Prolog: Recursion and Lists

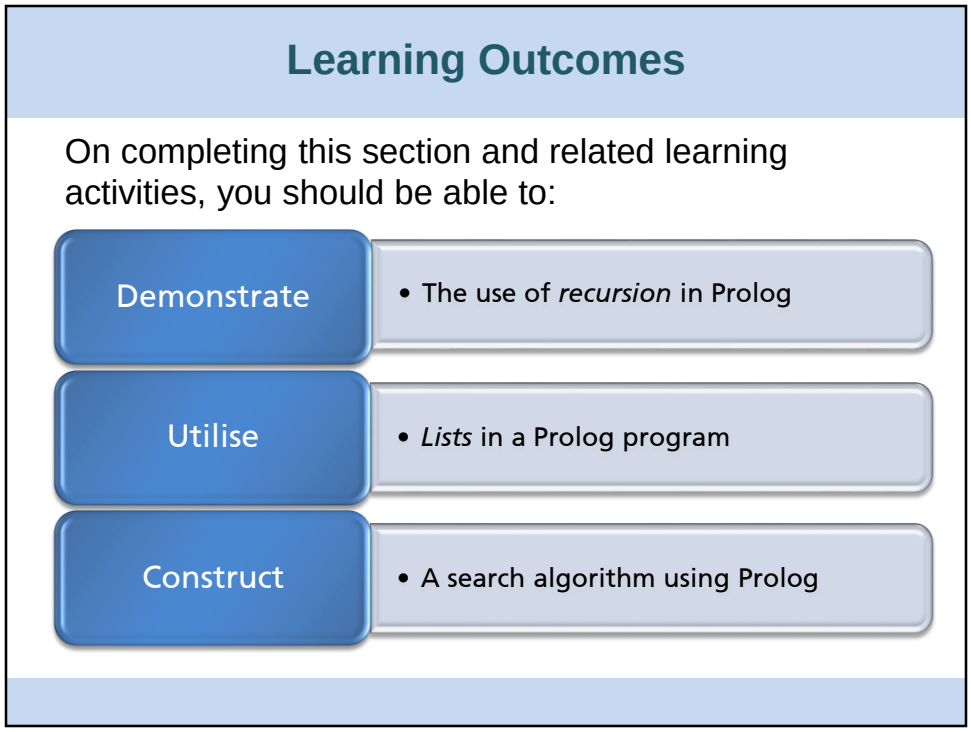
Workshop 3 Section 2

CT621 Artificial Intelligence - MScSED

In Section 1 of this Workshop, we learned about the following constructs, which are the basic components of Prolog:

- Facts
- Constants
- Variables
- Questions
- Rules

Before proceeding, you should be familiar with using these to make Prolog perform a range of tasks like those discussed in Part 1. In this part of the workshop, we'll be moving onto other essential Prolog concepts.



In Workshop 3 section 2, we'll meet two new topics, *recursion* and *lists* and we will be applying Prolog to implementing a search algorithm.
Recursion works in a similar fashion to the C Programming Language which you should already be familiar with.

The main difference between Prolog and C is that in C arguments only have one function (either input or output), whereas in Prolog arguments can be either input or output. This means that we can write a single piece of code that can perform a range of tasks.

Lists are the main data type in Prolog. They are similar to arrays in C, but they are more flexible and more powerful. One of the reasons for this is that there are no types in Prolog, so a list is just a sequence of elements.

Rules and Variables

- Consider the rule:
  - A person X may date Y if:
    Y is a person and
    X likes Y

```
may_date(X, Y) :-
likes(X, Y), person(Y).
```


Suppose we wanted to write a very simplistic dating program in Prolog code that defined the following:

“A person may date someone if that someone is a person and if they like the person”

We can code this as shown in the slide. Now consider how we can represent `person` and `likes`.

Using Rules and Variables

- `person(X)` and `likes(Y, Z)` can be:
  - Facts, like we've already seen
  - Rules, again like we've seen (rules can also be recursive, as we will see)
  - Both:
    a mixture of facts and rules

We have three possibilities, we can either represent them as **facts**, so we define who is a person and who likes what.

We can also represent them as rules, so we could write **rules** that would define either or both of these relationships.

The final possibility is to use both rules AND facts. So we can define some facts about who is a person, and we can also write rules that define who we may date. This is one of the powerful aspects of Prolog, facts and rules are interchangeable so we can mix facts and rules.

Rules and Variables: Sample Problem (Part 1)

- Consider the following fact and rules:

```
person(emma).
person(mary).
likes(emma, food).
likes(mary, wine).
likes(john, X ) :- likes(X ,wine).
may_date(X, Y) :- likes(X, Y), person(Y).
```


To illustrate this consider the example shown in this slide.

We've defined a single fact for `may_date`, however we've defined both facts and a rule for the relationship `likes`. The rule for `likes` is also recursive, that is it calls itself.

Note that the convention (when you have facts and rules with the same name) is to put the facts first and the rules second. This is particularly important when the rule is recursive, as if the convention is not followed, the programme may get stuck in an infinite recursion (like you encountered in C).

