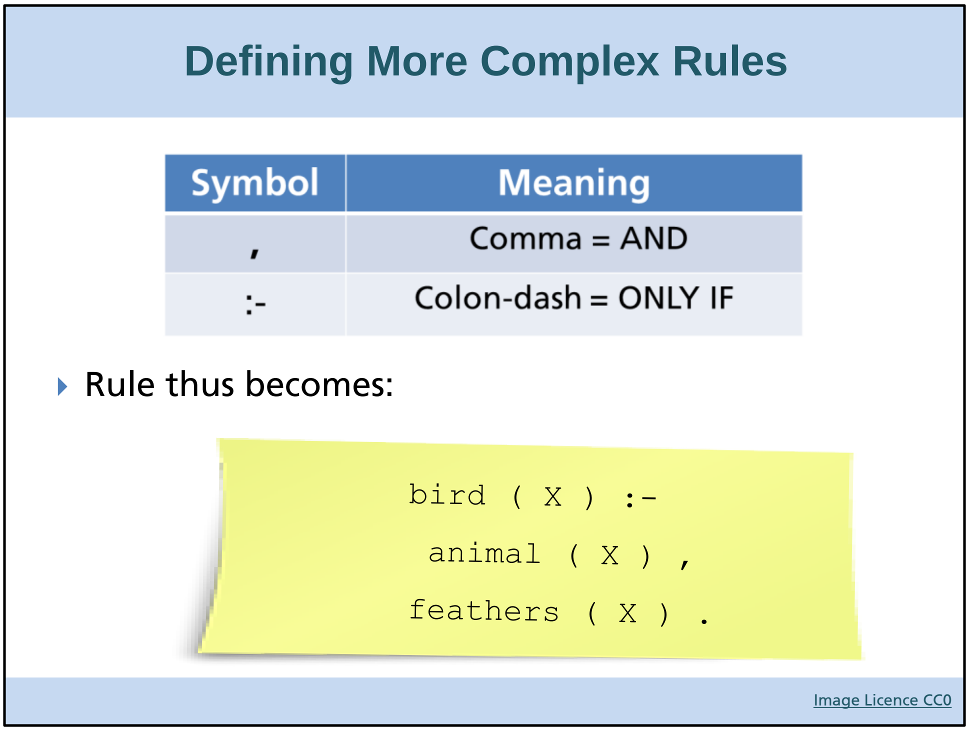
```
bird ( X ). % X is a bird

animal ( X ). % X is an animal

feathers ( X ). % X has feathers
```

Image Licence CC0

Let's focus on the bird example.



Defining More Complex Rules

Symbol	Meaning
,	Comma = AND
:-	Colon-dash = ONLY IF

- Rule thus becomes:

```
bird ( X ) :-
 animal ( X ) ,
feathers ( X ) .
```

Image Licence CC0

We know how to say “and”. We just use a comma.

