Prolog Tutorial

What is Prolog

* Prolog stands for programming in logic. In the logic programming paradigm, prolog language is most widely available. Prolog is a declarative language, which means that a program consists of data based on the facts and rules (Logical relationship) rather than computing how to find a solution. A logical relationship describes the relationships which hold for the given application.
* To obtain the solution, the user asks a question rather than running a program. When a user asks a question, then to determine the answer, the run time system searches through the database of facts and rules.
* The first Prolog was 'Marseille Prolog', which is based on work by Colmerauer. The major example of fourth-generation programming language was prolog. It supports the declarative programming paradigm.
* In 1981, a Japanese computer Project of 5th generation was announced. After that, it was adopted Prolog as a development language. In this tutorial, the program was written in the 'Standard' Edinburgh Prolog. Prologs of PrologII family are the other kind of prologs which are descendants of Marseille Prolog.
* Prolog features are 'Logical variable', which means that they behave like uniform data structure, a backtracking strategy to search for proofs, a pattern-matching facility, mathematical variable, and input and out are interchangeable.
* To deduce the answer, there will be more than one way. In such case, the run time system will be asked to find another solution. To generate another solution, use the backtracking strategy. Prolog is a weakly typed language with static scope rules and dynamic type checking.
* Prolog is a declarative language that means we can specify what problem we want to solve rather than how to solve it.
* Prolog is used in some areas like database, natural language processing, artificial intelligence, but it is pretty useless in some areas like a numerical algorithm or instance graphics.
* In artificial intelligence applications, prolog is used. The artificial intelligence applications can be automated reasoning systems, natural language interfaces, and expert systems. The expert system consists of an interface engine and a database of facts. The prolog's run time system provides the service of an interface engine.
* A basic logic programming environment has no literal values. An identifier with upper case letters and other identifiers denote variables. Identifiers that start with lower-case letters denote data values. The basic Prolog elements are typeless. The most implementations of prolog have been enhanced to include integer value, characters, and operations. The Mechanism of prolog describes the tuples and lists.
* Functional programming language and prolog have some similarities like Hugs. A logic program is used to consist of relation definition. A functional programming language is used to consist of a sequence of function definitions. Both the logical programming and functional programming rely heavily on recursive definitions.

**Applications of Prolog**

The applications of prolog are as follows:

* Specification Language
* Robot Planning
* Natural language understanding
* Machine Learning
* Problem Solving
* Intelligent Database retrieval
* Expert System
* Automated Reasoning

Prerequisite

Before learning Prolog,

Audience

Our Prolog tutorial is designed to help beginners and professionals.

Problems

We assure that you will not find any problem in this Prolog Tutorial. But if there is any mistake, please post the problem in a contact form.

Starting Prolog

Prolog system is straightforward. From one person to other person, the precise details of Prolog will vary. Prolog will produce a number of lines of headings in the starting, which is followed by a line. It contains just

**?-**

The above symbol shows the system prompt. The prompt is used to show that the Prolog system is ready to specify one or more goals of sequence to the user. Using a full stop, we can terminate the sequence of goals.

**For example:**

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**?- write('Welcome to Javatpoint'),nl,write('Example of Prolog'),nl.**

**nl** indicates 'start a new line'. When we press 'return' key, the above line will show the effect like this:

**Welcome to Javatpoint**

**Example of Prolog**

**yes**

**?- prompt** shows the sequence of goal which is entered by the user. The user will not type the prompt. Prolog system will automatically generate this prompt. It means that it is ready to receive a sequence of goals.

The above example shows a sequence of goals entered by the user like this:

**write('Welcome to Javatpoint'), write('Example of Prolog'), nl(twice).**

Consider the following sequence of goals:

**write('Welcome to Javatpoint'),nl,write('Example of Prolog'),nl.**

The above sequence of goals has to succeed in order to be succeeded.

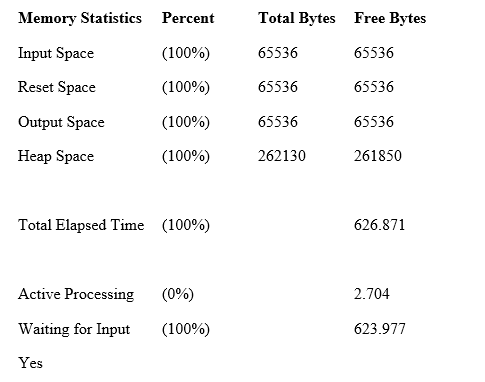
* **write('Welcome to Javatpoint')**  
  On the screen of the user, Welcome to Javatpoint has to be displayed
* **nl**  
  On the screen of the user, a new line has to be output
* **write('Example of Prolog')**  
  On the screen of the user, Example of Prolog has to be displayed
* **nl**  
  On the screen of the user, a new line has to be output

All these goals will simply achieve by the Prolog system by outputting the line of text to the screen of the user. To show that the goals have succeeded, we will output **yes**.

The Prolog system predefined the meanings of **nl** and **write**. Write and nl are called as built-in predicates.

**Halt** and **statistics** are the two other built-in predicates. In almost all Prolog versions, these predicates are provided as standard.

* **?- halt.**  
  The above command is used to terminate the Prolog system.
* **?- statistics.**  
  This command will cause the Prolog system statistics. This statistics feature is mainly used to experienced user. In statistics, the following things will generate:



The above output ends with Yes. Yes, specify that the goal has succeeded, as halt, statistics, and many other built-in predicates always do. When they evaluate, their values produce, which lies in the side-effects.

**'Query'** is a sequence of one or more goals. These goals are entered by the user at the prompt. In this tutorial, we are generally using the 'sequence of goals' term.

Prolog Programs

Using the built-in predicates, the sequence of goals, or specifying a goal at the system prompt would be of little value in itself. To write a Prolog program, firstly, the user has to write a program which is written in the Prolog language, load that program, and then specify a sequence of one or more goals at the prompt.

To create a program in Prolog, the simple way is to type it into the text editor and then save it as a text file like **prolog1.pl**.

The following example shows a simple program of Prolog. The program contains three components, which are known as clauses. Each clause is terminated using a full stop.

1. dog(rottweiler).
2. cat(munchkin).
3. animal(A) :- cat(A).

Using the built-in predicate '**consult'**, the above program can be loaded in the Prolog system.

Play Video

**?-consult('prolog1.pl').**

This shows that prolog1.pl file exists, and the prolog program is systemically correct, which means it has valid clauses, the goal will succeed, and to confirm that the program has been correctly read, it produces one or more lines of output. e.g.,

**?-**  
**# 0.00 seconds to consult prolog1.pl**  
**?-**

The alternative of 'consult' is 'Load', which will exist on the menu option if the Prolog system has a graphical user interface.

When the program is loaded, the clause will be placed in a storage area, and that storage area is known as the Prolog database. In response to the system prompt, specify a sequence of goals, and it will cause Prolog to search for and use the clauses necessary to evaluate the goals.

Terminology

In the following program, three lines show the clauses.

1. dog(rottweiler).
2. cat(munchkin).
3. animal(A) :- cat(A).

Using the full stop, each clause will be terminated. Prolog programs have a sequence of clauses. Facts or rules are described by these clauses.

Example of **facts** is **dog(rottweiler)**and**cat(munchkin)**. They mean that '**rottweiler** is a dog' and '**munchkin** is a cat'.

Dog is called a predicate. Dog contains one argument. Word '**rottweiler'** enclosed in bracket( ). Rottweiler is called an atom.

The example of rule is the final line of the program.

1. animal(A) :- dog(A).

The colon(:-) character will be read as 'if'. Here A is a variable, and it represents any value. In a natural way, the rule can be read as "If A is an animal, then A is a dog".

The above clause shows that the **rottweiler** is an animal. Such deduction can also make by Prolog:

**?- animal(rottweiler).**  
**yes**

To imply that **munchkin**is an animal, there is no evidence of this.

**?- animal(munchkin).**  
**no**

More Terminology

Evaluating a goal term determines whether or not it is satisfied. It also means that goal evaluates to true or false.

Note that when a user enters a goal, then sometimes it can be interpreted as a command. For example,

**?- halt.**                 'It is used to exit from the Prolog system.'

Sometimes it can be regarded as a question like,

**?- animal(rottweiler).**    &             'Is rottweiler an animal?'

The following program shows another example about animals. It comprises eight clauses. The comment is shown by all the text between the /\* and \*/.

1. /\* Another Program of Animal \*/
2. Dog(rottweiler).
3. cat(sphynx).        dog(poodle).
4. dog(bulldog).       cat(bengal).
5. dog(dobermann).
6. cat(himalayan). cat(singapura).
7. /\* This Prolog program consists of various clauses. It is always terminated using the full stop.\*/

Predicate dog and predicate cat both have four clauses. Assume that in a text file 'animal.pl', the program has been saved, and output is generated by loading the program and at the system prompt, we are entering a sequence of goals as follows:

**?- consult('animals1.pl').                 System prompt**  
**# 0.01 seconds to consult animals.pl                 animals.pl loaded using the consult**

**?- dog(rottweiler).**  
**yes**

**?- dog(boxer).**  
no

**?- dog(A).**  
**A = rottweiler                 pauses- return key is pressed by the user**

**?- dog(B).**  
**B = rottweiler;                 pauses ? user presses ;**  
**B = poodle;                 pauses ? user presses ;**  
**B = bulldog;                 pauses ? user presses ;**  
**B = dobermann                 No pause ? It will go onto next line**

**?- cat(A). A = sphinx;                 pause ? user presses;  
A = Bengal                 pauses ? user presses return**

**?- listening(dog).                 It will list all the clauses which define predicate dog**

**/\* dog/1 \*/**

**dog(rottweiler).  
dog(poodle).  
dog(bulldog).  
dog(dobermann).  
yes  
?-**

In this example, various new features of Prolog are introduced. The query is as follows:

**?- dog(A).**

It means that find the A's value, and it will be the name of the dog. The answer of Prolog is as follows:

**A = rottweiler**

Other possible answers of A are as follows, poodle, bulldog, dobermann. It will cause the Prolog pause, and because of this, we have to wait for the user to press the 'return' key before it outputs the system prompt ?-.

We can enter the next query as follows:

**?- dog(B).**

This query is the same as before. The above query means that 'find the B's value, and it will be the name of a dog'. The answer of Prolog is as follows:

**B = rottweiler**

Prolog will again pause. This time semicolon (;) key is pressed by the user. Now Prolog will find for an alternative value of B that satisfies the goal dog(B). It will reply as follows:

**B = poodle**

Prolog will again pause. The semicolon (;) key is again pressed by the user. Prolog produces a further solution as follows:

**B = bulldog**

Prolog will again pause. The semicolon (;) key is again pressed by the user. Prolog produces a further solution as follows:

**B = dobermann**

Prolog recognizes that there is no more available solution by not pausing, but the system prompt ?- by immediately going on to the output.

A new built-in predicate is introduced in this example. Specifying the goal

**?- listing(dog)**

In the above goal, Prolog will list all four clauses which define the predicate dog. They will define in the same order as they loaded into the database.

The use of variables in the query is shown by the following example. The sequence of goal is as follows:

**?-cat(A),dog(B).**

This will give us all possible combinations of a cat and a dog.

**?-cat(A),dog(B).**  
**A = sphinx,**  
**B = rottweiler;**

**A = sphinx,**  
**B = poodle;**

**A = sphinx,**  
**B = bulldog;**

**A = sphinx,**  
**B = dobermann;**

etc.

In contrast, the sequence of goal is as follow:

**?-cat(A), dog(A).**

This will give all animals which are both a cat and a dog (in the database, there is no such animal). Here A is 'any value' in both cat(A) and dog(A), but both must have the same value.

**?-cat(A),dog(A).**  
**no**

Types of Prolog

Prolog is used to provide the Tuples, lists, numbers, atoms, and patterns. In this section, we can define the type of objects that are passed as arguments.

Simple Types

This type of Prolog is implementation-dependent. The following table shows the implementation of a simple type of prolog:

|  |  |
| --- | --- |
| **TYPE** | **VALUES** |
| Boolean | true, fail |
| Variables | variables |
| Integer | integers |
| Atom | character sequence |
| Real | floating point number |

The Boolean constants are not passed as an argument. Variables describe the character string. The character strings start with a capital letter or upper case letter. Atoms are constants that have no numerical value. All the atoms start with a lower case letter or small letter.

Composition Types

The distinction between data and program are blurred in prolog. In the argument, data is often passed to predicates. In prolog, the most common data structure is lists. Lists are much like the stack in which we can only sequentially access the lists of elements, and much like the array in which we have a list of elements sequentially.

Play Video

Prolog is used to allow arbitrary patterns as data, and that pattern represents tuples. An array is not provided by Prolog. But single or multidimensional arrays may be represented as a list or list of lists. The array can also be represented by a set of facts in the database.

TYPE REPRESENTATION [ comma separated sequence of items ] list pattern sequence of items

Using the square brackets ([]+), a prolog list can represent. The following example shows a list of fruits:

[mango, grapes, orange]

The above list shows the elements mango, grapes, and orange. In prolog list, the elements are ordered. If there are no indexes, the elements will also be ordered. Using the patterns, the tuples can be represented.

**Example**

book(author(aaby,anthony),title(labmanual),data(1991))

Using the pattern matching, the elements of tuples can be accessed.

book(Title,Author,Publisher,Date).author(LName,FName).

Type Predicates

**T**he user has to determine the parameter type because prolog is a weakly typed language. To determine the parameter type in prolog, the following built-in predicate can be used.

|  |  |
| --- | --- |
| **PREDICATE** | **CHECKS IF** |
| atom(A) | A is an atom |
| atomic(A) | A is a number or an atom |
| number(N) | N is an integer or real value |
| var(V) | V is a variable |
| nonvar(NV) | NV is not a variable |
| integer(I) | I is an Integer |
| real(R) | R is a floating-point number |
| T =L | T is a term, L is a list |
| functor(T,F,A) | T is a term with functor F, and A is an arity |
| clause(H, T) | H :- T is a program rule |

In the above example, T =..L, function(T,F,A), and clause(H,T) are used in program manipulation.

**Here**

clause(H,T): It checks the database content.  
T =..L: It can manipulate terms.  
functor(T,F,A): It can also manipulate terms.

The following example shows the predicate function:

functor (T,F,A)

T is a term, F is its functor, and A is its arity.

?- functor(t(x,y,z),F,A).  
F = t  
A = 6  
yes

**Here,**

t: It is the function of the term  
3: It is the arity of the term

Prolog Syntax

The syntax of Prolog is as follows:

Symbols

Using the following truth-functional symbols, the Prolog expressions are comprised. These symbols have the same interpretation as in the predicate calculus.

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

Variable

Variable is a string. The string can be a combination of lower case or upper case letters. The string can also contain underscore characters that begin with an underscore or an upper-case letter. Rules for forming names and predicate calculus are the same.

**For example:**

Play Video

1. female
2. Male
3. X
4. y
5. mother\_of
6. \_father
7. Pro34

Facts

A fact is like a predicate expression. It is used to provide a declarative statement about the problem. In a Prolog expression, when a variable occurs, it is assumed to be universally quantified. Facts are specified in the form of the **head**. Head is known as the clause head. It will take in the same way as the goal entered at the prompt by the user.

