Lab 6 – 08226 Artificial Intelligence

This lab tutorial will introduce you to using recursion in your Prolog programs, and to introduce the trace facility within Prolog.

1.0 Recursion

Start SWI-Prolog-Editor from the Windows Menu system and create a new Prolog file called **Lab6.pl** and store it in **G:/08226/Lab 6/**.

In Prolog you **cannot** explicitly use any of the following loop structures:

REPEAT ... **UNITL** loops
FOR loops
DO loops
WHILE loops

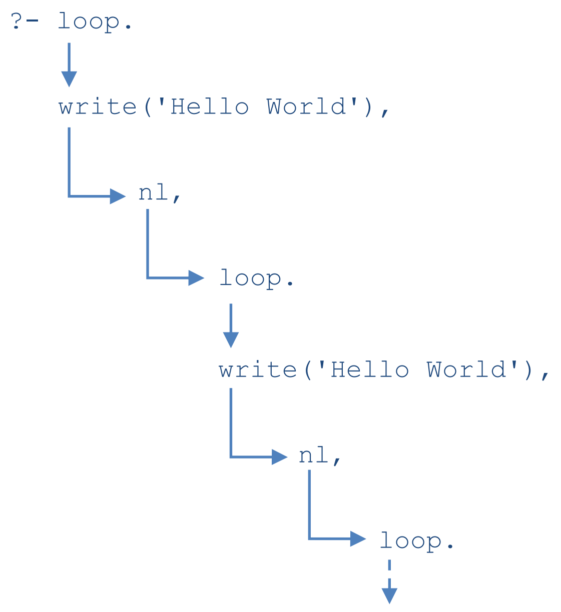
Therefore if you are required to have some form of repeating structure in your program you will have to use recursion.

Recursion in general programming is when a function or method calls itself. Fractal generation is one example of use of this technique. In Prolog recursion is when a rule uses itself as a sub-goal.

Consider the following:

```
loop :- write('Hello World'), nl, loop.
```

This rule (**loop**) will firstly display **Hello World** to the query window followed by moving the cursor to the next line. Then the final sub-goal is itself. Therefore the final sub-goal will display **Hello World** to the query window followed by moving the cursor to the next line. Then the final sub-goal is itself, and so on. Here is a diagram showing the recursive nature of the rule.



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

The major issue with this rule is that it will never end. This shows the importance to construct your recursive rules so that they **always** finish.

Enter the above rule (**loop**) into Prolog and **consult**. Run the program. To quit the Prolog program press **Esc**.

2.0 Recursion with Conditions

We require a way of continuing our recursive sub-goals only if a condition is satisfied. We could use the semi-colon (**or**) as follows:

```
loop :- write('Enter end. to finish'), nl,

        read(Word), (Word=end ; loop).
```

This rule (**loop**) will firstly display **Enter end. to finish** to the query window followed by moving the cursor to the next line. Then the rule requests an input from the user and stores it in the **Word** variable. Then the final sub-goal is split into two parts. Due to the **or** (semi-colon) symbol comparison either **Word = end** or **loop** has to be true to continue. Because only one side of the **or** symbol has to be true to continue, Prolog looks at the left hand side of the **or** symbol first. The variable **Word** is compared to the value **end**, and if it is the same then this is clearly true. Due to the **or** symbol only requires one of the side to be true it does not look at the right hand side and therefore the recursion finishes.

Enter the above code into Prolog and run it. Enter any value you wish (remember to end it full a full stop) and the recursion will continue unless you enter **end.** which will end the recursion.

We can also write the above final condition using the **and** (comer) symbol which may make more sense as then both sides of the **and** symbol have to be true for the condition to continue:

```
loop :- write('Enter end. to finish'), nl, read(Word),

        (not(Word=end), loop).
```

[The predicate not() returns true if the result of the predicate is false and visa versa. E.g. if X=X is true, then not(X=X) is false]

Apart from giving us a headache trying to understand what is going on, these two implementations of recursion are not particularly easy ways to write recursive calls especially when you are writing a large complicated rule. Therefore we usually put the final condition in a separate rule above the recursive rule so that Prolog can find it before the recursive rule.

Consider the following code:

```
loop(end).

loop(_) :- write('Enter end. to finish'), nl, read(Word), loop(Word).
```

We know have two rules. If the first rule is true (i.e. when the argument is end) then the second rule will not be executed, thus preventing the recursive calls. If the first rule is false (i.e. when the argument is anything other than end) then the second rule will be executed.

