**Declarative Programming: Handout 10**

**Yılmaz Kılıçaslan**
December 21, 2022

**Key Concepts:** Meta-Programming, Ordering of Clauses, Ordering of Goals, Left-Recursion Problem, `cut`, `fail`, Search Space, Negation as Failure, `assert`, `retract`.

# META-PROGRAMMING IN PROLOG

## 1 INTRODUCTION

If a certain level of representation somehow 'refers to itself', it becomes a **meta**-level. For example, a sentence like:

**(1) A SENTENCE WITH A META-LEVEL INTERPRETATION**

'This sentence is false.'

would be taken to be a meta-level representation when interpreted with the demonstrative (i.e., 'this') referring to the sentence itself.

In a similar but much weaker sense, an interpreter (or an assembler or compiler) can be thought of as a meta-level program (or a meta-porogram) as it takes as input and operates on something of its kind, a program:

**(2) PROGRAMS PROCESSING OTHER PROGRAMS**


Exploiting the flexibility of this sense, we will define 'meta-programming' as a programming activity that is carried out by taking into consideration the working principles of the interpreter / compiler / assembler etc. used.

In fact, from this point of view any programming activity can be considered as a meta-level activity because when writing a program we always keep in mind the working principles of the translator used to make the program understandable to the computer. However, mostly we are not aware of this dependence on the translator as it suffices to bear in mind only the most general principles. For instance, in logic programming, if we are lucky enough, we can do with the principles of logic, which we follow almost unconsciously.

But, sometimes there might be deviations from general principles. In logic programming this happens when non-logical (i.e., non-declarative) restrictions constrain the way we are writing a program. In these cases, we become obliged to write a program with an eye to the internal working of the interpreter that will interpret it. In a sense, such cases force us to be programming more self-consciously. We will refer to such sort of programming practice as *weak meta-programming*.

At some other times, it does not suffice us just to know the internal working of the interpreter but it becomes necessary for us to intervene with the working of the interpreter. This happens when the default way in which the operator works yields undesirable results. In those cases, we will need to tell the interpreter to deviate from the way it usually follows. We will call a programming practice resorting to such interventions *strong meta-programming*.

Sometimes, we can even want to change the original source code of our program. If we do this, we will consider ourserselves as doing *super-strong meta-programming*

Today, we will see how we can practice the three kinds of meta-programming defined as above. For weak meta-programming we will play around with the ordering of clauses and that of goals in Prolog. For strong and super-strong meta programming Prolog provides us with some extra-logical operators. We will examine how we can employ the `cut` (`!`) operator and the `fail/0` predicate to do strong meta-programming and the `assert/1` and `retract/1` predicates to do `super`-strong meta-programming.

## 2 Weak Meta-Programming by Ordering Clauses and Goals

### 2.1 How Ordering Matters in Prolog

Let us reconsider the Prolog program that we wrote last week to define some family relations (adopted from Sterling and Shapiro's (1986) Biblical family):

**(3) Prolog Program Defining Family Relations**

```
father(tarik,ibrahim).
father(tarik,naci).
father(tarik,harun).
father(ibrahim,ishak).
father(ibrahim,ismail).
father(harun,lut).
father(harun,melike).
father(harun,yasemin).
mother(sahra,ishak).
mother(melike,abide).
mother(abide,mustafa).
male(tarik).
male(ibrahim).
male(naci).
male(harun).
male(lut).
male(ishak).
male(ismail).
female(sahra).
female(melike).
female(yasemin).
female(zeliha).
female(nalan).
parent(X,Y):- father(X,Y).
parent(X,Y):- mother(X,Y).
ancestor(X,Y):- parent(X,Y).
ancestor(X,Y):- parent(X,Z),ancestor(Z,Y).
```

Recall that we have exactly ten solutions for the following simple query to the program above:

**(4)**

```
?- ancestor(tarik, X).
X = ibrahim ;
X = naci ;
X = harun ;
X = ishak ;
X = ismail ;
X = lut ;
X = melike ;
X = yasemin ;
X = abide ;
X = mustafa ;
false.
```

Here is a very simple observation about the ordering of Prolog clauses: Changing the ordering of some clauses may result in a change in the ordering of generated solutions:

