Department of Computer Science

CS - 306 : Programming Languages

Simple Prolog Interpreter in Java

Authors:
Nazar Kashif (IMT2018042)
Nipun Goel (IMT2018052)
Vinayak Agarwal (IMT2018086)

May, 2021

Table of Contents

1 Project

1.1 Abstract

This report documents our approach to build a simple yet very efficient and accurate Prolog interpreter in Java, and the various methodologies we used to perform Lexical Analysis and Parsing. It explains how we tried to bring out the same functionality, using our understanding of Data Structures and Algorithms instead of the conventional techniques. It also highlights the similarities and differences between the actual Prolog interpreter and our version of it.

1.2 Project Description

This Project is the implementation of a Simple Prolog Interpreter in Java.

Prolog is a logical and a declarative programming language. Logic Programming is one of the Computer Programming Paradigm in which the program statements express the facts and rules about different problems within a system of formal logic. In Prolog, the relations are written in the form of logical clauses (facts and rules), where head and body are present. In case of facts, body is always true. Computation is carried out by running a query over these relations.

Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

The implementation of Prolog Interpreter in Java gave us an opportunity to deepen our understanding of logic programming language and object oriented programming language. We designed several classes to simulate a Prolog interpreter as explained later.

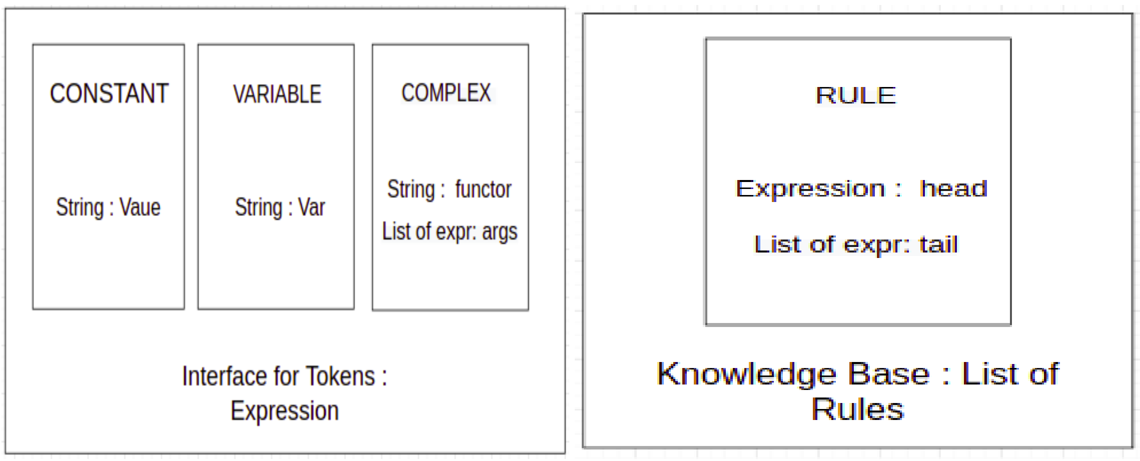
2 Solution Approach

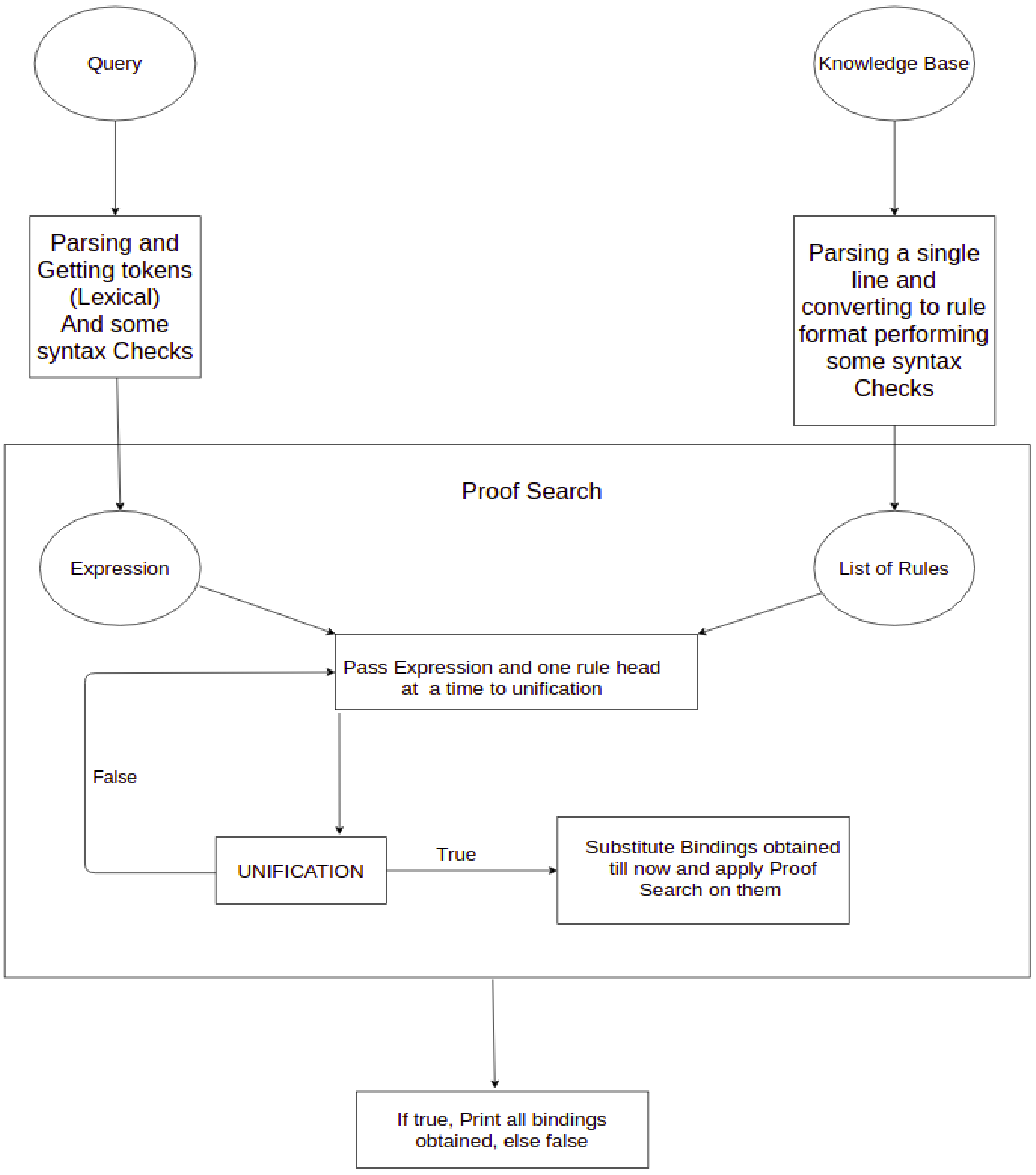
2.1 Overview

The Project consists of 4 major components namely,

1. Parsing and Lexical Analysis.
2. Unification
3. Bindings substitution
4. Proof Search

The structure and workflow of our Prolog interpreter can be seen below,

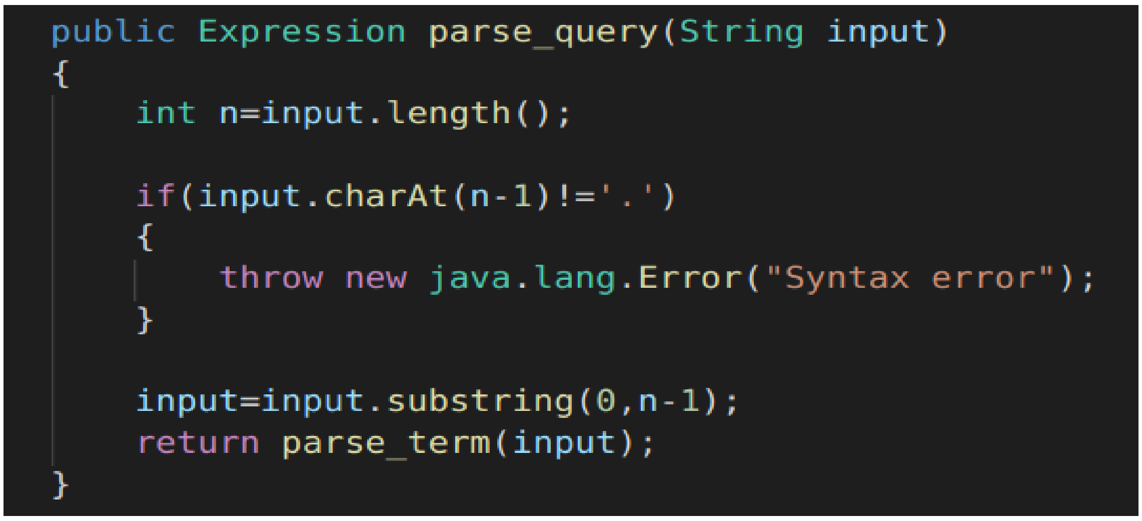

CONSTANT
String : Vaue
VARIABLE
String : Var
COMPLEX
String : functor
List of expr: args
Interface for Tokens : Expression
RULE
Expression : head
List of expr: tail
Knowledge Base : List of Rules


Query
Knowledge Base
Parsing and Getting tokens (Lexical) And some syntax Checks
Parsing a single line and converting to rule format performing some syntax Checks
Proof Search
Expression
List of Rules
Pass Expression and one rule head at a time to unification
False
UNIFICATION
True
Substitute Bindings obtained till now and apply Proof Search on them
If true, Print all bindings obtained, else false

2.2 Lexical analysis and Parsing

Lexical analysis is the first phase of the interpreter. It converts the high level input program into a sequence of Tokens. The input sequence is read, white spaces are ignored, and various predefined tokens are identified. Failure to classify some input literal in one of the predefined token classes throws an error that invalid token is identified. There is an Expression interface which is implemented by all token classes namely Constant, Variable and Complex class. Also a Rule class is created which has three attributes - Head , Body(tail) and list of operations between two consecutive expressions in body. Rule is a way to represent all terms from the knowledge base, namely facts and rules, to make operations easier and consistent in the future.

In parsing, it is ensured that the tokens obtained from the lexical analysis follow grammar rules described by Prolog. To accomplish this task,a Parser class is implemented which has the following methods - parse rule , parse term and parse query. Parse rule method is used for syntactic analysis of Rules in knowledge base, Parse term is used for syntactic analysis of the individual term which can be Constant, Variable and Complex and Parse query method is used for syntactic analysis of the user input query. Parse term method is also used by other methods recursively for syntactic analysis.



```java
public Expression parse_query(String input)
{
    int n=input.length();

    if(input.charAt(n-1)!='.')
    {
        throw new java.lang.Error("Syntax error");
    }

    input=input.substring(0,n-1);
    return parse_term(input);
}
```