**For example:**

1. holi.
2. cat(bengal).                /\* bengal is a cat \*/
3. dog(rottweiler).            /\* rottweiler is a dog \*/
4. likes(Jolie, Kevin).            /\* Jolie likes Kevin \*/
5. likes(A, Kevin).                /\* Everyone likes Kevin \*/
6. likes(Jolie, B).                /\* Jolie likes everybody \*/
7. likes(B, Jolie), likes(Jolie, B).       /\* Everybody likes Jolie and Jolie likes everybody \*/
8. likes(Jolie, Kevin); likes(Jolie, Ray). /\* Jolie likes Kevin or Jolie likes Ray \*/
9. not(likes(Jolie, pasta)).       /\* Jolie does not like pasta \*/

Queries

In Prolog, the query is the action of asking the program about the information which is available within its database. When a Prolog program is loaded, we will get the query prompt,

1. ?-

After this, we can ask about the information to the run time system. Using the above simple database, we can ask a question to the program like

1. ?- 'It is sunny'.

and it will give the answer

1. yes
2. ?-

The system responds to the query with yes if the database information is consistent to answer the query. Using the available database information, we can also check that the program is capable of proving the query true. No indicates that the fact is not deducible based on the available information.

The system answers no to the query if the database does not have sufficient information.

1. ?- 'It is cold'.
2. no
3. ?-

Rules

Rules extend the logic program capabilities. Rules are used to provide the decision-making process in Prolog. Rules are specified in the form:

1. head:- t1, t2, t3,….., tk. Where k>=1

The head is known as the clause of the head.

:- is known as the clause neck. It is read as 'if'. The body of the clause is specified by t1, t2, t3,…, tk. It contains one or more components, and it can be separated using the commas. A rule will read as 'head is true if t1, t2, t3,…., tk are all true'.

In the following program, first two lines indicate the facts and last two lines indicate the rules:

1. dog(rottweiler).    large(rottweiler).
2. cat(siamese).         large(siamese).
3. large\_animal(A) :- dog(A),large(A).
4. large\_animal(C) :- cat(C),large(C).

The above rules mean that 'large\_animal(A) is true if dog(A) is true, and large(A) is true, etc.'

The last line means that 'large\_animal(C) is true if cat(C) is true, and large(C) is true.

Data Objects in Prolog

In Prolog, data objects are also known as terms. In Prolog, the example of terms is bulldog, dog(rottweiler), A, and cat(A).

Terms have several different types, which are explained as follows:

1. Numbers

In Prolog, all versions allow the use of integers. Any sequence of numbers from 0 to 9 is written as numbers. The numbers are preceded by + or - sign.

**For example,**

Play Video

1. 278
2. -64
3. +8
4. 057

The use of numbers with decimal points is allowed by most versions of Prolog. Just like the integer, they are written. It contains a single decimal point. But this decimal point cannot exist before an optional plus(+) or minus(-) sign.

**For example:**

1. 4.32
2. -7.91
3. +8.45.

2. Atoms

Atoms do not have any numerical value. Atoms are like the constants. The atoms can be written in three different ways.

a. Any sequence of upper case or lower case letters, underscores, numerals, starting with a lower case letter. For example,

1. jackson
2. toady\_is\_Saturday
3. smart\_jack
4. x64\_YZ
6. but not
8. Weekend
9. Toady-is-Saturday
10. 64xyz

b. In single quotes, any sequence of characters can be enclosed. It includes lower case letters and spaces. For example,

1. 'today-is-Saturday'
2. 'Today is Saturday'
3. '64xyz'

c. Any sequence of one or more characters includes = @ > < # & + - \* /. For example,

1. <
2. ++-
3. <=
4. -++

3. Variable

A variable is a name that is used to stand for a term. The variable name can be any sequence of one or more upper case letters or lower case letters, underscore, and numerals. It can begin with an underscore or lower case letters.

**For example:**

1. A
2. jack
3. Javatpoint\_X
4. \_987X
6. but not
8. 82\_XYZ
9. Javatpoint-X
10. jack

Prolog Clauses

In Prolog, the program contains a sequence of one or more clauses. The clauses can run over many lines. Using a dot character, a clause can be terminated. This dot character is followed by at least one 'white space' character. The clauses are of two types: facts and rules.

Facts are specified in the form of the **head**. Head is known as the clause head. It will take in the same way as the goal entered at the prompt by the user. The head of a clause must be a compound term or an atom. Compound terms and atoms are collectively called as call terms.

Examples of facts are as follows:

1. holi.
2. likes(Russell, angelina).
3. likes(A, prolog).
4. cat(sphynx).

Rules are specified in the form:

Play Video

1. head:- t1, t2, t3,….., tk. Where k>=1

The head is known as the clause of head.

:- is known as the clause neck. It is read as 'if'. The body of the clause is specified by t1, t2, t3, tk. It contains one or more components, and it can be separated using the commas. The goal represents the components. The command is represented by 'and'.

A rule will be read as 'if t1, t2, t3,…., tk are all true, head is true'.

Examples of rules are as follows:

1. corona\_virus(A) :- virus(A), corona(A).
2. grandparent(A, B) :- father(A, C), parent(C, B).
3. go :- write('welcome to javatpoint'), nl.

The following program shows another animal program. This program includes facts and rules.

Examples of Animals Program 2 are as follows:

1. /\* Example of Animals Program 2 \*/
2. dog(rottweiler).        large(rottweiler).
3. cat(siamese).           large(siamese).
4. dog(dobermann).         dog(poodle).
5. dog(boxer).             large(boxer).
6. cat(bengal).
7. dog(weimaraner).        large(weimaraner).
8. dog(samoyed).       small(samoyed).
9. cat(cheetoh).           large(cheetoh).
10. cat(pomeranian).        large(pomeranian).
11. large(mike).
12. large\_animal(A) :- dog(A),large(A).
13. large\_animal(C) :- cat(C),large(C).

Atoms are rottweiler, siamese, dobermann, etc. which are indicated by initial lower case letters. Variables are A and B, which are indicated by initial lower case letters.

Facts are indicated by the first 18 clauses. Rules are indicated by the final two clauses.

Predicates in Prolog

There was a simple program that has five clauses. Head is a compound term for each first three clauses with functor parent. It has two arguments with 2 arity.

1. parent(emma, robert).
2. parent(A, B) :- father(A, B).
3. parent(A, B) :- mother(A, B).
4. father(kevin, mary).
5. mother(anne, mary).

The following program includes the clauses for which functor parent is contained by the head, but a different arity.

For example:

1. parent(kevin).
2. parent(A) :- son(A, B).
3. /\* A is parent if A has a son B \*/

In the same program, we can also use a parent as an atom.

Play Video

For example

1. animal(parent).

The above example can cause make confusion too.

An atom can appear as a rule head or as a fact. For example,

1. holi.
2. go :- parent(kevin, Y),
3. write('kevin has a daughter'),
4. write(Y), nl.

The above example regards as a predicate with no arguments, i.e., go/0.

In the previous **Prolog Clauses** file, we have an example of Animals Program 2, which has five predicates: dog/1, cat/1, large/1, small/1, and large\_animal/1. Facts are defined by the first 18 clauses, which represent the predicates dog/1, cat/1, large/1, and small/1 in 6, 4, 7

, and 1 clause respectively. The rules are defined by the final two clauses. Both clauses define large\_animal/1 predicate.

Declarative Interpretation of Rules

Declarative and procedural interpretation both describes the rules. The following example shows the declarative interpretation of the rules as follows:

1. chases(A, B) :- dog(A), cat(B), write(A),
2. write(' chases '), write(B), nl.

In the above example, 'chases(A, B) is true if dog(A) is true, and cat(B) is true, and write(A) is true, etc.'

In the procedural interpretation, "to satisfy chases(A, B), first satisfy dog(A), then satisfy cat(B), then satisfy write(A), etc."

Facts are interpreted declaratively as follows:

1. cat(bengal).

The above will be read as 'bengal is a cat'.

Loading clauses in Prolog

To load the clauses into the database, there are two built-in predicates like: consult/1 and reconsult/1. In both predicates, clauses available in a text file will be loaded into the database. The following example shows the difference between them. Let's assume, **file1.pl** file contains the following details:

1. dog(dane).
2. dog(bandog).
3. dog(mastiff).
4. dog(scottish).
5. dog(labrador).
6. cat(dwelf).
7. cat(chausie).
8. small(bulldog).
9. large(dane).
10. large(mary).
11. large(scottish).
12. large(labrador).
13. large(rottweiler).
14. large(boxer).

The **file2.pl** file contains the following details:

1. dog(bulldog).
2. dog(dane).
3. cat(cheetoh).
4. cat(bill).
5. large(fido).
6. large\_animal(A) :- dog(A), large(A).
7. large\_animal(C) :- cat(C), large(C).

Now we will enter the two goals as follows:

1. ?-consult('file1.pl').
2. ?-consult('file2.pl').

At the prompt, after succession these goals will put these clauses in the database.

Play Video

1. dog(dane).
2. dog(bandog).
3. dog(mastiff).
4. dog(scottish).
5. dog(labrador).
6. dog(bulldog).
7. dog(dane).
8. cat(dwelf).
9. cat(chausie).
10. cat(cheetoh).
11. cat(bill).
12. small(bulldog).
13. large(dane).
14. large(dwelf).
15. large(scottish).
16. large(labrador).
17. large(rottweiler).
18. large(boxer).
19. large(fido).
20. large\_animal(A):- dog(A),large(A).
21. large\_animal(C):- cat(C),large(C).

Effectively, the clauses which are loaded from the second file are added to those already loaded from the first file. The clauses are added predicate by predicate. In the above example, we can see that the dog(fido) appears twice in the database. In the Prolog system, there is nothing to prevent this.

By contrast, we will enter the two goals as follows:

1. ?- consult('file1.pl').
2. ?- reconsult('file2.pl').

At the prompt, after succession these goals will put these clauses in the database.

1. dog(bulldog).
2. sdog(dane).
3. cat(cheetoh).
4. cat(bill).
5. small(bulldog).
6. large(fido).
7. large\_animal(A) :- dog(A), large(A).
8. large\_animal(C) :- cat(C), large(C).

In file2.pl, the definition of predicate completely replaces any previous clauses in the database. Now in the usual way, the new predicates can load. The above example shows the following things:

1. The dog/1, cat/1, and large/1 definition will replace those already in the database.
2. In file1.pl, the small/1 definition will remain in the database.
3. In file2.pl, the large\_animal/1 definition is placed in the database.

The above example shows that reconsult is most unhelpful. But reconsult is routinely used in normal program development. Using various consult goals, some program developers choose several parts to load a large program. But the most common method of developing a program by the programmer is to load an entire program as a single file, test the file, and then make the changes in the file. In a new version of file, the programmer will save the changes using the same name, and then the programmer will reload the clauses from the file. To work it properly, we have to check that each time older versions of clauses are deleted. Using the consult, we can achieve this first time and then reconsult each subsequent time.

In various Prolog versions, consult and reconsult predicates are used so frequently. A simplified notation of Prolog describes that ['file1.pl'] standing for consult('file1.pl'), and [-file.pl] standing for reconsult('file1.pl').

Variables in Prolog

In the head or body of the clause, the variables are used. Variables are also used in goals, and those goals are entered at the system prompt.

**Variables in Goals**

In goals, the variable can be interpreted as meaning 'find variable's value that makes the goal satisfied'.

**For example:**

1. ?- large\_animal(X).

The above goal will be read as 'find the X's value such that large\_animal(X) is satisfied'.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

The following example, **Animal Programs 3** shows the third version of Animals Programs. This example is same as **Animals Program 2**, but it has some additional rules as shown below:

1. /\* Example of Animal Programs 3 \*/
2. /\* some additional rules \*/
3. chases(A, B) :-
4. dog(A), cat(B),
5. write(A), write(' chases '), write(B), nl.
6. /\* chases is a predicate. It has two arguments \*/
7. go :- chases(X, Y).
8. /\* go is a predicate. It has no arguments \*/

The following shows a goal:

1. ?- chases(A, B).

The above goal describes that to satisfy chases(A, B), find the value of A and B.

To satisfy the above goal, Prolog will search through all the clauses until a matching clause is found. The search for clauses will be done from top to bottom. In the body of that clause, it will work through the goals one by one, it will work from left to right, and attempting to satisfy each one in turn.

The following example shows the entering of some typical goals at the prompt and the output of Animal Programs 3.

1. ?- consult('animals3.pl').          System prompt
2. # 0.01 seconds to consult animals3.pl   animals3.pl
4. ?- chases(A, B)             In **this**, user backtracks to find two solutions only.
5. dane chases dwelf
6. A = dane,
7. B = dwelf;
9. dane chases chausie
10. A = dane,
11. B = chausie
13. ? - chases(D, labrador).            Nothing chases labrador
14. no
16. ?- go.                  No values of the variable are output. Due to **this**,
17. dane chases dwelf           the user has no opportunity to backtrack.
18. yes

We can't prevent to generate the same answer more than once by backtracking.

For example:

1. chases(dane, dwelf) : - fchasesm.
2. chases(dane, chausie).
3. chases(dane, dwelf) : - freallychasesm.
4. fchasesm.
5. freallychasesm.

Using the backtracking, two identical answers will be produced out of three by the query **?-chases(dane, A)**.

1. ?- chases(dane, A).
2. A = dwelf;
3. A = chausie;
4. A = dwelf;
5. ?-

**Lexical Scope of Variables**

The following clause shows that

1. parent(A, B) :- father(A, B).

In the above clause, the occurrence of A and B variables can be replaced consistently by any other variables like First\_person and Second\_person.

1. parent(First\_person, Second\_person) :-
2. father(First\_person, Second\_person).

The above can't change the program of the user or meaning of the clause.

**Universally Quantified Variables**

In the fact or rule, if a variable appears, it indicates that the fact or rule applies for all possible variable value.

**For example:**

1. large\_animal(A) :- dog(A), large(A).

**Existentially Quantified Variables**

Assume that the database contains various clauses:

1. person(emma, stone, female, 40, chemist).
2. person(denzel, washington, male, 53, solicitor).
3. person(tom, cruise, male, 39, doctor).
4. person(richard, gere, male, 43, architect).
5. person(angelina, jolie, female, 30, engineer).
6. person(hugh, jackman, male, 27, programmer).
7. man(X) :- person(X, Y, male, Z, S).

The above example has six clauses. These clauses define the definition of person/5 predicate. It has 5 arguments with an obvious interpretation like firstname, lastname, gender, age, and occupation of the person.

The rule is defined by the last clause. It has a predicate 'person'. The rules mean that 'for all X, X is a man if A is a man if X is a person whose sex is male.' In the head of the clause, variable X stands for 'for all X', and it is known as **universally quantified**.

Satisfying Goals in Prolog

In this section, we will look at how the goals are satisfied by the Prolog. If the user has a good understanding of this, he will write a powerful Prolog program in a compact way, and it will frequently use just a few clauses.

This process starts when at the system prompt the user specifies a sequence of goals.

For example:

1. ?- owns(A, B), dog(Z), write(A), nl.

The prolog system is used to satisfy each goal sequence by sequence. It will work from left to right. If the goal has variables like owns(A, B), then the Prolog system always involves binding them to value like A to Kevin and B to dane. All sequence of goals succeeds if all goals succeed turn by turn. All the values of variables are output by the Prolog system used in the sequence of goals.

