LAB # 8

Statement Purpose:

This lab will introduce students to Prolog. Prolog is a programming language, but a rather unusual one. ``Prolog'' is short for ``Programming with Logic'', and the link with logic gives Prolog its special character. At the heart of Prolog lies a surprising idea: don't tell the computer what to do. Instead, describe situations of interest, and compute by asking questions. Prolog will logically deduce new facts about the situations and give its deductions back to us as answers.

Activity Outcomes:

This lab teaches you the following topics:

- How to install and use the interface of Prolog
- How to create a simple Knowledge Base How
- to use simple queries

Instructor Note:

As pre-lab activity, read Chapter 1 from the book (Learn Prolog Now, Vol 7, by BlackBurn et. al.,) to know the basics of prolog programming.

1) Stage J (Journey)

Introduction:

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.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

How to install Prolog?

Solution:

Go to http://www.swi-prolog.org/ and download the standalone installer for your platform. After installing, go to /bin directory and open SWI-Prolog which will show the command prompt like interface as shown below,

Activity 2:

How to create a simple Knowledge Base?

Solution:

Go to the File menu and click on New. Type the filenames like test and click OK. This will create a new file name test.pl (every file in prolog is saved with .pl extension).

Activity 3:

How to add facts in a Knowledge Base?

Solution:

Knowledge Base 1 (KB1) is simply a collection of facts. 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).
```

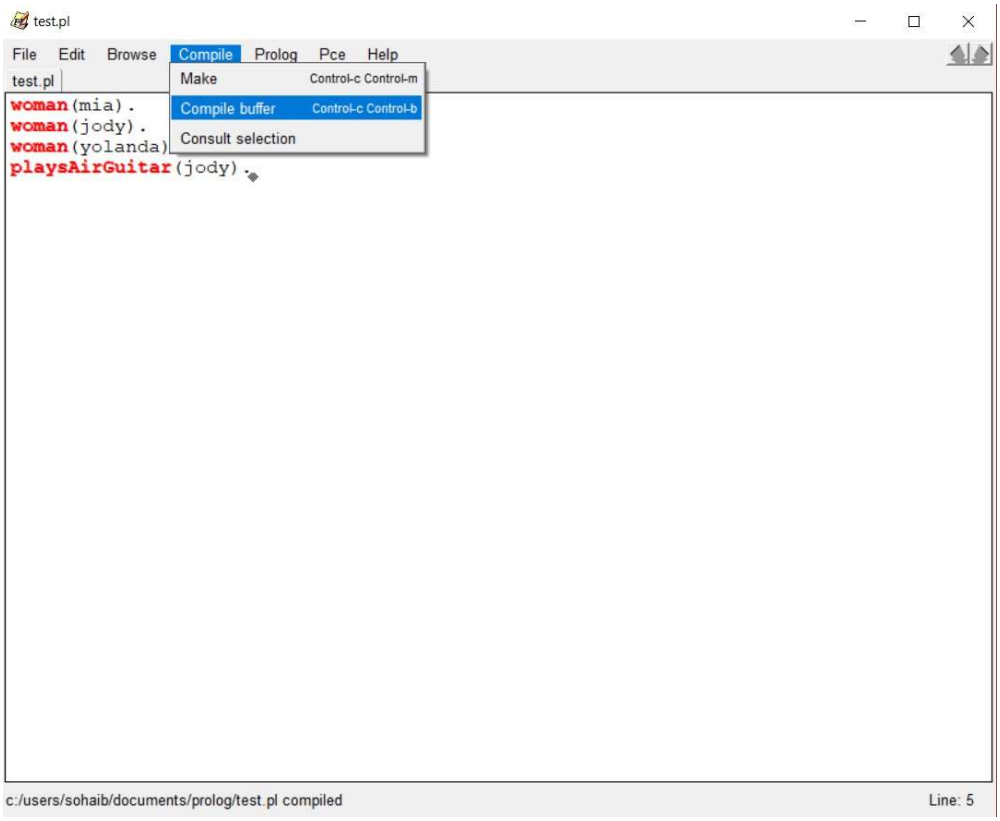
Do not forget to enter a sentence with a period (.).

Activity 3:

How to run a query within a Knowledge Base?

Solution:

How can we use KB1? By posing queries. That is, by asking questions about the information KB1 contains. But first compile the program using compile option as shown in the following figure,



Now we can ask Prolog whether Mia is a woman by posing the query in the command prompt.

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

Prolog will answer `true` 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).

Activity 3:

How to add rules alongside facts in a Knowledge Base?

Solution:

Here is KB2, our second knowledge base:

```
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 if 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`.

This fundamental deduction step is what logicians call modus ponens.

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)`.

Our next example shows that Prolog can chain together uses of modus ponens. Suppose we ask: `?- playsAirGuitar(yolanda).` Prolog would respond "yes". Why? Well, using 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). Nonetheless, Prolog can then use it just like an explicitly recorded fact. Thus, together with the rule `playsAirGuitar(yolanda) :- listensToMusic(yolanda)` it can deduce that `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:

```
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). 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"

Activity 4:

Use conjunction based rules in a Knowledge Base?

Solution:

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

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

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

has *two* items in its body, or (to use the standard terminology) two goals. What does this rule mean? The important thing to note is 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. Incidentally, the spacing used in this rule is irrelevant. For example, we could have written it as `playsAirGuitar(vincent):- happy(vincent),listensToMusic(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):-
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". For although 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).
```

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. But Prolog programmers usually write multiple rules, as extensive use of semicolon can make Prolog code hard to read. It should now be clear that Prolog has something do with logic: after all, the `:-` means implication, the `,` means conjunction, and the `;` means disjunction. (What about negation? That is a whole other story. We'll be discussing it later in the course.) Moreover, we have seen that a standard logical proof rule (modus ponens) plays an important role in Prolog programming. And in fact "Prolog" is short for "Programming in logic".

3) Stage v (verify)

Activity 1:

Create a knowledge base which defines your family tree and make a query that uses application of modus ponens to derive a fact which is not explicitly elaborated in the knowledge base.

4) Stage a2 (assess)

Assignment:

Activity 1:

Submit the above activity as your assignment.