**(5)**

```
father(tarik,ibrahim).
father(tarik,naci).
.
.
.
```

→

```
father(tarik,naci).
father(tarik,ibrahim).
.
.
.
```

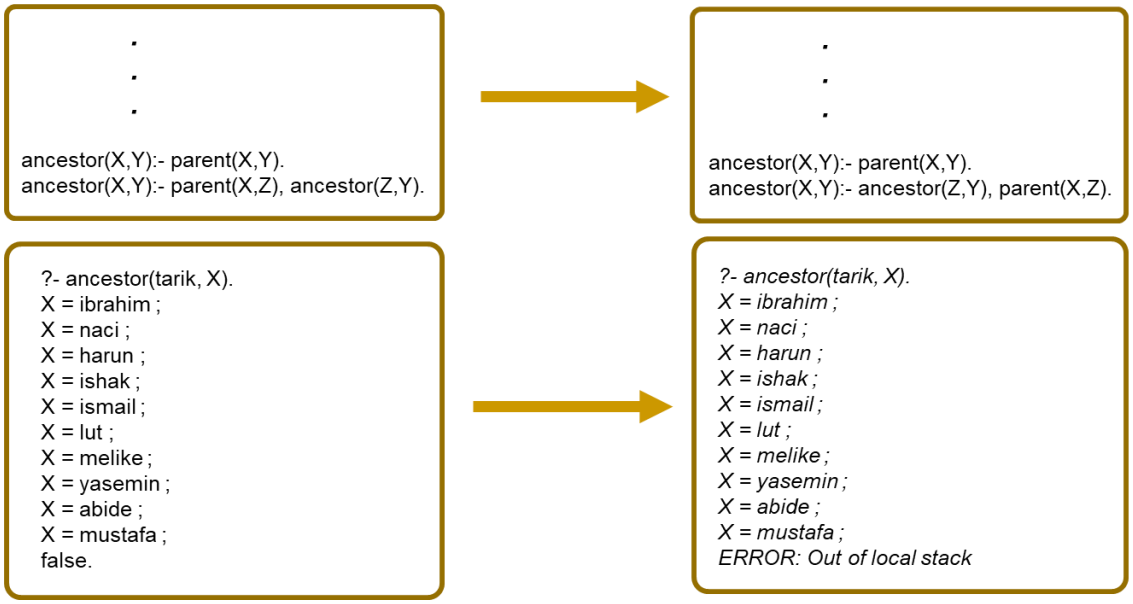
```
?- ancestor(tarik, X).
X = ibrahim ;
X = naci ;
X = harun ;
X = ishak ;
X = ismail ;
X = lut ;
X = melike ;
X = yasemin ;
X = abide ;
X = mustafa ;
false.
```

→

```
?- ancestor(tarik, X).
X = naci ;
X = ibrahim ;
X = harun ;
X = ishak ;
X = ismail ;
X = lut ;
X = melike ;
X = yasemin ;
X = abide ;
X = mustafa ;
false.
```

Consider now what happens if we change the order of the two goals in the body of the recursive ancestor rule:

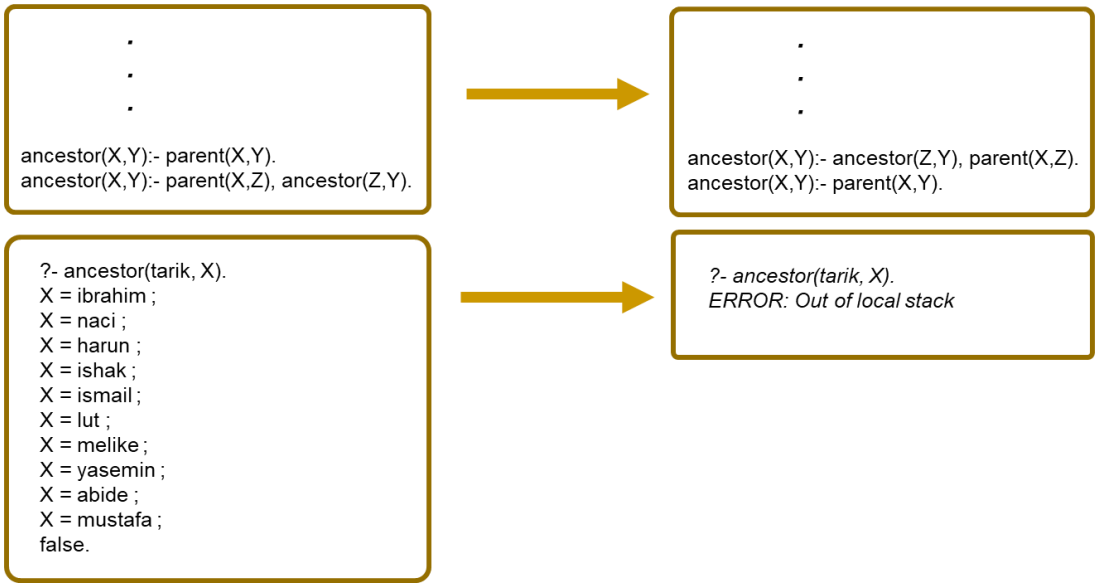
### (6) Re-Ordering the Goals in the Body of the Recursive Rule




