GURU TEGH BAHADUR INSTITUTE OF TECHNOLOGY

LAB MANUAL

Subject: Principles of Artificial Intelligence Lab

(Based on Prolog)

Paper Code: AIML-257

DEPARTMENT OF

AIML

What is Prolog?

Prolog is a logic programming language that is used in artificial intelligence. It is a declarative programming language expressing logic as relations, called facts and rules. A Prolog program consists of a collection of facts and rules; a query is a theorem to be proved. Here are some basic elements of Prolog:

Facts: These are statements that are true. They are written as a predicate followed by a period. For example, `likes(john, mango).` is a fact that states that John likes mango.

Rules: These are statements that define relationships between facts. They are written as a predicate followed by a body, which is a list of one or more predicates separated by commas and enclosed in parentheses, followed by a period. For example,

Grandfather(X, Y) :-father(X,Z),parent(Z,Y)

This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be the father of Z.

Queries: These are questions that we ask Prolog. They are written as a predicate followed by a question mark. For example, `?- likes(john, mango).` is a query that asks if John likes mango.

Key features of Prolog:

The key features of Prolog are as follows:

1. **Declarative Language:** Prolog is a declarative programming language, meaning that programs are written as sets of logical statements. This makes Prolog programs concise and easy to read.

2. **Predicate Calculus**: Prolog uses the language of predicate calculus, which allows for the representation of relationships between facts and rules.

3. **Handling Lists and Recursion**: Prolog naturally handles lists and recursion, making it well-suited for tasks that involve these concepts.

4. **Unification**: Prolog uses unification, which is the process of determining if given terms can represent the same structure. Unification is a fundamental concept in Prolog.

5. **Backtracking**: When a task fails, Prolog traces backward and tries to satisfy the previous task. This backtracking feature allows for flexible problem-solving.

6. **Efficiency**: Prolog is known for its efficiency in solving problems that would be difficult in other programming languages.

Symbols in Prolog

Using the following truth-functional symbols, the Prolog expressions are

English	**Predicate Calculus**	**Prolog**
If	-->	:-
Not	~	Not
Or	V	;
and	^	,

comprised. These symbols have the same interpretation as in the predicate calculus.

Two types of prolog programming are there based on the features of:

1. Facts, rules and queries
2. Knowledge based

Facts

We can define fact as an explicit relationship between objects, and properties these objects might have. So facts are unconditionally true in nature. Suppose we have some facts as given below −

- Tom is a cat
- Kunal loves to eat Pasta
- Hair is black
- Nawaz loves to play games
- Pratyusha is lazy.

So these are some facts, that are unconditionally true. These are actually statements, that we have to consider as true.

Following are some guidelines to write facts −

- Names of properties/relationships begin with lower case letters.
- The relationship name appears as the first term.
- Objects appear as comma-separated arguments within parentheses.
- A period "." must end a fact.
- Objects also begin with lower case letters. They also can begin with digits (like 1234), and can be strings of characters enclosed in quotes e.g. color(penink, ‘red’).
- phoneno(agnibha, 1122334455). is also called a predicate or clause.

Syntax

The syntax for facts is as follows −

```
relation(object1,object2...).
```

Example

Following is an example of the above concept −

```
cat(tom).
loves_to_eat(kunal,pasta).
of_color(hair,black).
loves_to_play_games(nawaz).
lazy(pratyusha).
```

Rules

We can define rule as an implicit relationship between objects. So facts are conditionally true. So when one associated condition is true, then the predicate is also true. Suppose we have some rules as given below −

- Lili is happy if she dances.
- Tom is hungry if he is searching for food.
- Jack and Bili are friends if both of them love to play cricket.
- will go to play if school is closed, and he is free.

So these are some rules that are **conditionally** true, so when the right hand side is true, then the left hand side is also true.

Here the symbol (:-) will be pronounced as “If”, or “is implied by”. This is also known as neck symbol, the LHS of this symbol is called the Head, and right hand side is called Body. Here we can use comma (,) which is known as conjunction, and we can also use semicolon, that is known as disjunction.

Syntax
rule_name(object1, object2, ...) :- fact/rule(object1,
 object2, ...)
Suppose a clause is like :
P :- Q;R.
This can also be written as
P :- Q.
P :- R.

If one clause is like :
P :- Q,R;S,T,U.

Is understood as
P :- (Q,R);(S,T,U).
Or can also be written as:
P :- Q,R.
P :- S,T,U.
Example
happy(lili) :- dances(lili).
hungry(tom) :- search_for_food(tom).
friends(jack, bili) :- lovesCricket(jack), lovesCricket(bili).
goToPlay(ryan) :- isClosed(school), free(ryan).
Queries

Queries are some questions on the relationships between objects and object properties. So question can be anything, as given below −

- Is tom a cat?
- Does Kunal love to eat pasta?

- Is Lili happy?
- Will Ryan go to play?

So according to these queries, Logic programming language can find the answer and return them.

Knowledge Base in Logic Programming

In this section, we will see what knowledge base in logic programming is.

Well, as we know there are three main components in logic programming − **Facts, Rules** and **Queries**. Among these three if we collect the facts and rules as a whole then that forms a **Knowledge Base**. So we can say that the **knowledge base** is a **collection of facts and rules**.

Now, we will see how to write some knowledge bases. Suppose we have our very first knowledge base called KB1. Here in the KB1, we have some facts. The facts are used to state things, that are unconditionally true of the domain of interest.

Knowledge Base 1

Suppose we have some knowledge, that Priya, Tiyasha, and Jaya are three girls, among them, Priya can cook. Let's try to write these facts in a more generic way as shown below −

```
girl(priya).
girl(tiyasha).
girl(jaya).
can_cook(priya).
```

Note − Here we have written the name in lowercase letters, because in Prolog, a string starting with uppercase letter indicates a **variable**.

Now we can use this knowledge base by posing some queries. "Is priya a girl?", it will reply "yes", "is jamini a girl?" then it will answer "No", because it does not know who jamini is. Our next question is "Can Priya cook?", it will say "yes", but if we ask the same question for Jaya, it will say "No".

Output

```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- change_directory('D:/TP Prolog/Sample_Codes').

yes
| ?- [kb1]
.
compiling D:/TP Prolog/Sample_Codes/kb1.pl for byte code...
```

```
D:/TP Prolog/Sample_Codes/kb1.pl compiled, 3 lines read - 489 bytes written,
10 ms

yes
| ?- girl(priya)
.

yes
| ?- girl(jamini).

no
| ?- can_cook(priya).

yes
| ?- can_cook(jaya).

no
| ?-
```

Let us see another knowledge base, where we have some rules. Rules contain some information that are conditionally true about the domain of interest. Suppose our knowledge base is as follows −

```
sing_a_song(ananya).
listens_to_music(rohit).

listens_to_music(ananya) :- sing_a_song(ananya).
happy(ananya) :- sing_a_song(ananya).
happy(rohit) :- listens_to_music(rohit).
playes_guitar(rohit) :- listens_to_music(rohit).
```

So there are some facts and rules given above. The first two are facts, but the rest are rules. As we know that Ananya sings a song, this implies she also listens to music. So if we ask “Does Ananya listen to music?”, the answer will be true. Similarly, “is Rohit happy?”, this will also be true because he listens to music. But if our question is “does Ananya play guitar?”, then according to the knowledge base, it will say “No”. So these are some examples of queries based on this Knowledge base.

Output

```
| ?- [kb2].
compiling D:/TP Prolog/Sample_Codes/kb2.pl for byte code...
D:/TP Prolog/Sample_Codes/kb2.pl compiled, 6 lines read - 1066 bytes written,
15 ms
```

```
yes
| ?- happy(rohit).

yes
| ?- sing_a_song(rohit).

no
| ?- sing_a_song(ananya).

yes
| ?- playes_guitar(rohit).

yes
| ?- playes_guitar(ananya).

no
| ?- listens_to_music(ananya).

yes
| ?-
```

Knowledge Base 3

The facts and rules of Knowledge Base 3 are as follows −

```
can_cook(priya).
can_cook(jaya).
can_cook(tiyasha).

likes(priya,jaya) :- can_cook(jaya).
likes(priya,tiyasha) :- can_cook(tiyasha).
```

Suppose we want to see the members who can cook, we can use one **variable** in our query. The variables should start with uppercase letters. In the result, it will show one by one. If we press enter, then it will come out, otherwise if we press semicolon (;), then it will show the next result.

Let us see one practical demonstration output to understand how it works.

