Unit 14 Programming in Prolog

Structure:

14.1 Introduction

In the previous unit, you learnt basics of robotics along with the contemporary uses and the hardware required for robots. Prologis a logic programming language widely utilized in Artificial Intelligence. In this unit you will be introduced to basics of programming language the Prolog. Prolog is a general purpose language associated with artificial intelligence and computational linguistics. Prolog is the major example of a fourth generation programming language supporting the declarative programming paradigm. The Japanese Fifth-Generation Computer Project, announced in 1981, adopted Prolog as a development language, and thereby focused considerable attention on the language and its capabilities.

Objectives:

After studying this unit, you should be able to

- define Prolog
- list the facts and rules in prolog
- describe the basic data structure and syntax of Prolog
- explain Backtracking and Recursion
- list and explain the features of Prolog.

14.2 Basics of Prolog

In this section, you will learn the basics of Prolog. It is a high-level programming language which enables the user to build programs by stating what they want the program to do rather than how it should do it. Prolog is a logical and a declarative programming language. The name itself, Prolog, is short for PROgramming in LOGic. Prolog's heritage includes the research on theorem provers and other automated deduction systems developed in the 1960s and 1970s. The inference mechanism of Prolog is based upon Robinson's resolution principle (1965) together with mechanisms for extracting answers proposed by Green (1968). These ideas came together forcefully with the advent of linear resolution procedures.

Having its roots in formal logic, and unlike many other programming languages, Prolog is declarative: The program logic is expressed in terms of relations, and execution is triggered by running *queries* over these relations. Relations and queries are constructed using Prolog's single data type, the *term*. Relations are defined by *clauses*. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications.

The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s, while the first compiler was written by David H. D. Warren in Edinburgh, Scotland. Prolog was one of the first logic programming languages, and remains among the most popular such languages today, with many free and commercial implementations available. While initially aimed at natural language processing, the language has since then stretched far into other areas like theorem proving, expert systems, games, automated answering systems, ontology and sophisticated control systems, and modern Prolog environments support the creation of graphical user interfaces, as well as administrative and networked applications.

A prolog program consists of a set of facts and a set of rules. There are no type declarations, initializations or any other stuff like that, just some facts and rules.

Some prolog facts are:

```
lectures(amita, ai).
lectures(john, databases).
female(amita).
age(amita, 29).        [Oh, OK, that was last year..]
office(amita, s134).
animal(lion).
animal(sparrow).
has_feathers(sparrow).
```

Facts consist of:

- A predicate name (or *functor*) such as lectures, female, and office. This must begin with a lower case letter.
- Zero or more arguments, such as alison, ai3, and s134.

Note that facts (and rules, and questions) must end with a full stop.

Some prolog rules are:

```
bird(X) :-
  animal(X),
  has_feathers(X).

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

You should read the prolog operator ":-'' as "if'', while ",'' can be read as meaning "and''. So the first rule says that "X is a bird if X is an animal and X has feathers'', while the second rule says that "Y is X's grandparent if Z is X's parent and Y is Z's parent".

All arguments beginning with a capital letter (such as X and Y) are variables. (Note that variables are NOT treated in the same manner as in conventional programming languages - for example, they don't have to have values). Any constant should NOT begin with a capital letter else it will be treated as a variable. So, given the fact capital (India, Delhi) both arguments would be treated as unbound variables.

“Running’’ a prolog program involves asking Prolog questions (having loaded in your set of facts and rules). For example, you could ask:

```
?- lectures(amita, ai).
```

And prolog would give the answer “yes’’. If we ask a sequence of questions we might get:

```
?- lectures(amita, ai).
yes
?- lectures(amita, databases).
no
```

Questions can have variables in them, which may get *instantiated* (i.e., get bound to particular values) when prolog tries to answer the question. Prolog will display the resulting bindings/instantiations of all the variables in the question. So we might have:

```
?- lectures(amita, Course).
Course = ai
```

We can also ask the question the other way around, like:

```
?- lectures(Someone, ai).
Someone = amita
```

Note that the variables Course and Someone can both take values of any type.

We can find out who lectures what by asking:

```
?- lectures(Someone, Something).
Someone = amita
Something = ai ;
Someone = john
Something = databases ;
no
```

By typing a semicolon (or, in Mac Prolog, clicking on “next’’) after Prolog prints out the first set of bindings, we can see if there are any other possible bindings. Prolog systematically goes through all its facts and rules and tries to find all the ways it can associate variables with particular values so that the initial query is satisfied. However, as an example of how rules are used, suppose we ask the question:

```
?- bird(B).
```

and we have the facts and rule:

```
animal(lion).
animal(sparrow).
has_feathers(sparrow).

bird(X) :-
  animal(X),
  has_feathers(X).
```

Prolog will respond with

```
B = sparrow.
```

Prolog *matches* bird(B) against the *head* of the rule (bird(X)), and sets as new questions first animal(B) and then has_feathers(B). Animal(B) can be satisfied by binding B to lion. However, has_feathers(lion) isn't true, so that doesn't work. Prolog goes back (*backtracks*) and tries B = sparrow. has_feathers(sparrow) is true, so that all works and prolog returns with B = sparrow as a possible solution.

Self Assessment Questions

1. Prolog is the major example of a _____ generation programming language supporting the declarative programming paradigm.
2. The inference mechanism of Prolog is based upon Robinson's _____ principle.
3. All arguments beginning with a capital letter are variables. (State True or False?)

14.3 Basic Data Structure and Syntax of Prolog

We have already seen how prolog programs consist of facts, rules and questions. Facts declare things that are always true. Rules declare things that are true depending on some conditions. They consist of a head and a body. For example the rule “a(X) :- b(X), c(X).’’ has head “a(X)’’ and body “b(X), c(X)’’. Its head is true if the goals in its body are all true. (Facts can be viewed as rules without a body). Facts and/or rules can be used to define *relations* (or *predicates*). These can be of different *arities* (ie, have different numbers of arguments). For example, lectures/2 is the binary relation lectures, and might be defined by the three facts lectures (amita, ai), tt lectures (manav, databases), and lectures (shuchita, hci).

Questions are used to find out if something is true (and what the associated variable bindings would be to make it true). Facts, rules and questions are all prolog *clauses* and must end with a full stop.

The fundamental data structure in Prolog is the ***term***. i.e., everything including program and data is expressed in form of term. Prolog terms include:***Atoms*** such as sparrow, x, 'Alison Cawsey', a102,-. These generally represent specific single entities in the world and cannot be separated into parts. They normally begin with a lower-case letter, but arbitrary characters can be used if they are quoted. Symbols (such as " – '') and sequences of symbols (e.g., ":- '') are also atoms.
Numbers such as 29, 1.2, 3000, -2.
Variables such as X, Person, _Var. Variables begin either with a capital letter or an underline character.
Structured Objects:
such as
book(title(lord_of_the_rings), author(tolkien))
bicycle(owner(alison), parts(gears(number(18), type(shimano))))

These consist of a *function* (e.g., book, bicycle), and some arguments. The arguments may in turn be any prolog term.

Complex structures like the ones above are useful if we want to carry around some information about a related collection of objects, or an object with many parts, for example. We can get at the parts of a complex term by matching it against a query. So, if we had the following Prolog program:

```
book(title(lord_of_the_rings), author(tolkien)).

has_famous_author(Title) :-
  book(title(Title), author(Author)),
  famous(Author).
```

and asked the query:

```
?- has_famous_author(lord_of_the_rings).
```

Prolog would first MATCH the query with the HEAD of the rule, so that Title is bound to lord_of_the_rings. Then it would try to satisfy book(title(lord_of_the_rings), author(Author)). This would be matched with the first fact, and Author would be bound to tolkien. Then it would try to satisfy famous(tolkien), which would match the second fact, so it would all be satisfied, and Prolog would return with yes.

More about Prolog Matching

As we have seen, Prolog tries to prove (answer) a query by looking for facts which match that query, or rules whose heads match the query and whose body can be proved. The way prolog matches terms is therefore crucial.

We have already seen how simple expressions such as lectures (amita,Course) can be matched to facts such as lectures (amita, ai) resulting in bindings such as Course=ai. In the above section we saw examples of a slightly more complex match, where terms with complex arguments were matched against each other, so

book(title(Name), author(Author))

matches:

book(title(lord_of_the_rings), author(tolkien))

with bindings:

Name = lord_of_the_rings
Author = tolkien

Prolog matches expressions in a purely structural way, with no evaluation of expressions, so:

```
?- 1+2 = 3
no
```

Note: In prolog "='' means "matches with''. You can test matches by typing in queries such as the one above.

Similarly the following won't match, as the two expressions have different structures:

```
?- X + 2 = 3 * Y.
no
```

However, the following match as they have the same structure:

```
?- X+Y = 1+2.
X = 1
Y = 2

?- 1+Y = X + 3.
X = 1
Y = 3
```

Note that there can be variables in both sides of a matched expression, as in the second example above.

For non-arithmetic examples, we have:

```
?- lectures(X, ai) = lectures(amita, Y).
X = amita
Y = ai

?- book(title(X), author(Y)) = book(Z, author(tolkien)).
Z = title(X)
Y = tolkien
```

Of course, prolog will normally be doing lots of matches in a row, as it tries to prove different subgoals in a rule. It then needs to use the variable bindings obtained in the matches so far when it does the next match. So we might have:

```
?- X+Y = 1+5, X=Y.
no
```

Note: We can ask several queries in a row, with a comma in between them, just like in the body of a Prolog rule.

X and Y get instantiated (bound) to 1 and 5 respectively in the first match (X+Y = 1+5), so the second match is effectively 1=5, which fails. However, the following will succeed:

```
?- X+Y = 1+5, Z = X.
Z = 1
Y = 5
X = 1

?- book(author(X)) = book(author(tolkien)),
   famous(X) = famous(tolkien).
X = tolkien

?- X=Y, Y = amita.
X = amita
Y = amita
```

Note that if two uninstantiated (unbound) variables are matched (and therefore instantiated to each other) then as soon as one becomes

instantiated to some term then the other automatically becomes instantiated to that term.

The algorithm for prolog's matching process is based on the *unification* algorithm proposed for automated theorem proving. Two terms match if you can instantiate variables to values in such a way that, if the variables in both terms were replaced by their instantiations, the two expressions would become identical.

14.4 Backtracking

At this point we should go through in more detail how Prolog answers queries (and therefore runs programs). Given a query to prove (answer), Prolog goes down its list of facts and rules, from top to bottom, looking for facts or rule heads which match the query (given any existing variable bindings). When it finds one, it stores the place in the fact/rule base it had got to in its search, so if the first match it finds is no good it can carry on and look for more possibilities. So, suppose we have the facts:

```
bird(type(sparrow), name(steve))).
bird(type(penguin), name(tweety))).
bird(type(penguin), name(percy)).
```

and give the query ?- bird(type(penguin), name(X)). Prolog will first try matching the query with the first fact, but fail because sparrow doesn't match penguin. It will then try matching with the second fact, and succeed with X = tweety. However, it will put a pointer to the place it got to, so if it turns out that a later query/subgoal fails, it will go back to the saved position, and look for more solutions (ie, X = percy).

Similarly, the leftmost expression is tried first, then the second expression from the left (using whatever variable matches were found for the first) and so on. So predicate expressions in a query are initially done in order, like lines of a program in a conventional language like Pascal.

But suppose that a predicate expression fails--that is, no fact matching it can be found. If the expression has variables that were bound earlier in the query line, the fault may just be in the bindings. So the interpreter automatically *backtracks* (goes back to the immediately previous expression in the query) and tries to find a different fact match. If it cannot, then *that*

predicate expression fails and the interpreter backtracks to the previous one, and so on.

Anytime the Prolog interpreter cannot find another matching for the leftmost expression in a query, then there's no way the query could be satisfied; it types out the word **no** and stops. Anytime on backtracking it can find a new matching for some predicate expression, it resumes moving right from there as it did originally.

The purpose of backtracking is to give “second chances’’ to a query, by revising earlier decisions. Backtracking is very important in artificial intelligence, because many artificial-intelligence methods use intelligent guessing and following of hunches. Guesses may be wrong, and backtracking is a good way to recover then.

Self Assessment Questions

4. _____ declare things that are true depending on some conditions.
5. The algorithm for prolog’s matching process is based on the *Non unification* algorithm proposed for automated theorem proving.(State True or False?)
6. The purpose of backtracking is to give _____ to a query, by revising earlier decisions.

14.5 Recursion

Almost any non-trivial prolog program involves recursive predicates - predicates that call themselves. The basic idea should be familiar from recursive function definitions in functional languages. However, as Prolog is not based on function application, the way recursive predicates are used and written is slightly different.

Suppose we want to write a Prolog procedure to determine whether someone is an ancestor of someone else. This has a natural recursive definition. X is Y’s ancestor if X is Y’s parent OR Z is Y's parent and X is Z's ancestor. This can be written as follows:

```
ancestor(Person, Ancestor) :-      % Rule 1: Base case
   parent(Person, Ancestor).

ancestor(Person, Ancestor) :-      % Rule 2: Recursive case
   parent(Person, Parent),
   ancestor(Parent, Ancestor).
```

Note how the *base case* of the recursive definition is a separate rule, preceding the recursive case. All recursive predicates must have a base case, otherwise they would either fail or recurse for ever.

Consider what happens if we also have the following facts:

```
parent(alison, david).        % fact 1
parent(alison, kathleen).     % fact 2
parent(david, harold).        % fact 3
parent(david, ida).
parent(kathleen, john).       % fact 5
```

and we ask:

```
?- ancestor(alison, harold).
```

Prolog will match the query against rule 1, and try to prove parent(alison, harold). This will fail, so Prolog will backtrack and try the second rule. parent(alison, Parent) first succeeds with Parent = david, so Prolog tries to prove ancestor(david, harold). Matching this new term against the head of the first rule, and successfully proving parent(david, harold), the whole query succeeds.

14.6 Arithmetic and Lists in Prolog

Two additional features of Prolog are arithmetic and lists. These give rules new capabilities. As we've seen already, rules can:

1) define new predicates in terms of existing predicates
2) extend the power of existing predicates (as with inheritance rules)
3) recommend what to do in a situation (as with the traffic lights program)

To these, we'll now add:

4) quantify and rank things
5) store, retrieve, and manipulate sets and sequences of data items

14.6.1 Arithmetic comparisons

Prolog has built-in arithmetic comparison predicates. They're written in the *infix* notation of mathematics. The predicate name comes between the arguments, like this:

3 > 4 means 3 is greater than 4
15 = 15 means 15 equals 15
X < Y means **X** is less than **Y**

Z >= 4 means **Z** is greater than or equal to 4
PPPP =< 3 means PPPP is less than or equal to 3

We'll usually put spaces around infix symbols to make them easier to see, but it's not required. As an example, here's the definition of a predicate that checks if a number is positive:

```
positive(X) :- X > 0.
```

With this definition in our database, it could be used like this:

```
?- positive(3).
yes
?- positive(-6).
no
```

Here's the definition of a predicate that checks if its first argument is a number lying in the range from its second to its third argument, assuming all arguments are bound:

```
in_range(X,Y,Z) :- X >= Y, X =< Z.
```

Using this definition, the query

```
?- in_range(3,0,10).
```

gives the response **yes**.

14.6.2 Arithmetic assignment

Like any computer language, Prolog has arithmetic computations and assignment statements. Arithmetic assignment is done by expressions with the infix **is** predicate. Querying these peculiar expressions has the side effect of binding some variable to the result of some arithmetic computation. For instance