We get the same ten solutions for the query '?- ancestor(tarik, X).' but this time we also get a stack overflow message if we ask for an alternative solution after these ten.

Even though the modifications above lead to some changes in the outputs, we can turn a blind eye to them as they are performative issues, after all; we still get the same set of responses in each case. However, Let us see now see what happens if we also change the order of the clauses of the ancestor/2 procedure (i.e., if we place the base clause after the recursive one):

### (7) Re-Ordering the Clauses of the ancestor/2 Procedure



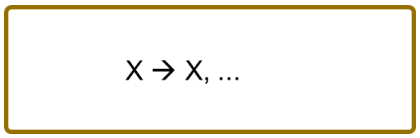

We will encounter a more undesirable response: a stack overflow message with no solutions provided.

Logic tells us that logically equivalent premises (logically equivalent programs in our case) yield the same conclusions (the same responses in the case of logic programming). However, it is clear from the examples above that Prolog might behave in a way that is not in accord with what logic says. If Prolog were a 'perfect' logic programming language (i.e., a programming language that models logic in a perfect way), for the same queries we would get exactly the same set of responses from the interpreter. However, this is not the case.

### 2.2 The Left-Recursion Problem

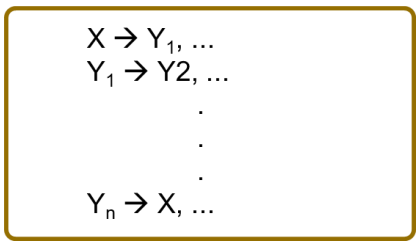
You might have guessed why a stack overflows in cases like those above. This happens simply because the interpreter goes into an infinite loop. A slightly more difficult question is what causes the interpreter to go into an infinite loop in such cases. Let us give the answer without beating around the bush. The program fragments in (6) and (7) suffer from the left-recursion problem. In a context-free grammar, a rule is left-recursive if the leftmost symbol on its right-hand side is the same as the nonterminal on its left-hand side:

**(8) SCHEMA FOR DIRECT LEFT RECURSION**




or, the leftmost symbol on the right-hand side can be made the same as the one on the left-hand side by some sequence of substitutions:

**(9) SCHEMA FOR INDIRECT LEFT RECURSION**




As Prolog follows a left-to-right strategy when handling the goals in the body of a rule, a rule whose syntax fits in one of the schemas above will ultimately lead to an infinite loop. This is what happens with the query '?- ancestor(tarik, X).' asked to the programs in (6) or (7). Since the base clause precedes the recursive one, this happens after generating all possible solutions in case the query is asked to (6). However, the worst scenario takes place in case the query is raised to (7): the adopted top-down strategy leads the interpreter to the left-recursive rule before it finds a chance to discover the solutions through the base rule.

### 2.3 How to Handle the Left-Recursion Problem

What follows from the discussion above is that when writing Prolog programs we should always keep the following rule of thumb in mind:

**(10) A RULE OF THUMB**

- base clauses should precede recursive ones and
- goals in the body of a recursive rule should be ordered in a non-left-recursive style.

In conclusion, if we want to be more effective Prolog programs, we need to raise to a level of meta-programmer. To do this, we should not be content only with a knowledge of general principles of first-order predicate logic but we should also know (and keep in mind when writing a program) the 'non-logical' ways the interpreter may behave, such as taking heed of the ordering of clauses and that of goals when interpreting a program.

## 3 STRONG META-PROGRAMMING BY INTERFERING WITH THE INTERPRETER

### 3.1 Using the `cut` (`!`) Operator

The system predicate `cut`, which is denoted in Prolog by `!`, serves to reduce the search space of Prolog computations by dynamically pruning the search tree. A search space in logic programming consists of all the paths through which the interpreter can search for a solution. The expected effect of the *cut* operator is to narrow down the search space. A *cut* is effective in the **procedure** it occurs in. It prunes all alternative solution paths to be generated by the clauses that appear **below** it and by the goals that appear to its **left**.

