**Introduction to Prolog.**

Prolog stands for "Programming in Logic". It’s a unique and powerful programming language that is based on formal logic and is widely used in fields such as artificial intelligence, knowledge representation, natural language processing, and constraint logic programming.

Prolog is based on a form of symbolic logic called Horn clause logic. Prolog programs are composed of a set of rules that define relationships between objects and can be used to make inferences about those relationships. The language is particularly well-suited to solving problems involving symbolic reasoning and pattern matching.

In this report, we will explore the main features of Prolog that make it distinctive and well-suited for these domains.

**1.1 Declarative Nature:**

One of the key features of Prolog is its declarative programming style, where programmers express relationships and rules as facts and rules. This allows for concise and expressive representation of complex relationships and rules, making it easy to model real-world problems in a natural and intuitive way. For example, consider the following Prolog code that represents personal cuisine:

**likes(huy, socola).**

**likes(tuong, cofffe).**

**likes(toan, water).**

**likes(phat, milk).**

* The following queries with Prolog answers.

**| ?- likes(tuong,cofffe).**

**true.**

**| ?- likes(toan,water).**

**true.**

**| ?- likes(phat,food).**

**false.**

In this example, the facts **likes** represent the relationships like of dislike a disk of a person.

This declarative style makes it easy to express complex relationships and rules in a concise and intuitive manner.

**1.2 Prolog's Predicate Notation for Logic.**

Prolog's predicate notation is a way of representing logical statements in the form of predicates. Predicates in Prolog are represented using a notation that is similar to mathematical notation.

A predicate is a statement that is either true or false. It is represented using a combination of variables, constants, and logical operators. In Prolog, a predicate is represented using the following notation:

**predicate(arg1, arg2, ..., argn)**

where predicate is the name of the **predicate**, and **arg1**, **arg2**, ..., argn are its arguments. The arguments can be either variables or constants.

For example, consider the following logical statement:

**All men are mortal.**

This statement can be represented using a predicate in Prolog as follows:

**mortal(X):- men(X).**

In this representation, **mortal** is the name of the predicate, **X** is a variable, and man(**X**) is another predicate. The **:-** symbol represents implication, and the statement can be read as "X is mortal if X is a man."

Similarly, the statement "Socrates is a man" can be represented using a predicate as follows:

**man(socrates).**

This statement can be read as "socrates is a man."

The querys with corresponding answer in Prolog:

**| ?- mortal(socarates).**

**true.**

**| ?- mortal(time).**

**false.**

|  |  |  |
| --- | --- | --- |
| **English** | **Predicate Calculus** | **PROLOG** |
| **and** | **^** | **,** |
| **or** | **v** | **;** |
| **if** | **-->** | **:-** |
| **not** | **~** | **not** |

Simple Symbols in Prolog

Using predicate notation, we can represent complex logical statements and perform reasoning on them using Prolog's built-in inference engine.

**1.3 List Handling and Recursion**

Prolog is a natural language for handling lists and recursion. Lists are fundamental data structures in Prolog, and Prolog provides built-in support for manipulating lists efficiently. Prolog allows developers to easily append, concatenate, reverse, and manipulate lists using predefined predicates. Prolog also naturally supports recursion, which is a powerful technique for solving problems that involve repetitive or recursive structures. Recursive predicates can be easily defined in Prolog, allowing for elegant and concise solutions to problems that require repeated computations or traversals of data structures.

**Takes a list of numbers and returns the sum of those numbers**.

**sum\_list([], 0).**

**sum\_list([X|Xs], Sum) :-**

**sum\_list(Xs, Rest),**

**Sum is X + Rest.**

The first line defines the base case: if the list is empty, the sum is 0.

The second line defines the recursive case: if the list is not empty, we split it into its head (the first element, X) and tail (the rest of the list, Xs), and recursively find the sum of the tail.

The third line calculates the sum of the entire list: it adds the head (X) to the sum of the tail (Rest), and unifies the result with Sum.

The query with corresponding answer in Prolog.