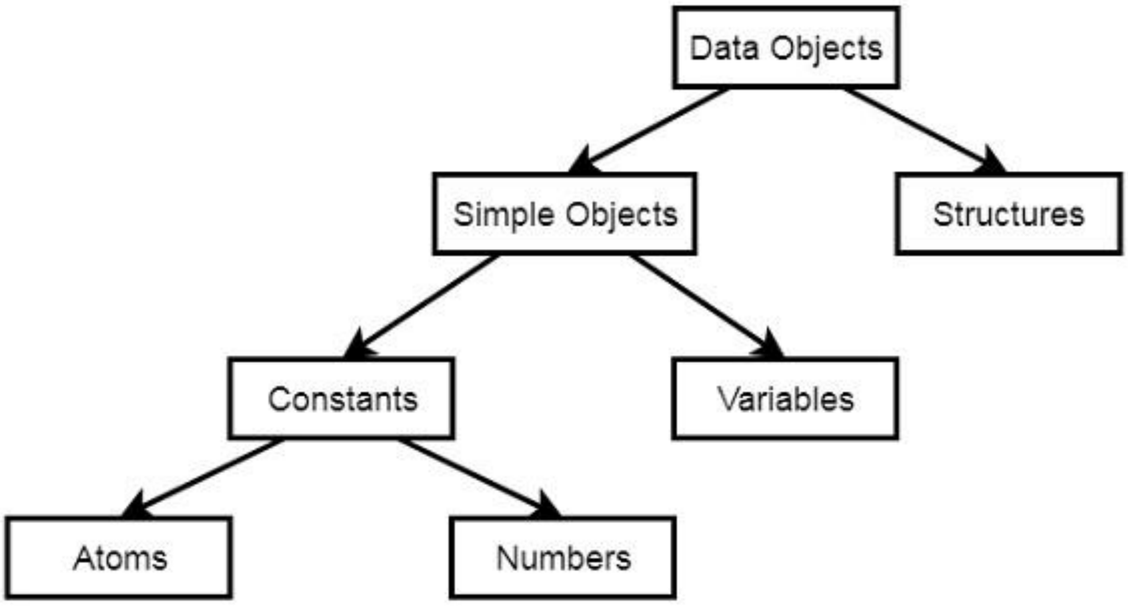
Output

```
| ?- [kb3].
compiling D:/TP Prolog/Sample_Codes/kb3.pl for byte code...
D:/TP Prolog/Sample_Codes/kb3.pl compiled, 5 lines read - 737 bytes written, 22 ms
warning: D:/TP Prolog/Sample_Codes/kb3.pl:1: redefining procedure can_cook/1
        D:/TP Prolog/Sample_Codes/kb1.pl:4: previous definition
```

```
yes
| ?- can_cook(X).

X = priya ? ;

X = jaya ? ;

X = tiyasha

yes
| ?- likes(priya,X).

X = jaya ? ;

X = tiyasha

yes
| ?-
```

Data objects in Prolog:

Data objects can be divided into few different categories as shown below −



Data objects in Prolog

Below are some examples of different kinds of data objects −

- Atoms − tom, pat, x100, x_45
- Numbers − 100, 1235, 2000.45
- Variables − X, Y, Xval, _X
- Structures − day(9, jun, 2017), point(10, 25)

Atoms and Variables

In this section, we will discuss the atoms, numbers and the variables of Prolog.

Atoms

Atoms are one variation of constants. They can be any names or objects. There are few rules that should be followed when we are trying to use Atoms as given below −

Strings of letters, digits and the underscore character, '_', starting with a lower-case letter. For example −

- azahar
- b59
- b_59
- b_59AB
- b_x25
- antara_sarkar

Strings of special characters

We have to keep in mind that when using atoms of this form, some care is necessary as some strings of special characters already have a predefined meaning; for example ':-'.

- <--->
- =======>
- ...
- .:.
- ::=

Strings of characters enclosed in single quotes.

This is useful if we want to have an atom that starts with a capital letter. By enclosing it in quotes, we make it distinguishable from variables −

- ‘Rubai'
- ‘Arindam_Chatterjee'
- ‘Sumit Mitra'

Numbers

Another variation of constants is the Numbers. So integer numbers can be represented as 100, 4, -81, 1202. In Prolog, the normal range of integers is from -16383 to 16383.

Prolog also supports real numbers, but normally the use-case of floating point number is very less in Prolog programs, because Prolog is for symbolic, non-numeric computation. The treatment of real numbers depends on the implementation of Prolog. Example of real numbers are 3.14159, -0.00062, 450.18, etc.

The variables come under the **Simple Objects** section. Variables can be used in many such cases in our Prolog program, that we have seen earlier. So there are some rules of defining variables in Prolog.

We can define Prolog variables, such that variables are strings of letters, digits and underscore characters. They **start with** an **upper-case** letter or an **underscore character**. Some examples of Variables are −

- X
- Sum
- Memer_name
- Student_list
- Shoppinglist
- _a50
- _15

Anonymous Variables in Prolog

Anonymous variables have no names. The anonymous variables in prolog is written by a single underscore character '_'. And one important thing is that each individual anonymous variable is treated as **different**. They are not same.

Now the question is, where should we use these anonymous variables?

Suppose in our knowledge base we have some facts — "jim hates tom", "pat hates bob". So if tom wants to find out who hates him, then he can use variables. However, if he wants to check whether there is someone who hates him, we can use anonymous variables. So when we want to use the variable, but do not want to reveal the value of the variable, then we can use anonymous variables.

So let us see its practical implementation −

Knowledge Base (var_anonymous.pl)

```
hates(jim,tom).
hates(pat,bob).
hates(dog,fox).
hates(peter,tom).
```

Output

```
| ?- [var_anonymous].
compiling D:/TP Prolog/Sample_Codes/var_anonymous.pl for byte code...
D:/TP Prolog/Sample_Codes/var_anonymous.pl compiled, 3 lines read - 536
bytes written, 16 ms

yes
| ?- hates(X,tom).

X = jim ? ;

X = peter

yes
| ?- hates(_,tom).

true ? ;

(16 ms) yes
| ?- hates(_,pat).

no
| ?- hates(_,fox).

true ? ;
```

```
no
| ?-
```

Prolog Version

In this tutorial, we are using GNU Prolog, Version: 1.4.5

Official Website

This is the official GNU Prolog website where we can see all the necessary details about GNU Prolog, and also get the download link.

http://www.gprolog.org/

Direct Download Link

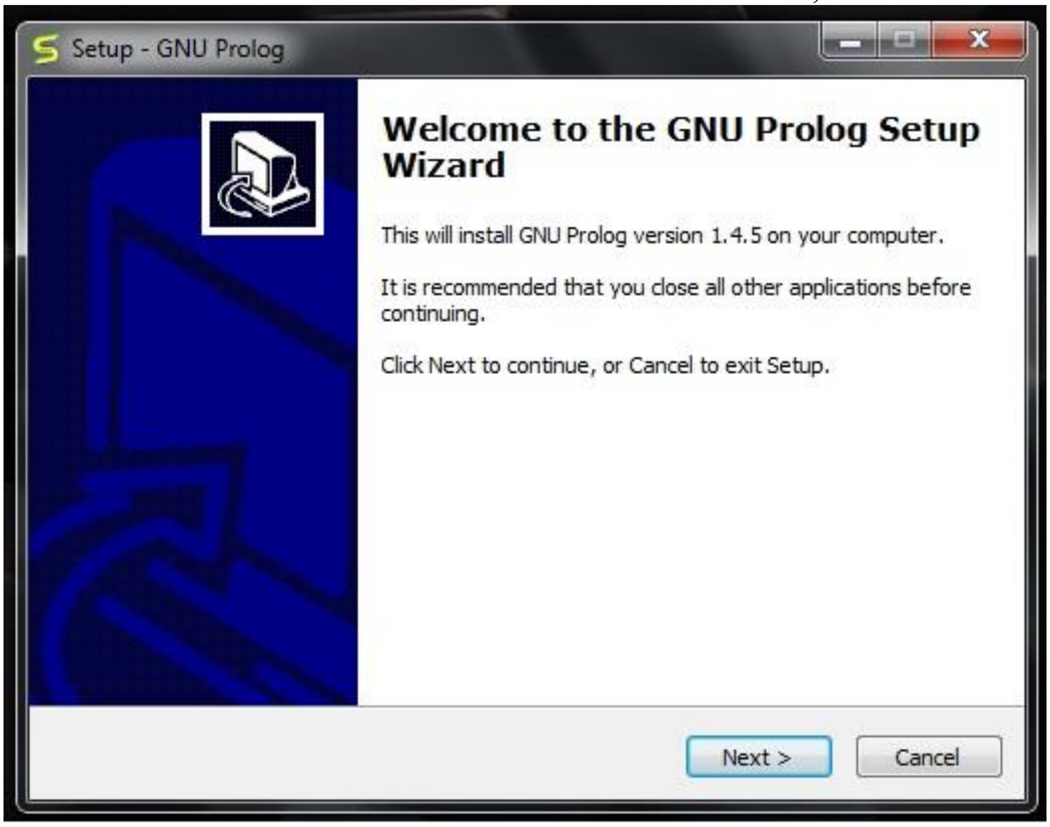
Given below are the direct download links of GNU Prolog for Windows. For other operating systems like Mac or Linux, you can get the download links by visiting the official website (Link is given above) −

http://www.gprolog.org/setup-gprolog-1.4.5-mingw-x86.exe (32 Bit System)

http://www.gprolog.org/setup-gprolog-1.4.5-mingw-x64.exe(64 Bit System)

Installation Guide

- Download the exe file and run it.
- You will see the window as shown below, then click on **next** −