```
X is ( 2 * 3 ) + 7
```

binds (assigns) **X** to the value 13 (2 times 3 plus 7). The thing to the left of the **is** must be a variable name, and the stuff to the right must be an algebraic formula of variables and numeric constants, something that evaluates to a number. The algebraic formula is written in standard infix form, with operations **+** (addition), **-** (subtraction), ***** (multiplication), and **/** (division). We'll often put spaces around these symbols to make them more readable. The algebraic formula can have variables only if they're bound to values, as in

```
Y is 2, X is Y * Y.
```

where **Y** is first bound to 2, and then **X** is bound to 4. A practical example is this definition of the square of a number, intended to be a function predicate:
square(X,Y) :- Y is X * X.
If this rule is in the Prolog database, then if we query
?- square(3,Y).
(that is, if we ask what the square of 3 is), the Prolog interpreter will type
Y=9

Notice that since predicate expressions aren't functions in Prolog, we can't write anything like
f(X,Y) + g(X,Z)
even if **f** and **g** are function predicates, because expressions only succeed or fail; expressions don't have "values". Instead, to add the two function values we must say something like
f(X,Y), g(X,Z), T is Y + Z

Another warning: don't confuse **=** with **is**. The **=** is a purely logical comparison of whether two things are equal. (Originally intended for numbers, it also works for words.) The **is** is an operation, an arithmetic assignment statement that figures out a value and binds a variable to it.

Reversing the "is''

A serious weakness of arithmetic, which makes it different from everything else in Prolog we've talked about so far, is that it isn't multi-way or reversible. For instance, if we have the preceding definition of **square** in our database, and we query

?- square(X,9).

wondering what number squared is 9, the interpreter will refuse to do anything because the right side of the **is** statement refers to an unbound variable. This is different from having a bunch of arithmetic facts in *prefix* form like
square(0,1).
square(1,1).
square(2,4).
square(3,9).
for which we could query **square(3,Y)** or **square(X,9)** or even **square(X,Y)** and get an answer. Similarly, for the preceding definition of **positive**, the query

?- positive(X).
Won't work: the interpreter can only do a **>** comparison when both things are bound to numbers. So it will complain and refuse to do anything.

14.6.3 Lists in Prolog

Another important feature of Prolog is linked-lists. Every argument in a predicate expression in a query must be anticipated and planned for. To handle sets and sequences of varying or unknown length, we need something else: linked-lists, which we'll henceforth call just *lists*.

Lists have always been important in artificial intelligence. Lisp, the other major artificial intelligence programming language, is almost entirely implemented with lists--even programs are lists in Lisp.

Square brackets indicate a Prolog list, with commas separating items. For example:
[monday,tuesday,wednesday,thursday,friday,saturday,sunday]

(Don't confuse square brackets "**[ ]**" with parentheses "**( )**"; they're completely different in Prolog. Brackets group lists and parentheses group arguments.) Lists can be values of variables just like words and numbers. Suppose we have the following facts:

weekdays([monday,tuesday,wednesday,thursday,friday]).
weekends([saturday,sunday]).

Then to ask what days are weekdays, we type the query
?- weekdays(Days).

and the answer is
Days=[monday,tuesday,wednesday,thursday,friday]

We can also bind variables to items of lists. For instance, if we query
?- weekends([X,Y]).
with the preceding facts in the database, we get
X=saturday, Y=sunday

But that last query requires that the weekends list have exactly two items; if we query
?- weekends([X,Y,Z]).
we get **no** because the query list can't be made to match the data list by some binding.

