Tutorial SHEET-1

Introduction to SWI Prolog

1. Introduction

Prolog is a logic programming language associated with artificial intelligence and computational linguistics. There are three basic constructs in Prolog: facts, rules, and queries. A collection of facts and rules is called a knowledge base (or a database). 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 one finds interesting. Prolog program is used by posing queries. That is, by asking questions about the information stored in the knowledge base.

1.1. Entering Queries

1.1. Syntax

All the facts, rules and queries in prolog are built of **terms**.

There are four kinds of terms 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.

1. Atoms
   An atom is either:
   - 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. For example: `butch, big_kahuna_burger, and m_monroe2.`

- An arbitrary sequence of characters enclosed in single quotes. For example `’Vincent’, ’The Gimp’, ’Five_Dollar_Shake’, ’&^%&#@$ &*’,` and `’ ’`. The character 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.

- A string of special characters. For example: `@= and ====>` and `;` and `:-` are all atoms. Some of these atoms, such as `;` and `:-` have a predefined meaning.

2. Numbers
   Prolog supports floating point numbers as well as integers. Their Prolog syntax is the obvious one: `23, 1001, 0, -365`, and so on.

3. 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 underscore. For example, `X, Y, Variable, _tag, X_526`, and `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*.

4. Complex Terms
   Constants, numbers, and variables are the building blocks that fit together to make complex terms.

   Complex terms are built out of a **functor** followed by a sequence of **arguments**. The arguments are put in ordinary brackets, separated by commas, and placed after the functor. There can be more than one argument for a complex term. **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.**

For example, `playsAirGuitar(jody)` is a complex term: its functor is `playsAirGuitar` and its argument is `jody`. Other examples are `loves(vincent,mia)` and `jealous(marcellus,W)`. Here both the complex terms have two arguments and the latter has the second argument as a variable.

It also allows us to to keep nesting complex terms inside complex terms indefinitely (that is, it is a *recursive definition*). For example `hide(X,father(father(father(butch))))` is a perfectly ok 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 so on.

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

1.2. Creating Knowledge Base and Entering Queries

1.2.1. Example 1

Knowledge Base can be created in a Prolog file, which can be created using any text editor.

The facts and rules contained in a knowledge base are called **clauses**. **Facts** are used to state things that are *unconditionally true* of the domain of interest.

For example, we can state that Mia, Jody, and Yolanda are women, and that Jody plays air guitar, using the following four facts:

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

This collection of facts is a knowledge base. Let it be named KB1. It is an example of a Prolog program. Note that the names `mia, jody`, and `yolanda`, and the properties `woman` and `playsAirGuitar`, have been written so that the first letter is in lower-case.
Save the file with the `.pl` extension in the working directory of Prolog as `kb1.pl`

For executing queries, open SWI Prolog console. Prolog will display its prompt, something like
```
?-
```
which indicates that it is ready to accept a query.

Type in the command listing, followed by a full stop, and hit return. That is, type
```
?- listing.
```
and press the return key.

Now, the listing command is a special in-built Prolog predicate that instructs Prolog to display the contents of the current knowledge base. But we haven't yet told Prolog about any knowledge bases, so it will just say
```
yes
```
This is a correct answer: as yet Prolog knows nothing — so it correctly displays all this nothing and says yes. You may get a little more (for example, the names of libraries that have been loaded).

Note that all queries end with a (.) full stop.
To execute queries on a domain of interest, you need a knowledge base. To create a knowledge base, first you need to know syntax.

Now to load the knowledge base, in the Prolog console type
```
?- [kb1].
```
This tells Prolog to consult the file `kb1.pl`, and load the contents as its new knowledge base.
If the file does not contain any errors, Prolog will read it in and print out a message saying that it is consulting the file `kb1.`pl, and then answer:
```
yes
```

Sometimes, Prolog may give an error message if the file is not found. You can change the working directory to where the file is stored by typing:

```
?- working_directory(CWD,'NewPath').
```

Try the `listing` command again. Prolog will now also display the contents of the loaded file.

`listing` can be used in other ways. For example, typing

```
?- listing(playsAirGuitar).
```

simply lists all the information in the knowledge base about the `playsAirGuitar` predicate

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 the knowledge base.