Rules and Variables: Sample Problem (Part 2)

- Evaluate the following question:

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

```
may_date(john, X ) :- likes ( john , X ),
                      person(X).
```


To evaluate our query, the first thing Prolog does is to search the database to try and find something that matches `may_date`. In this case it finds the last entry in our database. Prolog now tries to match our query to the head of the rule. Our question has two arguments and the rule also has two arguments, so this also matches.

The first argument in our question is a constant, so the first argument of our rule is instantiated to `john`. The second argument is a variable, so Prolog is going to try and find a value for this variable. Our rule is now effectively:

```
may_date( john , X) :- likes ( john , X ), person(X) .
```

So to satisfy our rule Prolog must show that `John` likes X and that `X` is a person . Our goal is now:

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

The only thing in our database that matches this goal, is the `likes` rule, which is true if we can find something that likes `wine`. This eventually succeeds by finding that `Mary` likes `wine`, so `X` is instantiated to `mary`. However, our second clause must also be matched. We search for `person(mary)`, which is found in the database. Therefore, the answer returned to our original question is:

```
X = mary
```

Self Review Question: Rules and Variables

Quiz - 1 Question

Last modified: Monday, October 15, 2018 at 10:36:29 AM

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

Edit in Quizmaker Edit Properties

Recursive Question

- Let's examine the Prolog rules that define the relationship **ancestor**
  - Assume that the only facts are of the form:

```
parent(X, Y). % X is a parent of Y
```

Now we are going to examine the use of recursion in Prolog to define the relationship **ancestor**. We will define this relationship using only the **parent** fact.

Answer to Recursive Question

- One solution is to consider an ancestor as either your parents, or an ancestor of your parents. This gives the following result:

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



```
ancestor ( X , Y ) :- parent ( X , Y ). % your
parents are your ancestors

ancestor ( X , Y ) :- parent ( Z , Y ) , % find
one of your parents
                      ancestor ( X , Z ). % now find
one of their ancestors
```

This simple piece of code will work for any database, as long as a path exists between you and your ancestor. The recursion works by repeatedly looking back one generation, until you find the person you are looking for.

Factorial Problem

- Let's examine the use of recursion to solve the factorial problem in Prolog
  - In mathematical notation the factorial of a non-negative number **n** is denoted as **n!** and is the product of all positive integers less than or equal to **n**

We will now look at using recursion in Prolog to solve the factorial problem.

The factorial of a non-negative number `n` is denoted as n where `n! = 1*2*3*...*n.`

Therefore, the factorial of 4! is 24 (1*2*3*4).

Calculating Factorial in Prolog

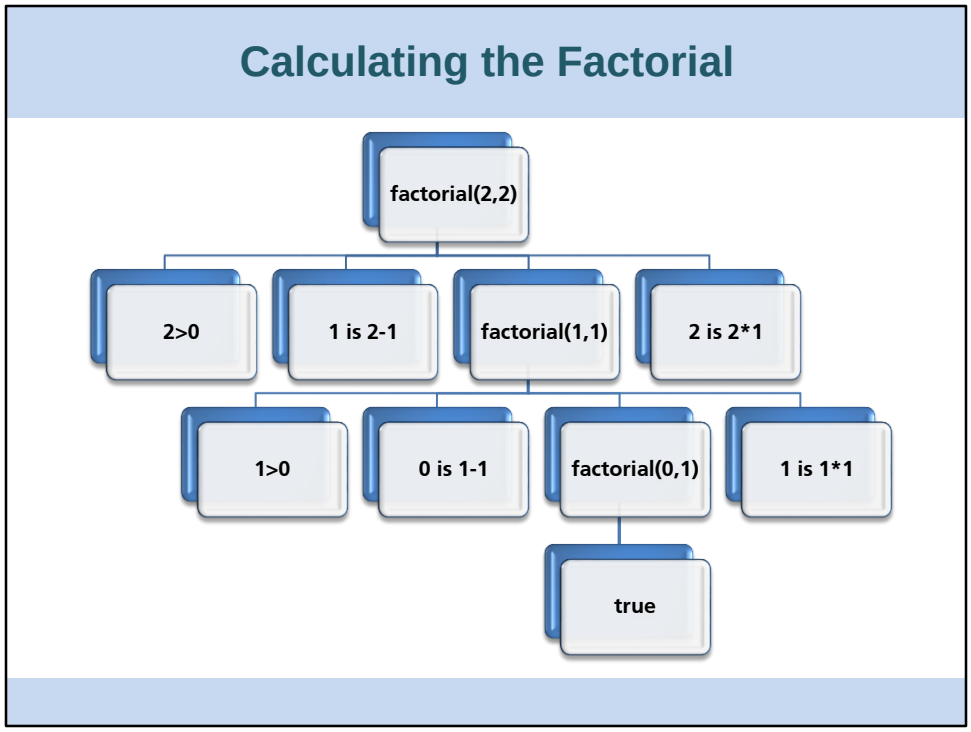
- An easy way of solving the factorial problem is to do it recursively
- Here is an example of how it can be done :

```
factorial(0,1).

factorial(A,B) :-  A > 0, C is A-1,
            factorial(C,D),
            B is A*D.
```


The first line in this program can be read as saying that "the factorial of 0 is 1“ .