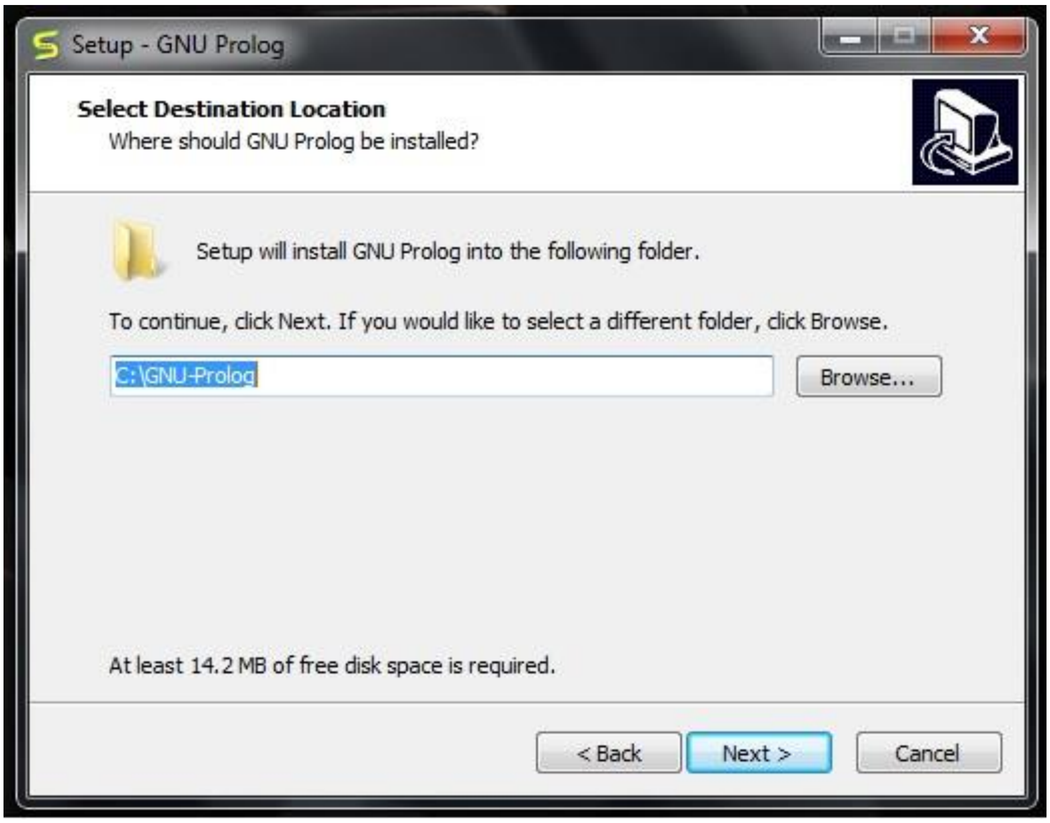
Welcome to the GNU Prolog Setup Wizard

This will install GNU Prolog version 1.4.5 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Select proper **directory** where you want to install the software, otherwise let it be installed on the default directory. Then click on **next**.


Setup - GNU Prolog

Select Destination Location

Where should GNU Prolog be installed?

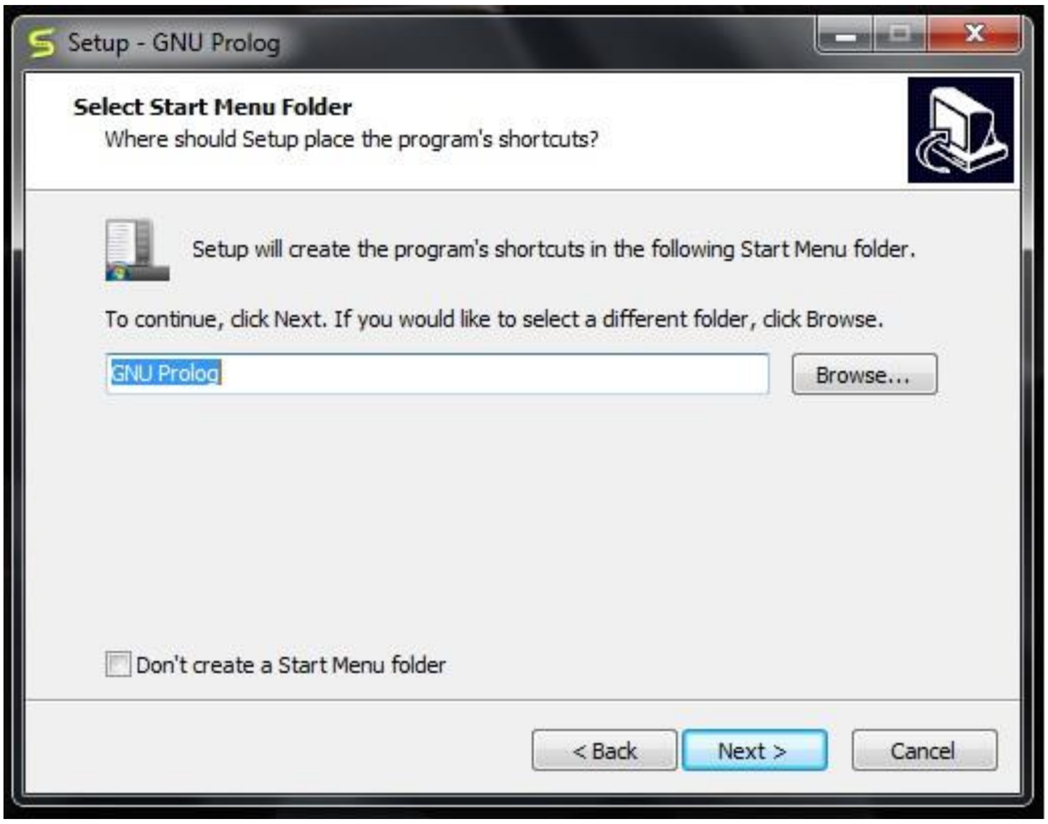
Setup will install GNU Prolog into the following folder.

To continue, click Next. If you would like to select a different folder, click Browse.

C:\GNU-Prolog

At least 14.2 MB of free disk space is required.

You will get the below screen, simply go to **next**.


Setup - GNU Prolog

Select Start Menu Folder

Where should Setup place the program's shortcuts?

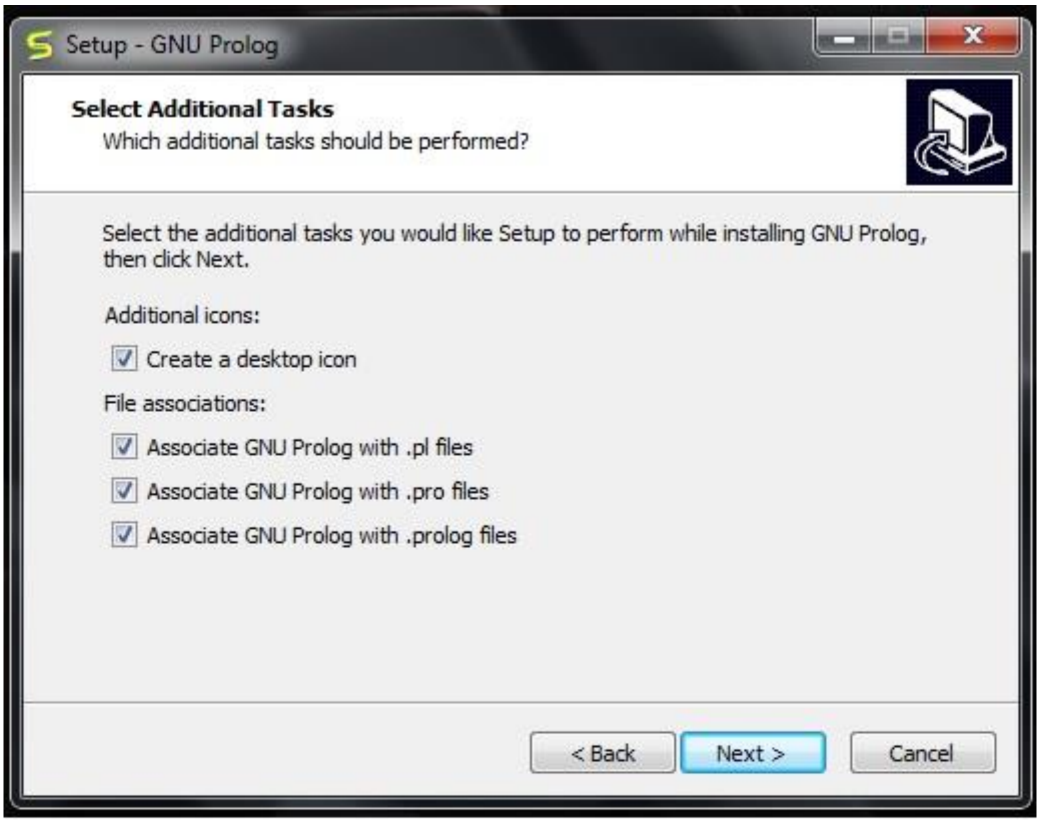
Setup will create the program's shortcuts in the following Start Menu folder.

To continue, click Next. If you would like to select a different folder, click Browse.

GNU Prolog

☐ Don't create a Start Menu folder

You can verify the below screen, and **check/uncheck** appropriate boxes, otherwise you can leave it as default. Then click on **next**.



Setup - GNU Prolog

Select Additional Tasks
Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing GNU Prolog, then click Next.