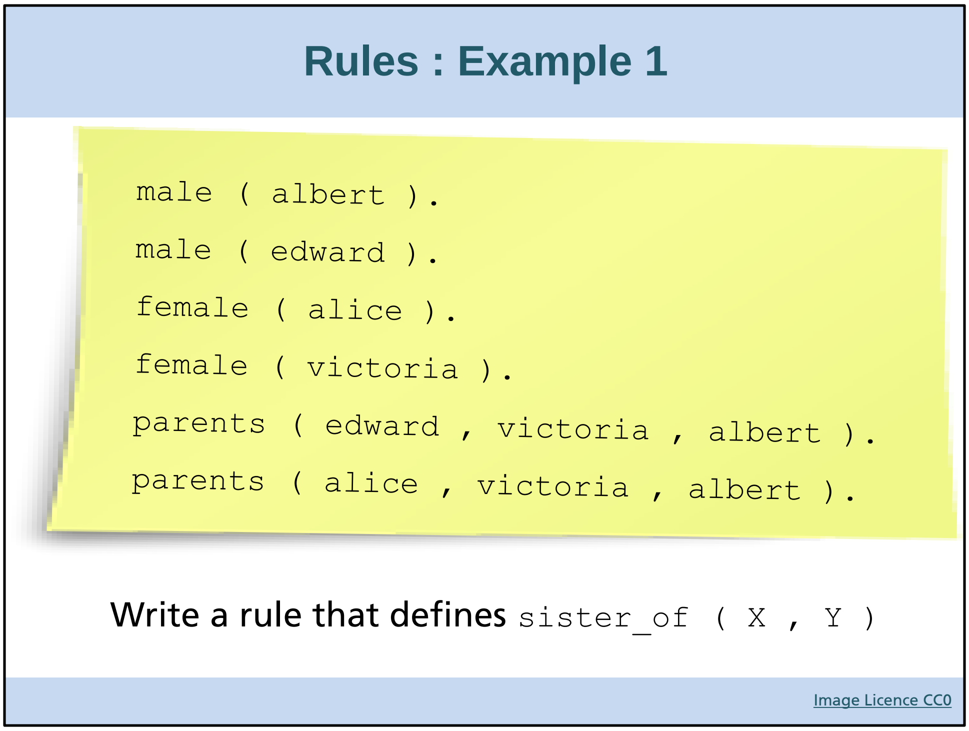
So the only thing we haven’t seen how to represent yet is “if”. To do this we use the symbol “**:-**”

By inserting these Prolog constructs into our rule we get:

```
bird ( X ) :-
     animal ( X ) ,
     feathers ( X ) .
```

This says, X is a bird if X is an animal and X has feathers. This is our first Prolog rule.

Note the full stop at the end of the rule.



Rules : Example 1

```
male ( albert ).
male ( edward ).
female ( alice ).
female ( victoria ).
parents ( edward , victoria , albert ).
parents ( alice , victoria , albert ).
```

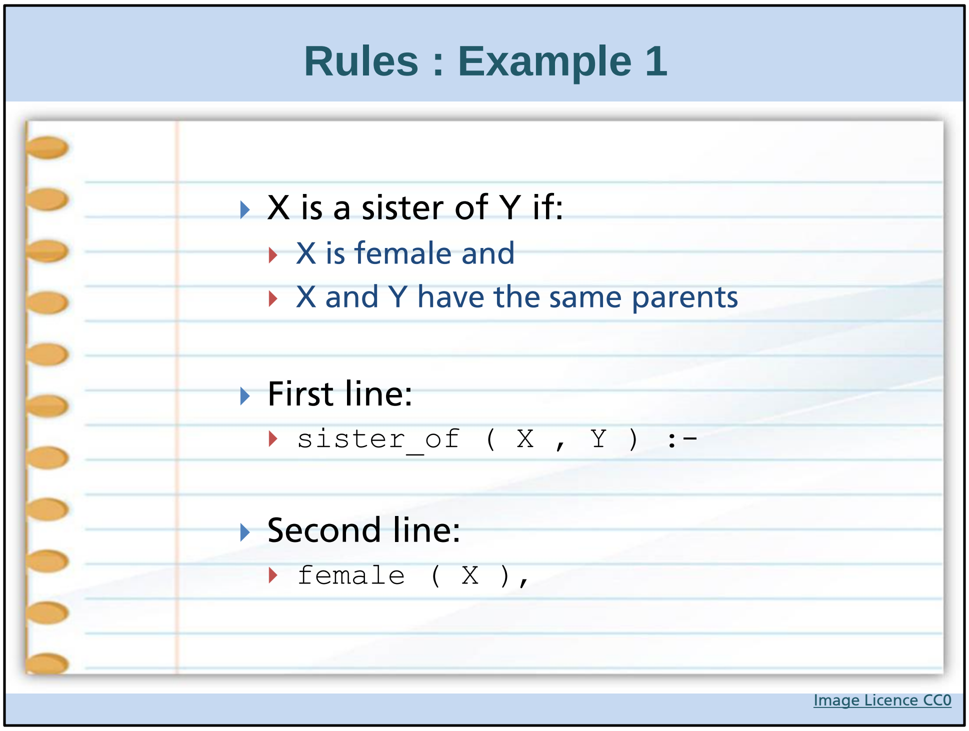
Write a rule that defines `sister_of ( X , Y )`

Image Licence CC0

The rest of the examples will be based upon familial relationships as these should be familiar to you. Make sure that you do not assume that Prolog knows anything about such relationships. For example if we know that Bob's parents are Emily and Bill, and we know that Emily is female, then Prolog cannot deduce that Bob is male. Unless, of course, we explicitly tell it, or write a rule to deduce it.