2.3 Unification

In Prolog, there are three kinds of term:-

1. Constants
2. Variables.
3. Complex terms.

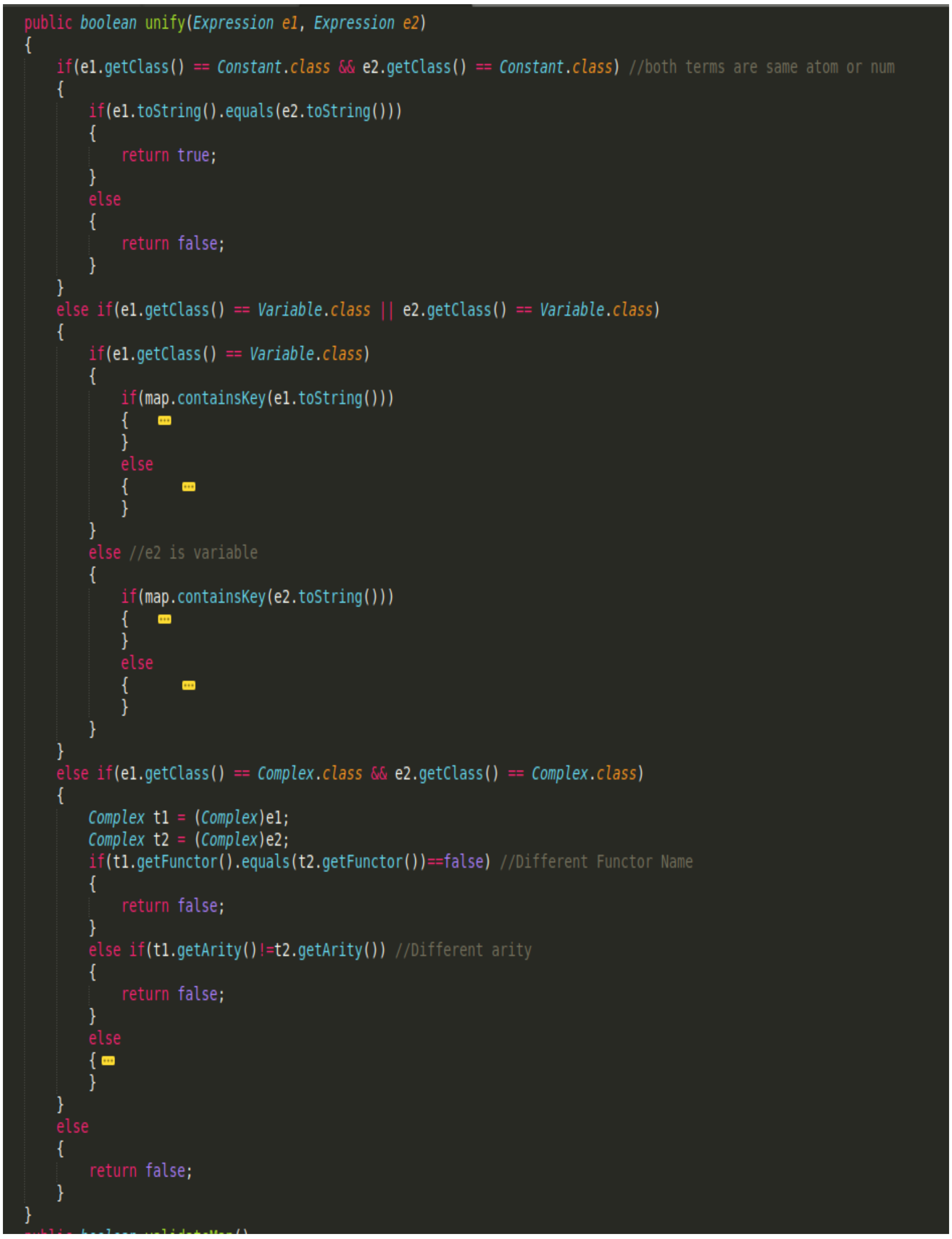
In Prolog, two terms unify if and only if

1. If term1 and term2 are constants, then term1 and term2 unify if and only if they are the same atom, or the same number.
2. If term1 is a variable and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2 . Similarly, if term2 is a variable and term1 is any type of term, then

term1 and term2 unify, and term2 is instantiated to term1 . (So if they are both variables, they're both instantiated to each other, and we say that they share values.)

3. If term1 and term2 are complex terms, then they unify if and only if: They have the same functor and arity, and all their corresponding arguments unify, and the variable instantiations are compatible.

4. Two terms unify if and only if it follows from the previous three clauses that they unify.



```
public boolean unify(Expression e1, Expression e2)
{
    if(e1.getClass() == Constant.class && e2.getClass() == Constant.class) //both terms are same atom or num
    {
        if(e1.toString().equals(e2.toString()))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else if(e1.getClass() == Variable.class || e2.getClass() == Variable.class)
    {
        if(e1.getClass() == Variable.class)
        {
            if(map.containsKey(e1.toString()))
            {
            }
            else
            {
            }
        }
        else //e2 is variable
        {
            if(map.containsKey(e2.toString()))
            {
            }
            else
            {
            }
        }
    }
    else if(e1.getClass() == Complex.class && e2.getClass() == Complex.class)
    {
        Complex t1 = (Complex)e1;
        Complex t2 = (Complex)e2;
        if(t1.getFunctor().equals(t2.getFunctor())==false) //Different Functor Name
        {
            return false;
        }
        else if(t1.getArity()!=t2.getArity()) //Different arity
        {
            return false;
        }
        else
        {
        }
    }
    else
    {
        return false;
    }
}
```