Enter the above code in Prolog and consult. Enter the following query:

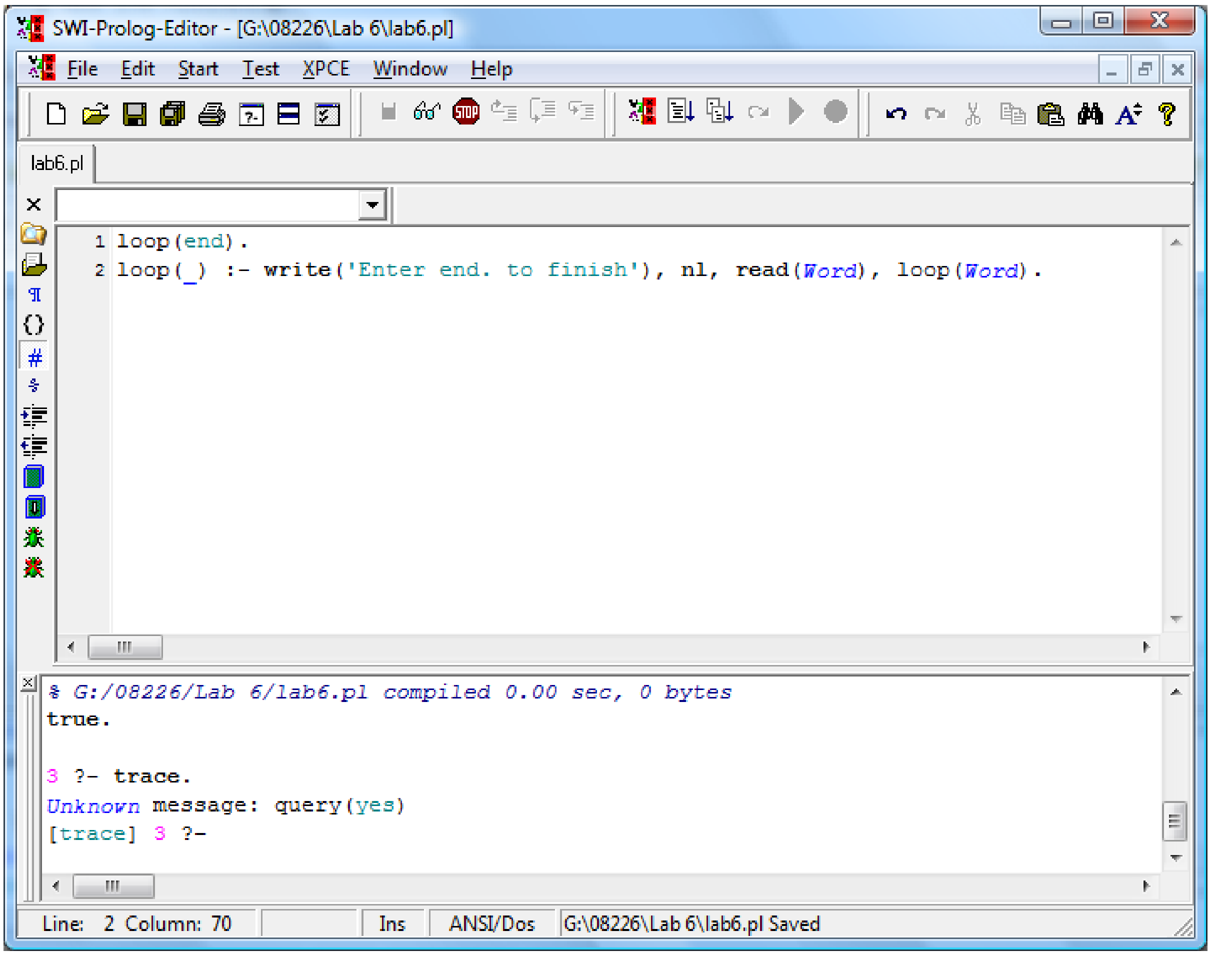
```
?- loop(anything).
```

When prompted, you can enter any value other than **end.** to continue with the recursive calls, or enter **end.** to finish the recursive calls.

3.0 Tracing

It is often useful to use a facility within Prolog that will allow us to trace what is happening within our program.