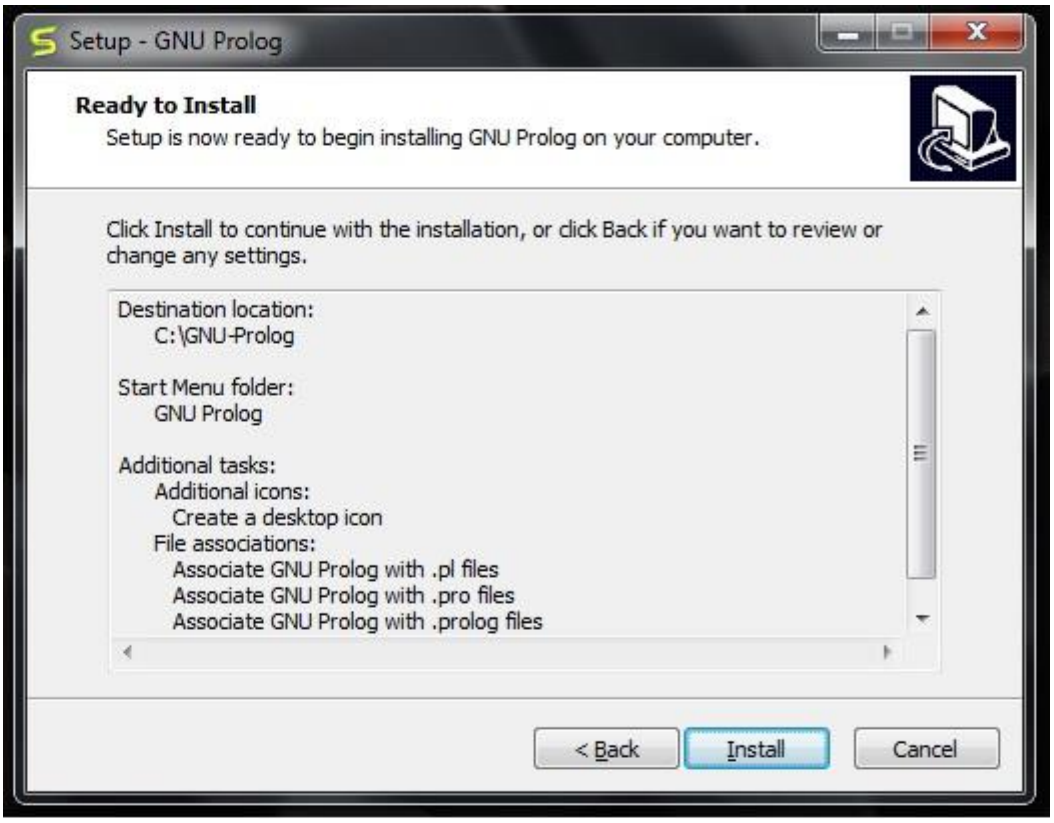
Additional icons:

☑ Create a desktop icon

File associations:

☑ Associate GNU Prolog with .pl files
☑ Associate GNU Prolog with .pro files
☑ Associate GNU Prolog with .prolog files

In the next step, you will see the below screen, then click on **Install**.



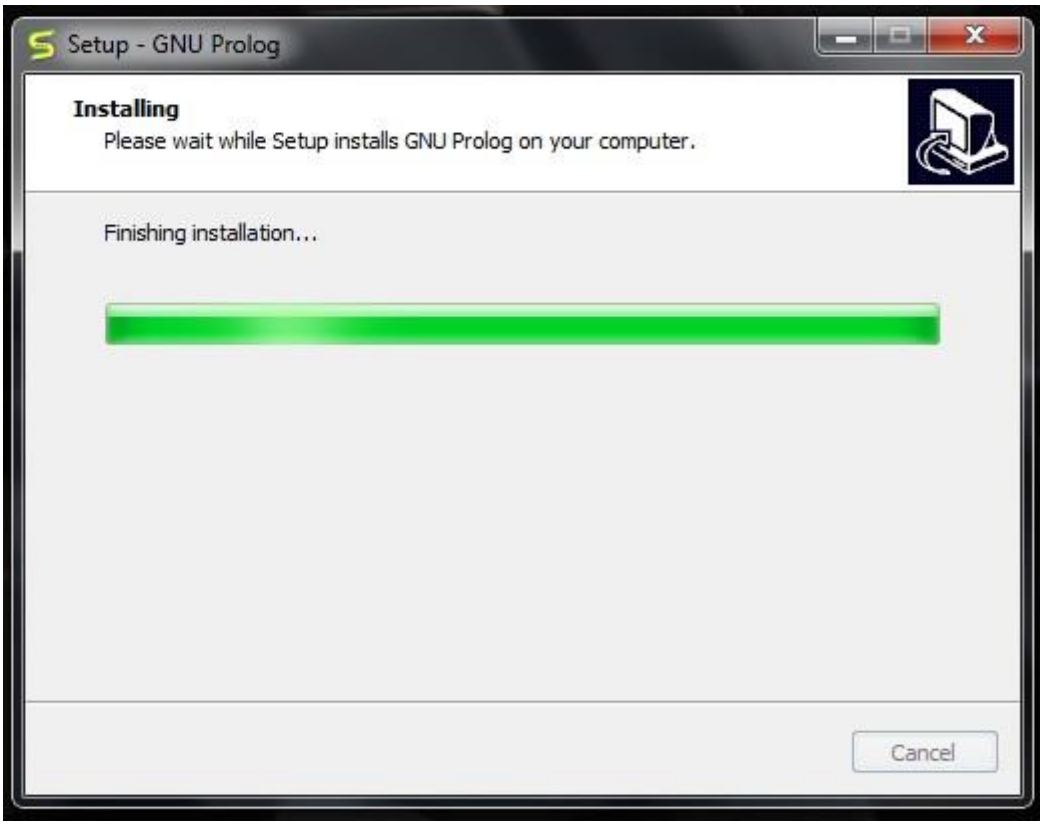
Setup - GNU Prolog

Ready to Install
Setup is now ready to begin installing GNU Prolog on your computer.

Click Install to continue with the installation, or click Back if you want to review or change any settings.

```
Destination location:
    C:\GNU-Prolog

Start Menu folder:
    GNU Prolog

Additional tasks:
    Additional icons:
      Create a desktop icon
    File associations:
      Associate GNU Prolog with .pl files
      Associate GNU Prolog with .pro files
      Associate GNU Prolog with .prolog files
```

Then wait for the installation process to finish.

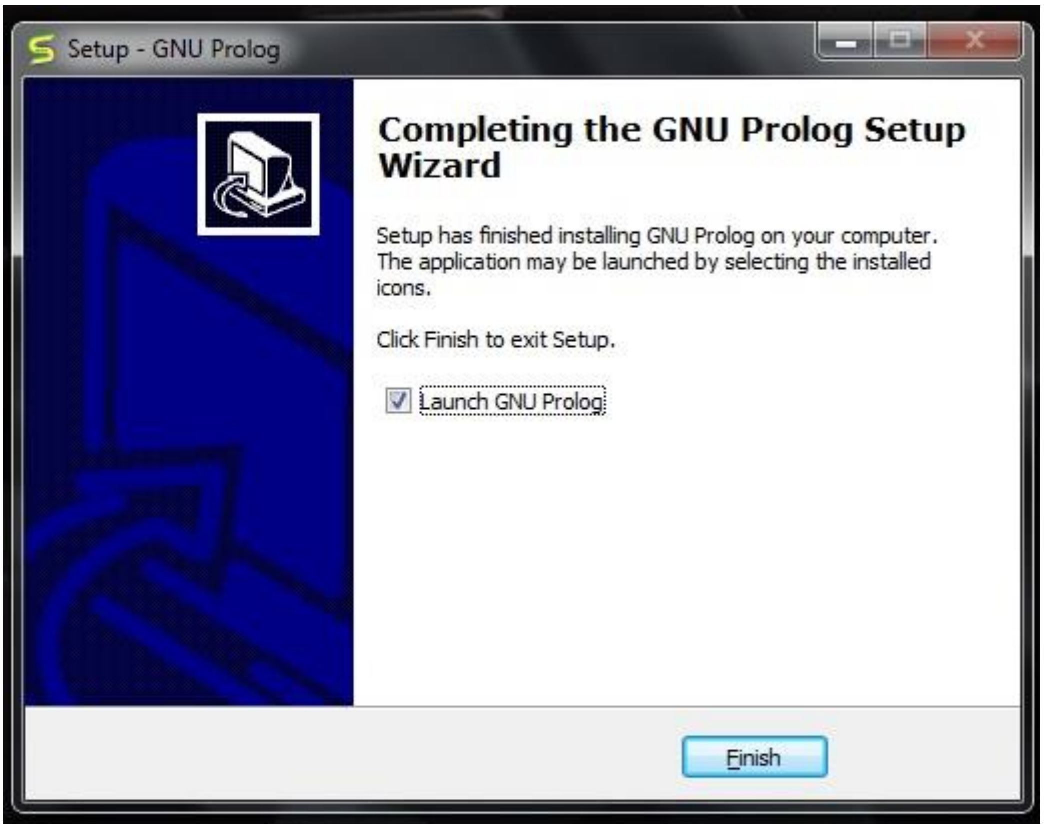

Setup - GNU Prolog

Installing

Please wait while Setup installs GNU Prolog on your computer.

Finishing installation...

Finally click on **Finish** to start GNU Prolog.