Let us exemplify how the *cut* operator works by referring to our biblical family (cf. (3)). Remember that we get ten different solutions if the query '`?- ancestor(tarik, X).`' is issued for this program (cf. (4)). Let us see some examples that illustrate how the *cut* operator can reduce the number of solutions for this query. Consider first the following example:

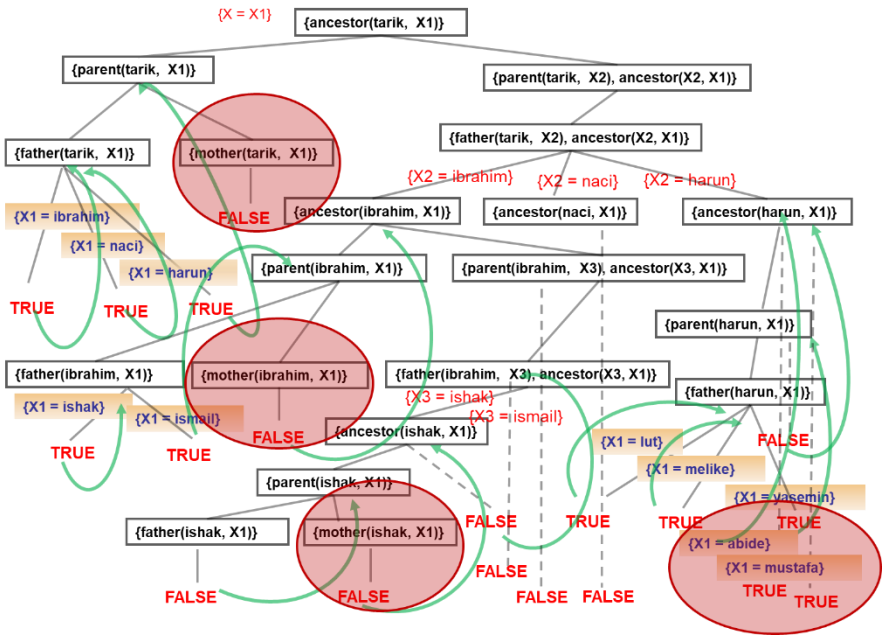
**(11) AN EXAMPLE CONTAINING THE *cut* OPERATOR**

```
        .
        .
        .

parent(X,Y):- !,father(X,Y).
parent(X,Y):- mother(X,Y).

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

```
?- ancestor(tarik, X).
X = ibrahim ;
X = naci ;
X = harun ;
X = ishak ;
X = ismail ;
X = lut ;
X = melike ;
X = yasemin ;
false.
```

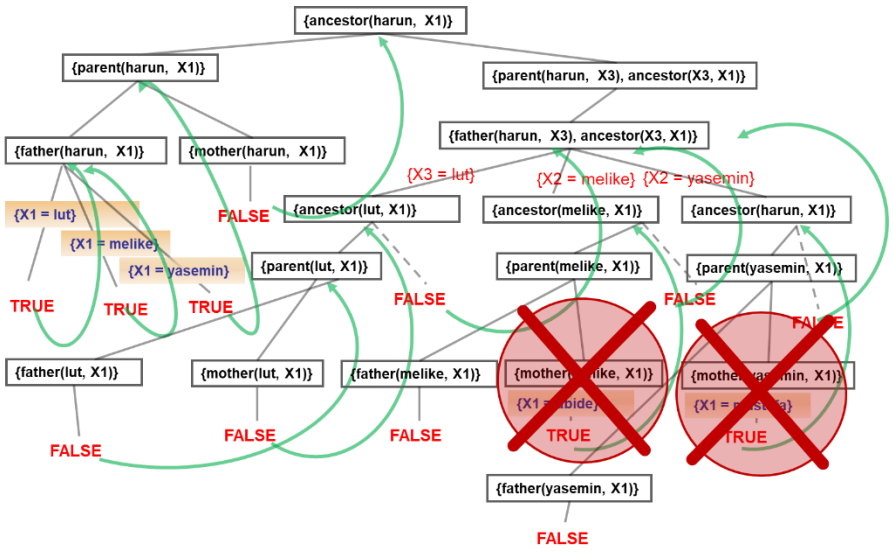
Notice we have this time eight alternative responses for the query, not ten. Obviously, two of the responses (namely, the last two ones in (4)) have been cut out of the original list. Taking the position of the *cut* operator into account, it is not difficult to see that the responses removed from the list are those to be generated by the second clause in the `parent` procedure. The effect of that occurrence of the *cut* operator is to chop out those parts of the tree built via the reduction of the `mother` goal. The red-circled parts below are some of them:

**(12)**




The solutions not appearing in the list narrowed down by the *cut* operator are those in the bottom-right circle. Below is a zoomed-in view of the fragment of the search tree that contains that part:

**(13)**



The `mother` goals in the circled parts have been blocked by the *cut* operator, causing the solutions there (namely, *abide* and *mustafa*) not to be included in the response-list.

Here are two more examples where the *cut* operator is used, along with the generated query-answer pairs :

**(14) TWO OTHER EXAMPLES CONTAINING THE *cut* OPERATOR**

a)