Using the same program as above, select the **tracemode** button . You should then see something like the following:


SWI-Prolog-Editor - [G:\08226\Lab 6\lab6.pl]
File Edit Start Test XPCE Window Help
lab6.pl
Line: 2 Column: 70 Ins ANSI/Dos G:\08226\Lab 6\lab6.pl Saved

```
loop(end).
loop(_) :- write('Enter end. to finish'), nl, read(Word), loop(Word).
```

```
% G:/08226/Lab 6/lab6.pl compiled 0.00 sec, 0 bytes
true.

3 ?- trace.
Unknown message: query(yes)
[trace] 3 ?-
```

Enter the following query:

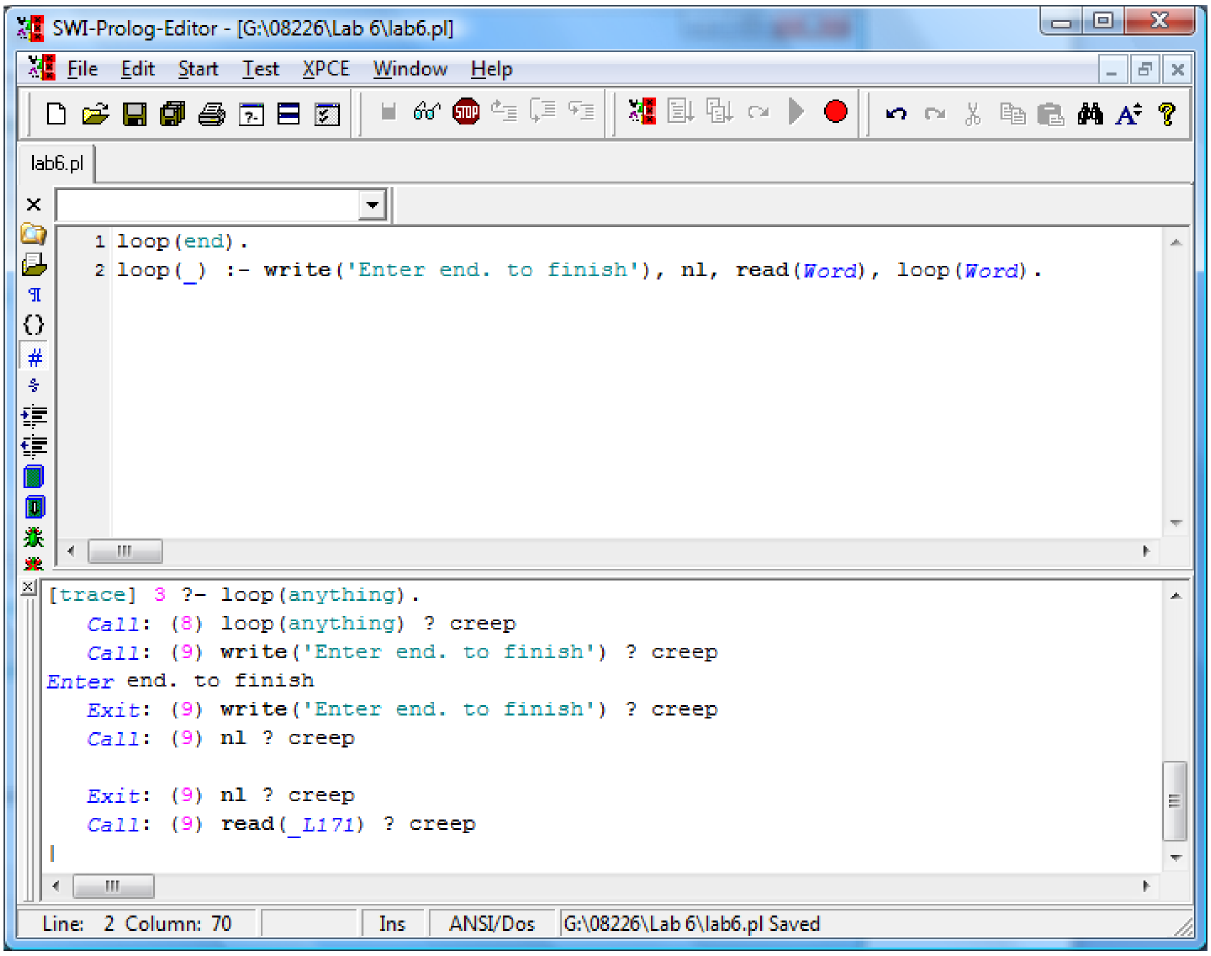
```
?- loop(anything).
```

You will be presented with a **Call** trace. This is telling you that the program has entered a new part of the program, in this case the **loop(anything)** that we entered. Press the **Enter** key to go to the next trace.

You will now be presented with another **Call** trace. The program has entered the **write** predicate. Press the **Enter** key and the **write** predicate will be executed and then you will be presented with an **Exit** trace. This is showing you that the **write** predicate has been completed and the program is about to exit the predicate. Press the **Enter** key to go to the next trace.