**?- sum\_list([6,8,6,8], Sum).**

**Sum = 28.**

**?- sum\_list([], Sum).**

**Sum = 0.**

**?- sum\_list([5], Sum).**

**Sum = 5.**

**The Tower of HaNoi uses Prolog language.**

Chart

Description automatically generated

Initialize State with 3 disk and 3 peg (Source: [Tower of Hanoi - Coding Ninjas](https://www.codingninjas.com/codestudio/problems/tower-of-hanoi_981323))

**move(1,X,Y,\_) :-**

**write('Move top disk from '),**

**write(X),**

**write(' to '),**

**write(Y),**

**nl.**

**move(N,X,Y,Z) :-**

**N>1,**

**M is N-1,**

**move(M,X,Z,Y),**

**move(1,X,Y,\_),**

**move(M,Z,Y,X).**

Prolog solves with case N=3.

**?- move(3,left,right,center).**

**Move top disk from left to right**

**Move top disk from left to center**

**Move top disk from right to center**

**Move top disk from left to right**

**Move top disk from center to left**

**Move top disk from center to right**

**Move top disk from left to right .**

**move(1,X,Y,\_)** can be read as: To move one disk from peg X to peg Y, write a message indicating the movement of the top disk from X to Y, and add a newline character at the end". The \_ in the rule indicates that the value of the fourth argument is not used.

**move(N,X,Y,Z)** can be read as: To move N disks from peg X to peg Y, where peg Z is available as an intermediate pole, first move the top M disks from peg X to peg Z, then move the largest disk from peg X to peg Y, and finally move the M disks from peg Z to peg Y.

The rule uses recursion to solve the problem for smaller and smaller values of N until the base case is reached. The **M** variable is used to keep track of the number of disks that are being moved in each recursive call.

As you can see, Prolog is a very simple and natural for dealing with lists and recursion problems.

**1.4 Unification**

In Prolog, unification is used to match goals with facts and rules in the program's database. It is a key process in the resolution of queries, and it enables the program to find the correct answer to a query by matching the query with the program's database. When a query is made, Prolog attempts to unify the query with the clauses in its database.

Unification is the process of comparing two terms, and if they are identical, binding the variables in the terms to the same values. In Prolog, a term can be a variable, an atom, or a compound term. A compound term is a term that is made up of one or more subterms, which can be variables, atoms, or other compound terms.

When Prolog attempts to unify two terms, it does so by comparing the terms from left to right. If the terms are both atoms or both variables, they are considered to be identical if they have the same name. If the terms are compound terms, Prolog compares their functor, or name, and then recursively compares their subterms. If the functors are the same and the subterms unify, then the terms are considered to be identical.

Consider the followings rules to determines the outcome of unification:

Given two terms T1 and T2 which are to be unified:

* Firstly, if both T1 and T2 are constants (i.e., atoms or numbers), then if they are the same, the unification succeeds. Otherwise, it fails.
* Secondly, if T1 is a variable, it can be instantiated to T2. Similarly, if T2 is a variable, it can be instantiated to T1.
* Thirdly, if T1 and T2 are complex term with same numbers of arguments, find the principal functor F1 to T1 and principal functor F2 of T2. If these are the same, then take the ordered set of arguments of <A1,A2,….AN> of T1 and the ordered set of arguments <<B1,B2,….BN> > of T2. For each pair of arguments AM and BM from same the position in the term, AM must unify with BM
* If none of these rules apply, unification fails.

The built in Prolog operator '=' can be used to unify two terms. Below are some examples of its use:

**| ?- a = a. %** Two identical atoms unify

**true.**

**| ?- knows(John,X) = knows(X,Steve)** % In this case there is no unification,

**false.** % because X can not assign two value at the same time.

**| ?- knows(John,X) = knows(Y,Steve)**

**X = Steve,**

**Y = John.**

**| ?- [a,b,c] = [X,Y,Z]. %** Lists unify just like other terms

**X=a,**

**Y=b,**

**Z=c.**

**| ?- [4,5,6,8] = [X|Y]. %** Unification using the '|' symbol can be used

**X=4,** % to find the head element, X, and tail list, Y,

**Y=[5,6,8].** % of a list

**1.5 Built-in Parallelism.**

Built-in Parallelism in Prolog refers to the ability of a Prolog system to execute multiple goals in parallel, either by using multiple processors or by interleaving the execution of the goals on a single processor. Built-in parallelism is a feature that some Prolog systems provide in order to improve the performance of Prolog programs that involve multiple independent computations.