In the database shown here, we define some basic facts about the gender of some people and who the parents of two of them are. For the parents facts, we assume that the fact `parents(Child,Parent1,Parent2)` means that Parent1 and Parent2 are the parents of Child.

What we want to do is write a rule that allows us to say that **X is the sister of Y**.



Rules : Example 1

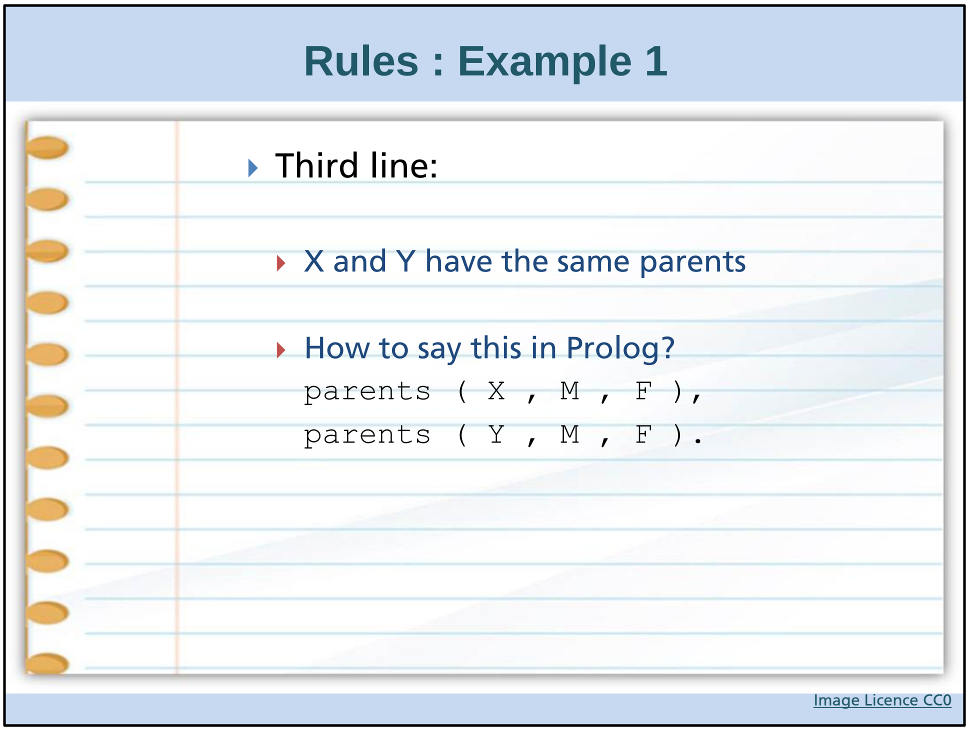
- X is a sister of Y if:
  - X is female and
  - X and Y have the same parents
- First line:
  - `sister_of ( X , Y ) :-`
- Second line:
  - `female ( X ),`

The first thing we need to do is decide what the relationship **sister of** means.

To simplify things we define a sister as someone who is female, and has the same parents as you. More formally we get:

X is a sister of Y if:
X is female and
X and Y have the same parents

To convert this to Prolog, we simply replace each part with its Prolog equivalent. The first two lines are fairly straightforward and are shown in this slide. The third line is trickier and is covered in the next slide.