```
.
.
.

parent(X,Y):- father(X,Y),!.
parent(X,Y):- mother(X,Y).

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

```
?- ancestor(tarik, X).
X = ibrahim ;
X = ishak ;
false.
```

b)

```
.
.
.

parent(X,Y):- father(X,Y).
parent(X,Y):- mother(X,Y).

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

```
?- ancestor(tarik, X).
X = ibrahim ;
X = naci ;
X = harun ;
false.
```

### 3.2 Using the `fail` Predicate

Wherever it is used the predicate `fail/0` fails and, therefore, forces the interpreter to backtrack. For example, a query like the one in (15b) would yield the indicated result when sent to the program in (15a):

**(15) A PROGRAM-QUERY PAIR WITH `fail/0`:**

a)

```
pred(a):- fail.
pred(b).
```

b)

```
?- pred(X).
X = b ;
false.
```

The predicate `fail/0` is a system predicate and, hence, there are no clauses defined for it.

A question that might naturally come to mind is this: If this predicate is always doomed to fail, why do we have it? It may not sound too useful to force backtracking. Nonetheless, there are some ways in which the `fail/0` predicate can be useful. Just to mention one, it can be used to define a procedural loop. We can, for instance, use it to define an endless loop:

**(16) A PROCEDURAL LOOP WITH `fail/0`**

```
loop:- writeln('A new line!'), fail.
loop:- loop.
```

The procedure above will write the same message on the screen unceasingly:

**(17)**

```
?- loop.
A new line!
A new line!
A new line!
   .
   .
   .
```

However, the real usefulness of the `fail/0` predicate can be appreciated when used in combination with the cut operator. In the following section, we will illustrate, with an example, how the `cut-fail` combination can serve to define so-called 'negation-as-failure'.

### 3.3. Using the `cut-fail` Combination

Consider those cases where we need to take exceptions into consideration. One of the best features of Prolog is that it enables us to make generalizations via rules. For instance, Ali might love dogs. We can express this in Prolog with a rule like the following:

**(18)**

```
loves(ali, X):- dog(X).
```

However, in real life generalizations may come with exceptions. Ali might, in general, be loving dogs but not so when it comes to bulldogs. So, how can we state that Ali loves dogs but not bulldogs. Can we, for example, express the exception by writing the clause below just after (18)?

**(19)**

```
loves(ali, X):- bulldog(X), fail.
```

The answer is negative. This will not work because the first clause already subsumes those cases where Ali is stated also to love bulldogs. To illustrate, let us assume that we have added the following clauses to our program:

**(20)**

```
dog(X):- labrador(X).
dog(X):- siberian(X).
dog(X):- bulldog(X).
dog(X):- terrier(X).
dog(X):- cocker(X).
labrador(fido).
siberian(ivan).
bulldog(rodrigo).
terrier(carlos).
cocker(ioup).
```

Here are the responses to be generated for a query asking for those that Ali loves:

**(21)**

```
?- loves(ali,X).
X = fido ;
X = ivan ;
X = rodrigo ;
X = carlos ;
X = ioup ;
No
```

The third answer is undesirable because it includes `rodrigo`, which is a bulldog. But, if we change the order of the `loves` clauses and put a cut operator immediately before the fail/0 predicate, we will somewhat come closer to what we want to achieve:

**(22)**

```
loves(ali,X):- bulldog(X),!,fail.
loves(ali,X):- dog(X).
```

At least, we will get correct answers if ask the same thing for each individual dog separately:

**(23)**