Setup - GNU Prolog

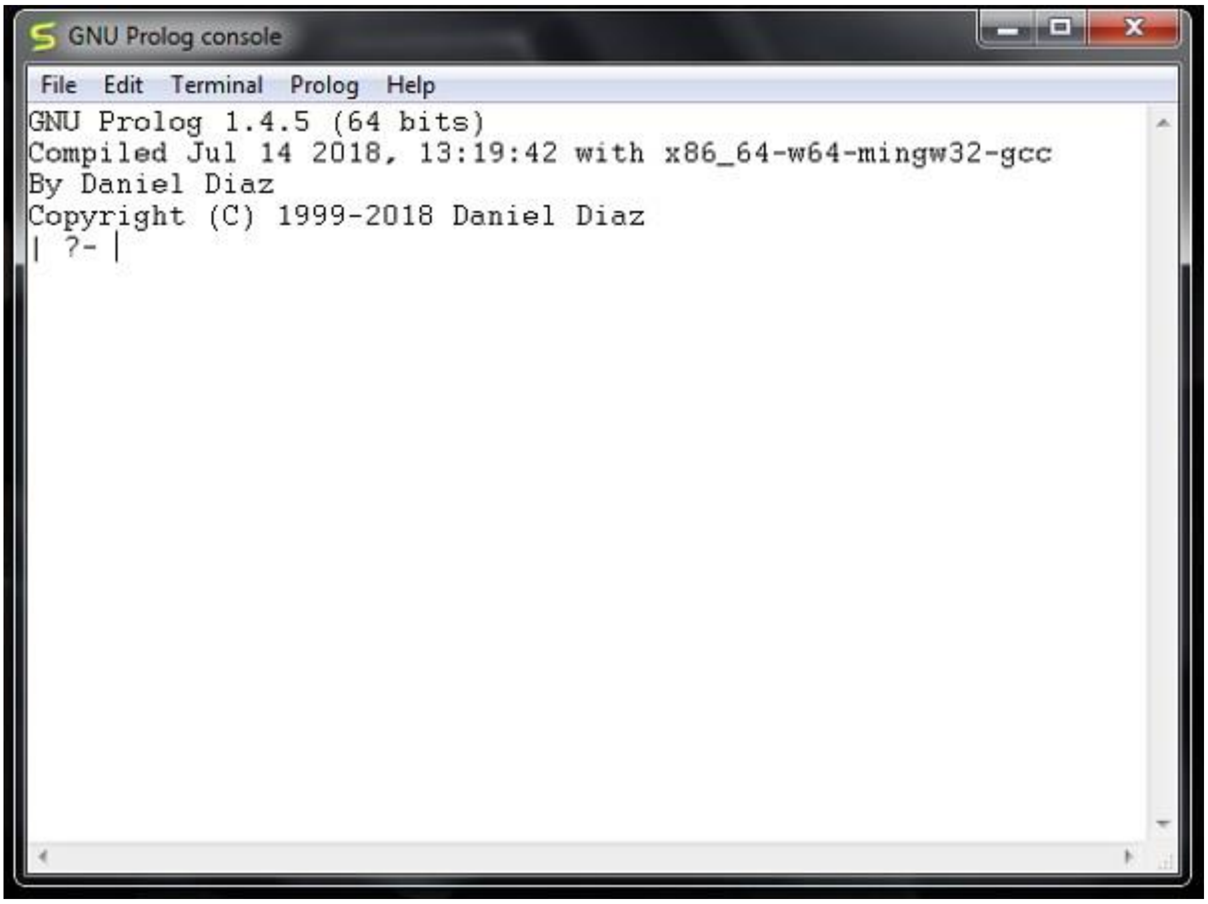
Completing the GNU Prolog Setup Wizard

Setup has finished installing GNU Prolog on your computer. The application may be launched by selecting the installed icons.

Click Finish to exit Setup.

☑ Launch GNU Prolog

The GNU prolog is installed successfully as shown below −



```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- |
```

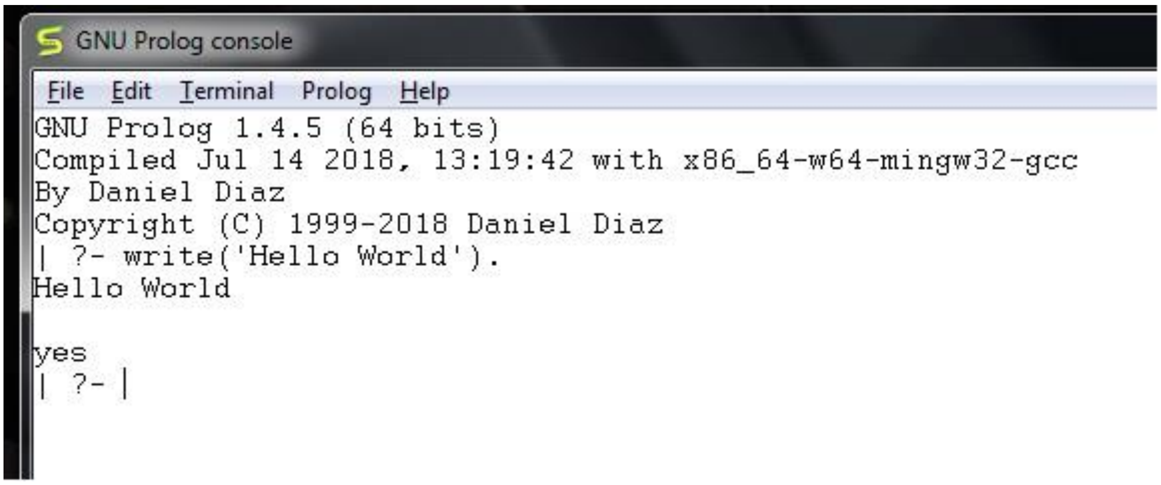
Hello World Program

After running the GNU prolog, we can write hello world program directly from the console. To do so, we have to write the command as follows −

write('Hello World').

Note − After each line, you have to use one period (.) symbol to show that the line has ended.

The corresponding output will be as shown below −



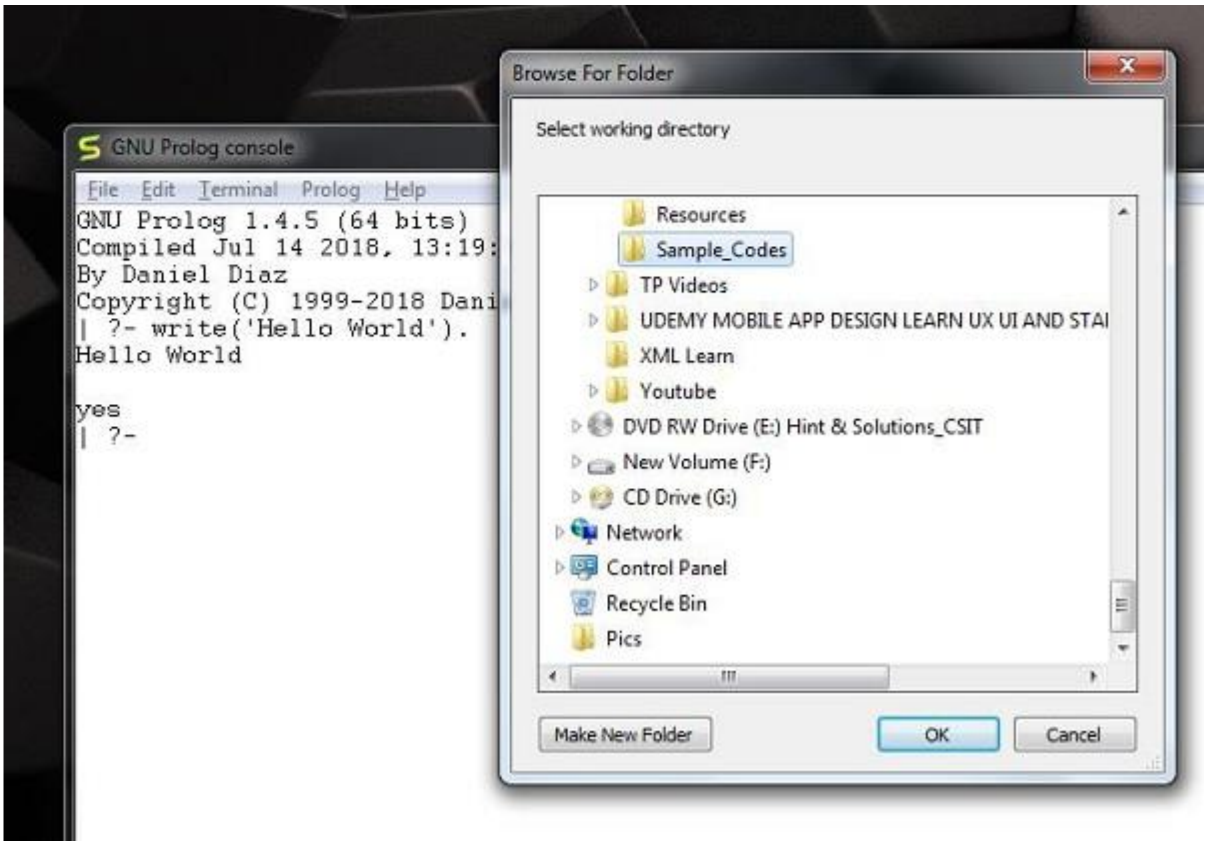
```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- write('Hello World').
Hello World

yes
| ?- |
```

Now let us see how to run the Prolog script file (extension is *.pl) into the Prolog console.