Rules : Example 1

- Third line:
  - X and Y have the same parents
  - How to say this in Prolog?

```
parents ( X , M , F ),
parents ( Y , M , F ).
```

Image Licence CC0

The final line of our rule is slightly more complex. The others can be defined by a simple factual relationship, but how do we say that two people have the same parents?

We know that we have the parents facts, which tell us who someone's parents' are. All we need to do is find out who the parents of X are and who the parents of Y are and check to see if they are the same. How can we check that they are the same?

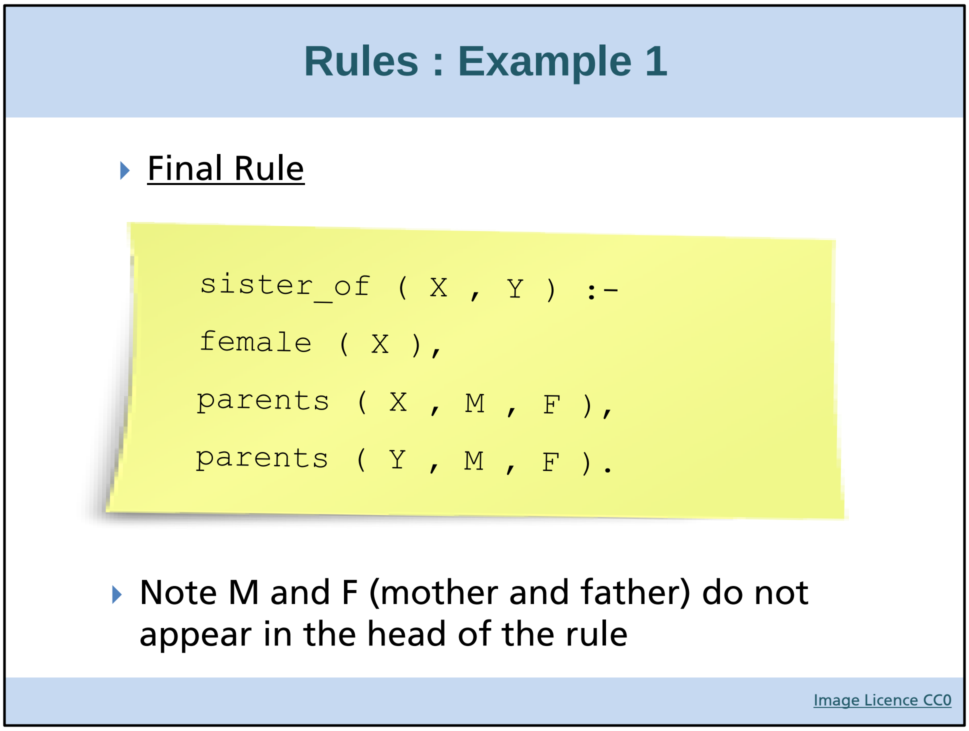
To find the parents of X, we could say:

```
    parents ( X , M , F )
```

so M and F would be the parents of X. When Prolog matches this in the database, M and F will be instantiated to the values that represent the parents of X. To check if Y has the same parents we can just say:

```
    parents ( Y , M , F )
```

As Y, M and F will all have values by this point, this effectively says “Are M and F the parents of Y?”.



Rules : Example 1

- Final Rule

```
sister_of ( X , Y ) :-
female ( X ),
parents ( X , M , F ),
parents ( Y , M , F ).
```