```
?- loves(ali,fido).
Yes
?- loves(ali,rodrigo).
No
```

However, this time we will come up with an even more undesirable answer for the query in (20):

**(24)**

```
?- loves(ali,X).
No
```

Even the felicitious positive answers have become unavailable. They were to be produced by the second `loves` clause that is already blocked by the `cut` operator occurring in the first `loves` clause.

The good news is that we can break this blockage by placing the cut operator in a separate clause together with the `fail/0` predicate, as, for example, in (24):

**(25) A NEW NEGATION PREDICATE**

```
degil(P):- P, !, fail.
degil(_).
```

It is not coincidental that the predicate is named as `degil`. As you know, 'değil' is the (widescope) negation operator in Turkish. Similarly, the procedure in (25) is a form of negation, as illustrated by the following program-query pairs:

**(26)**

```
pred(a):- fail.
pred(b).
```

```
?- degil(pred(a)).
true.
?- degil(pred(c)).
true.
?- degil(pred(b)).
false.
```

Now, if we replace the procedure in (22) with the one below:

**(27)**

```
loves(ali,X):- dog (X),
               degil(bulldog(X)).
```

we will finally get the desired result:

**(28)**

```
?- loves(ali, X).
X = fido ;
X = ivan ;
X = carlos ;
X = ioup ;
No
```

The form of negation offered by the `cut-fail` combination, as in (25), is called negation as failure. The standart built-in negation operator that comes with Prolog (`not/0` or `\+/0`) just performs that kind of negation. The following is equivalent to (27):

**(29)**

```
loves(ali,X):- dog(X),
               not(bulldog(X)).
```

Note, however, that you must not make the mistake of thinking that negation as failure works just like logical negation. You already know that from a logical standpoingt it does not matter how we order the goals in the body of a rule. So, we might expect the following rule to behave as (29) (or (27)):

**(30)**

```
loves(ali,X):- not(bulldog(X)),
               dog(X).
```

But, not necessarily it will. This is simply because the built-in negation operator, `not`, is not the same as the logical negation operator. It has a procedural aspect, which you can grasp if you closely examine how the cut-fail combination works.

## 4 Super-Strong Meta-Programming by Interfering with the Interpreter

### 4.1 Using the `assert/1` Predicate

The system predicate `assert/1` allows for adding clauses to a program at runtime. For instance, the following query will serve to add the fact `father(kaya, suna)` to your program:

**(31)**

```
?- assert(father(kaya, suna)).
true.
```

After making this assertion, we will have the following query-response pairs for the enlarged program:

**(32)**

```
?- father(kaya, suna).
true.

?- father(X, suna).
X = kaya.
```

### 4.2 Using the `retract/1` Predicate

The predicate `retract/1` is the complement of `assert/1`. It allows for removing clauses from a program at runtime. For example, the query below will serve to remove the fact `father(kaya, suna)` from the program:

**(33)**

```
?- retract(father(_, _)).
true.
```

After this, we will have the following query-response pairs for the program:

**(34)**

```
?- father(kaya, suna).
false.

?- father(X, suna).
false.
```

## 5 CONCLUSION

Efficient programming requires some knowledge of the working principles of the translator (e.g., interpreter, compiler, or assembler) used. We have called programming practice performed with such knowledge in mind **meta-programming**. We saw with examples how **weak** meta-programming can be carried out by ordering clauses and / or goals in Prolog to avoid unwanted results; how **strong** meta-programming can be performed using the `cut` operator and the `fail/0` predicate; and how **super-strong** meta-programming can be done using the `assert/1` and `retract/1` predicates.

**EXERCISES**

**A. Which ones are meta-level representations? (5 points each)**

1. **Statement A1 is meaningless.**
2. **What the teacher said last week was nonsense.**
3. **Pentasyllabic**
4. **Edible**
5. **Incomplete**
6. **Awkwardnessful**
7. 
8. 
9. 

**B. Does the ordering of clauses and goals in a program logically matter? (5 points)**

**C. Might the ordering of clauses and goals matter in terms of what responses we get when we query a Prolog program? (5 points)**

**D. Place the *cut* operator in various other positions in the program in (3) and observe how the responses are affected each time.**

## References

Sterling, L. and Shapiro E. (1986) The Art of prolog. Cambridge, Massachusetts: MIT Press.