Play Video

1. ?- owns(A, B), dog(B), write(A), nl.
2. kevin
3. A = kevin,
4. B = dane

The sequence of goals will fail if it is not possible to satisfy all the goals.

1. ?- owns(A, B), dog(B), write(A), nl.
2. no

We have an issue, and that is what Prolog will do if the first sequence goal succeeds, and the second sequence of goal fails.

Call Terms

In Prolog, every goal must be a term, but it doesn't have any kind of term. The term may be a compound term, list, variables, atoms, not a number, or other types of term which is provided by some particular implementation of Prolog. This is a restriction type of term, and that is known as call term. In the bodies of rule, the goals, and heads of the clauses must also be call terms.

Every goal like go, nl, write('Welcome to javatpoint'), dog(A) has a corresponding predicate as go/0, n1/0, write/1, and dog/1, respectively. The write and nl predicates are known as a functor. It has a number of arguments which are known as arity.

To satisfy the goal, Prolog matches the goal with the heads of clauses in the database, and the search will be done from top to bottom.

**For example:**

1. ?- dog(A).

The above goal will match with the fact as follows:

1. dog(dane).

It will give the output as follow:

1. A = dane

To evaluate the user-defined predicate, there is a fundamental principle in Prolog that is if the facts and rules in the database are unable to satisfy the goal, the goal will fail. It has no intermediate position like 'unknown' or 'not proven'. This is equivalent to making a very strong assumption, which is known as closed world assumption about the database: if the facts and rules in the database are unable to prove any conclusion, it will be false. There is no other information.

Unification in Prolog

We will give a goal to evaluate and Prolog will work through the clauses in the database. In this, Prolog attempts to match the goal with each clause. The matching process works from left to right. The goal will fail if no match is found. If a match is found, the action will take.

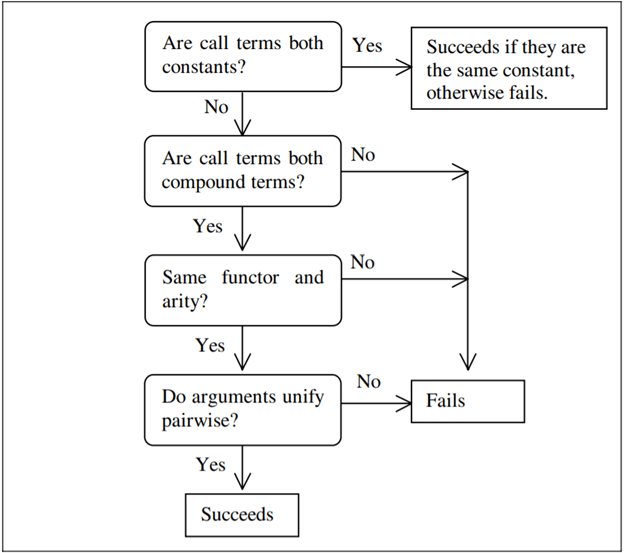
Prolog uses the unification technique, and it is a very general form of matching technique. In unification, one or more variables being given value to make the two call terms identical. This process is called binding the variables to values. For example, Prolog can unify the terms cat(A), and cat(mary) by binding variable A to atom mary that means we are giving the value mary to variable A. Prolog can unify person(Kevin, dane) and person(L, S) by binding L and S to atom kevin and dane, respectively.

In starting, all variables have no value. In unification, once a variable bound to the value, it can be made unbound again and then perhaps be bound to a new value using the backtracking.

Unifying Call Terms

In the following, flowchart can summarize the process.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)



To consider this, we have three cases. In the first case, an atom is unified with another atom, and it is the easiest way. If two atoms are same, this will only succeed, so

* The atoms dane unifies and dane succeeds.
* The atom dane unifies and 'dane' also succeeds.
* The atom dane unifies and kevin fails.

In the **second case,** an atom is unified with a compound term like **dane**with **likes(Kevin, henry)**. This second case always fails.

In the **third case,** the two compound terms are unified, and it the most common case. For example **dog(A) with likes(kevin, B)** or **likes(A, B)** with **likes(Kevin, henry).** In two compound terms, if functor and arity are same, then the unification fails. Predicate is the same, so unifying dog(A) and likes(Kevin, B) fail.

Same Functor and arity unify the two compound terms like parent(A, B, C) with the head parent(Kevin, tom, 30). It requires the head and clause arguments, which are unified 'pairwise', and it will work from left to right. In the two compound terms, the first arguments are unified, and then their second arguments are unified as so on. So, variable A is unified with atom kevin, then variable B with tom, and then C with 30. If all pairs of arguments in the two compound terms unify the unification, it will succeed just like in this case. It fails if not.

The compound term has any kind of argument like variables, list, numbers, atoms, and compound terms. The following example shows some typical unification:

1. parent(A, B, C)
3. parent(kevin, henry, 30)
4. It will succeed with A, B, and C variables bound to kevin, henry, and 30, respectively.
5. parent(kevin, B, 25)
7. parent(A, henry, 30)
8. The 25 cannot be unified with 30, so it will fail.
9. pred1(A, B, [x, y, z])
11. pred1(X, prolog, Y)
12. The variable A bound to variable X, B bound to atom prolog and variable Y bound to list[x, y, z], so it will succeed.

Repeated Variables

In a compound term, if a variable appears more than once, it will become the slightly more complicated case.

1. pred2(A, A, male)
3. pred2(canada, cat, X)
4. ?

The above has two compound terms in which the first argument is unified successfully. So, the **A** bound to **canada**. In the first compound term, all other values of A are also bound to **canada**. When the two second arguments are examined by Prolog, they are no longer A and **cat** but **canada** and **cat**. These atoms are different, and Prolog fails to unify.

1. pred2(A, A, male)
3. pred2(canada, cat, X)
4. The variable A cannot unify with canada and cat, so it will fail.

All bound variables are replaced by their value before Prolog unified any pair of arguments.

A successful unification is shown by the following example, and it involves repeated variables.

1. pred3(A, A, male)
3. pred3(canada, canada, X)
4. The variable A bound to atom canada and variable X bound to atom male, so it will succeed.

The following example describes a repeated variable in one of the arguments in the compound term.

1. pred(male, female, mypred(A, A, B))
3. pred(L, S, mypred(no, yes, maybe))
4. It will fail.

The variable **L** unifies with atom **male,** and variable **S** unifies with atom **female**. Now the two third arguments are unified by the Prolog, i.e., **mypred(A, A, B)** and **mypred( no, yes, maybe).** Firstly variable A unifies with atom no. The variable **A** successfully bound to **no** so it will succeed. Now, Prolog compares the two second arguments. As A is bound to **no**, instead of A and **yes** the second arguments are now **no** and **yes**, so unification fails.

In the following example, unification succeeds because the second argument **mypred** is now **no** rather than **yes**.

1. pred(male, female, mypred(A, A, B))
3. pred(L, S, mypred(no, no, maybe))
4. L and S variable bound to atom male and female and variables A and B bound to atom no and maybe,

Evaluating Goals in Prolog

We will give a goal like **cat(A)** and **go**. Prolog searches from to bottom in the database. It examines the clauses that have heads with the same functor and arity. It finds that the head unifies with the goal or not. The goal fails if Prolog is unable to make successful unification. If it does, the output depends on whether the clause is a fact or rule.

The goal will immediately succeed if the clause is a fact. If the clause is a rule, one by one, the goal is evaluated in the body of the rule from left to right. The original goal succeeds if they all succeed.

A phrase 'a goal matches a clause' is used by the Prolog. It means that it unifies with the head of the clause. The following example shows a goal:

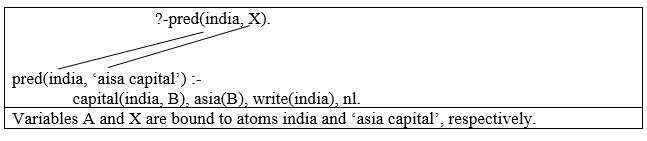
?-pred(india, X)

In the database, the first clause is with predicated pred/2. Using the following rule, the head can unify with the above goal. We will call this rule as **Rule 1**.

Play Video

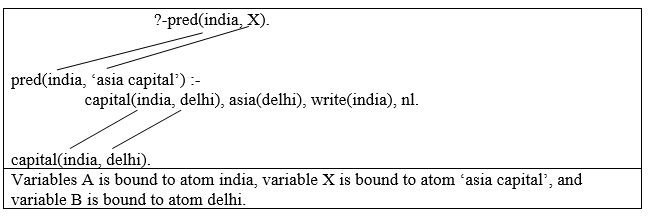
1. pred(A, 'asia capital') :-
3. capital(A, B), asia(B), write(A), nl.

In unification, the variable **A** is bound to atom **india,** and variable **X** is bound to atom '**asia capital'**. When A is bound to india, it affects all occurrence of A in the rule. The diagrammatical representation of this is as follow:



Now in Rule 1, Prolog examines the goals one by one. This process will work from left to right. To succeed the original goal, all the goals have to be satisfied in order. Evaluating each of these goals is the same as evaluating the original goals of the user.

Now will assume that the first clause, which is matched by goal capital(india, B) is the fact capital(india, delhi). In the body of Rule 1, the first goal is satisfied with variable B bound to delhi. Due to this binding, all the occurrence of B is affected in the body of Rule 1. Now we have the following:



In the body of Rule 1, we will try to satisfy the second goal. The rewritten form is asia(delhi).

1. asia(delhi) :- write('PM Modi is Best'), nl.

Now we will call this above rule as **Rule 2**.

In the body of Rule 2, Prolog tries to satisfy the goals: write('PM Modi is Best') and nl. This will process successfully, and as a side effect, the following line shows as output:

PM Modi is Best

In the body of Rule 1, the first two goals have been satisfied. Two more rewritten form of goals are write(india) and nl. Both the goals succeed, and as a side effect, the following line shows as output:

india

In the body of Rule 1, all the goals have now succeeded. That means the goal that form its head succeeds like pred(india, 'asia capital').

?-pred(india, X)

This specifies that original goals succeed, which is entered by the user, with variable X bound to atom 'asia capital'.

Prolog system produces the output as follows:

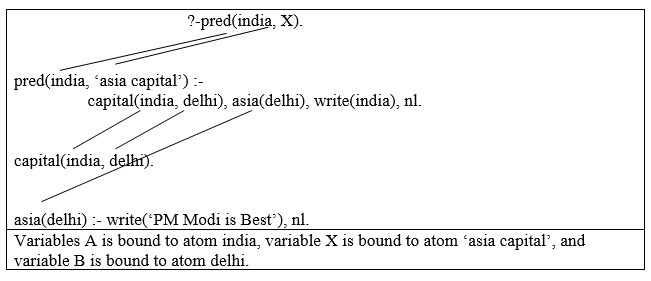
?-pred(india, X).

PM Modi is Best

india

X = 'asia capital'

Prolog system focus on the evaluation of goals with the head of clauses using the unification. In the process of satisfying the goal of the user, we will create a linkage between the goal, clause's head, and the goals in the body of rules. Although the description of this process is very lengthy, but visualizing the linkages is very easy.



A goal of the user is shown below:

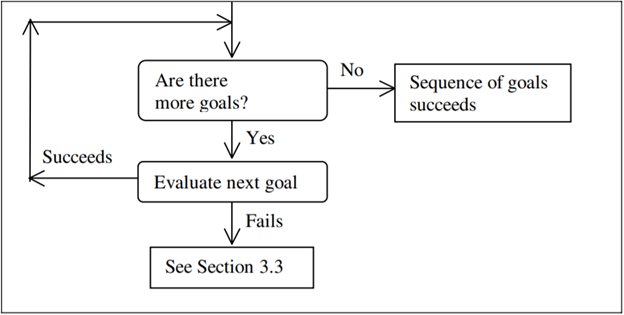
?- pred(india, X)

In the above diagram, the user's goal has been placed on the right side. That is because it has much in common with a goal in the rule's body. The following example shows the goals which are entered by the user at the prompt.

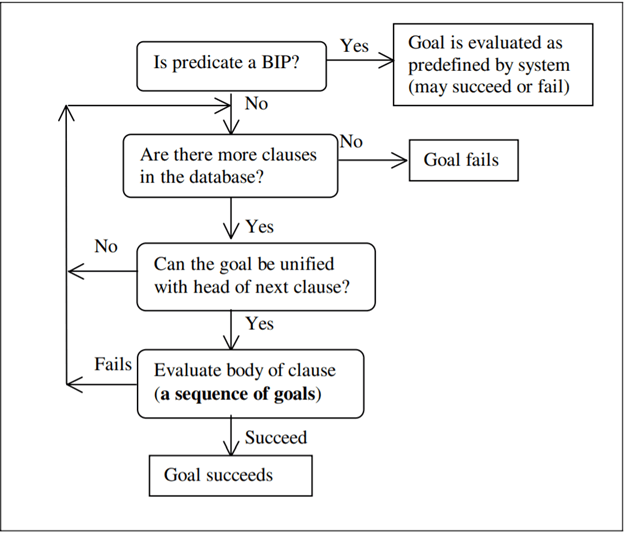
?- owns(A, B), cat(B), write(A), nl.

The above sequence of goals will treat in the same way as goals in the imaginary rule. It will say succeed:- **owns(A, B), cat(B), write(A), nl.**

The following flowchart shows the process of evaluating a goal.



**Fig: Evaluating a Sequence of Goals**



**Fig: Evaluating a Goal**

The left part shows what happens if any of the goals fails in evaluation. If the goal fails, Prolog tries to find another way of satisfying the most recently satisfied previous goal. This process is called backtracking. Unification and backtracking together comprise the mechanism. Prolog uses that mechanism to evaluate the goal, whether the goal is entered in the body rule or by the user at the prompt.

Backtracking in Prolog

In the process of backtracking, we will go back to the previous goal, and after that, we will try to find another way to satisfy the goal. In this section, we will give two detailed ways to satisfy a sequence of goals using the process of backtracking and unification.

For example:

The following example shows a family relationship between a group of people. In the following clauses, the **mother/2** predicate defines **10** facts, **father/2** predicate defines the **9** facts, and **parent/2** predicate defines the **6** clauses.

1. [M1]                mother(mia, liam)
3. [M2]                mother(mia, haley)
5. [M3]                mother(jessica, chris)
7. [M4]                mother(jessica, mark)
9. [M5]                mother(eva, ben)
11. [M6]                mother(haley, jake)
13. [M7]                mother(bennet, alicia)
15. [M8]                mother(michelle, jennifer)
17. [M9]                mother(alicia, chris)
19. [M10]              mother(alicia, britt)
21. [F1]                 father(liam, michael)
23. [F2]                 father(josh, haley)
25. [F3]                 father(mark, henry)
27. [F4]                 father(mark, alicia)
29. [F5]                 father(josh, chris)
31. [F6]                 father(taylor, michelle)
33. [F7]                 father(josh, mark)
35. [F8]                 father(james, luke)
37. [F9]                 father(james, miller)
39. [P1]                 parent(olivia, matt)
41. [P2]                 parent(olivia, smith)
43. [P3]                 parent(A, B) :- write('mother?'), nl, mother(A, B), write('mother!'), nl.
45. [P4]                 parent(X, Y) :- write('father?'), nl, father(X, Y), write('father!'), nl..
47. [P5]                 parent(alexa, jessa)
49. [P6]                 parent(alexa, dave)

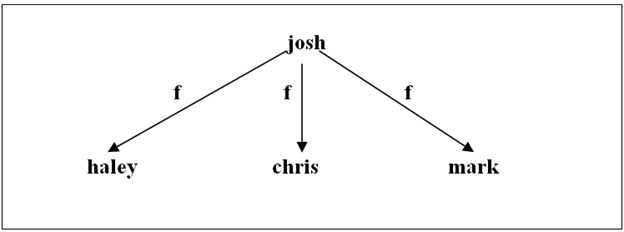
Facts are as follows:

1. mother(jessica, chris).
3. father(josh, chris).