The second line declares that the factorial of `A` is `B` if:

1. `A>0` and
2. `C` is `A-1` and
3. The factorial of `C` is `D` and
4. `B` is `A*D`.



The tree provides a depiction of how the program would solve for the question factorial**(2,X)**. All of the leaves of the tree are true by evaluation and the lowest link in the tree corresponds to the very first clause of the program for factorial**(0,1)**.

?- factorial(2,X).
X = 2
Yes

In this example Prolog will try to calculate **2!**, then **1!** and **0!**. The result will be **0!*1*2**. Prolog will respond **2**.

Lists in Prolog

- Sequence of elements
- Elements can be atoms - constants
- Elements can be variables
  - mostly used within rules
  - allows 'gaps' to be left in list
- Elements can be lists
  - allows construction of complex structures

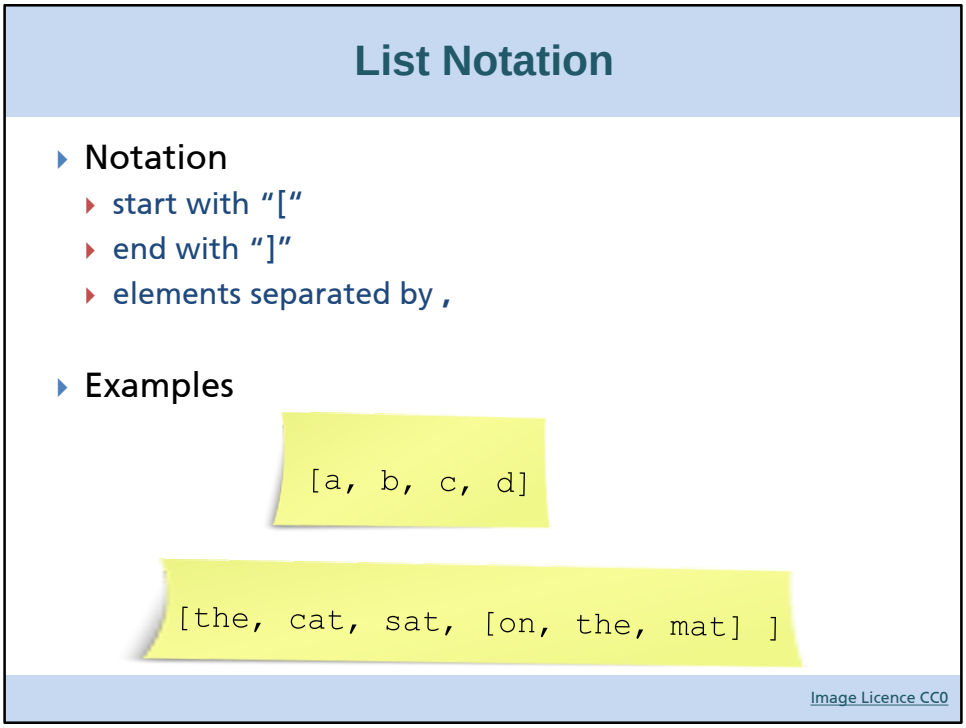
We now move on to examining lists. Recursion is fundamental when it comes to processing lists, so we'll also extend our examination of recursion while we look at lists.

Lists are similar to arrays in C, in that they are an ordered sequence of elements. However, unlike C, the elements are not of a specific type. In fact the elements of a list can be one of three things:

•**Constants** – e.g. a list of names

•**Variables** – Prolog variables can be used within rules to leave holes, that it can later fill

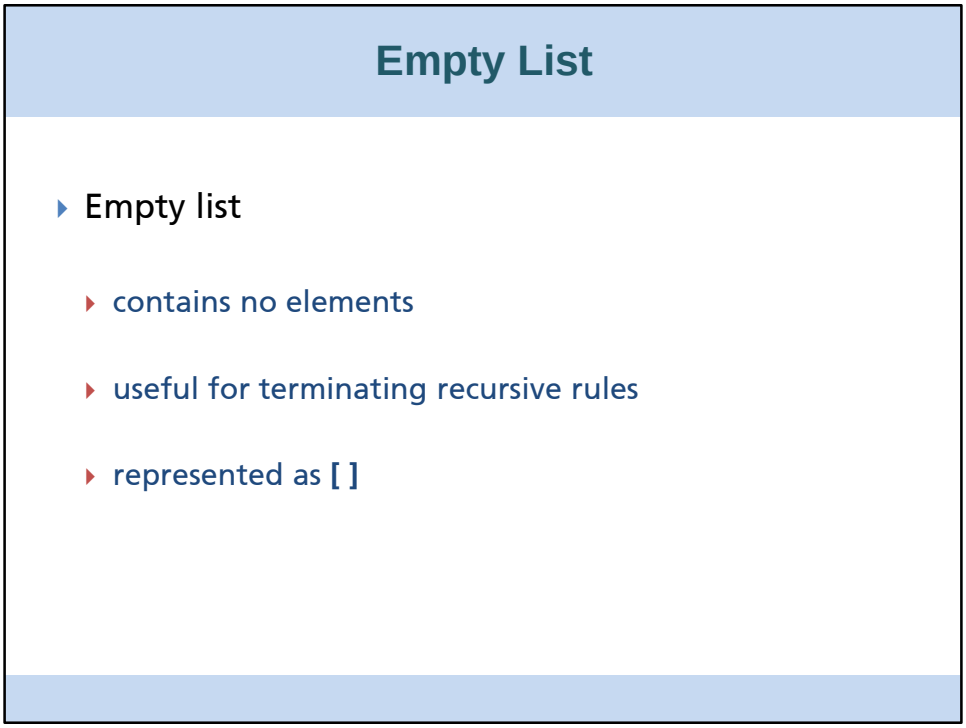
•**Lists** – perhaps surprisingly, the elements can also themselves be lists – we'll see more of this later



