Lab Task:

Install SWI-Prolog and SWI-Prolog Editor on your laptops/PCs.

Lab 11 - Part 2 - First Order Logic

In this lab, we'll learn about a different way of programming using a declarative programming language called Prolog.

Introduction to Prolog

There are only three basic constructs in Prolog: facts, rules, and queries. A collection of facts and rules is called a knowledge base (or a database) and Prolog programming is all about writing knowledge bases. That is, **Prolog programs simply are knowledge bases:** collections of facts and rules which describe some collection of relationships that we find interesting.

So how do we use a Prolog program? By posing queries. That is, by asking questions about the information stored in the knowledge base.

Syntax table

Syntax	**Meaning**
:-	Is implied by If (implication)
,	and (conjunction)
;	or (disjunction)
Names starting with capital letters e.g. X	Variables
Names starting with small letters e.g. woman	Atoms
\+	not
<	is less than

>	is greater than
=	is equal to
>=	is greater than or equal to
=<	is less than or equal to
=:=	equality check
=\=	inequality check

<u>Knowledge Base 1</u>

Knowledge Base 1 (KB1) is simply a collection of facts. Facts are used to state things that are unconditionally true of some situation of interest. For example, we can state that Mia, Jody, and Yolanda are women, that Jody plays air guitar, and that a party is taking place, using the following five facts:

```
woman(mia).
woman(jody).
woman(yolanda).
playsAirGuitar(jody).
party.
```

This collection of facts is KB1. It is our first example of a Prolog program. Note that the **names** mia, jody, and yolanda, the **properties** woman and playsAirGuitar , and the **proposition** party have been written so that the first letter is in lower-case. **This is important**; we will see why later.

How can we use KB1? By posing queries. That is, by asking questions about the information KB1 contains. Here are some examples. We can ask Prolog whether Mia is a woman by posing the query:

```
?-  woman(mia).
```

Prolog will answer

```
yes
```

for the obvious reason that this is one of the facts explicitly recorded in KB1. Incidentally, we don't type in the ?- . This symbol (or something like it, depending on the implementation of Prolog you are using) is the prompt symbol that the Prolog interpreter displays when it is waiting to evaluate a query. We just type in the actual query (for example woman(mia)) followed by . (a full stop). The full stop is important. If you don't type it, Prolog won't start working on the query.

Similarly, we can ask whether Jody plays air guitar by posing the following query:

```
?-  playsAirGuitar(jody).
```

Prolog will again answer yes, because this is one of the facts in KB1. However, suppose we ask whether Mia plays air guitar:

```
?-  playsAirGuitar(mia).
```

We will get the answer

```
no
```

Why? Well, first of all, this is not a fact in KB1. Moreover, KB1 is extremely simple, and contains no other information (such as the rules we will learn about shortly) which might help Prolog try to infer (that is, deduce) whether Mia plays air guitar. So Prolog correctly concludes thatplaysAirGuitar(mia) does not follow from KB1.

Here are two important examples. First, suppose we pose the query:

```
?-  playsAirGuitar(vincent).
```

Again Prolog answers no. Why? Well, this query is about a person (Vincent) that it has no information about, so it (correctly) concludes thatplaysAirGuitar(vincent) cannot be deduced from the information in KB1.

Similarly, suppose we pose the query:

```
?-  tattooed(jody).
```

Again Prolog will answer no. Why? Well, this query is about a property (being tatooed) that it has no information about, so once again it (correctly) concludes that the query cannot be deduced from the information in KB1. (Actually, some Prolog implementations will respond to this query with an error message, telling you that the predicate or procedure tattooed is not defined; we will soon introduce the notion of predicates.)

Needless to say, we can also make queries concerning propositions. For example, if we pose the query

```
?-  party.
```

then Prolog will respond

```
yes
```

and if we pose the query

```
?-  rockConcert.
```

then Prolog will respond

```
no
```

exactly as we would expect.