You will be presented with a **Call** trace for the **nl** predicate. Press the **Enter** key and the **nl** predicate will be executed and then you will be presented with an **Exit** trace. This is showing you that the **nl** predicate has been completed and the program is about to exit the predicate. Press the **Enter** key to go to the next trace.

You will be presented with a **Call** trace for the **read** predicate. The value inside of the parenthesis that begins with an underscore is the basically the register (memory address) of the variable. Press the **Enter** key and the **read** predicate will be executed. You should now have the following:



SWI-Prolog-Editor - [G:\08226\Lab 6\lab6.pl]

```
loop(end).
loop(_) :- write('Enter end. to finish'), nl, read(Word), loop(Word).
```

```
[trace] 3 ?- loop(anything).
   Call: (8) loop(anything) ? creep
   Call: (9) write('Enter end. to finish') ? creep
Enter end. to finish
   Exit: (9) write('Enter end. to finish') ? creep
   Call: (9) nl ? creep

   Exit: (9) nl ? creep
   Call: (9) read(_L171) ? creep
|
```

Line: 2 Column: 70 | Ins | ANSI/Dos | G:\08226\Lab 6\lab6.pl Saved

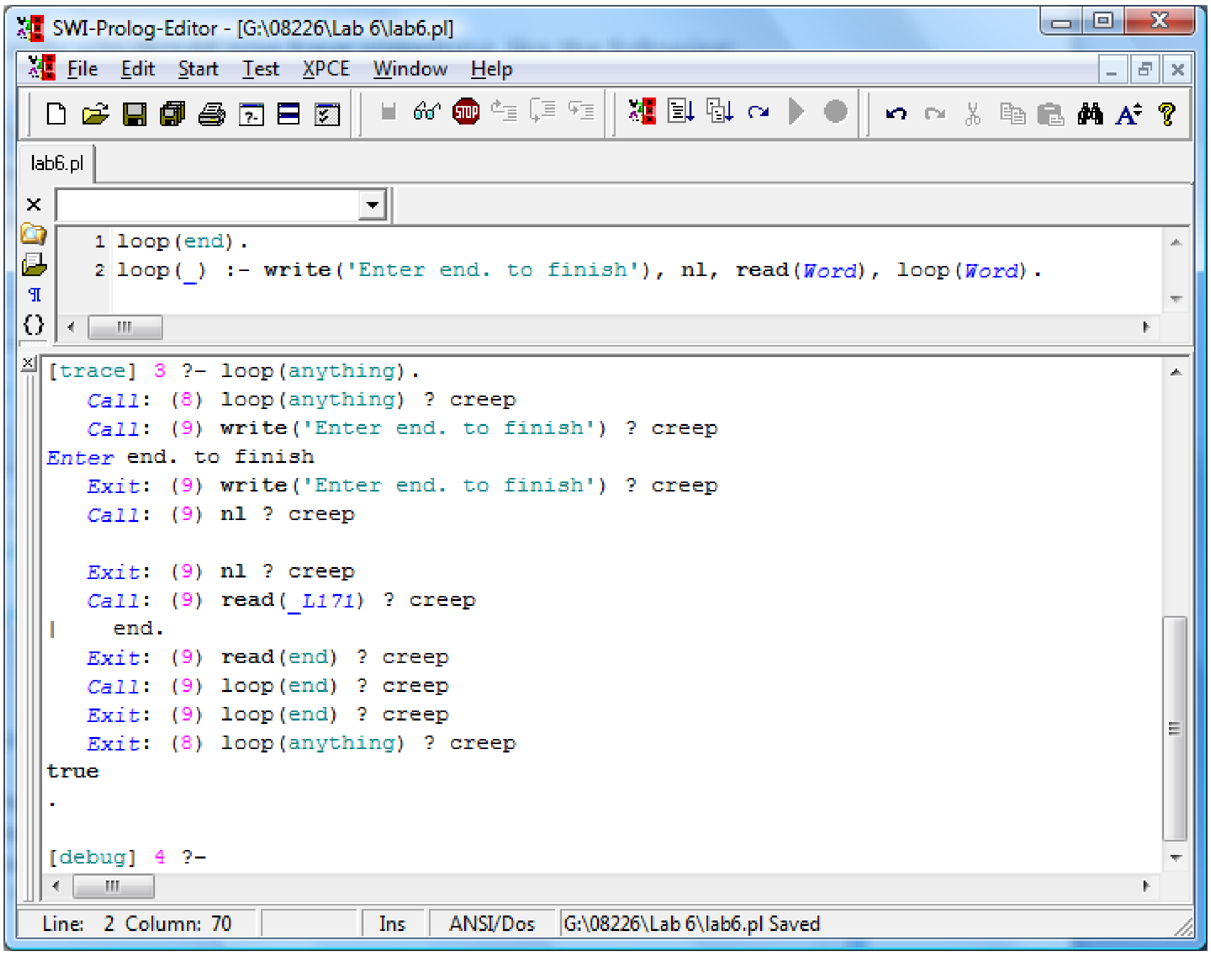
Now you can enter the value **end.** for the read predicate. Then you will be presented with an **Exit** trace. This is showing you that the **read** predicate has been completed and the program is about to exit the predicate. Press the **Enter** key to go to the next trace.