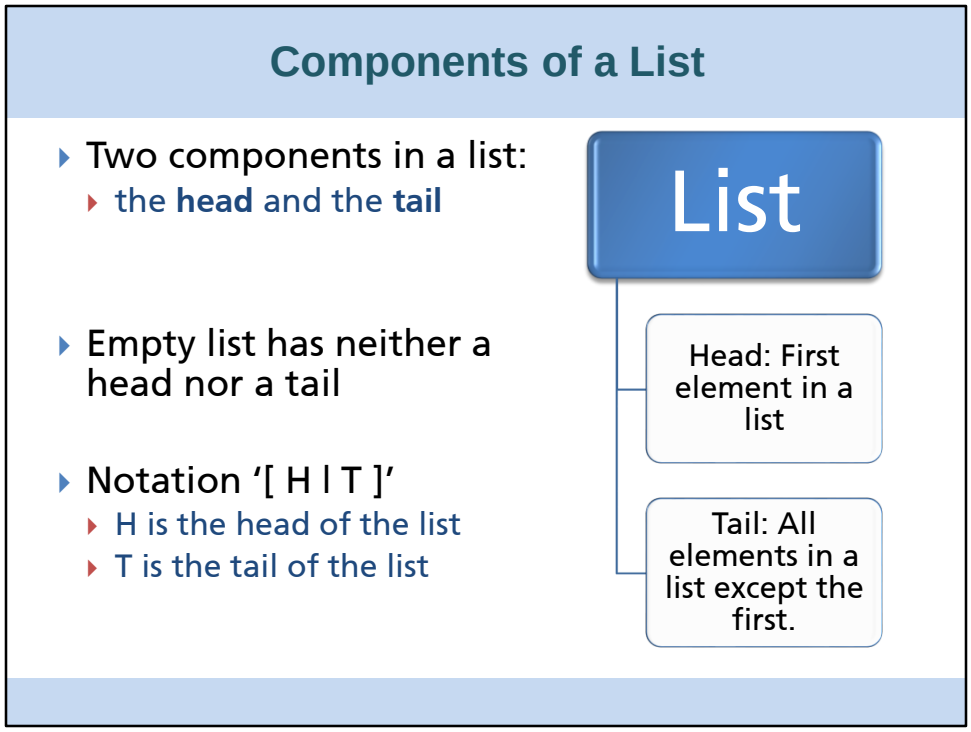
How are lists represented in Prolog?

Each list starts with an open square bracket "[" and ends with a close square bracket "]". The body of the list is a sequence of elements separated by commas ",".

The two example lists shown here both have 4 elements, the first one just has four constant elements (the letters **a**..**d**). The second list also has 4 elements, however this time the first three are constants **the**, **cat**, **sat** , whilst the fourth is a list **[on, the, mat]** .



A special case of a list is the empty list, or a list that contains no elements. The empty list is particularly useful for terminating recursive rules on lists, e.g. keep removing and processing elements in a list until there are no more elements (the empty list). It is represented simply by closed square brackets **[]**, quite literally a list that contains no elements.



We've seen how we can represent and define lists - we now need to see how we can use them.
Unlike arrays in C, each element cannot be directly accessed. Only the first element in a list can be simply accessed (the others require more complex code). To process a list, Prolog allows us to split the list into two parts, the **head** and the **tail**.

The head of a list is just the first element of a list, whist the tail of a list is the rest of the list (excluding the head).
In order to have a head and a tail, a list must have at least one element, so by definition the empty list has neither a head nor a tail.

The symbol "|" is used to split a list into a head and a tail. By convention the variable **H** is used to represent the head and the variable **T** is used to represent the tail. Of course you can use any suitable variable names.
So **[H | T]** splits a list into a head (H) and a tail (T).

Components of a List: Example

- Examples

LIST	HEAD	TAIL
[a, b, c]	a	[b,c]
[]	(none)	(none)
[[the, cat], sat]	[the, cat]	[sat]
[the, [cat, sat]]	the	[[cat, sat]]
[one]	one	[]

We'll use these examples to illustrate the concepts of a head and a tail of a list.

In the first example **[a , b , c]** the head of the list is the constant **a** and the tail of the list is the list **[b , c]**.

In the second example **[]**, the empty list has neither a head nor a tail.