Before running *.pl file, we must store the file into the directory where the GNU prolog console is pointing, otherwise just change the directory by the following steps −

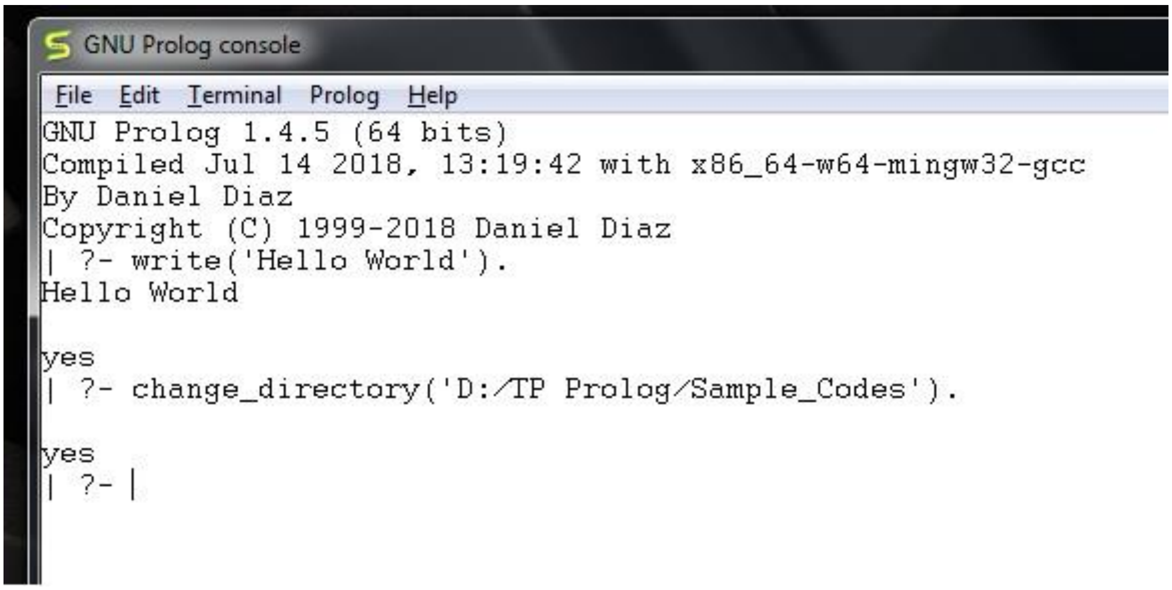
Step 1 − From the prolog console, go to File > Change Dir, then click on that menu.

Step 2 − Select the proper folder and press **OK**.



```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:
By Daniel Diaz
Copyright (C) 1999-2018 Dani
| ?- write('Hello World').
Hello World

yes
| ?-
```

Now we can see in the prolog console, it shows that we have successfully changed the directory.



```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- write('Hello World').
Hello World

yes
| ?- change_directory('D:/TP Prolog/Sample_Codes').

yes
| ?- |
```

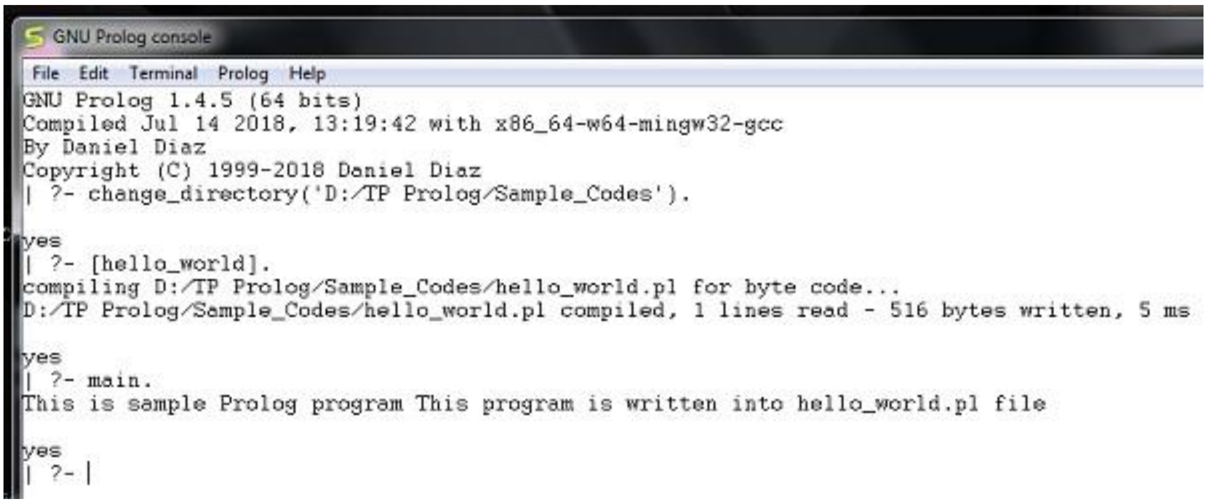
Step 3 − Now create one file (extension is *.pl) and write the code as follows −

main :- write('This is sample Prolog program'),
write(' This program is written into hello_world.pl file').

Now let’s run the code. To run it, we have to write the file name as follows −

[hello_world]

The output is as follows −



```
GNU Prolog 1.4.5 (64 bits)
Compiled Jul 14 2018, 13:19:42 with x86_64-w64-mingw32-gcc
By Daniel Diaz
Copyright (C) 1999-2018 Daniel Diaz
| ?- change_directory('D:/TP Prolog/Sample_Codes').

yes
| ?- [hello_world].
compiling D:/TP Prolog/Sample_Codes/hello_world.pl for byte code...
D:/TP Prolog/Sample_Codes/hello_world.pl compiled, 1 lines read - 516 bytes written, 5 ms

yes
| ?- main.
This is sample Prolog program This program is written into hello_world.pl file

yes
| ?- |
```

Lists and Sequence in Prolog

In Prolog, the list builder uses brackets[...]. A list is referred by the notation [A | B] in which, A is the first element, and whose tail is B. The following example shows the three definitions, where the first element of the list is refereed by the 'car', the tail of the list is referred by 'cdr', list constructor is referred by the 'cons'.

1. car([A | B], A).
2. cdr([A | B], B).
3. cons[A, S, [A | S]).

Where,

- A is the head(car) of [A | B].
- B is the tail(car) of [A | B].
- Put A at the head and B as the tail constructs the list [A | S].

However, the definitions of the above explicit are unneeded. The Prolog team [A|B] refers that A is the head of list and B is its tail. A will bound to the first element of the list, and B will bound to the tail of list if the list can be unified with the team of prolog '[A|B]'.

In this section, many of the predicates are built-in for many interpreters of Prolog.

The predicate 'member/2' definition is described as follows:

1. member(A, [A | S]).
2. member(A, [B | S]) :- member(A, S).

The clauses can be read as follows:

- A is a list member whose first element is A.
- A is a list member whose tail is S if A is a member of S.

We can use this program in many ways. We can also test the membership as follows:

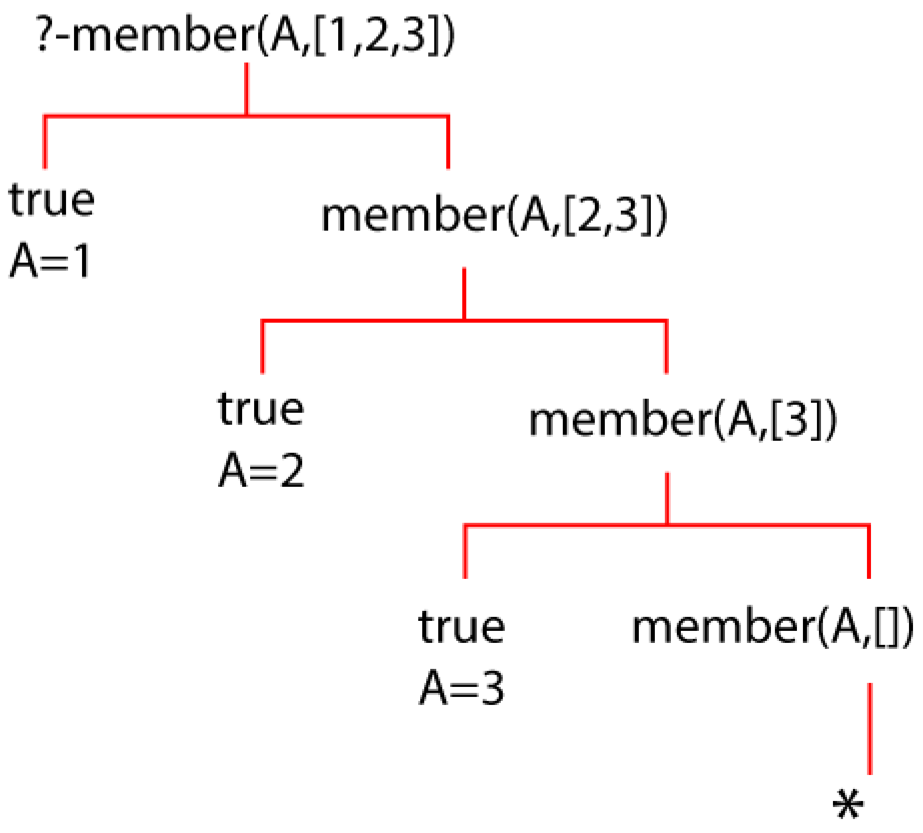
1. ?- member(2, [1, 2, 3]).
2. yes

We can also generate the list member as follows:

1. ?- member(A, [1,2,3]).
2. A = 1 ;
3. A = 2 ;
4. A = 3 ;

5. no

Here, the following derivation tree is used to show how all of the answers are generated by this last goal.



Each left branch corresponds to a match against the first clause for 'member'. Each right branch corresponds to a match against the second clause. On the lowest right branch, the subgoal 'member(A, [])' will not match any 'member' clause head.

Members have many other uses. This example query is as follows:

1. ?- member([3, B], [[1, a], [2, m], [3, z], [4, v], [3, p]]).
2. B = z ;
3. B = p ;
4. no

In the above query, we intend to search to find the elements which are paired with a specified element. In a list, we can find elements in another way, and these elements will satisfy some constraints:

1. ?- member(A, [23, 45, 67, 12, 222, 19, 9, 6]), B is A*A, B<100

```
A=9 B=81 ;
A=6 B=36 ;
no
```

The member definition is written as follows:

```
member(A, [A | _]).
member(A, [_ | S]) :- member(A, S).
```

The "don't care" variable is shown by '_' (underscore). The "don't care" variable is also known as an anonymous variable. In general, anonymous variables have names in which underscore is the first character.