2.4 Substituting bindings :

This method returns the expression for which it is called, with all variables substituted with their respective bindings. It makes use of all the bindings, which are stored in a map which we get after unification. It works differently for each token class. The interface modelled by us, contains a function, which does this for us, and each class has their own implementation of this. It allows us to call this function as a class method, which saves us from the trouble of always checking the class of the object. This tries to mimic the property of **Dynamic Polymorphism**, which allows for seamless functionality.
For class Constant, we simply return a copy of the object itself, as a constant cannot be binded to anything else.
For class Variable, we check if the binding from the map is a Constant, a Variable or a Complex term. If the binding is a constant, we substitute the variable with this constant. If it is a variable, we obtain call this method recursively on that binding, and assign it to our Variable. If the term is a Complex, then we get the substituted form for the Complex Term, and assign it to our variable. If the binding is null, we simply return a copy of the Variable itself. For every case, where we return the object itself, we return a copy, to avoid the problem of **Shallow Copy**, that is prevalent in Java.
For class Complex, we simple go through its arguments, and call the same method on them.
Note:- To handle infinite recursion for bindings of form X-¿Y and Y-¿X, we keep a set of previously visited expressions, and if it is revisited , it signifies that a cycle is there. This idea was inspired form a very popular tree algorithm **Depth First Search**, which does a similar thing to avoid infinite recursion.

2.5 Proof Search

Proof Searching is the core of how Prolog, or rather Logic Programming, are able to perform high level tasks, without any harsh computations or effort. Proof search in Prolog is used to validate , whether a given query maps to a certain rule(s) or fact(s) in our knowledge base, and also returns the various Variable bindings that , on substitution in the query, will converge to True according to our Knowledge Base.
In our implementation of Proof Search, we pass a query to it, and it has access to the set of rules in the Knowledge Base (facts are represented as fact :- TRUE). Then it tries to validate or solve the query through the following steps:

- It tries to unify the query with each of the head from the rules. (Head is the expression before the ":-" in the Knowledge base).
- If it does not unify , it moves on to the next rule. But is it does unify, we then substitute the bindings obtained from this unification, substitute them in the sub-rules, and then call Proof Search recursively on the sub-rules. This goes on till a rule is encountered which has a tail as TRUE, which implies the rule is actually a fact. Then it stops, returns true, and the bindings also.
- To handle the "and" , ""or" operations between the sub-rules, we also keep a boolean variable, which applies this operation with the result from the Proof Search of a sub-rule.

Note:- The recursion tree which will form as a result of this proof search, analogously represents the Search tree that is formed when Prolog processes a query.

2.6 Interface

Interface is the final component of this interpreter. It mimics the interface of the original Prolog interpreter. Input can be of two types namely loading knowledge base using consult command and query. For input of type loading knowledge base, list of rules is obtained by parsing each rule of the knowledge base line by line and for input of type query, object of query class is created which takes two parameters namely query and list of rules then finally output of the query is obtained by calling the solve method of query object. Detailed instructions to compile and run this interpreter can be found be Methods to use section. User has to enter "Exit" to close the interpreter.

3 Results and Observation

3.1 Methods to Use

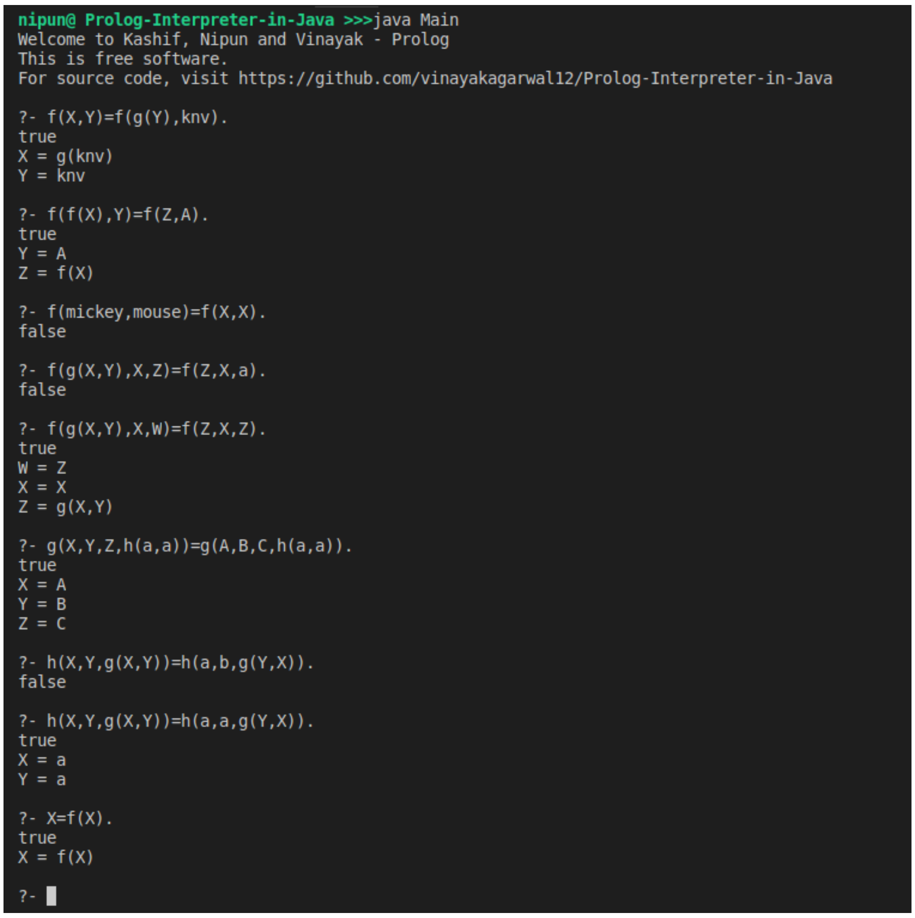
1. First download the source code from github.
2. To compile the code, enter the below command on the terminal.
   make interpreter
   and to run the code, enter the below command in the terminal.
   ./interpreter
   Now, the Prolog interpreter is running, enter the query in this format