In the third example, **[[the , cat] , sat]**, the head of the list is the list **[the , cat]** (as this is the first element) and the tail of the list is the list **[sat]**.

In the fourth example, **[the , [cat , sat]]**, the head of the list is the constant **the** and the tail of the list is the list **[[cat , sat]]**. Notice that the tail in this case is a list which has a single element, which is itself a list.

In the last example, **[one]**, the head of the list is the constant **one** and the tail of the list is the empty list **[]**. **Note that the tail of a list is always a list, even if it contains no elements.**

Membership of a List

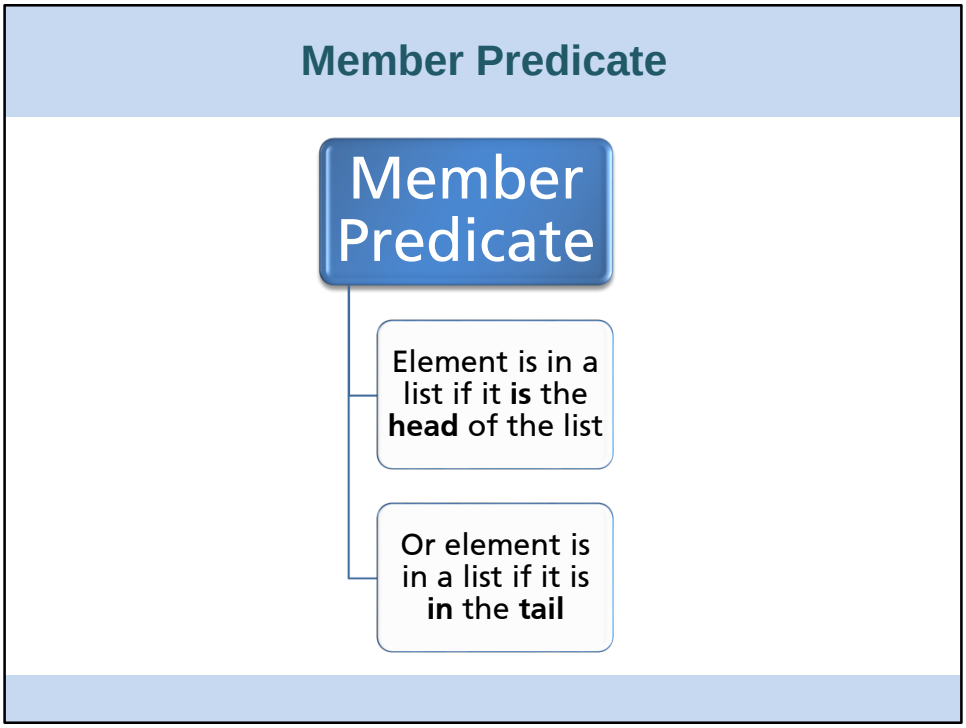
- Is given element a member of the list?
- For example:

```
member ( a, [ a, b, c, d ] ).
```

```
member ( bob, [ allan, bob, dave ] ).
```

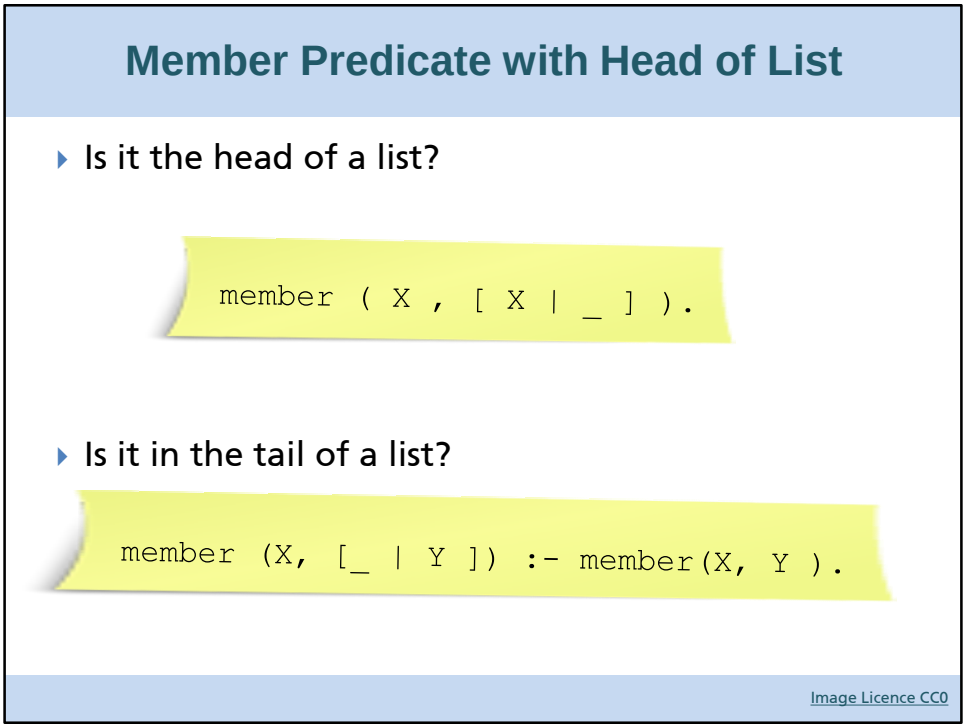

A common thing we may want to do, is to see if a particular element is in a list, or to see if an element is a member of a list.