The following definition for 'takeout' is related to 'member' as follows:

```
takeout(A, [A | S], S).
takeout(A, [F | S], [F | T]) :- takeout(A, S, T).
```

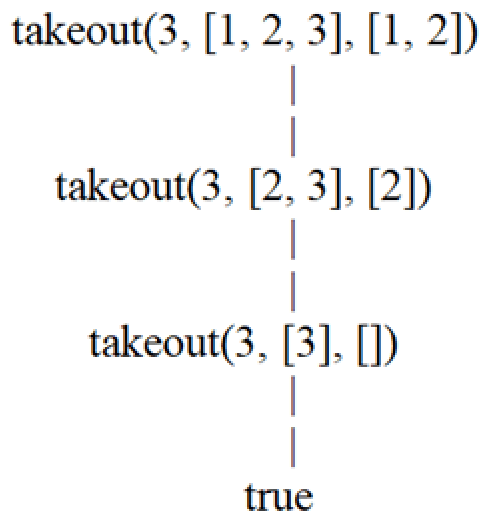
In English, we can paraphrase these clauses as follows:

- The result will be S if A is taken out of [A | S].
- The result will be [A | S] if A is taken out of the tail of [A | S].

For example,

```
?- takeout(A, [1, 2, 3], L).
A=1 L=[2, 3] ;
A=2 L=[1, 3] ;
A=3 L=[1, 2] ;
no
```

In the definition of 'takeout', using any anonymous variables is not appropriate. The consequence of the definition is 'takeout(3, [1, 2, 3], [1, 2])' which is shown by following clause tree.



We will get the following goal:

```
?- takeout(3, X, [a, b, c])
X = [3, a, b, c] ;
X = [a, 3, b, c] ;
X = [a, b, 3, c] ;
X = [a, b, c, 3] ;
no
```

The above example explains that 'takeout(A, C, X)' can also be interpreted as "insert A into X to produce C". We can also define:

```
putin(A, L, S) :- append(B, S, L).
```

The following definition shows the concatenating or appending of two Prolog lists:

```
append([A | B], C, [A | X]) :- append(B, C, X).
append([], A, A).
```

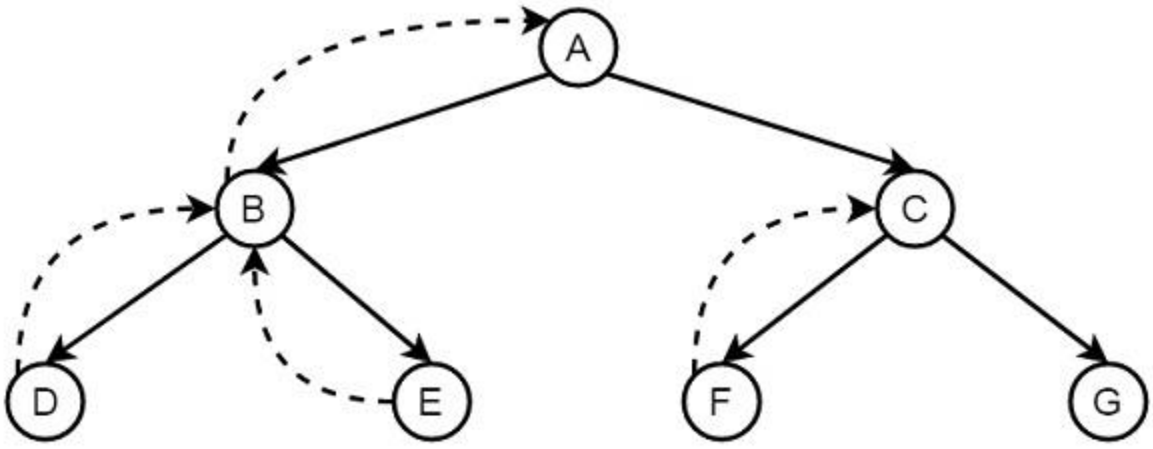
Various possible goals are as follows:

```
?- append([1, 2, 3], [4, 5], [1, 2, 3, 4, 5]).
yes

?- append([1, 2, 3], [4, 5], P).
P = [1, 2, 3, 4, 5]
```

6.
7. ?- append([1, 2, 3], X, [1, 2, 3, 4, 5]).
8. X = [4, 5]
9. ... and so on.

Prolog - Backtracking

Let us see how backtracking takes place using one tree like structure −



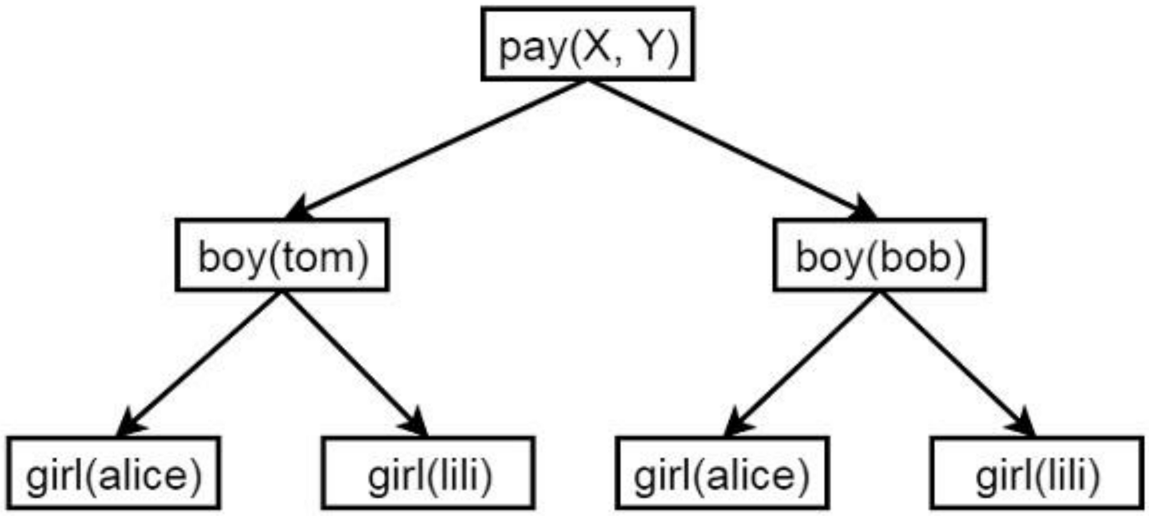
Suppose A to G are some rules and facts. We start from A and want to reach G. The proper path will be A-C-G, but at first, it will go from A to B, then B to D. When it finds that D is not the destination, it backtracks to B, then go to E, and backtracks again to B, as there is no other child of B, then it backtracks to A, thus it searches for G, and finally found G in the path A-C-G. (Dashed lines are indicating the backtracking.) So when it finds G, it stops.

Now, consider a situation, where two people X and Y can pay each other, but the condition is that a boy can pay to a girl, so X will be a boy, and Y will be a girl. So for these we have defined some facts and rules −

Knowledge Base

```
boy(tom).
boy(bob).
girl(alice).
girl(lili).

pay(X,Y) :- boy(X), girl(Y).
```

Following is the illustration of the above scenario −



As X will be a boy, so there are two choices, and for each boy there are two choices alice and lili. Now let us see the output, how backtracking is working.