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
```

This is because it is not a fact in KB1. Moreover, KB1 is extremely simple, and contains no other information (such as the rules) which might help Prolog try to **infer** (that is, **deduce** whether Mia plays air guitar. So Prolog correctly concludes that `playsAirGuitar(mia)` does not follow from KB1.

Similarly, suppose we pose the query:

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

The answer will be

```
ERROR: Unknown procedure: tattooed/1 (DWIM could not correct
goal)
```

Not only can we use Prolog to deduce or infer whether a fact is true or false, but we can also use Prolog to find out which

1.2.2. Example 2

Now let us make a new knowledge base KB2 with

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

KB2 contains two facts, `listensToMusic(mia)` and `happy(yolanda)`. The last three items are rules.

Rules state information that is *conditionally true* of the domain of interest. For example, the first rule says that Mia plays air guitar if she listens to music, and the last rule says that Yolanda listens to music *if* she is happy. 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`.

We can view a fact as a rule with an empty body.

Let's consider an example. We will ask Prolog whether Mia plays air guitar:

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

Prolog will respond "yes". Why? Well, although `playsAirGuitar(mia)` is not a fact explicitly recorded in KB2, KB2 does contain the rule

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

Moreover, KB2 also contains the fact `listensToMusic(mia)`. Hence Prolog can use **modus ponens** to deduce that `playsAirGuitar(mia)` is true.

Suppose we ask:

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

Prolog would respond “yes”. It uses the *fact* `happy(yolanda)` and the *rule*
`listensToMusic(yolanda):- happy(yolanda),`
Prolog can deduce the new fact `listensToMusic(yolanda)`. This new fact is not explicitly recorded in the knowledge base — it is only *implicitly present* (it is inferred knowledge). Thus Prolog can chain together uses of modus ponens.

KB2 consists of three **predicates** (or **procedures**). The three predicates are:

```
listensToMusic
happy
playsAirGuitar
```

The `happy` predicate is defined using a single clause (a fact). The `listensToMusic` and `playsAirGuitar` predicates are each defined using two clauses (in both cases, two rules). 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 interrelated.

Prolog allows 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,marcellus,mia)`).
However, if we did this, Prolog would treat the two place `love` and the three place `love` as completely different predicates.

When we need to talk about predicates 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

```
listensToMusic
happy
playsAirGuitar
```

we should really say that it defines predicates

```
listensToMusic/1
```

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

1.2.3. Example 3

Let KB3 be another knowledge base:

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

Here, there are two facts, namely `happy(vincent)` and `listensToMusic(butch)`, and three rules.
KB3 defines the same three predicates as KB2 (namely `happy`, `listensToMusic`, and `playsAirGuitar`)
Note that the rule

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

has two items in its body, or (to use the standard terminology) two goals. Here, the comma `,` that separates the goal `listensToMusic(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 information `listensToMusic(vincent)`, and this fact cannot be deduced either.

So KB3 only fulfils one of the two preconditions needed to establish `playsAirGuitar(vincent)`, and our query fails.

The spacing used in this rule around the comma is irrelevant.

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

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

This is a way of stating that Butch plays air guitar if either 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” even though the first of these rules will not help (KB3 does not allow Prolog to conclude that `happy(butch)`), KB3 does contain `listensToMusic(butch)` and this means Prolog can apply modus ponens using the rule

```
playsAirGuitar(butch):-
     listensToMusic(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);
     listensToMusic(butch).
```

The semicolon `;` is the Prolog symbol for OR. But Prolog programmers usually write multiple rules, as extensive use of semicolon can make Prolog code hard to read.

1.2.4 Example 4

KB4 is as follows:

```
woman(mia).
woman(jody).
woman(yolanda).
loves(vincent,mia).
loves(marcellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).
jealous(X,Y) :- loves(X,Z),loves(Y,Z).
```

KB4 contains three facts about `woman`, four facts about the `loves` relation and one rule. It also contains three variables: `X`, `Y` and `Z`.

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.

We can also use variables to enter queries such as:
```
?- woman(X).
```
This query essentially 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 match (or unify) the expression `woman(X)` with the information KB4 contains. Now the first item in the knowledge base is `woman(mia)`. So, Prolog matches `X` to mia, thus making the query agree perfectly with this first item. 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
```
It actually gave us the **variable binding, or instantiation** that led to success.

There is information about other women in the knowledge base. We can access this information by typing the following simple query
```
?- ;
```
So it responds:
```
X = jody
```
If we press ; a second time, Prolog returns the answer
```
X = yolanda
```

Since no further responses are possible, Prolog replies with ‘no’ if ; is pressed a third time.

Another query,

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

gives `X = mia` as the only answer.

Suppose we pose the query:

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

This query asks: can you find an individual W such that Marcellus is jealous of W?

Prolog will return the value

```
W = vincent
```

2. <u>Exercises</u>

1. Represent the following in Prolog:
    1) Butch is a killer.
    2) Mia and Marcellus are married.
    3) Zed is dead.
    4) Marcellus kills everyone who gives Mia a footmassage.
    5) Mia loves everyone who is a good dancer.
    6) Jules eats anything that is nutritious or tasty.

2. Given 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?

```
1)wizard(ron).
2)witch(ron).
3)wizard(hermione).
```

```
4)witch(hermione).
5)wizard(harry).
6)wizard(Y).
7)witch(Y).
```