It would be useful to ask questions like “Is bob a member of the list `[ allan , bob , dave ]` ?”



To determine if an element is a member of a list or not, there are two possible cases.

An element is a member of a list if either the element is the head of the list, or it is a member of the tail of the list.



To check whether the element we are looking for is the head of the list, we just need to compare the element with the head of the list (this is the base case of our recursion), so:

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

Says X is a member of the list if it is the head of the list. The underscore "_" is used to represent something, when we don't care what value the component has. In this case, we are only interested in the head of the list and not the tail, so we can use the underscore to highlight this. Unlike variables, each instance of "_" in a rule is NOT assumed to represent the same value. The underscore is useful, as it allows you to focus on the important aspects of the code and not get bogged down in meaningless variables.

How do we check whether the element is in the tail of our list?

To check whether the element is in the tail of our list, we first need to extract the tail and then recursively call the member predicate:

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

This code breaks our list into a head (_) and a tail `(Y)`. X is a member of the original list if it is a member of `Y`.
This code repeatedly compares the element we are looking for with the head of the list. If the head is what we are looking for, the first predicate matches. Otherwise we delete the head and carry on searching the rest of the list.
If the element we are looking for is not in the list, we will eventually reach the point where we have an empty list. As both of our predicates break the list into a head and a tail, they will both fail (as the empty list does not have a head or a tail).

It is important with recursion that the base case comes first, so if you use this code, it must be given in the order shown here.

Member Predicate: Example 1

- Evaluate the following:

```
member ( a, [ a, b, c, d ] ).
```


Over the next few slides we will examine a number of examples that use lists.

The first example, `member( a, [ a, b, c, d ] ).` asks is 'a' a member of the list `[ a , b , c , d ]`?

Prolog searches the database for something that matches the member predicate. The first thing that it finds is the base case:

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

This has two arguments, like our question, and so it matches. The first argument in our question is a constant and so the variable `X` is instantiated to '`a`', . The second argument is a list, which is split into a head and a tail. The head of the list has the same variable name as the value we are looking for. So for this clause to match, the first element of the list has to be the value we are looking for. In this case, 'a' is the head of the list, so the values match and Prolog concludes that '`a`' is a member of the list.

Let us now look at the second example.

Member Predicate: Example 2

- Evaluate the following:

```
member (bob, [allan, bob, dave ] ).
```


Looking at the second example: `member ( bob, [ allan, bob, dave ] ).`

Prolog again tries to match this question with the base case. In this case, the value we are looking for is 'bob' and the head of the list is 'allan', which are not the same, so the base case does not match. Prolog then looks for other matches and finds the second member predicate:

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

`X` is instantiated to `bob` and `Y` is instantiated to the tail of our list `([ bob , dave ]).` As Prolog always works from top to bottom of the database, the only way we could have reached this point is if the value we are looking for is not the head of the list. We therefore no longer care what value the head is so we use '_' for it.
The rule now effectively reads:

```
member ( bob , [ _ | [ bob , dave ] ] ) :- member ( bob , [ bob , dave ] ).
```

This matches if bob is a member of `[ bob , dave ]` which it is, so Prolog finds that `bob` is in the list.
Finally, let's look at the last example …

Member Predicate: Example 3

- Evaluate the following:

```
member ( X, [ a, b, c, d] ).
```


The last example is: `member ( X, [ a, b, c, d] ).`

This is an example of the flexibility of Prolog. The code we have written was designed to check whether an element is a member of a list or not. What we have done now is ask the question "What element is a member of `[ a , b , c , d]`?". So now the first argument is now used to return a value, rather than to input a value.

So how does Prolog handle this? It works in much the same way., It searches the database for a predicate called member and finds the base case. As this has the same arity (2) as the question, a partial match is achieved. Prolog now tries to use this to answer our question. As the first argument is a variable, it doesn't change anything in our predicate, but the second argument is broken into a head and a tail, with `X` representing the head. `X` is therefore instantiated to '`a`'. As `X` is also the first argument, the value '`a`' is returned as an answer to our question, so we get:

```
X = a
```

If we now ask for more answers, by typing ";" Prolog tries to find other answers. It moves onto the second predicate member, which removes the head of the list and effectively makes our question:

```
member  ( X , [ b , c , d ]).
```

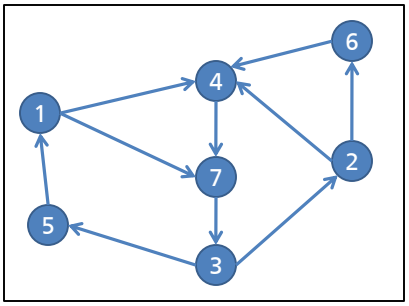
This will again match the case base. If we repeatedly ask for more answers, we get:

```
:- member ( X , [ a , b , c , d ]).

X = a;

X = b;

X = c;

X = d;

No
```