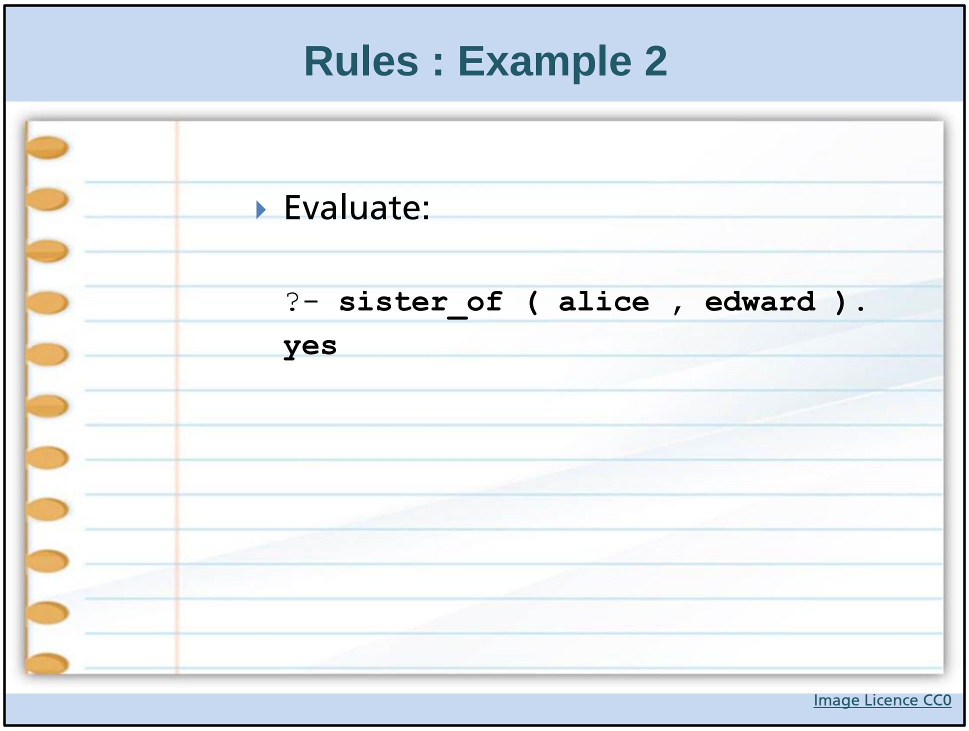
- Note M and F (mother and father) do not appear in the head of the rule

Image Licence CC0

Now that we’ve written all of the rule we can see the final version.

An interesting aspect of this rule, is that the variables `M` and `F` only appear in the body of the rule, not in its head. `M` and `F` are effectively local variables to that rule, and so cease to exist once the rule has finished.

Rules contain *head* and *body* with the following syntax:
head :- body.
Read as “*head is true if body is true*



Rules : Example 2

- Evaluate:

```
?- sister_of ( alice , edward ).
yes
```

Image Licence CC0

Suppose we ask the question:

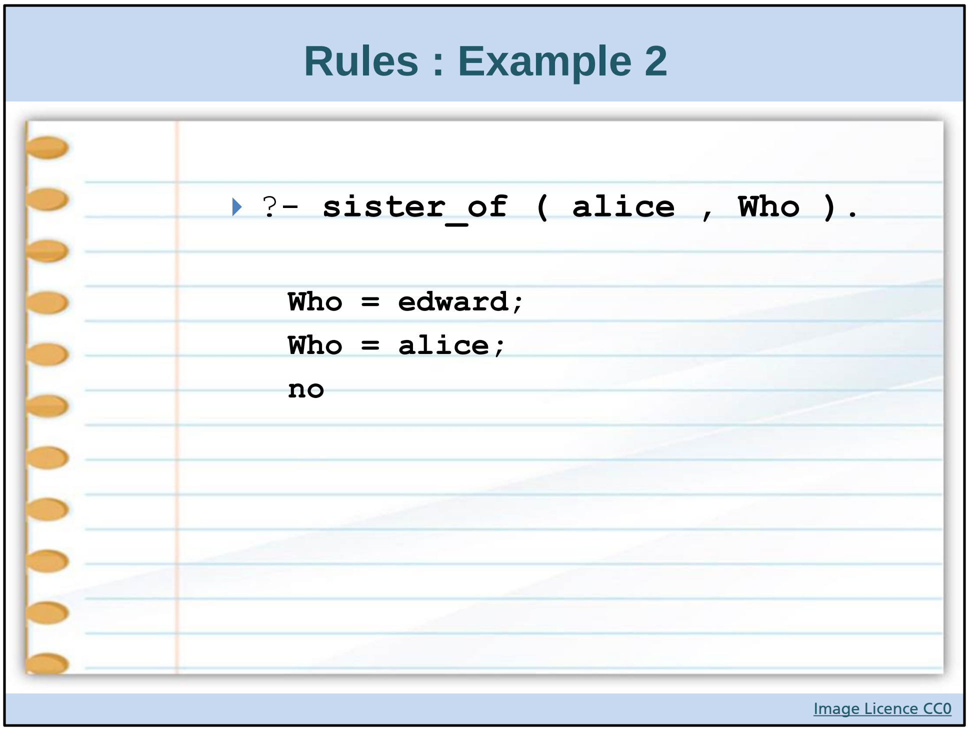
```
     sister_of ( alice , edward ).
```