You will be presented with a **Call** trace for the **loop(end)** that we entered. Press the **Enter** key and you will be presented with an **Exit** trace for the **loop(end)**. This is showing you that the **loop(end)** rule has been completed and the program is about to exit the rule. Press the **Enter** key to go to the next trace.

You may expect that our program will finish now. Instead we have gone back to the **loop(anything)** that started our program. This is because we have backtracked to the previous recursive call. Therefore if you have had several recursive calls then Prolog will have to backtrack several times to exit the program. Press the **Enter** key to go to the next trace.

The program has almost finished and has return **true**. Press the **Enter** key to finish the program trace.

If you have traced the program correctly you should now have something like the following:


SWI-Prolog-Editor - [G:\08226\Lab 6\lab6.pl]
File Edit Start Test XPCE Window Help
lab6.pl

```
loop(end).
loop(_) :- write('Enter end. to finish'), nl, read(Word), loop(Word).
```

```
[trace] 3 ?- loop(anything).
   Call: (8) loop(anything) ? creep
   Call: (9) write('Enter end. to finish') ? creep
Enter end. to finish
   Exit: (9) write('Enter end. to finish') ? creep
   Call: (9) nl ? creep

   Exit: (9) nl ? creep
   Call: (9) read(_L171) ? creep
|    end.
   Exit: (9) read(end) ? creep
   Call: (9) loop(end) ? creep
   Exit: (9) loop(end) ? creep
   Exit: (8) loop(anything) ? creep
true
.

[debug] 4 ?-
```

Line: 2 Column: 70
Ins
ANSI/Dos
G:\08226\Lab 6\lab6.pl Saved

4.0 Recursion Exercise 1

Write a Prolog program using rules in the form **multiple_loops/1** that displays the message **'looping'** a particular number of times given by the user, e.g. if the user enters **multiple_loops(3).** into the query window, the program will display **'looping'** 3 times.

Check your answer here: **Recursion Exercise 1 Answer**

5.0 Recursion Exercise 2

Write a Prolog program using rules in the form **factorial/2** that calculates and returns the factorial of a particular number given by the user, e.g. if the user enters **factorial (3,Factorial).** into the query window, then when the program has finished Prolog will return the variable value of Factorial = 6.

*[The factorial equation is described as n! This means that an integer is multiplied by all of the previous integers lower than itself, e.g. 2! = 2*1, 3! = 3*2*1, 4! = 4*3*2*1]*

Check your answer here: **Recursion Exercise 2 Answer**

6.0 Recursion Exercise 1 Answer

Write a Prolog program using rules in the form **multiple_loops/1** that displays the message **'looping'** a particular number of times given by the user, e.g. if the user enters **multiple_loops(3).** into the query window, the program will display **'looping'** 3 times.

```
multiple_loops(0).

multiple_loops(N) :- write('looping'), nl,

                     M is N-1, multiple_loops(M).
```

Return to your exercise here: **Recursion Exercise 1**

7.0 Recursion Exercise 2 Answer

Write a Prolog program using rules in the form **factorial/2** that calculates and returns the factorial of a particular number given by the user, e.g. if the user enters **factorial (3,Factorial).** into the query window, then when the program has finished Prolog will return the variable value of Factorial = 6.

*[The factorial equation is described as n! This means that an integer is multiplied by all of the previous integers lower than itself, e.g. 2! = 2*1, 3! = 3*2*1, 4! = 4*3*2*1]*

```
factorial(1,1).

factorial(N,Fact) :- M is N-1,

                     factorial(M,New_Fact),

                     Fact is New_Fact*N.
```

Return to your exercise here: **Recursion Exercise 2**