The above facts mean that 'jessica is the mother of chris' and 'josh is the father of chris', respectively. In the above example, we added [M1] labels only for reference purposes. These labels are not part of clauses.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

The following diagram shows the facts of the following example. Here, **f** stands for father.



Example:

Given query

?- parent(josh, Child), write('The child is '), write(Child), nl.

In this, Prolog tries to satisfy all sequence of goals. Firstly, for variable Child, Prolog finds one or more possible values. It starts with the first goal **parent(josh, Child)**. Prolog tries to unify the first goal with the head of each clause, which defines the predicate **parent/2** in turn. To unify this, Prolog works from top to bottom. Clause [P1] and [P2] firstly search, but Prolog fails to match the goal with either of them. Now, Prolog searches the clause [P3], and this time Prolog is successful in unifying the goal with the clause's head, with A is bound to **josh**, and variable B is bound to variable **Child**.

1. ?- parent(josh, Child), write('The child is '), write(Child), nl.
3. [P3] parent(josh, B) :- write('mother?'), nl, mother(josh, B), write('mother!'), nl.
5. It will succeed with A is bound to josh, and variable B is bound to variable Child.

In the body of rule [P3], Prolog works through the goals and trying to succeed the goal in turn. The goals write('mother?'), and nl is successfully evaluated by Prolog and then produce the following line of text as output:

mother?

Now Prolog comes to the third goal, that is mother(josh, B). But this third goal is not unified with the head of any clauses [M1] to [M10], which defines the predicate mother/2. So, the third goal fails.

Now the Prolog system backtracks. In the body of clause [P3], Prolog goes back to the most recently satisfying goal. It works from right to left. The most recent satisfy goal is **nl**, and we will try to satisfy it.

The **n1/0** built-in predicate is unsatisfiable, that means when we evaluate it while backtracking, it always fails.

Now in the body of [P3], Prolog moves to the left and finds the goal write('mother?'). But this goal always fails because the **write/1** predicate is also unsatisfiable.

In the body of rule [P3], there are no goals available, working from left to right. So, rule **[P3]** is rejected by the Prolog system. Now we have the following:

1. ?- parent(josh, Child), write('The child is '), write(Child), nl.
3. is unbound.

In this case, the most recently evaluated goal is parent(josh, Child), and we will try to satisfy this goal.

Now Prolog goes back to the most recently evaluated previously goal, which is parent(josh, Child), and then we try to satisfy this goal. To find the clauses that define the predicate parent/2, Prolog searches the database from the point it has reached previously, i.e., [P3] clause. Firstly, it examines the [P4] clause and successfully unifies the goal with its head. Variable X is bound to josh, and variable Y is bound to variable Child.

1. ?- parent(josh, Child), write('The child is '), write(Child), nl.
3. [P4] parent(josh, Y) :- write('father?'), nl, father(josh, Y), write('father!'), nl.
4. It will succeed with X is bound to josh, and variable Y is bound to variable Child.

In the body of rule [P4], Prolog works through the goals and trying to succeed the goal in turn. The first two goals of Rule [P4] succeed by Prolog and produce the following line of text as output:

father?

Now Prolog attempts to satisfy the third goal, that is father(josh, Y). Prolog searches the clauses and finds that clause that defines the father/2 predicate in turn. The search works from top to bottom.

The system finds the first clause [F2], and it is a fact. Now, the variable Y is bound to haley. Variable Y is bound to variable Child so, variable Child is also bound to atom haley.

1. ?- parent(josh, Child), write('The child is '), write(Child), nl.
3. [P4] parent(josh, haley) :-
4. write('father?'), nl, father(josh, haley), write('father!'), nl.
6. [F2] father(josh, haley).
7. It will succeed with X is bound to josh, and variable Y and Child bound to each other and then to atom haley.

In the body of rule [P4], there are two further goals, i.e., write('father!') and nl. They both succeed and produce the following line of text as output:

father!

Now in the body of rule [P4], all the goals have succeeded, so the head of the clause, i.e., parent(josh, haley) succeeds. Therefore, in the user's query, the goal parent(josh, Child) succeeds.

In the sequence, the first goal has been satisfied, which is entered by the user. In the sequence, there are three more goals, i.e., write('The child is '), write(Child), and nl. They all succeed and produce the following line of text as output:

The child is haley

In the user's query, all the goals have been successfully satisfied. The Prolog system outputs the value of all variables, which are used in the query.

?- parent(josh, Child), write('The child is '), write(Child), nl.

mother?

father?

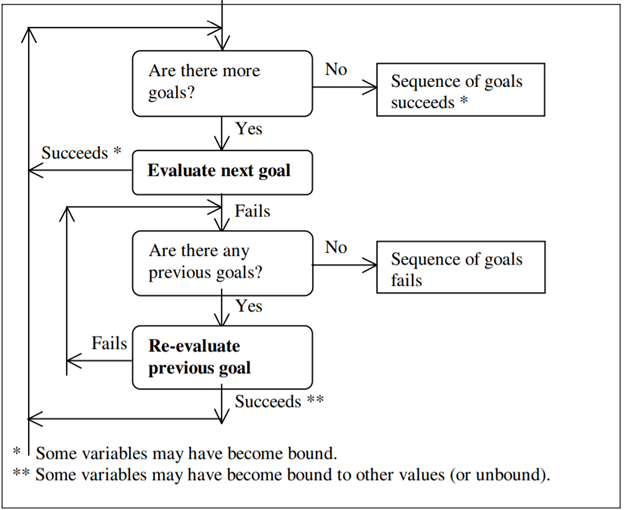
father!

The child is haley

Child = haley

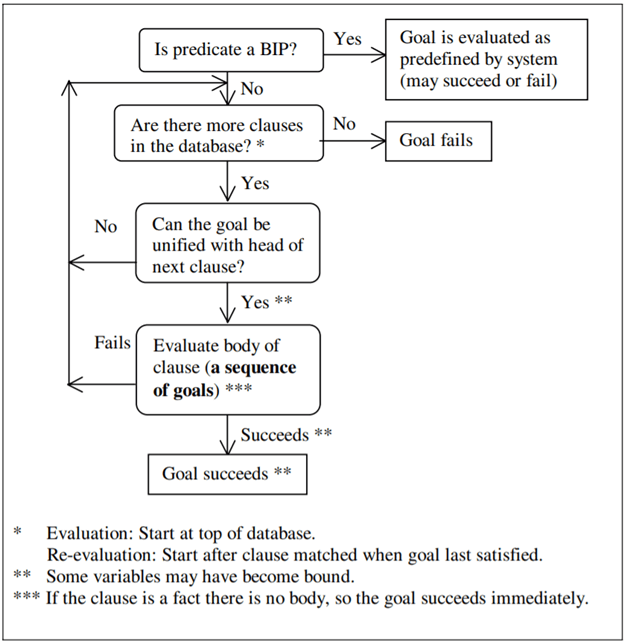
Satisfying Goals Summary

The following diagram shows the methods which were described in the previous section.



**Figure: Evaluating a Sequence of Goals**

Prolog evaluates the goal in turn. To do this, it works from left to right. All sequence of goals succeeds if they all succeed. If one goal fails, Prolog goes back to the previous goals one by one in the sequence and trying to satisfy them from right to left. All the sequence fails if they all fail. Whenever one goal succeeds, Prolog starts to work the goals again from left to right.



**Figure: Evaluating/Re-evaluating a Goal**

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

Summary of Evaluating / Re-evaluating a Goal

To find the goal, Prolog searches all the clauses from top to bottom. Prolog searches in the database until one is found. The goal succeeds if the Prolog matches a clause, and it is a fact. Evaluate the sequence of goals in the body of rule if it is a rule. Prolog continues to search the database for further matches if it is not a rule. The goal fails if the Prolog reaches the end of the database.

Operators in Prolog

notation contains a number of arguments in parenthesis like **likes(hary, jack).**

Any user-defined predicate which has two arguments can be converted into an infix operator as an alternative. In this, we can write the functor between the two arguments, and they have no parenthesis like

1. hary likes jack

Any user-defined predicate which has one argument can be converted into a prefix operator. In this, we can write the functor before the arguments, and they have no parenthesis like

1. isa\_cat bombay

Instead of

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. isa\_cat(bombay)

Alternatively, we can convert a unary operator into a postfix operator. In this, we can write the functor after the argument.

1. bombay isa\_cat

The rules can also use the operator notation to aid readability. Some users of Prolog may find a rule like

1. likes(hary,A) :- is\_female(A),owns(A,B), isa\_dog(B).

If the above rule is written as follow, it is easier to understand:

1. hary likes A :- A is\_female, A owns B, B isa\_cat.

If preferred, the operator can use the standard notation 'functor and argument'. 'Mixed' notation is also permitted if likes/2, is\_female/1, owns/2, and isa\_dog/1 are all operators and a valid form of the previous rule.

1. likes(hary,X) :- is\_female(A), A owns B ,isa\_cat(B).

At the system prompt, if the user enters a goal using **op** predicate, any user-defined predicate which has one or two arguments can convert into an operator. The op predicate has three arguments.

**For example**

1. ?- op(145, xfy, likes).

'Operator precedence' is shown by the **first argument**. It is an integer from 0 upwards. On the implementation, the range of numbers depends. If the number is lower, the precedence will be higher. When the operator is used more than once in a term, the values of operator precedence will be used to determine the order of operator. In most cases, we will use an arbitrary value like 145.

The **second argument** is represented by one of the three atoms:

**xfy** shows that it is a binary predicate, and it is to be converted into an infix operator.

**fy** shows that it is a unary predicate, and it is to be converted into a prefix operator.

**xf** shows that it is a unary operator, and it is to be converted into a postfix operator.

The name of the predicate is shown by the **third argument**. That predicate is to be converted into an operator.

We can also convert a predicate into an operator by placing a line like

1. ?- op(145, xfy, likes).

In a Prolog program, **consult** or **reconsult** is used to load a file. Note that it must include the prompt(two character ?-). When a goal is used in this way, the whole line is called as a directive. In this case, before the first clause that uses the **likes** operator, the directive must be placed in a file.

Various built-in predicates have been pre-defined as an operator. To compare the numerical value, these include relational operator like < that denote 'greater than' and > that denotes 'less than'.

In the body of the rule, the following terms may be included.

1. A>3
2. B<C
3. X=Y

The built-in predicate is also used the bracket notation that is defined as an operator.

**For example**

1. >(A, 3) instead of A>3.

Arithmetic in Prolog

In the previous sections, the examples are non-numerical. In this section, we will use is/2 built-in predicate. This predicate is predefined as an infix operator. The is/2 predicate is placed between the two arguments.

If the first argument is an unbound variable, the predicate is/2 are mostly used. The goal A is -4.2 which shows that A is bound to number -4.2, and the goal succeed.

Arithmetic expression or number can be expressed by the second argument.

**For example**

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. A = 6 \* B + C - 3.2 + S - T / 4

In arithmetic expression, any variables must already be bound. The value of these variables must be numerical. The value of arithmetic expression bounds the variable of first argument. If it is not, an error message will be generated as result.

1. ?- A is 5.7 + 2.9 \* 3
2. A = 14.4
4. ?- B is 6, C is B + 2.
5. B = 6,
6. C = 8

In arithmetic expression, + - \* / symbols are special type of infix operator, and these operators are also known as arithmetic operators. In Prolog, operators are used as predicates but here operators are functions and these operators return a numerical value.

Arithmetic expressions can include variables, numbers, operators, and arithmetic functions. These will be written in parentheses with their arguments. These will return numerical values just like the arithmetic operators.

**For example:** Square bracket of 25.

1. ?- A is sqrt(25).
2. A = 5

The minus(-) arithmetic operator is used as a binary infix operator, which is used to describe the difference of two numerical values like A - 2. It is also used as a unary prefix operator, which is used to describe the negative of a numerical value like

1. ?- A is 7, B is -A - 3.
2. A = 7,
3. B = -10

Arithmetic functions and Arithmetic operators available in Prolog are shown in the following table:

1. A + B                                 sum of A and B
3. A - B                                 difference of A and B
5. A \* B                                 product of A and B
7. A / B                                  quotient of A and B
9. A // B                                 'Integer quotient' of A and B
11. A ^ B                                   A to the power of B
13. - A                                       negative of A
15. sin(A)                                 sine of A
17. cos(A)                                cosine of A
19. abs(A)                                absolute value of A
21. sqrt(A)                               square root of A
23. max(A, B)                          larger of A and B

**Example:**

1. ?- A is 30, B is 3, C is A + B + A \* B + sin(A).
2. A = 30,
3. B = 3,
4. C = 123.5

The **is** predicate is used in the normal way. The first argument can be a bound variable or a number with numerical value. In two arguments, the numerical values are calculated. If these values are equal, the goal succeeds. It fails if these values are not equal.

1. ?- A is 5, A is 4+1.
2. A = 7
4. ?- 15 is 9 + 6 - 13 + 20
5. no
7. ?- 22 is 9 + 6 - 13 + 20
8. yes

The goal A is A + 1 will always fail, whether or not A is bound.

1. ?- A is 7, A is A + 1.
2. no

A different approach is used to increase a value by one.

1. /\* Incorrect Version \*/
3. increase(S) :- -S is S + 1.

**?- increase(4).**

**no**

1. /\* Correct version \*/
3. Increase(S, T) :- -T is S + 1.

**?- increase(4, A).**

**A = 5**

Relational Operators

The relational operators are =:=, >, <, >=, =/=, =<. The relational operators compare the values of two arguments. If the first argument's value is equal to, greater than, less than, greater than or equal to, not equal to, less than or equal to the value of second argument, the goal succeeds. Both arguments can be arithmetic expression, bound variable or numbers.

1. ?- 60 - 5 + 10 =:= 85 - 10\*2.
2. yes
4. ?- 59=\=63.
5. yes

Equality Operators in Prolog

To test the equality and inequality, Prolog has three types of relational operators. The value of arithmetic expression can be compared by the first type of relational operator. The terms can be compared by other two types of relational operator.

Equality Operator (=:=)

Given Arithmetic expression

1. E1 =:= E2

If E1 and E2 evaluate to the same value, the above E1 =:= E2 succeed.

**For example:**

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. ?- 10+3 =:= 5\*4-7.
2. yes
3. ?- sqrt(25)+10 =:= 3\*7-6
4. yes

Prolog uses checkeven/1 predicate to identify whether an integer is odd or even.

1. checkeven(X) :- Y is X//2, X =:= Y\*2.
2. ?- checkeven(7).
3. no
4. ?- checkeven(18)
5. yes
6. ?- checkeven(-27)
7. no
8. ?- checkeven(-10)
9. yes

The // is the division operator. It divides the first argument to the second argument and the result of this division truncates to the nearest integer between it and zero. So 7//2 is 3, 18//2 is 9, -27//2 is -13, -10//2 is -5. Divide the integer by 2 and multiplies it by 2 will give the original integer if it is even, otherwise it is not.