Prolog starts by trying to find something in the database that matches the name of our question (sister_of). It finds the rule that we have just written, which has two arguments, just like our question, so a match is found.
The heading of the rule (sister_of(X,Y)) has two variables and our question has two constants (alice and edward). As the variables do not currently have any value, they are instantiated to the constants. So upon matching the question to our rule, X now has the value alice and Y now has the value edward. Our rule is now effectively:

```
sister_of( alice , edward ) :-
           female ( alice ),
           parents ( alice , M , F ),
           parents ( edward , M , F).
```

Prolog now tries to satisfy each condition in turn (in exactly the same way as we've already seen). The first condition is shown to be true as the fact female (alice) is in the database. As the first condition is true, Prolog moves onto the second condition. It now tries to find out who Alice's parents are. By searching the database, it finds that they are Victoria and Albert. The variable M now has the value victoria and the variable F now has the value edward.

As a match for the second condition was found, the final condition is now tried. It now reads parents (edward , victoria, albert). This fact is in the database, so the condition is satisfied and the rule has shown that Alice is Edwards sister, so the value yes is returned.



Rules : Example 2

```
?- sister_of ( alice , Who ).

  Who = edward;
  Who = alice;
  no
```

Image Licence CC0

In this question, a variable is used as well as a constant, so we are effectively asking, who is alice the sister of.

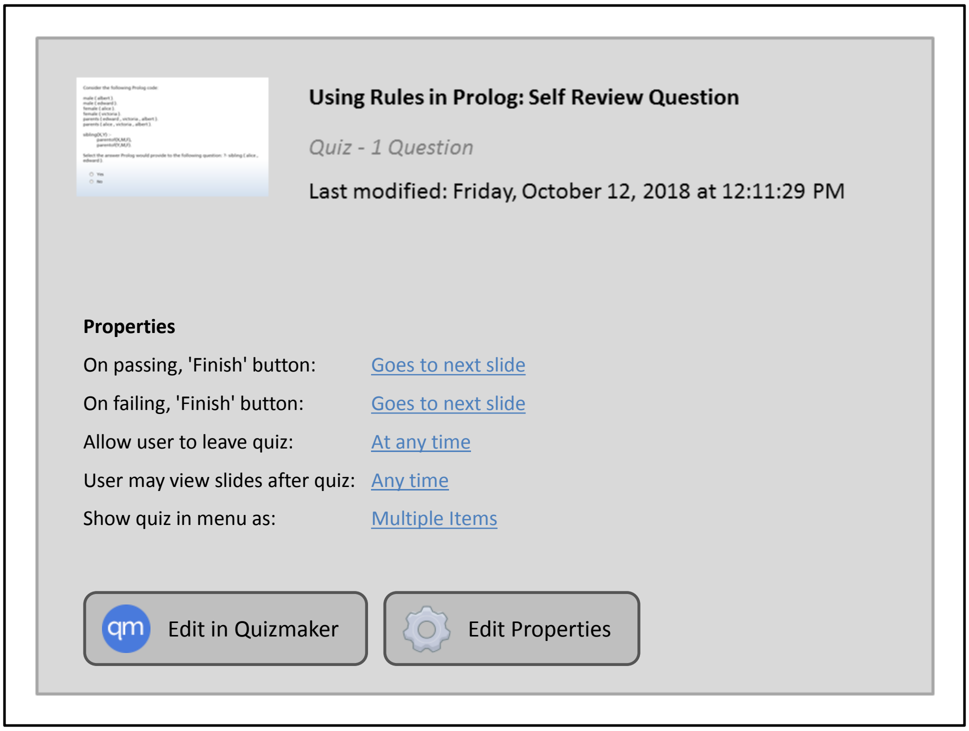
Prolog answers this question in a similar way to the last one. When it matches the question with the rule, the variable X is instantiated to `alice`, whilst Y is left uninstantiated. Y is effectively being used as a variable argument. This is a very powerful feature of Prolog, arguments can be used as both input and output parameters, we'll cover more of this later.