Knowledge Base 2

Here is KB2, our second knowledge base:

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda):-  happy(yolanda).
playsAirGuitar(mia):-  listens2Music(mia).
playsAirGuitar(yolanda):-  listens2Music(yolanda).
```

There are two facts in KB2, listens2Music(mia) and happy(yolanda) . The last three items it contains are rules.

Rules state information that is conditionally true of the situation of interest. For example, the first rule says that Yolanda listens to music if she is happy, and the last rule says that Yolanda plays air guitar if she listens to music. More generally, the :- should be read as “if”, or “is implied by”. The part on the left hand side of the :- is called the head of the rule, the part on the right hand side is called the body. So in general rules say: if the body of the rule is true, then the head of the rule is true too. And now for the key point:

If a knowledge base contains a rule head :- body, and Prolog knows that body follows from the information in the knowledge base, then Prolog can infer head.

This fundamental deduction step is called modus ponens.

Let’s consider an example. Suppose we ask whether Mia plays air guitar:

```
?-  playsAirGuitar(mia).
```

Prolog will respond yes. Why? Well, although it can’t find playsAirGuitar(mia) as a fact explicitly recorded in KB2, it can find the rule

```
playsAirGuitar(mia):-  listens2Music(mia).
```

Moreover, KB2 also contains the fact listens2Music(mia) . Hence Prolog can use the rule of modus ponens to deduce thatplaysAirGuitar(mia) .

Our next example shows that Prolog can chain together uses of modus ponens. Suppose we ask:

```
?-  playsAirGuitar(yolanda).
```

Prolog would respond yes. Why? Well, first of all, by using the fact happy(yolanda) and the rule

```
listens2Music(yolanda):-  happy(yolanda).
```

Prolog can deduce the new fact listens2Music(yolanda) . This new fact is not explicitly recorded in the knowledge base — it is only implicitlypresent (it is inferred knowledge). Nonetheless, Prolog can then use it just like an explicitly recorded fact. In particular, from this inferred fact and the rule

```
playsAirGuitar(yolanda):-  listens2Music(yolanda).
```

it can deduce playsAirGuitar(yolanda) , which is what we asked it. Summing up: any fact produced by an application of modus ponens can be used as input to further rules. By chaining together applications of modus ponens in this way, Prolog is able to retrieve information that logically follows from the rules and facts recorded in the knowledge base.

The facts and rules contained in a knowledge base are called clauses. Thus KB2 contains five clauses, namely three rules and two facts. Another way of looking at KB2 is to say that it consists of three predicates (or procedures). The three **predicates** are:

```
listens2Music
happy
playsAirGuitar
```

The happy predicate is defined using a single clause (a fact). The listens2Music and playsAirGuitar predicates are each defined using two clauses (in one case, two rules, and in the other case, one rule and one fact). It is a good idea to think about Prolog programs in terms of the predicates they contain. In essence, the predicates are the concepts we find important, and the various clauses we write down concerning them are our attempts to pin down what they mean and how they are inter-related.

One final remark. **We can view a fact as a rule with an empty body**. That is, we can think of facts as conditionals that do not have any antecedent conditions, or degenerate rules.

Knowledge Base 3

KB3, our third knowledge base, consists of five clauses:

```
happy(vincent).
listens2Music(butch).
playsAirGuitar(vincent):-
      listens2Music(vincent),
      happy(vincent).
playsAirGuitar(butch):-
      happy(butch).
playsAirGuitar(butch):-
      listens2Music(butch).
```

There are two facts, happy(vincent) and listens2Music(butch) , and three rules.

KB3 defines the same three predicates as KB2 (namely happy , listens2Music , and playsAirGuitar) but it defines them differently. In particular, the three rules that define the playsAirGuitar predicate introduce some new ideas. First, note that the rule

```
   playsAirGuitar(vincent):-
          listens2Music(vincent),
          happy(vincent).
```

has two items in its body, or (to use the standard terminology) two goals. So, what exactly does this rule mean? The most important thing to note is the comma , that separates the goal listens2Music(vincent) and the goal happy(vincent) in the rule's body. This is the way logical conjunction is expressed in Prolog (that is, the comma means and). So this rule says: "Vincent plays air guitar if he listens to music and he is happy".

Thus, if we posed the query

```
   ?-  playsAirGuitar(vincent).
```

Prolog would answer no. This is because while KB3 contains happy(vincent) , it does not explicitly contain the informationlistens2Music(vincent) , and this fact cannot be deduced either. So KB3 only fulfils one of the two preconditions needed to establishplaysAirGuitar(vincent) , and our query fails.

Incidentally, the spacing used in this rule is irrelevant. For example, we could have written it as

```
   playsAirGuitar(vincent):-  happy(vincent),
                             listens2Music(vincent).
```

and it would have meant exactly the same thing. Prolog offers us a lot of freedom in the way we set out knowledge bases, and we can take advantage of this to keep our code readable.

Next, note that KB3 contains two rules with exactly the same head, namely:

```
   playsAirGuitar(butch):-
          happy(butch).
   playsAirGuitar(butch):-
          listens2Music(butch).
```

This is a way of stating that Butch plays air guitar either if he listens to music, or if he is happy. That is, listing multiple rules with the same head is a way of expressing logical disjunction (that is, it is a way of saying or). So if we posed the query

?- playsAirGuitar(butch).

Prolog would answer yes. For although the first of these rules will not help (KB3 does not allow Prolog to conclude that happy(butch)), KB3does contain listens2Music(butch) and this means Prolog can apply modus ponens using the rule

```
   playsAirGuitar(butch):-
          listens2Music(butch).
```

to conclude that playsAirGuitar(butch) .

There is another way of **expressing disjunction** in Prolog. We could replace the pair of rules given above by the single rule

```
playsAirGuitar(butch):-
      happy(butch);
      listens2Music(butch).
```

That is, the **semicolon ;** is the Prolog symbol for **or** , so this single rule means exactly the same thing as the previous pair of rules. Is it better to use multiple rules or the semicolon? That depends. On the one hand, extensive use of semicolon can make Prolog code hard to read. On the other hand, the semicolon is more efficient as Prolog only has to deal with one rule.

It should now be clear that **Prolog has something to do with logic**: after all, the **:- means implication**, the **, means conjunction**, and the **; means disjunction**. (What about **negation**? That is a whole other story.) Moreover, we have seen that a standard logical proof rule (**modus ponens**) plays an important role in Prolog programming. So we are already beginning to understand why “Prolog” is short for “Programming with logic”.

Knowledge Base 4

Here is KB4, our fourth knowledge base:

```
woman(mia).
woman(jody).
woman(yolanda).

loves(vincent,mia).
loves(marsellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).
```

Now, this is a pretty boring knowledge base. There are no rules, only a collection of facts. Ok, we are seeing a relation that has two names as arguments for the first time (namely the loves relation), but, let’s face it, that’s a rather predictable idea.

No, the novelty this time lies not in the knowledge base, it lies in the queries we are going to pose. In particular, for the first time we’re going to make use of variables . Here’s an example:

```
?-  woman(X).
```

The X is a variable (in fact, any word beginning with an upper-case letter is a Prolog variable, which is why we had to be careful to use lower-case initial letters in our earlier examples). Now **a variable isn’t a name, rather it’s a placeholder for information**. That is, ***this query asks Prolog: tell me which of the individuals you know about is a woman.***

Prolog answers this query by working its way through KB4, from top to bottom, trying to unify (or match) the expression woman(X) with the information KB4 contains. Now the first item in the knowledge base is woman(mia) . So, Prolog unifies X with mia , thus making the query agree perfectly with this first item. (Incidentally, there's a lot of different terminology for this process: we can also say that Prolog instantiates X to mia , or that it binds X to mia .) Prolog then reports back to us as follows:

```
   X   =   mia
```

That is, it not only says that there is information about at least one woman in KB4, it actually tells us who she is. It didn't just say yes, it actually gave us the variable binding (or variable instantiation) that led to success.

But that's not the end of the story. The whole point of variables is that they can stand for, or unify with, different things. And there is information about other women in the knowledge base. We can access this information by typing a semicolon:

```
   X   =   mia   ;
```

Remember that ; means or , so this query means: are there any alternatives ? So Prolog begins working through the knowledge base again (it remembers where it got up to last time and starts from there) and sees that if it unifies X with jody , then the query agrees perfectly with the second entry in the knowledge base. So it responds:

```
   X   =   mia   ;
   X   =   jody
```

It's telling us that there is information about a second woman in KB4, and (once again) it actually gives us the value that led to success. And of course, if we press ; a second time, Prolog returns the answer

```
   X   =   mia   ;
   X   =   jody   ;
   X   =   yolanda
```

But what happens if we press ; a third time? Prolog responds no. No other unifications are possible. There are no other facts starting with the symbol woman . The last four entries in the knowledge base concern the love relation, and there is no way that such entries can be unified with a query of the form woman(X) .

Let's try a more complicated query, namely

```
   ?-  loves(marsellus,X),  woman(X).
```

Now, remember that , means and , so this query says: is there any individual X such that Marsellus loves X and X is a woman ? If you look at the knowledge base you'll see that there is: Mia is a woman (fact 1) and Marsellus loves Mia (fact 5). And in fact, Prolog is capable of working this out. That is, it can search through the knowledge base and work out that if it unifies X with Mia, then both conjuncts of the query are satisfied. So Prolog returns the answer