Inequality Operator (=\=)

Given arithmetic expression

1. E1 =\= E2

If E1 and E2 do not evaluate to the same value, arithmetic expression E1 =\= E2 succeeds.

**For example:**

1. ?- 24 =\= 17+4
2. yes

Terms Identical

Given Goal

1. Term1 == Term2

In the infix operator ==, both arguments must be terms. If Term1 and Term2 are identical, the above goal succeeds.

**For example:**

1. ?- likes(A, prolog) == likes(A, prolog)
2. A = \_
4. ?- likes(A, prolog) == likes(B, prolog)
5. no
6. (Variables A and B are different)
8. ?- A is 25, pred1(A) == pred1(25).
9. A = 25
11. ?- A == 0.
12. no
14. ?- 5+4 == 2+7.
15. no

When the arithmetic expression value is used with is/2, the arithmetic expression value can only evaluate. Here the term is defined by 5+4 with arguments 5 and 4 and functor +. This term is totally different from 2+7 term.

Term Not Identical (\==)

Given goal

1. Term1 \== Term 2

The above goal is used to test whether Term1 and Term2 are not identical. If Term1 == Term2 fails, the Term1 \== Term2 succeeds. Otherwise the goal fails.

**For example:**

1. ?- pred1(A) \== pred1(B).
2. A = \_,
3. B = \_

The above output shows that variables A and B are different variables and both are unbound.

Term Identical With Unification (=)

The == operator is similar to equality operator = but with vital difference. If Term1 and Term2 unify, the Term1==Term2 succeeds.

**For example:**

1. ?- likes(A, mary) = likes(prolog, B).
2. A = prolog,
3. B = mary
4. (Variables A and B are bound to atoms prolog and mary respectively and makes the two terms identical.)
6. ?- pred(A) = pred(25).
7. A = 25
8. (Variable A bound to 25, and makes two terms identical.)
10. ?- likes(A, prolog) = likes(B, prolog).
11. A = B = \_
12. (Binding of A and B makes the terms identical.)
14. ?- A = 5, A =:= 5.
15. A = 5
16. (A=5 clauses A bound to atom 5. The goal A =:= 5 succeed, which confirms that A has the value 5.)
18. ?- 5+4 = 2+7.
19. no
20. (Under ==, the reason is explained.)
22. ?- 5+A = 5+3.
23. A = 3
24. (Variable A bound to atom 3 and makes the two terms identical. Both the terms are 5+3, not the number 8.)
26. ?- likes(A, john) = likes(B, prolog)
27. no
28. (No unification makes the atoms john and prolog identical.)

Non-Unification (\=)

Given goal

1. Term1 \= Term2

If Term1 = Term2 fails, the above goal succeeds. That means two terms cannot be unified. Otherwise the above goal fails.

**For example:**

1. ?- 5+4 \= 2+7.
2. yes
4. ?- likes(A, mary) \= likes(prolog, B).
5. no
6. (Variables A and B bound to atoms mary and prolog respectively and makes the terms identical.)
8. ?- likes(A, mary) \= likes(A, prolog)
9. A = \_

Logical Operator in Prolog

The description of the two operators is explained in this section. It takes arguments, and those arguments are called terms.

Not operator

To provide the negation, **not/1** prefix operator can be placed before any goal. If the original goal fails, the negation goal succeeds. If the original goal succeeds, the negation goal fails.

The use of not/1 is explained in the following example. Suppose a single clause contained in the database.

1. dog(boxer).

**For example:**

Play Video

1. ?- not dog(boxer).
2. no
3. ?- dog(bombay).
4. no
5. ?- not dog(bombay).
6. yes
7. ?- A is 5, A is 5.
8. A = 5
9. ?- A=5, not A is 5.
10. no

Disjunction Operator

The **';/2'** is the disjunction operator. It is an infix operator, which represents 'or'. It takes two arguments, and both the arguments are goals. If either Goal1 or Goal2 succeeds, the Goal1;Goal2 succeeds.

**For example:**

1. ?- 9<4; 10 is 7+3.
2. Yes
3. ?- 5\*5 =:= 25; 12=5+4.
4. Yes

Input and Output Terms in Prolog

Prolog provides the facility to enable the input and output either of character or of terms. It is simpler to use terms. Initially, we will assume that all input is from the keyboard of user and all output is at the screen of the user. In this section, we will describe input and output using external files like CR-ROM or hard disk. Just like many other built-in predicates, the predicates of input and output are unsatisfiable that means while backtracking they always fail.

Output Terms

To provide the input terms, we will use a built-in predicate **write/1**. This predicate takes a single argument. When we evaluate the predicate, the term will be written to the current output stream. By default, the current output stream is the screen of user.

In this tutorial, we have been used many times a built-in predicate **n1/0**. This predicates takes no arguments. When we evaluate nl predicate, this will cause a new line to be output to the current output stream.

**For Example:**

1. ?- write('examples of output term'),nl.
2. examples of output term
3. yes
5. ?- write(30),nl.
6. 30
7. yes
9. ?- write(pred1(x, y, z)),nl.
10. pred1(x, y, z)
11. yes
13. ?- write([x,y,z[a,b,c,d]]),nl.
14. [x,y,z[a,b,c,d]]
15. yes
17. ?- write('example of nl'),nl,nl,write('javatpoint'),nl,write('end of example')
18. example of nl
20. javatpoint
21. end of example

If the atoms have quoted on input, because of write predicate they are not quoted on output. If we want to output the quotes, we can use the **writeq/1** predicate.

**For example:**

1. ?- writeq('example of output stream'),nl.
2. 'example of output stream'
3. yes
5. ?- writeq(cat),nl.
6. cat
7. yes
9. ?- writeq(30)
10. 30
11. yes
13. ?- writeq('cat'),nl.
14. cat
15. yes

Input Terms

To provide the input terms, we will use a built-in predicate **read/1**. This predicate takes a single argument and that argument must be a variable. When we evaluate this predicate, due to this the next term will be read from the current input stream. By default, the current input stream is the keyboard of the user.

In the input stream, at least one white space character or new line and a dot('.') follows the term. The white space and dot are not part of the term, but they are read in.

Note that if the user input is required from the keyboard, a prompt character like colon will be displayed. Before Prolog accepts the input from user, we have to press the 'return' key.

When [Prolog](https://www.javatpoint.com/prolog) evaluates a **read** goal, the input term is unified with the variable of the argument. If the variable of the argument is unbound, it is bound to the input value.

**For example:**

1. ?- read(A).
2. : 30.
3. A = 30
5. ?- read(A).
6. : mary.
7. A = mary
9. ?- read(B).
10. : 'example of input term'.
11. B = 'example of input term'
13. ?- read(A).
14. : pred(x,y,z).
15. A = pred(x,y,z)
17. ?- read(C).
18. : [x,y,pred(u,v,w),[c,b,a]].
19. C = [x,y,pred(u,v,w),[c,b,a]]

If the variable of argument is already bound, the goal succeeds if and only if the previously bound value and input term are identical.

**For example:**

1. ?- A= hary,read(A).
2. : jack.
3. no
5. ?- A=hary,read(A).
6. : hary.
7. A = hary

Input and Output Using Characters

The input term and output term are straightforward. Sometimes it is complex to use the full stops and quotes, and it is not always suitable. For example, to define a predicate that would read characters in series from the keyboard and count the number of vowels is really tedious. For this kind of problem, a better approach is to input a character at a time. To apply this approach, we have to know the ASCII value of the characters. ASCII (American Standard Code for Information Interchange) value is an integer from 0 to 255.

All non-printing characters and printing characters have their corresponding ASCII value. ASCII value of characters, which is less than or equal to 32 is called as control characters or white space characters.

The following table shows the ASCII values of printable characters and some others.

|  |  |  |
| --- | --- | --- |
| **Number** | **Character** | **Description** |
| 9 | Tab |  |
| 10 | end of record |  |
| 32 |  | space |
| 33 | ! | exclamation mark |
| 34 | “ | quotation mark |
| 35 | # | number sign |
| 36 | $ | dollar sign |
| 37 | % | percent sign |
| 38 | & | ampersand |
| 39 | ‘ | apostrophe |
| 40 | ( | left parenthesis |
| 41 | ) | right parenthesis |
| 42 | \* | asterisk |
| 43 | + | plus sign |
| 44 | , | comma |
| 45 | - | hyphen |
| 46 | . | period |
| 47 | / | slash |
| 48 | 0 | 0 digit |
| 49 | 1 | 1 digit |
| 50 | 2 | 2 digit |
| 51 | 3 | 3 digit |
| 52 | 4 | 4 digit |
| 53 | 5 | 5 digit |
| 54 | 6 | 6 digit |
| 55 | 7 | 7 digit |
| 56 | 8 | 8 digit |
| 57 | 9 | 9 digit |
| 58 | : | colon |
| 59 | ; | semicolon |
| 60 | < | less-than |
| 61 | = | equals-to |
| 62 | > | greater-than |
| 63 | ? | question mark |
| 64 | @ | at sign |
| 65 | A | uppercase A |
| 66 | B | uppercase B |
| 67 | C | uppercase C |
| 68 | D | uppercase D |
| 69 | E | uppercase E |
| 70 | F | uppercase F |
| 71 | G | uppercase G |
| 72 | H | uppercase H |
| 73 | I | uppercase I |
| 74 | J | uppercase J |
| 75 | K | uppercase K |
| 76 | L | uppercase L |
| 77 | M | uppercase M |
| 78 | N | uppercase N |
| 79 | O | uppercase O |
| 80 | P | uppercase P |
| 91 | Q | uppercase Q |
| 82 | R | uppercase R |
| 83 | S | uppercase S |
| 84 | T | uppercase T |
| 85 | U | uppercase U |
| 86 | V | uppercase V |
| 87 | W | uppercase W |
| 88 | X | uppercase X |
| 89 | Y | uppercase Y |
| 90 | Z | uppercase Z |
| 91 | [ | left square bracket |
| 92 | \ | backslash |
| 93 | ] | right square bracket |
| 94 | ^ | Caret |
| 95 | \_ | underscore |
| 96 | ` | grave accent |
| 97 | A | lowercase a |
| 98 | B | lowercase b |
| 99 | C | lowercase c |
| 100 | D | lowercase d |
| 101 | E | lowercase e |
| 102 | F | lowercase f |
| 103 | G | lowercase g |
| 104 | H | lowercase h |
| 105 | I | lowercase i |
| 106 | J | lowercase j |
| 107 | K | lowercase k |
| 108 | L | lowercase l |
| 109 | M | lowercase m |
| 110 | N | lowercase n |
| 111 | O | lowercase o |
| 112 | P | lowercase p |
| 113 | Q | lowercase q |
| 114 | R | lowercase r |
| 115 | S | lowercase s |
| 116 | T | lowercase t |
| 117 | U | lowercase u |
| 118 | V | lowercase v |
| 119 | W | lowercase w |
| 120 | X | lowercase x |
| 121 | Y | lowercase y |
| 122 | Z | lowercase z |
| 123 | { | left curly brace |
| 124 | | | vertical bar |
| 125 | } | right curly brace |
| 126 | ~ | tilde |

Output and Input of Characters in Prolog

Outputting Characters

We can use the built-in predicate **put/1** to provide the output of characters. This predicate takes a single argument, and that argument is a number from 0 to 255 or an expression. When we evaluate **put** goal, due to this, a single character will output to the current output stream. This character corresponds to a numerical value and produces the ASCII value of its argument.

**For example:**

1. ?- put(74),nl.
2. J
3. yes
5. ?- put(104),nl.
6. h
7. yes
9. ?- put(92),nl.
10. \
11. yes

Inputting Characters

We can use two built-in predicates **get/1** and **get0/1** to provide the input of a single character. The predicate **get0** takes a single argument as a variable. When we evaluate get0 goal, due to this, a character will be read from the current input stream.

Note that if the user input is required from the keyboard, a prompt character like colon will be displayed. Before Prolog accepts the input from the user, we have to press the 'return' key.

When Prolog evaluates a **get0** goal, the variable is unified with the ASCII value of the input character. If the variable of the argument is unbound, it is bound to the ASCII value.

**For example:**

1. ?- get0(A),nl.
2. : f
3. A = 102
5. ?- get0(B),nl.
6. : R
7. B = 82
9. ?- get0(C),nl.
10. : =
11. C = 61

If the variable of the argument is already bound, the goal succeeds if and only if the ASCII value of input character and numerical value is the same.

**For example:**

1. ?- get0(A),nl.
2. : f
3. A = 102
5. ?- X is 64, get0(X).
6. : @
7. X = 64
9. ?- A=cat,get0(A).
10. : @
11. no
13. ?- A=62.3,get0(A).
14. : =
15. no
17. ?- A=62,get0(A).
18. : >
19. A = 62

The **get** predicate takes a single argument as a variable. When we evaluate **get** goal, due to this, the next non-white-space character will be read from the current input stream. A non-white-space character means a character with an ASCII value less than or equal to 32. Same as get0 predicate, the variable and ASCII value of the character unifies.

**For example**

1. ?- get(A).
2. : P
3. A = 80
5. ?- get(X).
6. :        P
7. X = 80
9. ?- get(A).
10. : ?
11. A = 63

Examples of Using Characters

In the **first example**, we will show how to read the character in a series from the keyboard. It finishes with \* and outputs their corresponding ASCII values one per line.

The **readin** predicate is defined recursively. When we evaluate this predicate, due to this, we will input a single character, and variable A will be bound to its ASCII value. In the process(X) goal, the action depends on whether A contains the value 42 or not, which signifies a \* character. If not, A's value is output and followed by a new line and again call to readlin predicate. If it has, the evaluation of process(X) goal stops. When Prolog reads the \* character, the process will stop. In the following example, 69, 120, 97, 109, etc. shows the ASCII value of characters E, x, a, m, etc.

1. readlin :- get0(A), process(A).
2. process(42).
3. process(A) :- A=\= 42, write(A), nl, readlin.
4. ?- readlin.
5. : Example of Prolog\*
6. 69
7. 120
8. 97
9. 109
10. 112
11. 108
12. 101
13. 32
14. 111
15. 102
16. 32
17. 80
18. 114
19. 111
20. 108
21. 111
22. 103
23. yes

In the **second example**, a number of characters are output instead of the ASCII value of input character. For this, we will use the **count** predicate. This predicate takes two arguments. The first argument read as 'the number of characters counted so far', and the second argument read as 'the total number of characters before \*'.

1. go(Total) :- count(0, Total).
2. count(Oldcount, Result) :-
3. get0(A), process(A, Oldcount, Result).
4. process(42, Oldcount, Oldcount).
5. process(A, Oldcount, Result) :-
6. A =\= 42, New is Oldcount+1, count(New, Result).
7. ?- go(T).
8. : javatpoint Tutorial\*
9. T = 19
11. ?- go(T)
12. : \*
13. T = 0

File Input and Output in Prolog

File Output: Changing the Current Output Stream

* The **tell/1** predicate is used to change the current output stream. This predicate has a single argument as variable or atom, which represents the name of a file like **tell('outfile.txt').**
* When **tell** goal is evaluated, due to this, the name of the file becomes the current output stream. The specified name file is first created if the file is not already open. If the existing file has the same name, it is deleted.
* When we select a new current output stream, the file which corresponds to the previous current output stream remains open. Using the **told** predicate, we can close only the current output stream.
* The **user** is the default current output stream, i.e., terminal of the user. Using either **told(user)** or **told** predicate, we can restore this value.
* The **told/0** predicate is a built-in predicate that has no argument. When the **told** goal is evaluated, it closes the current output file and resets the current output stream to the user, i.e., terminal of the user.
* The **telling/1** predicate is a built-in predicate that has one argument. This argument must be an unbound variable. When the **telling** goal is evaluated, due to this, the variable to be bound to the current output stream name.