There are several ways that Prolog systems can provide built-in parallelism:

1. **And-parallelism**: In this form of parallelism, the Prolog system evaluates multiple independent goals in parallel. For example, if a program has two independent subgoals, the Prolog system can evaluate them in parallel, potentially speeding up the overall execution time of the program.
2. **Or-parallelism**: In this form of parallelism, the Prolog system evaluates multiple alternative goals in parallel. For example, if a program has two alternative subgoals, the Prolog system can evaluate them in parallel, potentially finding a solution more quickly than if it evaluated them sequentially.
3. **Tabling**: In tabling, the Prolog system caches the results of subgoals that have already been evaluated, and reuses them when necessary. This can improve the performance of Prolog programs that involve repeated computations or backtracking.

Overall, built-in parallelism is a powerful feature that can significantly improve the performance of Prolog programs, especially those that involve independent computations or backtracking.

However, not all Prolog systems support built-in parallelism, and even those that do may require special programming techniques to take full advantage of it.

**2. Implement Prolog language.**

**2.1 Choosing a Prolog Programming Environment:**

There are several Prolog programming environments available, such as SWI-Prolog, GNU Prolog, SICStus Prolog, and others. For this report, we will use **SWI-Prolog**, which is a widely used and open-source Prolog implementation available for various platforms including Windows, macOS, and Linux.

**2.2 Installing SWI-Prolog.**

To implement Prolog language in SWI-Prolog, we need to first install the SWI-Prolog system on our machine. The installation process may vary depending on the operating system, but generally involves downloading the SWI-Prolog distribution from the official website (<https://www.swi-prolog.org/Download.html>). In this report we will use Stable version.

Graphical user interface, website

Description automatically generated

Step 1: Choose version (we will use Stable release version in this report).

Graphical user interface, text

Description automatically generated

Step 2: Choose the version that is suitable for your computer.

Graphical user interface, application, Word

Description automatically generated

*Step3: Run the exe file you downloaded and do personalization.*

Text

Description automatically generated

Step 4 (Done): Run SWI-Prolog.

Text, letter

Description automatically generated

“Hello World” with Prolog.