```
   X   =   mia
```

The business of unifying variables with information in the knowledge base is the heart of Prolog. As we'll learn, there are many interesting ideas in Prolog — but when you get right down to it, it's Prolog's ability to perform unification and return the values of the variable bindings to us that is crucial.

Knowledge Base 5

Well, we've introduced variables, but so far we've only used them in queries. But variables not only can be used in knowledge bases, it's only when we start to do so that we can write truly interesting programs. Here's a simple example, the knowledge base KB5:

```
loves(vincent,mia).
loves(marsellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).

jealous(X,Y):-  loves(X,Z),  loves(Y,Z).
```

KB5 contains four facts about the loves relation and one rule. (Incidentally, the blank line between the facts and the rule has no meaning: it's simply there to increase the readability. As we said earlier, Prolog gives us a great deal of freedom in the way we format knowledge bases.) But this rule is by far the most interesting one we have seen so far: it contains three variables (note that X , Y , and Z are all upper-case letters). What does it say?

In effect, it is defining a concept of jealousy. It says that an individual X will be jealous of an individual Y if there is some individual Z that X loves, and Y loves that same individual Z too. (Ok, so jealousy isn't as straightforward as this in the real world.) The key thing to note is that this is a general statement: it is not stated in terms of mia , or pumpkin , or anyone in particular — it's a conditional statement about everybody in our little world.

Suppose we pose the query:

```
?-  jealous(marsellus,W).
```

This query asks: can you find an individual W such that Marsellus is jealous of W ? Vincent is such an individual. If you check the definition of jealousy, you'll see that Marsellus must be jealous of Vincent, because they both love the same woman, namely Mia. So Prolog will return the value

```
W  =  vincent
```

Prolog Syntax

Now that we've got some idea of what Prolog does, it's time to go back to the beginning and work through the details more carefully. Let's start by asking a very basic question: we've seen all kinds of expressions (for example jody , playsAirGuitar(mia) , and X) in our Prolog programs, but these have just been examples. It's time for precision: exactly what are facts, rules, and queries built out of?

The answer is terms, and there are four kinds of term in Prolog: atoms, numbers, variables, and complex terms (or structures). Atoms and numbers are lumped together under the heading constants, and constants and variables together make up the simple terms of Prolog.

Let's take a closer look. To make things crystal clear, let's first be precise about the basic characters (that is, symbols) at our disposal. The upper-case letters are A , B ,…, Z ; the lower-case letters are a , b ,…, z ; the digits are 0 , 1 , 2 ,…, 9 . In addition we have the _ symbol, which is called underscore, and some special characters , which include characters such as + , - , * , / , < , > , = , : , . , & , ~ . The blank space is also a character, but a rather unusual one, being invisible. A string is an unbroken sequence of characters.

Atoms

An atom is either:

1. A string of characters made up of upper-case letters, lower-case letters, digits, and the underscore character, that begins with a lower-case letter. Here are some examples: butch , big_kahuna_burger , listens2Music and playsAirGuitar .
2. An arbitrary sequence of characters enclosed in single quotes. For example 'Vincent ', 'The Gimp ', 'Five_Dollar_Shake ', '&^%&#@$ &* ', and '   '. The sequence of characters between the single quotes is called the atom name. Note that **we are allowed to use spaces in such atoms; in fact, a common reason for using single quotes is so we can do precisely that.**
3. A string of special characters. Here are some examples: @= and ====> and ; and :- are all atoms. As we have seen, some of these atoms, such as ; and :- have a pre-defined meaning.

Numbers

Real numbers aren't particularly important in typical Prolog applications. So although most Prolog implementations do support floating point numbers or floats (that is, representations of real numbers such as 1657.3087 or π) we say little about them in this book.

But integers (that is: …,-2, -1, 0, 1, 2, 3,…) are useful for such tasks as counting the elements of a list, and we'll discuss how to manipulate them in Chapter 5 . Their Prolog syntax is the obvious one: 23, 1001, 0, -365, and so on.

Variables

A variable is a string of upper-case letters, lower-case letters, digits and underscore characters that starts either with an upper-case letter or with an underscore. For example, X , Y , Variable , _tag , X_526 , List , List24 , _head , Tail , _input and Output are all Prolog variables.

The variable _ (that is, a single underscore character) is rather special. It's called the anonymous variable , and we discuss it in Chapter 4 .

Complex terms

Constants, numbers, and variables are the building blocks: now we need to know how to fit them together to make complex terms. Recall that complex terms are often called structures.

Complex terms are build out of a functor followed by a sequence of arguments. The arguments are put in ordinary parentheses, separated by commas, and placed after the functor. Note that the functor has to be directly followed by the parenthesis; you can't have a space between the functor and the parenthesis enclosing the arguments. The functor must be an atom. That is, variables cannot be used as functors. On the other hand, arguments can be any kind of term.

Now, we've already seen lots of examples of complex terms when we looked at the knowledge bases KB1 to KB5. For example, playsAirGuitar(jody) is a complex term: its functor is playsAirGuitar and its argument is jody . Other examples are loves(vincent,mia) and, to give an example containing a variable, jealous(marsellus,W) .

But the definition allows for more complex terms than this. In fact, it allows us to keep nesting complex terms inside complex terms indefinitely (that is, it is allows recursive structure). For example