We can work with lists of arbitrary length by the standard methods for linked-pointer list manipulation. We can refer to any list of one or more items as **[X|Y]**, where **X** is the first item and **Y** is the rest of the list (that is, the list of everything but the first item in the same order). In the language Lisp, **X** is called the *car* and **Y** is called *cdr* of the list. We'll call "**|''** the *bar* symbol. Note that **[X|Y]** is quite different from **[X,Y]**; the first can have any nonzero number of items, whereas the second must have exactly two items. Note also that **X** and **Y** are different data types in **[X|Y]**; **X** is a single item, but **Y** is a list of items. So **[X|Y]** represents an uneven division of a list.

Here are some examples with the previous weekdays and weekends facts.

?- weekdays([A|L]).
A=monday, L=[tuesday,wednesday,thursday,friday]
?- weekdays([A,B,C|L]).
A=monday, B=tuesday, C=wednesday, L=[thursday,friday]
?- weekends([A,B|L]).
A=saturday, B=sunday, L=[ ]
The "**[ ]''** is the list of zero items, the *empty list,* called *nil* in the language Lisp.

Self Assessment Questions

7. All recursive predicates must have a _____ case.
8. Two additional features of Prolog are _____ and _____.
9. Lists can be values of variables just like words and numbers.(Say Yes or No?

14.7 Summary

In this unit you have learnt the basics of programming language called Prolog. You have learnt the facts and rules of prolog. We also discussed the concept of backtracking, recursion and arithmetic and lists in prolog. Prolog is a logical and a declarative programming language. The program logic is expressed in terms of relations, and execution is triggered by running queries over these relations. A prolog program consists of a set of facts and a set of rules. Facts declare things that are always true. Rules declare things that are true depending on some conditions. The fundamental data structure in Prolog is the term. Almost any non-trivial prolog program

involves recursive predicates - predicates that call themselves. Lists have always been important in artificial intelligence.

14.8 Terminal Questions

1. What is Prolog? Give prolog facts and rules.
2. Explain data structure in Prolog.
3. What is purpose of backtracking? Why is it important in artificial intelligence?
4. Write a note on Recursion.
5. Explain Arithmetic and list in Prolog

14.9 Answers

Self Assessment Questions

1. fourth
2. resolution
3. True
4. Rules
5. False
6. second chances
7. base
8. arithmetic, lists
9. Yes

Terminal Questions

1. Prolog is a logical and a declarative programming language. (Refer section 14.2 for details)
2. **Term** is a basic data structure in Prolog. (Refer section 14.3 for detail)
3. The purpose of backtracking is to give “second chances’’ to a query, by revising earlier decisions. (Refer section 14.4 for detail)
4. Almost any non-trivial prolog program involves recursive predicates - predicates that call themselves. (Refer section 14.5 for detail)
5. Two additional features of Prolog are arithmetic and lists. These give rules new capabilities. Prolog has built-in arithmetic comparison predicates. To handle sets and sequences of varying or unknown length, we need *lists*. (Refer section 14.6 for detail)

Acknowledgements, References and Suggested Readings

1. Jackson, P. (1992). *Introduction to Expert Systems*. M. A.: AWP.
2. Nilsson, N. J. (1990). *Principles of AI*. Narosa Publ. House.
3. Patterson, D. W. (1992). *Introduction to AI and Expert Systems.* PHI.
4. Ramani, S., & Sasikumar, M. (1994). *Rule Based Expert Systems*. Narosa Publishing House.
5. Rich, E., & Knight, K. (1991). *Artificial Intelligence*, 2nd Edition.T.M.H.
6. Schalkoff, R. J. (1992). *Artificial Intelligence- an Engineering Approach*. Singapore: McGraw Hill Int. Ed.

E-References:

1. http://www.humansfuture.org/artificial_intelligence_tutorial.php.htm
2. http://www.learnartificialneuralnetworks.com/ai.html
3. http://www.iep.utm.edu/art-inte/
4. http://www.ru.lv/~peter/zinatne/ebooks/Artificial.Intelligence.and.Expert.Systems.for.Engineers.pdf
5. http://www.scribd.com/doc/22001035/Artificial-Intelligence-and-Knowledge-Representatio
6. http://pcquest.ciol.com/content/technology/2008/108080201.asp