**Output to a File**

The above definition of **tell** says that 'any existing file which has the same name is deleted'. Some application has another possibility like the file is not deleted, and any output is placed after the end of existing file contents. In any practical implementation of [Prolog](https://www.javatpoint.com/prolog), 'append' and 'overwrite' options are available, but it will not use the tell predicate, it may involve using the different predicate.

File Input: Changing the current Input Stream

* The **see/1** predicate is used to change the current input stream. This predicate has a single argument as variable or atom, which represents the name of a file like **see('myfile.txt')**.
* When **see** goal is evaluated, due to this, the name of the file becomes the current input stream. The specified name file is first open(only for read access) if the file is not already open. An error will be generated if we are unable to open a given name of the file.
* When we select a new input stream, the file which corresponds to the current input stream remains open. Using the **seen** predicate, we can only close the current input stream.
* The **user** is the default current input stream, i.e., terminal of the user. Using either **see(user)** or **seen/0** predicate, we can restore this value.
* The **seen/0** predicate is a built-in predicate that has no argument. When the **see** goal is evaluated, it closes the current input file and reset the current input stream to the user, i.e., terminal of the user.
* The **seeing/1** predicate is a built-in predicate that has one argument. This argument must be an unbound variable. When the **seeing** goal is evaluated, due to this, the variable to be bound to the current input stream name.

**Reading from Files: End of Files**

When we evaluate the **read(A)** goal, and if the end of file is encountered, variable A will be bound to atom end\_of\_file.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

When we evaluate the **get0(A)** or **get(A),** and if the end of file is encountered, variable A will be bound to [ASCII](https://www.javatpoint.com/ascii-full-form) value. The range of ASCII values is 0 to 255. This will typically be -1, but on the implementation of Prolog, it may vary one to another.

**Reading from Files: End of Record**

On the basis of the Prolog version, there may be incompatibility for the input of character between reading the end of record from a file and from a terminal of the user.

At the user's terminal, the end of a line of input will be indicated using the character with 13 ASCII value. In a file, the end of record will generally be indicated using the two ASCII values that are 13 followed by 10.

The below program reads a series of characters from the keyboard and prints that characters one per line.

1. readline :- get0(A), process(A).
2. process(13).
3. process(A) :- A=\= 13, put(A), nl, readline.

Instead of **write**, we use **put** that test for **ASCII** value 13. In this, we don't need to not use the \* character to indicate 'end of input'.

1. ?- readline.
2. : Prolog test
3. P
4. r
5. o
6. l
7. o
8. g
10. t
11. e
12. s
13. t
14. yes

Examples of Files

Example 1:

In the following program, we define a **readterms** predicate. This predicate is used to read the first four terms from the given files. It outputs those terms to another specified file, one per line.

1. readterms(Infile, Outfile) :-
2. see(Infile), tell(Outfile),
3. read(X1), write(X1), nl, read(X2), write(X2), nl
4. read(X3), write(X3), nl, read(X4), write(X4), nl
5. see, told.

Assume that textfile.txt contains the three lines of contents as follow:

1. 'first term'.   'second term'.
2. 'third term'.
3. 'fourth term'. 'fifth term'.

The following output will be generated using the **readterms**.

1. ?- readterms('textfle.txt', 'outfile.txt').
2. yes

The above creates a file that contains a four line of text.

Play Video

1. first term
2. second term
3. third term
4. fourth term

The above shows that the **readterms** definition is correct as far as it goes. Due to the final two terms, i.e., **seen** and **told**, the current input stream and current output stream to be set to the user. If a large program uses **readterms** as a subgoal, and when the current input stream and current output stream was called, it was not necessarily both user, then the above definition creates a problem.

When we evaluate the goal **readterms**, to restore the original input and output streams as a final step, it becomes a good programming practice. In the body of a rule, we can achieve the input using the **see(P)** and **seeing(P).** We will put these predicate before and after the other terms. In the starting, **P** binds to the current input stream name. In the last, it resets the current output stream to **P**.

In the body of a rule, we can achieve the output using the **tell(X)** and **telling(X).** We will put these predicate before and after the other terms. In the starting, **X** binds to the current output stream name. In the last, it resets the current output stream to **X**.

By the above conventions, the revised definition of **readterms** is described below:

1. readterms(Infile, Output) :-
2. seeing(P), see(Infile), telling(X), tell(Outfile),
3. read(X1), write(X1), nl, read(X2), write(X2), nl,
4. read(X1), write(X1), nl, read(X2), write(X2), nl,
5. seen, see(P), told, tell(X).

Example 2:

To copy the input of characters, we define a **copychars** predicate at the terminal of the user to a specified file until **!** character is entered.

In the following program, the value of current input and output stream is saved and restored by **copychars**. The **copy\_character** task is left, which is defined recursively in the same way to **readin** in the first example of **Output and Input characters** of **Example of using characters**.

1. copychars(Outfile) :- telling(X), tell(Outfile),
2. copy\_characters, told, tell(X).
3. copy\_characters :- get0(A), process(A).
4. /\* ASCII value of ! character is 33 \*/
5. process(33).
6. process(A):-A=\=33, put(A), copy\_characters.

In the following way, use **copychars**:

1. ?- copychars('myfile.txt').
2. : javatpoint
3. yes

This will place the [javatpoint](https://www.javatpoint.com/) character in the myfile.txt file.

Loops in Prolog

The looping facility is contained in most of the programming languages. Looping is used to enable a set of instructions to be repeatedly executed either a fixed number of times or until a given condition met. Prolog has no looping facility, but we can obtain a similar effect. Using this effect, we can evaluate a sequence of goals repeatedly. In various ways, this can be done like built-in predicates, recursion, backtracking, or a combination of these.

Looping a fixed number of times

To execute the instruction a fixed number of times, many programming languages provide 'for loop'. In [Prolog](https://www.javatpoint.com/prolog), there is no such facility available directly, but using recursion, we can obtain a similar effect, which is shown in the following programs:

**Example 1:**

This program outputs the integer value from a specified value down to 1.

Play Video

1. loop(0).
2. loop(N) :- N>0, write('value of N is: '), write(N), nl.
3. S is N-1, loop(S).

In the above example, we define the **loop** predicate in terms of itself. The **second clause** read as 'to loop from N, first write the N's value, then subtract one to provide S, then loop from M'. This process will terminate using the **first clause**: 'when the argument is 0, then stop or do nothing'. Here the first clause is called as terminated condition.

1. ?- loop(4).
2. value of N is: 4
3. value of N is: 3
4. value of N is: 2
5. value of N is: 1
6. yes

In the second clause, we are using two goals S is N-1, loop(S) for the loop predicate. Prolog will not work on the obvious alternative of the loop(N-1). When evaluating goals with **is** functor, or with relational operators, only then Prolog evaluates expressions like N-1. If we are using N-1 as an argument of the predicate, it will mean that the term with arguments N and 1, and with infix operator -(minus sign).

**Example 2:**

The following program generates an integer value from first to last inclusive.

1. output\_values(Last\_value, Last\_value) :- write(Last\_value),nl,
2. write('javatpoint Tutorial'), nl.
3. output\_value(First\_value, Last\_value) :- First\_value =\= Last\_value, write(First\_value),
4. nl,N is First\_value+1, output\_values(N,Last\_value).

The above **output\_value** has two arguments. It can be read as 'output the integers from First to Last inclusive'. The above loop terminates when the value of both arguments are same.

1. ?- output\_values(3,9).
2. 3
3. 4
4. 5
5. 6
6. 7
7. 8
8. 9
9. javatpoint Tutorial
11. yes

**Example 3:**

In this program, we will define a predicate that can find the sum of integers from 1 to N. Let's assume N=100.

For this, we start with 1, then add 2, then add 3, ….., then add 100. If we re-expressed this process declaratively in terms of itself, the process will be much easier to program.

100 plus the sum of the first 99 integers is the sum of the first 100 integers.

99 plus the sum of the first 98 integers is the sum of the first 99 integers.

98 plus the sum of the first 97 integers is the sum of the first 98 integers.

…………………………………………………………………………….

3 plus the sum of the first 2 integers is the sum of the first 3 integers.

2 plus the sum of the first 1 integer is the sum of the first 2 integers.

1 is the sum of the first one integer.

To consider this process, we have two distinct cases, the general case and the terminating case. In general case: 'the sum of first N integers is the sum of first N-1 integers, plus N'. In terminating case: 'the sum of first 1 integer is 1'.

1. /\* sum of integers from 1 to N inclusive \*/
2. sumto(1, 1).
3. sumto(N, M) :- N>1, N1 is N-1, sumto(N1, M1), M is M1+N.
4. ?- sumto(100, N).
5. N= 5050
6. ?- sumto(1, 1).
7. yes

For holding the value of N-1, it is essential to use the additional N1 variable. Writing sumto(N-1, M1), etc. instead would work not work correctly. N-1 is not a numerical value, but it is a term.

**Example 4:**

This program is used to read the first 6 terms from the given file and then write these terms to the current output stream. Just like Example 1, it uses a method 'counting down'.

1. read\_six(Infile) :- seeing(M), see(Infile),
2. process\_terms(6), seen, see(M).
3. process\_terms(0).
4. process\_terms(N) :- N>0, read(A), write(A), nl, N1 is N-1,
5. process\_terms(N1).

Looping until a Condition is Satisfied

To execute a set of instructions repeatedly until a given condition is met, many programming languages provide an 'until loop'. Again, Prolog has no looping facility, but we can obtain a similar effect in several ways.

Recursion

The following program shows the use of recursion. It reads the terms which are entered by the user from the keyboard, and it produces term as the output on the screen until the end is encountered.

1. go:- loop(start)
2. loop(end).
3. loop(A) :- A\=end, write('The value is'), read(Word),
4. write('Input value is '), write(Word), nl, loop(Word).
5. ?- go.
6. The value is: javatpoint.
7. Input value is javatpoint
8. The value is: tutorial.
9. Input value is tutorial
10. The value is: end.
11. Input value is end
12. yes

In the above example, the program can be written in a single clause using the ;/2 disjunction operator.

1. loop:- write('The value is'), read(Word),
2. write('Input value is '), write(Word), nl,
3. (Word=end; loop).

If variable Word is bound to the atom end, the disjunction goal **(Word=end;loop)** succeeds. If it is not, the system recursively attempts to satisfy the goal **loop**.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. ?- loop
2. The value is: javatpoint.
3. Input value is javatpoint
4. The value is: tutorial.
5. Input value is tutorial
6. The value is: end.
7. Input value is end
8. yes

The following recursive program repeatedly prompts the user to enter a term until the user enters yes or not.

1. get\_answer(Ans) :- write('Enter answer to question').
2. nl, get\_answer1(Answer).
3. get\_answer1(Answer) :-
4. write('yes or no answer'),
5. read(X),
6. ((valid(X), Answer=X, write('The correct answer is '),
7. write(X), nl); get\_answer1(Answer)).
8. valid(yes).  valid(no).
9. ?- get\_answer(Myanswer).
10. Enter answer to question
11. yes or no answer: possibly.
12. yes or no answer: maybe.
13. yes or no answer: no.
14. The correct answer is no
15. Myanswer = no

Use of 'repeat' predicate

In the Prolog program, the easiest way to provide the type of looping is not always recursion. Another method to provide the looping is built-in predicate **repeat**.

The goal **repeat** does not mean that it repeats anything. Whenever it called, it merely succeeds. The value of **repeat** also succeeds while backtracking. Due to this effect, the order of evaluating goals is changed by any other succeeding goal from 'right to left' back to 'left to right'. This predicate can also create a looping effect.

The following program prompts the user repeatedly to enter a term until the user enters either yes or no. This program is an alternative to the previous program.

1. get\_answer(Answer) :-
2. write('Enter answer to question'), nl,
3. repeat, write('yes or no answer'), read(Answer),
4. valid(Answer), write('The correct answer is '), write(Answer), nl.
5. valid(yes).  valid(no).

In the body of **get\_answer**, the first five goals will always succeed. When we evaluate the 5th goal **read(Answer),** it will prompt the user to specify the term. The goal **valid((Answer)** will fail and say unsure if the input of term is not yes or no. Now [Prolog](https://www.javatpoint.com/prolog) will backtrack over **read(Answer)** and **write('yes or no answer')**. Both the goals are unsatisfiable that means while backtracking, it will always fail.

Now backtracking will go to the **repeat** predicate and succeed. Due to this, the evaluation will again proceed forward(left-to-right) and **write('yes or no answer')** and **read(Answer)** both succeeding. Now Prolog will further evaluate the goal **valid(Answer)**.

On the basis of value of **Answer**, which is input by the user, the goal **valid(Answer)** will succeed or fail. If this goal succeeds, the final three goals will succeed that is **write('The correct answer is '),** **write(Answer),** and **nl**. If this goal fails, Prolog will backtrack to the predicate repeat. The overall effect shows that **write('yes or no answer')** and **read(Answer)** both goals are repeatedly called until goal **valid(Answer)** is satisfied, which is the terminating condition.

1. ?- get\_answer(A).
2. Enter answer to question
3. Yes or no answer: unsure.
4. Yes or no answer: possibly.
5. Yes or no answer: yes.
6. The correct answer is yes
7. A = yes

In the body of a clause, the left of **repeat** goals will never be reached on backtracking.

The following program reads a sequence of terms from the specified file. It outputs those terms to the current output stream until we encountered the end of term.

1. readterms(Infile) :-
2. seeing(X), see(Infile).
3. repeat, read(Y), write(Y), nl, Y = end,
4. seen, see(user).

The following program defines a loop between the goals **repeat** and **Y=end**, just like the previous program.

The following line contains in the myfile.txt file.

1. 'first term'.
2. 'second term'.
3. 'third term'.
4. 'fourth term'.
5. 'fifth term'.
6. 'sixth term'.
7. 'seventh term'.
8. 'eighth term'.
9. end.

If we call readterms, the following output will be produced:

1. ?- readterms('myfiles.txt').
2. first term
3. second term
4. third term
5. fourth term
6. fifth term
7. sixth term
8. seventh term
9. eight term
10. end
11. yes

Backtracking with Failure

While backtracking or 'standard' evaluation left-to-right, the **fail** predicate always fails, as the name implies. We can take advantage of this by combining it with Prolog's automatic backtracking to find all the clauses in the database with a specified property.

Searching the database of Prolog

Suppose, the clauses contained in the database as follows:

1. dog(pug).
2. dog(boxer).
3. dog(rottweiler).

Using the predicate **alldogs**, we can process each clause of the **dog** in turn as follows:

1. alldogs :- dog(A), write(A), write('  is a dog'), nl, fail.
2. alldogs.