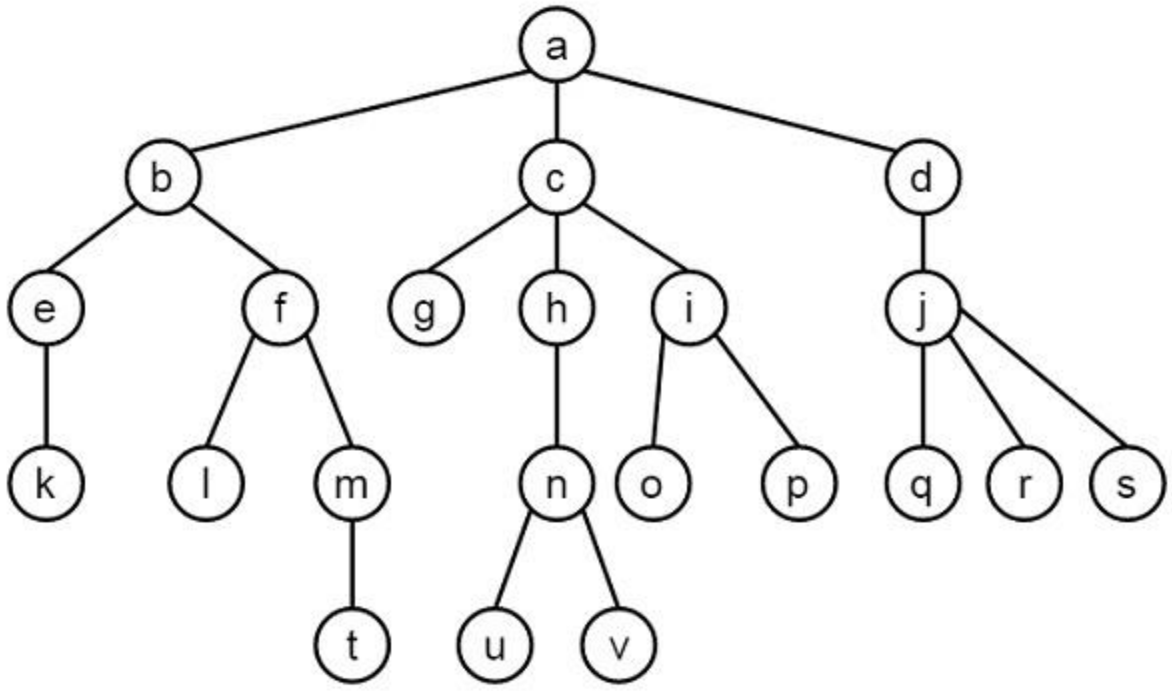
Output

```
| ?- [backtrack].
compiling D:/TP Prolog/Sample_Codes/backtrack.pl for byte code...
D:/TP Prolog/Sample_Codes/backtrack.pl compiled, 5 lines read - 703 bytes
written, 22 ms

yes
| ?- pay(X,Y).

X = tom
Y = alice ?

(15 ms) yes
| ?- pay(X,Y).

X = tom
Y = alice ? ;

X = tom
Y = lili ? ;

X = bob
Y = alice ? ;

X = bob
Y = lili
```

```
yes
| ?- trace.
The debugger will first creep -- showing everything (trace)

(16 ms) yes
{trace}
| ?- pay(X,Y).
   1 1 Call: pay(_23,_24) ?
   2 2 Call: boy(_23) ?
   2 2 Exit: boy(tom) ?
   3 2 Call: girl(_24) ?
   3 2 Exit: girl(alice) ?
   1 1 Exit: pay(tom,alice) ?

X = tom
Y = alice ? ;
   1 1 Redo: pay(tom,alice) ?
   3 2 Redo: girl(alice) ?
   3 2 Exit: girl(lili) ?
   1 1 Exit: pay(tom,lili) ?

X = tom
Y = lili ? ;
   1 1 Redo: pay(tom,lili) ?
   2 2 Redo: boy(tom) ?
   2 2 Exit: boy(bob) ?
   3 2 Call: girl(_24) ?
   3 2 Exit: girl(alice) ?
   1 1 Exit: pay(bob,alice) ?

X = bob
Y = alice ? ;
   1 1 Redo: pay(bob,alice) ?
   3 2 Redo: girl(alice) ?
   3 2 Exit: girl(lili) ?
   1 1 Exit: pay(bob,lili) ?
X = bob
Y = lili

yes
{trace}
| ?-
```

Tree data structure of Prolog and case study :

Suppose we have a tree as shown below −



We have to implement this tree using prolog. We have some operations as follows −

- op(500, xfx, 'is_parent').
- op(500, xfx, 'is_sibling_of').
- op(500, xfx, 'is_at_same_level').
- And another predicate namely leaf_node(Node)

In these operators, you have seen some parameters as (500, xfx, <operator_name>). The first argument (here 500) is the priority of that operator. The 'xfx' indicates that this is a binary operator and the <operator_name> is the name of the operator.

These operators can be used to define the tree database. We can use these operators as follows −

- **a is_parent b, or is_parent(a, b).** So this indicates that node a is the parent of node b.
- **X is_sibling_of Y or is_sibling_of(X,Y).** This indicates that X is the sibling of node Y. So the rule is, if another node Z is parent of X and Z is also the parent of Y and X and Y are different, then X and Y are siblings.

- **leaf_node(Node).** A node (Node) is said to be a leaf node when a node has no children.
- **X is_at_same_level Y, or is_at_same_level(X,Y).** This will check whether X and Y are at the same level or not. So the condition is when X and Y are same, then it returns true, otherwise W is the parent of X, Z is the parent of Y and W and Z are at the same level.

As shown above, other rules are defined in the code. So let us see the program to get better view.

Program

```
/* The tree database */

:- op(500,xfx,'is_parent').

a is_parent b. c is_parent g. f is_parent l. j is_parent q.
a is_parent c. c is_parent h. f is_parent m. j is_parent r.
a is_parent d. c is_parent i. h is_parent n. j is_parent s.
b is_parent e. d is_parent j. i is_parent o. m is_parent t.
b is_parent f. e is_parent k. i is_parent p. n is_parent u.
n
is_parent v.
/* X and Y are siblings i.e. child from the same parent */

:- op(500,xfx,'is_sibling_of').

X is_sibling_of Y :- Z is_parent X,
                  Z is_parent Y,
                  X \== Y.
leaf_node(Node) :- \+ is_parent(Node,Child). % Node grounded

/* X and Y are on the same level in the tree. */

:-op(500,xfx,'is_at_same_level').
X is_at_same_level X .
X is_at_same_level Y :- W is_parent X,
                     Z is_parent Y,
                     W is_at_same_level Z.
```

Output

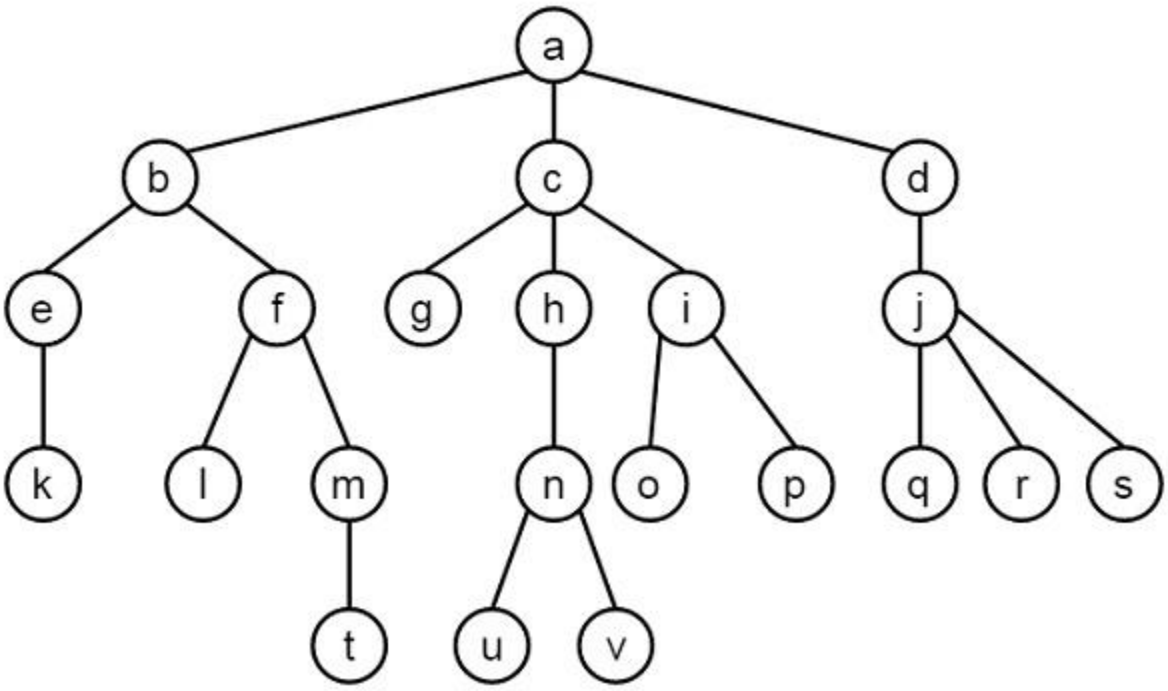
```
| ?- [case_tree].
compiling D:/TP Prolog/Sample_Codes/case_tree.pl for byte code...
D:/TP Prolog/Sample_Codes/case_tree.pl:20: warning: singleton variables
[Child] for leaf_node/1
```

```
D:/TP Prolog/Sample_Codes/case_tree.pl compiled, 28 lines read - 3244 bytes
written, 7 ms

yes
| ?- i is_parent p.

yes
| ?- i is_parent s.

no
| ?- is_parent(i,p).

yes
| ?- e is_sibling_of f.

true ?

yes
| ?- is_sibling_of(e,g).

no
| ?- leaf_node(v).

yes
| ?- leaf_node(a).

no
| ?- is_at_same_level(l,s).

true ?

yes
| ?- l is_at_same_level v.

no
| ?-
```

More on Tree Data Structure

Here, we will see some more operations that will be performed on the above given tree data structure.

Let us consider the same tree here −



We will define other operations −

- path(Node)
- locate(Node)

As we have created the last database, we will create a new program that will hold these operations, then consult the new file to use these operations on our pre-existing program.

So let us see what is the purpose of these operators −

- **path(Node)** − This will display the path from the root node to the given node. To solve this, suppose X is parent of Node, then find path(X), then write X. When root node 'a' is reached, it will stop.
- **locate(Node)** − This will locate a node (Node) from the root of the tree. In this case, we will call the path(Node) and write the Node.

Program

Let us see the program in execution −

```
path(a).                        /* Can start at a. */
path(Node) :- Mother is_parent Node, /* Choose parent, */
          path(Mother),         /* find path and then */
          write(Mother),
          write(' --> ').

/* Locate node by finding a path from root down to the node */
locate(Node) :- path(Node),
            write(Node),
        nl.
```

Output

```
| ?- consult('case_tree_more.pl').
compiling D:/TP Prolog/Sample_Codes/case_tree_more.pl for byte code...
D:/TP Prolog/Sample_Codes/case_tree_more.pl compiled, 9 lines read - 866
bytes written, 6 ms

yes
| ?- path(n).
a --> c --> h -->

true ?

yes
| ?- path(s).
a --> d --> j -->

true ?

yes
| ?- path(w).

no
| ?- locate(n).
a --> c --> h --> n

true ?

yes
| ?- locate(s).
a --> d --> j --> s

true ?

yes
| ?- locate(w).

no
| ?-
```