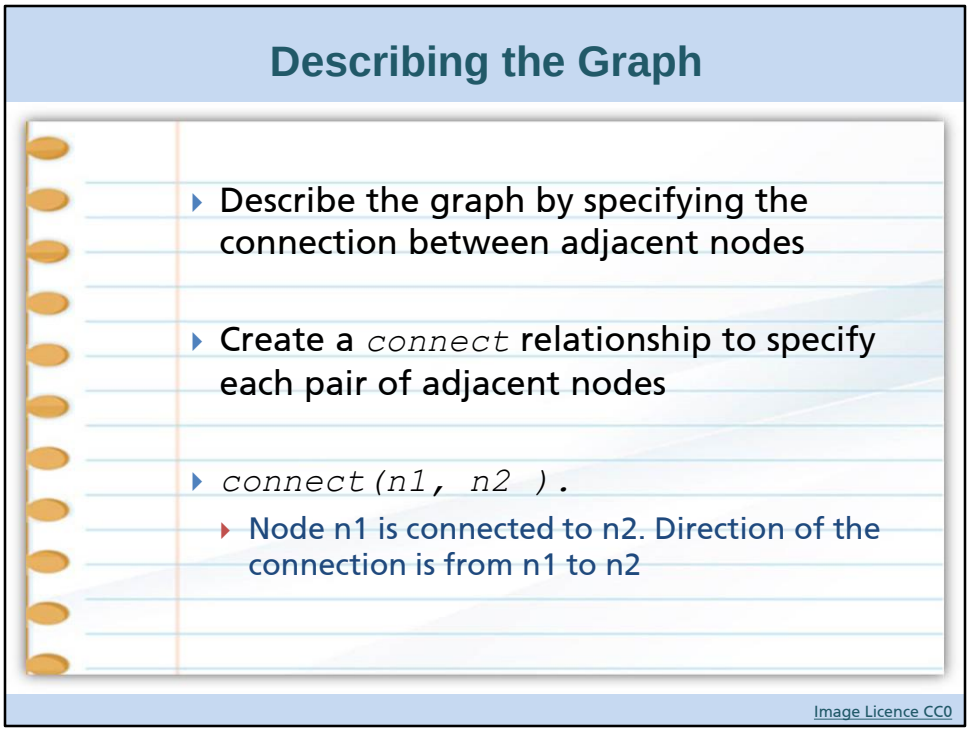
This has returned each of the elements of the list in sequence. So by using the same piece of code we have been able to ask different questions.

Search Algorithm Problem

- The remaining slides will focus on constructing a program to find a path from a source node to a destination node in a directed graph
  - Program should avoid cycles. Therefore, it should never visit the same node twice



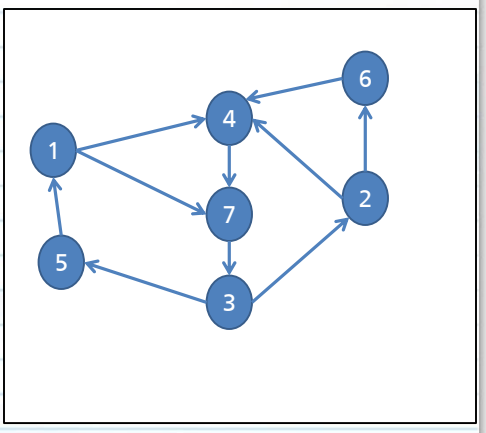
The remaining slides in this section will illustrate how we can utilise both lists and recursion in Prolog to build a relatively simple search algorithm.



The first step in solving the problem is to describe the directed graph by specifying the connection between adjacent nodes. This can be done by creating a relationship that we will call `connect` which specifies the directed connection between two nodes in the graph.
The directed graph that we will describe consists of 7 nodes and 10 individual connections.

Defining Connections

```
1) connect(1,4).
2) connect(1,7).
3) connect(6,4).
4) connect(4,7).
5) connect(2,4).
6) connect(7,3).
7) connect(3,2).
8) connect(2,6).
9) connect(3,5).
10) connect(5,1).
```



We can describe the graph shown in the slide with the use of the connection relationship. The slide contains a `connect` relationship for each of the ten connections within the graph. For example, the relationship `connect(1,4)` specifies that the directed graph has a link between node `1` and node `4` and that the direction of the link is from node `1` to node 4.

Clause Template

- Provide a function called path that will find the path from one node to another
  - Must avoid cycles (i.e. going around in circle)
  - A template for the clause is:
    - `path(Source, Destination, Visited, Path)`

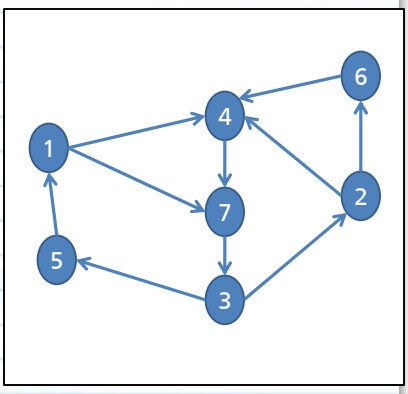


Image Licence CC0

•`Source` is the node we begin our search from. This can be any of the 7 nodes in the graph

•`Destination` is the node within the graph that we wish to reach.