```
   hide(X,father(father(father(butch))))
```

is a perfectly acceptable complex term. Its functor is hide , and it has two arguments: the variable X , and the complex term father(father(father(butch))) . This complex term has father as its functor, and another complex term, namely father(father(butch)) , as its sole argument. And the argument of this complex term, namely father(butch) , is also complex. But then the nesting bottoms out, for the argument here is the constant butch .

As we shall see, such nested (or recursively structured) terms enable us to represent many problems naturally. In fact, **the interplay between recursive term structure and variable unification is the source of much of Prolog's power**.

Arity:

The **number of arguments that a complex term has is called its arity**. For example, woman(mia) is a complex term of arity 1, and loves(vincent, mia) is a complex term of arity 2.

Arity is important to Prolog. Prolog would be quite happy for us to define two predicates with the same functor but with a different number of arguments. For example, we are free to define a knowledge base that defines a two-place predicate love (this might contain such facts as love(vincent,mia)), and also a three-place love predicate (which might contain such facts as love(vincent,marsellus,mia)). However, if we did this, Prolog would treat the two-place love and the three-place love as different predicates. Later we shall see that it can be useful to define two predicates with the same functor but different arity.

When we need to talk about predicates and how we intend to use them (for example, in documentation) it is usual to use a suffix / followed by a number to indicate the predicate's arity. To return to KB2, instead of saying that it defines predicates

```
listens2Music
happy
playsAirGuitar
```

we should really say that it defines predicates

```
listens2Music/1
happy/1
playsAirGuitar/1
```

And Prolog can't get confused about a knowledge base containing the two different love predicates, for it regards the love/2 predicate and the love/3 predicate as distinct.

Lab Task:

1. Are there any other jealous people in KB5?

2. Suppose we wanted Prolog to tell us about all the jealous people: what query would we pose? Do any of the answers surprise you? Do any seem silly?

3. Which of the following sequences of characters are atoms, which are variables, and which are neither?

   a. vINCENT
   b. Footmassage
   c. variable23
   d. Variable2000
   e. big_kahuna_burger
   f. 'big kahuna burger'
   g. big kahuna burger
   h. 'Jules'
   i. _Jules
   j. '_Jules'

4. Which of the following sequences of characters are atoms, which are variables, which are complex terms, and which are not terms at all? Give the functor and arity of each complex term.

   a. loves(Vincent,mia)
   b. 'loves(Vincent,mia)'

c. Butch(boxer)
d. boxer(Butch)
e. and(big(burger), kahuna(burger))
f. and(big(X), kahuna(X))
g. _and(big(X), kahuna(X))
h. (Butch kills Vincent)
i. kills(Butch Vincent)
j. kills(Butch,Vincent)

5. How many facts, rules, clauses, and predicates are there in the following knowledge base? What are the heads of the rules, and what are the goals they contain?

```
woman(vincent).
woman(mia).
man(jules).
person(X):-  man(X);  woman(X).
loves(X,Y):-  father(X,Y).
father(Y,Z):-  man(Y),  son(Z,Y).
father(Y,Z):-  man(Y),  daughter(Z,Y).
```

6. Represent the following in Prolog:
   a. Butch is a killer.
   b. Mia and Marsellus are married.
   c. Zed is dead.
   d. Marsellus kills everyone who gives Mia a foot massage.
   e. Mia loves everyone who is a good dancer.
   f. Jules eats anything that is nutritious or tasty.

7. Suppose we are working with the following knowledge base:

```
wizard(ron).
hasWand(harry).
quidditchPlayer(harry).
wizard(X):-  hasBroom(X),  hasWand(X).
hasBroom(X):-  quidditchPlayer(X).
```

How does Prolog respond to the following queries?

   a. wizard(ron).
   b. witch(ron).
   c. wizard(hermione).
   d. witch(hermione).
   e. wizard(harry).
   f. wizard(Y).

   g. witch(Y).