The first two conditions of the rule fire in exactly the same was as before (alice is still female and victoria and albert are still her parents). This time, however, the last condition is `parents ( Y , victoria , albert ),` effectively who has Victoria and Albert as parents.

Prolog searches through the database to try to find a match. It finds that the first parent's relation matches with the value of `edward` for Y. All of the conditions have now been met and edward is returned, giving `Who = edward` as the result. If we now type a ";", to ask for more answers, Prolog attempts to find another solution.

It does this by trying to resatisfy the last condition (if it can't it works back through the other conditions trying to resatisfy them). In this case it does find another match, the second instance of the fact parents, with a value for Y of `alice`.

So Prolog has deduced that Alice is her own sister. Clearly there is a problem with our rule. However this can be overcome by checking that X and Y are different. You'll see how to do this when you attempt the lab exercises.



Consider the following Prolog code:

```
male ( albert ).
male ( edward ).
female ( alice ).
female ( victoria ).
parents ( edward , victoria , albert ).
parents ( alice , victoria , albert ).

sibling(X,Y) :-
        parentof(X,M,F),
        parentof(Y,M,F).
```

Select the answer Prolog would provide to the following question: ?- sibling (alice , edward).

○ Yes
○ No

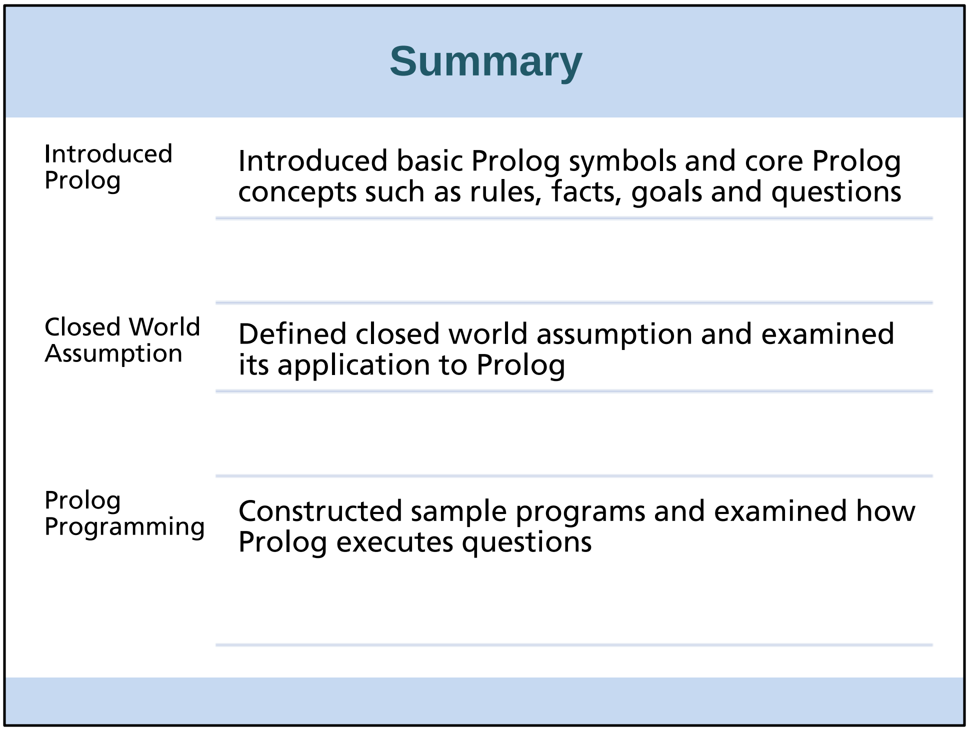
Using Rules in Prolog: Self Review Question