When we call the **alldogs**, due to this **dog(A)** will be matched with the clauses of **dog** in the database. In the starting, variable A will be bound to atom pug and produced the output as 'pug is a dog'. In the first clause of predicate **alldogs**, the final goal will cause evaluation to fail. Then, Prolog will backtrack over two goals **write** and **nl**, that goals are unsatisfiable until Prolog reaches to **dog(A)**. Then the second time this goal succeeds, and due to this, variable A will be bound to boxer.

Play Video

This process will continue until pug, boxer, and rottweiler have all been output, and when evaluation again fails. In the database, there is no further dog, this time call to **dog(A)** will also fail. Due to this, the first clause of **alldogs** will fail. Now the second clause of **alldogs** is examined by Prolog, and this will succeed. After that, the evaluation will stop.

Due to this effect, [Prolog](https://www.javatpoint.com/prolog) will generate a loop through the database. To satisfy the goal **dog(A),** Prolog will find all possible values of **A**.

1. ?- alldogs.
2. pug is a dog
3. boxer is a dog
4. rottweiler is a dog.
5. yes

For the predicate **alldogs**, the second clause is very important. It is used to ensure that the goal succeeds after the database has been searched. It will succeed with only the first line, any call to predicate **alldogs** will eventually fail.

1. alldogs :- dog(A), write(A), write('  is a dog'), nl, fail.
2. ?- alldogs.
3. pug is a dog
4. boxer is a dog
5. rottweiler is a dog.
6. no

The next program is used to search the database, which contains the clauses. The clauses contain the name, occupation, residence, and age of the number of people.

Suppose these five clauses contained in the database as follows:

1. person(tom, cruise, engineer, canada, 30).
2. person(angelina, jolie, doctor, england, 27).
3. person(tom, cruise, teacher, australia, 45).
4. person(vin, diesel, actor, brazil, 41).
5. person(johnny, depp, teacher, colombia, 38).

Using the **allteachers** predicate, we can find the name of all the teachers.

1. allteachers :- person(Firstname, Lastname, \_, \_, teacher),
2. write(Firstname), write(' '), write(Lastname), nl,
3. fail.
4. allteachers.

In this case, the effect of using backtracking with failure is to find all teachers in the database.

1. ?- allteachers.
2. tom cruise
3. johnny depp
4. yes

Both the teachers are found, and if we omit the second clause of **allteachers**, the evaluation of **allteachers** will end with failure. If the goal is entered at the system prompt, this has no importance. In the body of a rule, if **allteachers** were used as a goal, it would ensure that it always succeeded.

To search the database, it is not always necessary to use 'backtracking with failure'. For example, we will find all the people in the database using the **somepeople/0** predicate. For this, we will down to jolie, using only standard backtracking.

1. somepeople :- person(Firstname, Lastname, \_, \_, \_),
2. write(Firstname), write(' '), write(Lastname), nl,
3. Lastname=jolie.
4. somepeople.

If the variable Lastname is bound to jolie, the goal **Lastname= jolie** will succeed. If it is not, it will fail. The effect is to search the database down to and include the clause of person with second argument jolie.

1. ?- somepeople.
2. tom cruise
3. angelina jolie
4. yes

The Cut Predicate

In this section, we will provide two examples of predicate definitions. These definitions look correct, but when we use it with backtracking, it will be erroneous.

Example 1:

In this example, we will use a predicate **larger**. In the first two arguments, it takes the larger value. It returns the answer as the third argument value.

1. larger(X, Y, X) ;- X>Y.
2. larger(X, Y, Y).

In this, we search the clauses from 'top to bottom'. If X is less than or equal to Y, the second clause will only be assumed to apply. In the following example, the first two arguments have 7 and 5 value, and when we test the definition with 7 and 5, it gives a correct answer as follows:

1. ?- larger(7, 5, A).
2. A = 7

At this stage, if the system is forced by the user to backtrack, it will examine the second clause for larger, and after that, it will generate an incorrect second answer.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. ?- larger(7, 5, A).
2. A = 7;
3. A = 5
4. ?-

Example 2:

In this example, we will use the definition of **sumto/2** predicate. This definition looks correct, but it has a serious flaw.

The goal **sumto(N, S)** is used to calculate the sum of integers from 1 to N. It generates the result as the Y's value.

1. sumto(1, 1).
2. sumto(N, S) :- N1 is N-1, sumto(N1, S1),
3. S is S1+N.
4. ?- sumto(3, S).
5. S = 4

If backtracking is forced, due to this, the system will crash and generate a cryptic error message like 'stack overflow'. When whilst evaluates the goal sumto(3, S), the solution for sumto(1, S) is found by the Prolog. The first clause is rejected while backtracking. Now, the system uses the second clause to satisfy the goal. This causes it to subtract one from one, and then the system evaluates the goal sumto(0, S). When the system does this in turn, it evaluates sumto(-1, S1), then sumto(-2, S1), then sumto(-3, S1), and so on. This process will stop when the system runs out of memory.

In the definition of predicates, **Example 1** and **Example 2** could both be remedied using the additional goals. For example, for this, we will change the definition of larger in the second clause as follows:

1. larger(X, Y, Y) :- X=<Y.

We will also change the definition of **sumto** in the second clause as follows:

1. sumto(N, S) :- N>1, N1 is N-1, sumto(N1, S1), S is
2. S1+N.

It is considerably more difficult to identify such additional terms in other cases.

Using the **cut**, we can avoid unwanted backtracking. The **cut** is represented by **!**. In the body of a rule, when ! goal evaluates for the first time, it always succeeds. It always fails while backtracking. The further evaluation of the current goal always fails, so it prevents this.

**Example 1 (revised)**

1. larger(X, Y, X) :- X>Y, !.
2. larger(X, Y, Y).
3. ?- larger(7, 5, A)  .
4. A = 7
5. ?-

**Example 2 (revised)**

1. sumto(1, 1) :- !.
2. sumto(N, S) :- N1 is N-1, sumto(N1, S1),
3. S is S1+N.
4. ?- sumto(5, S).
5. S = 15
6. ?-

When we use backtracking over cut, it is abandoned to evaluate the current clause of **sumto** or **larger**. It is used to prevent the evaluation of any other clauses for that predicate.

Example 3:

This incorrect program uses a **classify/2** predicate. This predicate is used to classify a number, which can be zero, positive, and negative. In the following program, the first clause deals with zero value of the first argument. The second clause is used to deal with a negative value. The third clause is used to deal with a positive value.

1. classify(0, zero).
2. classify(X, negative) :- X<0.
3. classify(X, positive).
4. ?- classify(0, X).
5. X = zero;
6. X = positive
8. ?- classify(-5, Y).
9. Y = negative;
10. Y = positive
11. ?-

By changing the third clause, the above can be rectified as follows:

1. classify(X, positive) : X>0.

Or we can also use the cut as follows:

**Example 3 (revised)**

1. classify(0, zero) :- !.
2. classify(X, negative) :- X<0, !.
3. classify(X, positive).
4. ?- classify(0, X).
5. X = zero
7. ?- classify(-5, X)
8. X = negative
9. ?-

So, we have rectified all the incorrect programs rather than using cut. We add an additional goal to one of the clauses. This approach is much better than cut. A more difficult case is shown by the following program.

Example 4:

In the following program, we are going to use a predicate **go**. This predicate is used to prompt the user repeatedly for input until the user enters a positive number. In the definition of predicate **classify**, the lack of cuts leads to incorrect answers.

1. classify(0, zero).
2. classify(X, negative) :- X<0.
3. classify(X, positive).
4. go :- write(start), nl.
5. repeat, write('specify a positive number'), read(X),
6. classify(X, Type), Type=positive,
7. write('Positive number is '), write(X), nl.
8. ?- go.
9. start
10. specify a positive number: 30.
11. positive number is 30
12. yes
14. ?- go.
15. start
16. specify a positive number: 0.
17. positive number is 0
18. yes
20. ?- go.
21. start
22. specify a positive number: -5.
23. positive number is -5
24. yes

The expected behavior of **go** is given by changing the definition of **classify** to one, which is defined in the above Example 3 (revised).

**Example 4 (revised)**

1. classify(0, zero) :- !.
2. classify(X, negative) :- X<0, !.
3. classify(X, positive).
4. go :- write(start), nl,
5. repeat,
6. write('specify a positive number'), read(X),
7. classify(X , Type),
8. type = positive,
9. write('positive number is '), write(X), nl.
10. ?- go.
11. start
12. specify a positive number: 0.
13. specify a positive number: -5.
14. specify a positive number: -9.
15. specify a positive number: 30.
16. positive number is 30
17. yes

Cut with Failure

In this section, we will specify another use of '**cut**'. It is used to specify exceptions to general rules. In the following example, we have names of birds in the database as follows:

1. bird(crow).
2. bird(sparrow).
3. bird(parrot).
4. bird(penguins).
5. bird(dove).
6. bird(robin).
7. bird(turkey).
8. bird(hawk).
9. bird(goose).
10. bird(swallow).
11. bird(pigeon).
12. bird(woodpecker).

The following shows the natural rule to add this:

1. can\_fly(A) :- bird(A).

The above rule means that 'all birds can fly'.

The above rule is very general. We cannot ensure that the goal **cry\_fly(penguins)** will always fail? So, we are going to change the predicate **can\_fly** definition in this approach as follows:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. can\_fly(penguins) :- fail.
2. can\_fly(A) :- bird(A).

However, the desired result is not provided by the above:

1. ?- can\_fly(parrot).
2. yes
4. ?- can\_fly(penguins).
5. yes

The head of the first **can\_fly** clause and the goal **can\_fly(penguins)** are matched with each other. In the body of that clause, we are trying to satisfy the goal, and the goal obviously **fails**. Now, the system looks at the second **can\_fly** clause. The head and the goal match with each other, and the goal is also satisfied in the body of the clause, i.e., **bird(A),** so the goal **can\_fly(penguins)** succeeds. But this is not the desired result.

We can achieve the desired result by replacing the clause **can\_fly** by

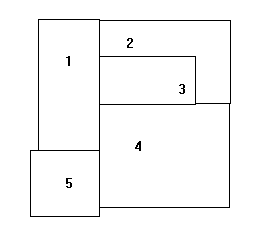
1. can\_fly(penguins) :- !, fail.
2. can\_fly(A) :- bird(A).
3. ?- can\_fly(parrot).
4. yes
6. ?- can\_fly(penguins).
7. no

As before, the head of the first **can\_fly** clause and the goal **can\_fly(penguins)** are matched with each other. In the body of that clause, we are trying to satisfy the goal, the goal obviously **fails**. But here the cut prevents it from backtracking the system, so the goal **can\_fly(penguins)** fails.

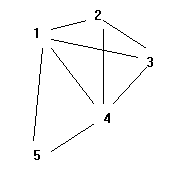
**Cut with failure** is the combination of **fail** and goals **!**.

Map Coloring in Prolog

In mathematics, the famous problem was coloring adjacent planar regions. Two adjacent regions cannot have the same color no matter whatever color we choose. Two regions which share some boundary line are considered adjacent to each other. In the following map, the region is shown by some numerical names.



The above numerical name shows which regions are adjacent. Now we are going to consider the following graph:



In the above graph, the original boundaries are erased. Now, we have an arc between the two regions name. The arc shows that both regions are adjacent in the original drawing. All the original adjacency information is conveyed by the adjacency graph. Using the following facts or unit clause, the adjacent information can represent in Prolog.

1. adjacent(1,2).         adjacent(2,1).
2. adjacent(1,3).         adjacent(3,1).
3. adjacent(1,4).         adjacent(4,1).
4. adjacent(1,5).         adjacent(5,1).
5. adjacent(2,3).         adjacent(3,2).
6. adjacent(2,4).         adjacent(4,2).
7. adjacent(3,4).         adjacent(4,3).
8. adjacent(4,5).         adjacent(5,4).

In prolog, if we load these clauses, we will get the following behavior for some goals.

Play Video

1. ?- adjacent(2, 3)
2. yes
3. ?- adjacent(5, 3)
4. no
5. ?- adjacent(3, 1)
6. no
7. ?- adjacent(3,2)
8. no

In prolog, one could declare coloring for the regions. It will also use unit clauses.

1. color(1, orange, x).    color(1, orange, y).
2. color(2, pink, x).  color(2, pink, y).
3. color(3, purple, x).    color(3, purple, y).
4. color(4, red, x).   color(4, pink, y).
5. color(5, pink, x).  color(5, purple, y).

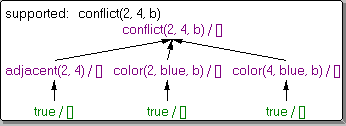
Here 'x' and 'y' colorings are encoded. In prolog, two adjacent regions which have the same color show the definition of conflictive coloring. The following example shows the prolog rule or clause to that effect.

1. conflict(R1, R2, Coloring) :-
2. adjacent(R1, R2),
3. color(R1, Color, Coloring),
4. color(R2, Color, Coloring).

Prolog distinguishes the two definitions of 'conflict'. 1st definition defines 1 logical parameter. The 2nd definition defines 3 parameters. Now we have,

1. ?- conflict(R1, R2, y)
2. R1=2    R2=4
3. ?- conflict(R1, R2, y), color(R1, Z, y).
4. R1= 2   R2=4    Z = blue

The above example shows that region 2 and region 4 both are adjacent. It also shows that regions 2 and 4 both are blue. The consequences of prolog program are shown by grounded instance conflict(2, 4, y). One way to demonstrate such a consequence is to draw a program clause tree. In which, root of the tree contains the consequence, to branch the tree, it uses the clause of the program, and lastly, it will produce a finite tree which has all true leaves. Using the fully grounded instance of the clause, we can construct the clause tree, which is shown as follows:

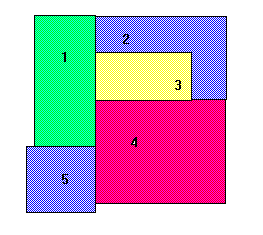


In the above tree, the bottom leftmost branch corresponds to the unit clause.

adjacent(2,4)

In prolog, this branch is equivalent to the clause  
adjacent(2,4) :- true

In prolog, the consequence of the program is not shown by 'conflict(1, 3, b)' because we can construct a finite clause tree using grounded clauses of P, which contains all 'true' leaves. Likewise, a consequence of the program is shown by 'conflict(a)'. To compute all possible colorings, we will develop a program in prolog. The famous four-color conjecture states that to color regions of the tree, we don't require more than four colors. The following example shows that we require at least four colors:



Two Factorial Definitions

It is used to find out the factorial functions. The stating of these definitions is as follows:

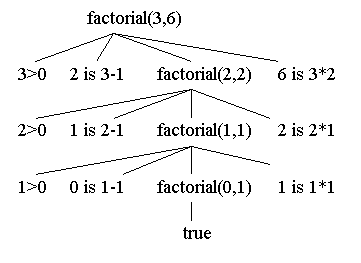
1. factorial(0,1).
3. factorial(N,F) :-
4. N>0,
5. N1 is N-1,
6. factorial(N1,F1),
7. F is N \* F1.

In prolog, this program has two clauses. The first clause has no body, and it is a unit clause. The second clause of prolog has a body. On the right-hand side of '-', the body of the second clause exists. The symbol '-' will be read as "if". The body consists of literals and that literals can be separated using the commas ','. The symbol '.' will be read as "and". If the clause is a unit clause, then the head of the class will be the whole class. Otherwise, the part of the clause which appears to the left of the colon in ':-' will be the head of the class. The first clause (unit clause) is used to show that "factorial 0 is 1". The second clause is used to show that "factorial of N is F if N>0, N1 is N-1, the factorial of N1 is F1 and F is N\*F1".