•`Visited` is a list of the nodes we have already visited. We keep a record of the nodes already listed to eliminate the possibility of cycles occurring. (For example in our graph we might end up going from node `3`, to node `5`, to node `1` to node `7` and back again to node `3`.)

•`Path` is the list of all nodes on the final path between the Source node and the Destination node (and including both the Source and Destination nodes).

Search Algorithm Termination

- The search for a path terminates when there is no other node to visit
- Therefore the base case is:

```
path(Node, Node, _, [Node]).
```


How will you know when the search for a path is finished?

Search Algorithm

```
path(Source, Destination, Visited, [Source| Path])
:-   connect(Source, X),
       not(member(X, Visited)),
          path(X, Destination, [X | Visited], Path).
```


Firstly the method checks that there is a connection between the `Source` node and `X`, where `X` is the first node in the path from `Source` to `Destination`.

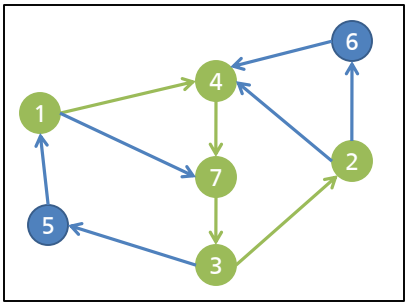
The next line `not(member(X, Visited))` verifies that the node `X` is not already contained in the list `Visited`.

Remember the `Visited` list contains a list of all nodes that we have visited in our route so far. While the member operator checks that `X` is a member of the list `Visited`, the use of the `not` operator before it means it can be read as `X` is not a member of the list `Visited`.

The final line is a recursive call to the `path` method. Notice that the current node `X`, now becomes the `Source` node in the new call to `path`. Also note that the current node `X` is appended to the start of the `Visited` list to keep track of all nodes we visit when exploring the directed graph.

Test the Search Algorithm

- Test the program.
  - Find a path from node 1 to node 2. Notice there aren't any cycles in the path




```
?- path(1, 2, [1], Path).
Path = [1, 4, 7, 3, 2] .
```


The `path` method can now be used to test the functionality of the program. We ask Prolog the question `?-path(1, 2, [1], Path).` to determine a route between `nodes 1` and node `2`, which you can see in the graph are not adjacent.
It returns the result `Path = [1, 4, 7, 3, 2].`

So the search algorithm has found a non-cyclic route between the two nodes. The route begins at node `1`, moves to node `4`, then to node `7`, then to node `3` and then arrives at node `2`.
You can run through the problem in more detail step by step by using the `trace` function.
At the command line type in:

```
?- trace.
```

You will notice your prompt now changes to:

```
[trace] 1 ?-
```

Next type in the Prolog question.

```
[trace] 1 ?- path(1, 2, [1], Path).
```

This will output a step by step description of each step the algorithm takes. The following is the output from the above question:

```
[trace] 1 ?- path(1, 2, [1], Path).
   Call: (6) path(1, 2, [1], _G544) ? creep
   Call: (7) connect(1, _G622) ? creep
   Exit: (7) connect(1, 4) ? creep
   Call: (7) not(member(4, [1])) ? creep
   Exit: (7) not(user:member(4, [1])) ? creep
   Call: (7) path(4, 2, [4, 1], _G615) ? creep
   Call: (8) connect(4, _G634) ? creep
   Exit: (8) connect(4, 7) ? creep
   Call: (8) not(member(7, [4, 1])) ? creep
   Exit: (8) not(user:member(7, [4, 1])) ? creep
   Call: (8) path(7, 2, [7, 4, 1], _G627) ? creep
   Call: (9) connect(7, _G646) ? creep
   Exit: (9) connect(7, 3) ? creep
   Call: (9) not(member(3, [7, 4, 1])) ? creep
   Exit: (9) not(user:member(3, [7, 4, 1])) ? creep
   Call: (9) path(3, 2, [3, 7, 4, 1], _G639) ? creep
   Call: (10) connect(3, _G658) ? creep
   Exit: (10) connect(3, 2) ? creep
   Call: (10) not(member(2, [3, 7, 4, 1])) ? Creep
   Exit: (10) not(user:member(2, [3, 7, 4, 1])) ? creep
   Call: (10) path(2, 2, [2, 3, 7, 4, 1], _G651) ? creep
   Exit: (10) path(2, 2, [2, 3, 7, 4, 1], [2]) ? creep
   Exit: (9) path(3, 2, [3, 7, 4, 1], [3, 2]) ? creep
   Exit: (8) path(7, 2, [7, 4, 1], [7, 3, 2]) ? creep
   Exit: (7) path(4, 2, [4, 1], [4, 7, 3, 2]) ? creep
   Exit: (6) path(1, 2, [1], [1, 4, 7, 3, 2]) ? creep
Path = [1, 4, 7, 3, 2]
```

Summary

Prolog and Recursion	Introduced the concept of recursion in Prolog and supported with a number of examples
Use of List in Prolog	Examined how to manage and manipulate lists in Prolog
Prolog Search Algorithm	Constructed a simple Prolog program that searches through a directed graph to find a route from a source node to a destination node

References

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

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