Quiz - 1 Question

Last modified: Friday, October 12, 2018 at 12:11:29 PM

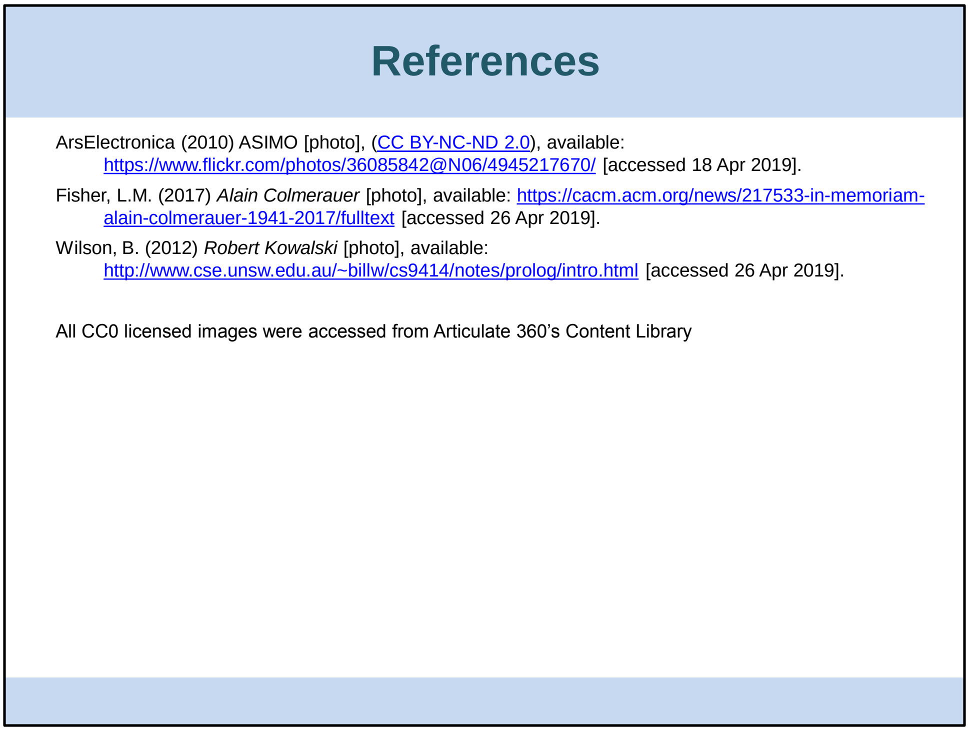
Properties

On passing, 'Finish' button:	Goes to next slide
On failing, 'Finish' button:	Goes to next slide
Allow user to leave quiz:	At any time
User may view slides after quiz:	Any time
Show quiz in menu as:	Multiple Items



Summary

Introduced Prolog	Introduced basic Prolog symbols and core Prolog concepts such as rules, facts, goals and questions
Closed World Assumption	Defined closed world assumption and examined its application to Prolog
Prolog Programming	Constructed sample programs and examined how Prolog executes questions



References

ArsElectronica (2010) ASIMO [photo], (CC BY-NC-ND 2.0), available: https://www.flickr.com/photos/36085842@N06/4945217670/ [accessed 18 Apr 2019].

Fisher, L.M. (2017) *Alain Colmerauer* [photo], available: https://cacm.acm.org/news/217533-in-memoriam-alain-colmerauer-1941-2017/fulltext [accessed 26 Apr 2019].

Wilson, B. (2012) *Robert Kowalski* [photo], available: http://www.cse.unsw.edu.au/~billw/cs9414/notes/prolog/intro.html [accessed 26 Apr 2019].

All CC0 licensed images were accessed from Articulate 360’s Content Library