In prolog, if we find the factorial of 3, it will respond with a value of W.

1. ?- factorial(3, W).
2. W= 6

Now we are going to construct the following clause tree for factorial(3, W). The clause tree has an instance of variables, but it does not have any free variables. In the original program, each branch can be determined by a clause under a node. Using the instance of head of clause, the node of the tree can be determined. In the clause tree, the children of the node can be determined by the body literals of the clause.



After evaluation, we can say that all of the arithmetic leaves are true. In the above tree, the lowest link corresponds to the very first clause of the factorial program. The first clause of the program will be written as

1. factorial(0,1) :- **true**.

?- true is the goal of a prolog that always succeeds.

At the root of the tree, the clause tree in the program provides the meaning of the program for the goal. The factorial(3, 6) has a clause tree, and all of the leaves of factorial(3,6) are true that's why factorial(3, 6) is a consequence of the prolog program. The literal factorial(7, 3) has no clause tree and has all true leaves that's why factorial(7, 3) is not a consequence of the prolog program. For the literal factorial(7, 3), the meaning of the program shows that it is false.

1. ?- factorial(4, 24).
2. yes
3. ?- factorial(7, 3).
4. no

The clause trees of the program are also known as AND-trees. If the subtrees are rooted at literals, the root to be a consequence of the program and all the subtrees will also be consequences of the program. Now we know that clause trees give us a meaning.

Now we are going to differentiate between the prolog derivation tree and program clause tree. In prolog, the **derivation tree** is based on the variable-binding mechanism. It is also based on the order that is considered by subgoals. The **clause trees** of the program are static in nature. The clause trees are drawn for a program. The clause tree is simply a declarative reading of the program.

How variables are bound in prolog is shown by the execution of prolog trace to satisfy goals. The on and off of the prolog tracer is shown in the following sample:

1. ?- trace.
2. % The debugger will first creep -- showing everything (trace).
4. yes
5. [trace]
6. ?- factorial(3,X).
7. (1) 0 Call: factorial(3,\_8140) ?
8. (1) 1 Head [2]: factorial(3,\_8140) ?
9. (2) 1 Call (built-in): 3>0 ?
10. (2) 1 Done (built-in): 3>0 ?
11. (3) 1 Call (built-in): \_8256 is 3-1 ?
12. (3) 1 Done (built-in): 2 is 3-1 ?
13. (4) 1 Call: factorial(2, \_8270) ?
14. ...
15. (1) 0 Exit: factorial(3,6) ?
16. X=6
17. [trace]
18. ?- notrace.
19. % The debugger is switched off
21. yes

To find the Prolog goal 'factorial(3, X)', the following animation tree describes another look at the derivation tree. Click on the button 'Step' to start the animation.

In this section, two factorial definitions referred by this title. Here, we show the same predicate name but with three variables.

1. factorial(0,F,F).
3. factorial(N,A,F) :-
4. N > 0,
5. A1 is N\*A,
6. N1 is N -1,
7. factorial(N1,A1,F).

Specify the following goal for this version.

1. ?- factorial(5,1,F).
2. F=120

In the definition, accumulating parameter is shown by the second parameter.

Loading and Editing Programs in Prolog

For loading programs in Prolog, the standard predicates are notations of bracket loader '[ …]', 'consult', 'reconsult'. For example,

1. ?- consult('lists.pro').

The above goal opens the **lists.pro** file. It also helps to load the clause in **lists.pro** file into memory.

The Prolog program is deficient in two main ways. While loading the code, if there will be any syntax error, it shows that source code has a **syntax error**. While testing the program, if the programmer discovers any error, it shows that the program has a **logical error**. In Prolog, the program of the current version is usually considered the prototype for the correct version in the future. To edit the current version, retest, and reload it, this prototype gives us a common practice. The proper working of the rapid prototyping approach shows that the programmer has given a lot of time and effort to analyze the problem. The failure of the rapid prototyping approach shows that now we need to take up a pencil and paper, and we should rethink the requirements and start over.

In Prolog, we could call our editor using the following code and then upon returning from the editor.

Play Video

1. ?- edit('lists.pro'). %% User-defined edit,

To reload the program clause into the memory, one could use the following goal:

1. ? reconsult('lists.pro')002E

This will automatically replace the old definitions. Old clauses and new clauses have to remain in memory together if the user had used 'consult' rather than 'reconsult'. It actually depends upon the system of Prolog.

We will use 'reconsult', if more than one file has been loaded into the memory, and we need only one file to be reloaded. If the predicates are defined by the reload files and that predicates are not defined in the remaining file, the clauses that were originally loaded from the other files will not be disturbed by the reload.

The following example shows the bracket notation and it is very handy.

1. ?- ['file1.pro',file2.pro',file3.pro'].

The above will load all three files into the memory of Prolog.

To edit the programs, many systems of Prolog depend upon the programmer that contains a favorite text editor. The following program in Prolog calls TextEdit on Mac.

1. edit(File) :-
2. name(File, FileString),
3. name('open -e', TextEditString),     %% It edit the line **for** editor
4. append(TextEditString, FileString, EDIT),
5. name(E, Edit),
6. shell(E).

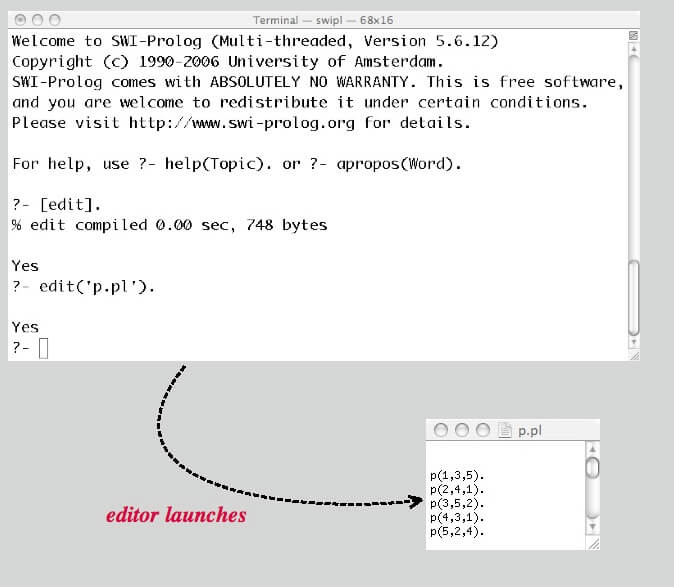
Its source will be loaded to use this editor. Here, we will assume that it is local to the session of Prolog.

1. ?- [edit].
2. yes

and then we will use the 'edit' goal. Now we will assume that file to edit is local to the session of Prolog.

1. ?- edit('p.pl').

Using the file loader, this TextEdit starts up. Edit the program, and using the same filename, save the program.



In the session of Prolog, we can reconsult the new version after editing and saving the programs in Prolog.

1. ?- reconsult('p.pl').\

The user interactively supplies to load clauses using the goals.

1. ?-consult(user).
2. ?-reconsult(user).
3. ?-[user].

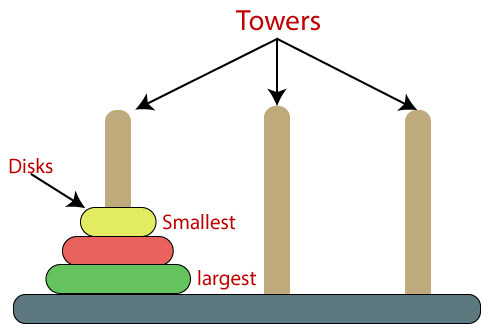
The user then interactively type in the clause at the end of clauses using 'stop'. To end the input, we can use ^Z.

Towers of Hanoi Puzzle in Prolog

We can move the disks to another rod, using the tower of Hanoi puzzle. The movement goes left to right using the center peg. This center peg is like an auxiliary holding peg.

It contains three rods and the different sizes of disks. In this puzzle, at a time, only one disk can be moved. These disks are placed in ascending order that means the smallest disk places upon the largest disk.

The following image contains 3 disks. In 7 moves, the following puzzle can be solved.



To solve the puzzle, the following recursive prolog program is as follows. This program contains two clauses.



1. move(1, A, B,\_) :-
2. write('move top disk from'),
3. write(A),
4. write(' to '),
5. write(B),
6. p1.
7. move(P, A, B, C) :-
8. P>1,
9. S is P-1,
10. move(S, A, B, C),
11. move(1, A, B,\_),
12. move(S, C, B, A).

The variables which begin with an underscore or filled in by '\_' are known as 'don't care' variables. In prolog, any structure can freely match by these variables.

The following steps are shown when case N=3 is solved by the prolog.

1. ?- move(3, left, right, center).
2. From left to right move top disk
3. From left to center move top disk
4. From right to center move top disk
5. From left to right move top disk
6. From center to left move top disk
7. From center to right move top disk
8. From left to right move top disk
10. yes

In the program, the movement of a single disk can be described by the first clause. How the solution will recursively obtain is declared by the second clause. In the second clause, N=3, A=left, B=right, and C=center accounts.

1. move(3,left,right,center) **if**
3. move(2,left,center,right) and ] \*
4. move(1,left,right,center) and
5. move(2,center,right,left). ] \*\*

The reading of this declarative clause is correct. The declarative interpretation of recursive clause is closely related to the procedural reading. The following code shows the procedural interpretation:

1. In order to satisfy the goal ?- move(3,left,right,center) **do** **this** :
3. satisfy the goal ?-move(2,left,center,right), and then
4. satisfy the goal ?-move(1,left,right,center), and then
5. satisfy the goal ?-move(2,center,right,left).

The declarative readings for N=2 are as follows:

1. move(2,left,center,right) **if** ] \*
3. move(1,left,right,center) and
4. move(1,left,center,right) and
5. move(1,right,center,left).
6. move(2,center,right,left) **if** ] \*\*
8. move(1,center,left,right) and
9. move(1,center,right,left) and
10. move(1,left,right,center).

In the above code, the body of last two implications is substituted for the heads. The prolog goal generates the solution, and one can see it.

1. move(3,left,right,center) **if**
3. move(1,left,right,center) and
4. move(1,left,center,right) and \*
5. move(1,right,center,left) and
6. ---------------------------
7. move(1,left,right,center) and
8. ---------------------------
9. move(1,center,left,right) and
10. move(1,center,right,left) and \*\*
11. move(1,left,right,center).

In prolog, the three major operations are described as follows:

1. The head of a goal matches against the goals.
2. A new sequence of goals becomes the body of the rule repeatedly. Until
3. We take some simple action, or some condition is satisfied.

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.

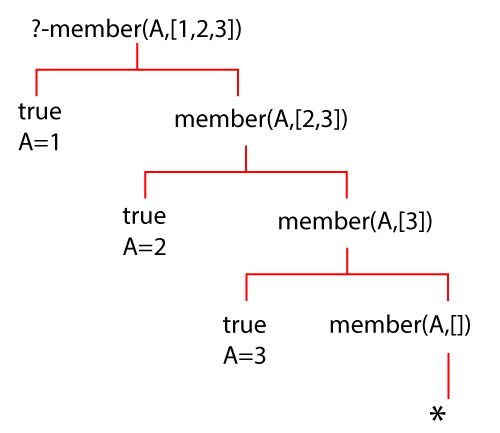
We can use this program in many ways. We can also test the membership as follows:

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
2. A=9 B=81 ;
3. A=6 B=36 ;
4. no

The member definition is written as follows:

1. member(A, [A | \_]).
2. 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:

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

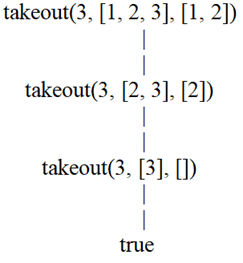
In English, we can paraphrase these clauses as follows:

* 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,

1. ?- takeout(A, [1, 2, 3], L).
2. A=1 L=[2, 3] ;
3. A=2 L=[1, 3] ;
4. A=3 L=[1, 2] ;
5. 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:

1. ?- takeout(3, X, [a, b, c])
2. X = [3, a, b, c] ;
3. X = [a, 3, b, c] ;
4. X = [a, b, 3, c] ;
5. X = [a, b, c, 3] ;
6. 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:

1. putin(A, L, S) :- append(B, S, L).

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

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

Various possible goals are as follows:

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

Simple Input-Output in Prolog

We can read the data using the following predicates of Prolog:

1. seeing(File)
2. see(File)
3. read(Data)
4. seen

To provide the input source, File= 'user ' will choose the keyboard. Otherwise, File should be the existing file disk name. The following program shows a program to read the file data and print it to the screen as follows:

1. browse(File) :-
2. seeing(Old),        /\*It will save for later \*/
3. see(File),      /\* It will open this file \*/
4. repeat,
5. read(Data),         /\* It will read from the file \*/
6. process(Data),
7. seen,           /\* It will close the file \*/
8. seen(Old),      /\* It will show the previous read source \*/
9. !           / \* It means stop now \*/
11. process(end-of-file) :- ! .
12. process(Data) :- write(Data), nl, fail.

We can issue the following goal to inspect the source file for 'browse'. Assume that name of browse is browse.pro:

1. ?- browse('browse.pro').

When 'seeing(Old)' is satisfied by the Prolog, the variable Old will bound to the port and that port is the currently coming read-inputs. When 'see(File)' is satisfied by the Prolog, the file will open whose name is bound as value to 'File' and stream or read port is associated with the file.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

In the program, the 'Data' variable should take value. The sequence of numbers or a Prolog program can contain by the 'File'. In the following section, a file of numbers will be formatted as follows:

1. 32.2        /\* = real 32.2 \*/
2. 31.         /\* = integer 31 \*/
3. 4.0.        /\* = real 4.0 \*/

The following example shows the interactive version of the program of browser:

1. browse :-
2. seeing(Old),        /\* It will save for later \*/
3. see(user),
4. write('specify the file name to browse: '),
5. read(File),
6. see(File),
7. repeat,
8. read(Data),         /\* It will read from File \*/
9. process(Data),
10. seen,           /\* It will close the file \*/
11. see(Old)        /\* It will previous read source \*/
12. !.          /\* It means stop now \*/

Here we will show how to use this version to pay attention to the form that the user supplies the filename.

1. ?- browse.
2. Specify the file name to browse: 'abc.pl'.
3. …

In Prolog, to write the data, we can use the following predicates:

1. telling(File)
2. tell(File)
3. write(Data)
4. nl
5. writeln(Data)
6. told

We will open the output destination screen using the File = 'User'. The following Prolog program is used to save the current program in memory to a file.

1. my\_save(ToFile) :-
2. telling(Old),       /\* It will current write output \*/
3. tell(To File),      /\* It will open this file \*/
4. listing,            /\* It will list all clauses in memory \*/
5. told,           /\* It will close ToFile \*/
6. tell(Old).      /\* resume this output \*/

When 'telling(Old)' is satisfied by the Prolog, the variable 'Old' will bound to the port and that port is associated with the current destination of write-output. When 'tell(ToFile)' is satisfied by the Prolog, the name of the file bound to 'ToFile' will be opened, and stream or output port will be associated with the file. Using the stream or port, the subsequent writing will go to that file. Then 'told' will close the file.