**To run program in SWI-Prolog:**

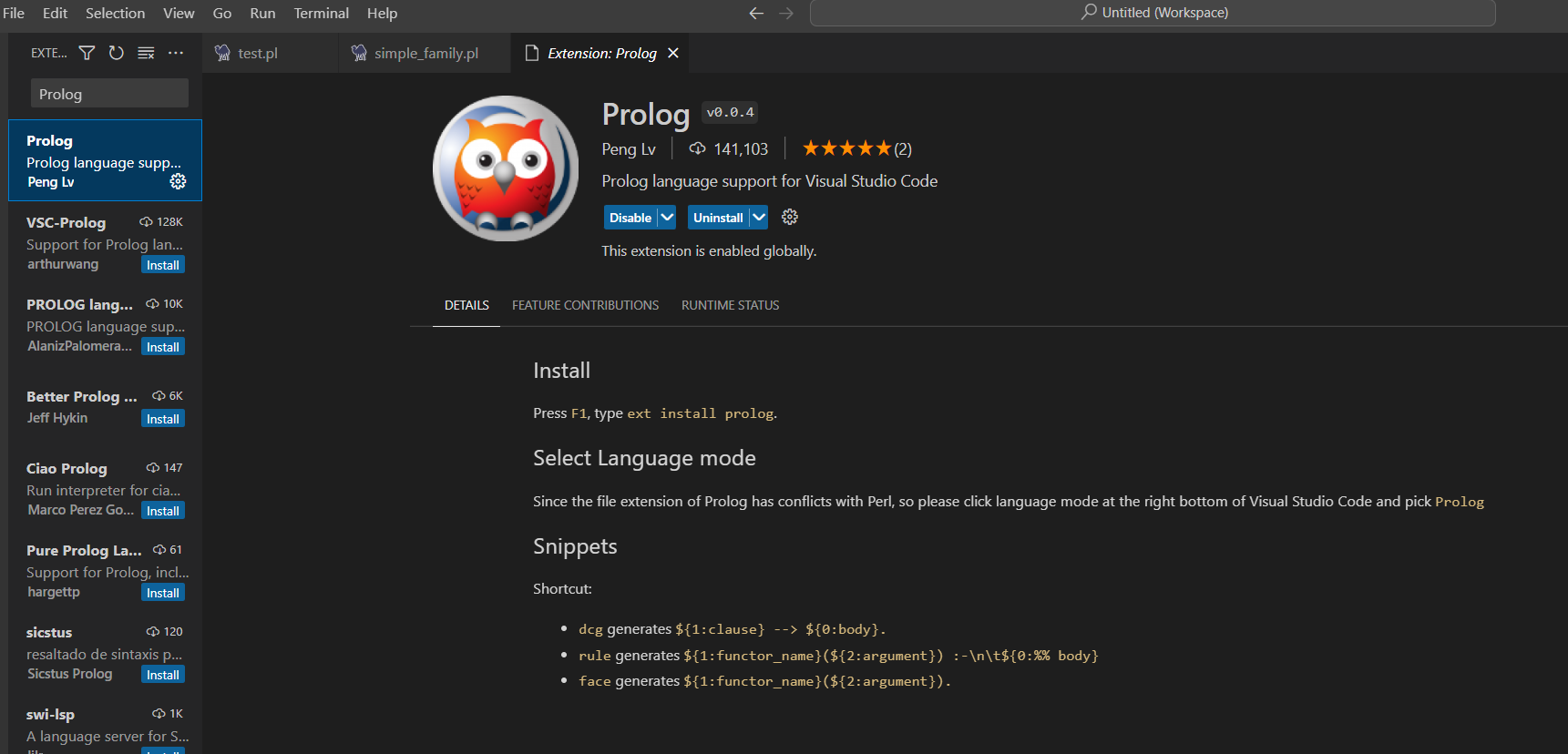
**Step 1**: Load your program into SWI-Prolog. You can do this by clicking on the File menu and selecting the "Consult..." option.

**Step 2**: In the "Consult File" dialog box that appears, navigate to the directory where your program is saved and select the file. Click on the "Open" button to load the program into SWI-Prolog.

**Step 3:** Once your program is loaded, you can execute it by typing the name of the predicate that you want to run into the Prolog console and pressing Enter. If your program doesn't have a predicate named "main", you will need to call a predicate that will start your program.

If the program is executed successfully, you should see the output in the Prolog console.

**2.2.1 Implement Prolog on Visual Studio Code**



Step 1: Install extension Prolog on Visual Studio Code



Step 2: Vào thư mục swipl/bin và lấy đường dẫn file swipl-win.exe

A screenshot of a computer

Description automatically generated with medium confidence

Step 3: Open file PROFILE and add “set-alias swipl “pth”” with pth is path in step 2.

Text

Description automatically generated

Step 4 (Done): Open Terminal in VScode and type “swipl” command and the same operation as on SWI-LOG.

**Simple Program with Prolog:**

**father(Parent, Child) :-**

**parent(Parent, Child),**

**male(Parent).**

**mother(Parent, Child) :-**

**parent(Parent, Child),**

**female(Parent).**

**child(Child, Parent) :-**

**parent(Parent, Child).**

**parent(henry,stark). %Henry is parent of Stark**

**parent(tom,stark).**

**parent(tom,liz).**

**parent(stark,ann).**

**parent(stark,pat).**

**parent(pat,jim).**

**female(henry).**

**female(liz).**

**female(ann).**

**female(pat).**

**male(tom).**

**male(stark).**

**male(jim).**

**Query with corresponding answer in Prolog**

**?- father(X,stark).**

**X = tom.**

**?- mother(X,jim).**

**X = pat.**

**?- child(pat,tom).**

**false.**