   (a) For updating knowledge base type consult(kb). in the interpreter (where kb is the file name of knowledge base).
   (b) For asking query type query in the interperter.

   Further usage details can be seen from below section.

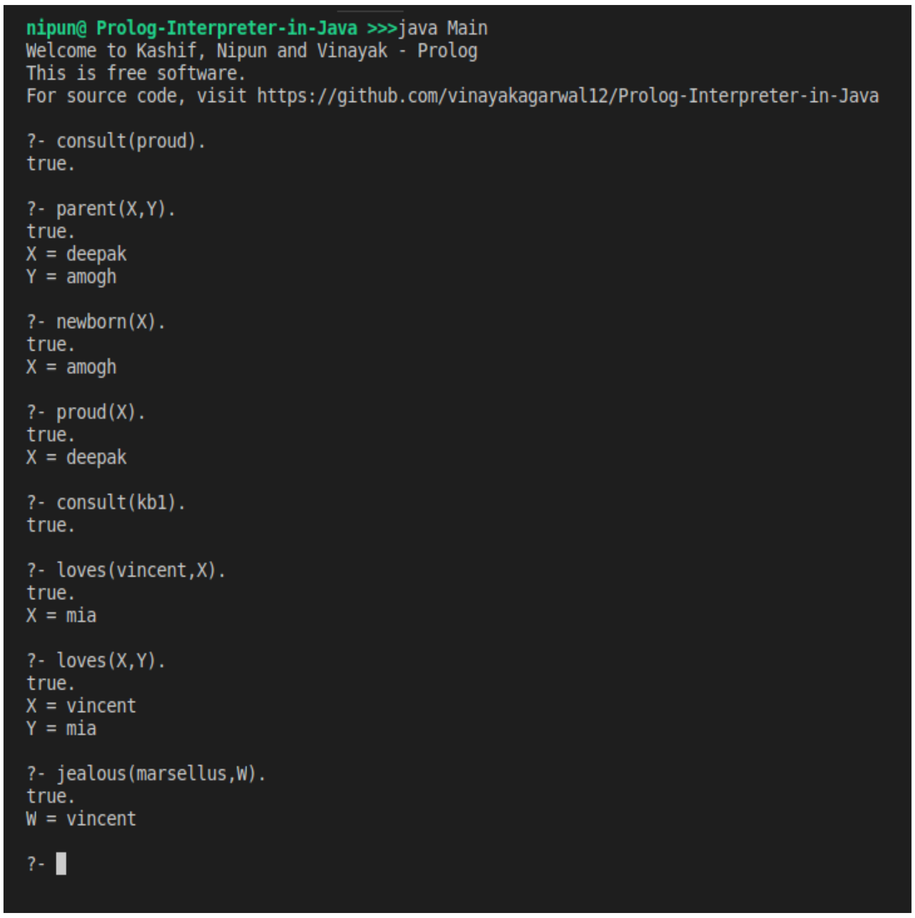
3.2 Test Cases Results

For unification :



```
nipun@ Prolog-Interpreter-in-Java >>>java Main
Welcome to Kashif, Nipun and Vinayak - Prolog
This is free software.
For source code, visit https://github.com/vinayakagarwal12/Prolog-Interpreter-in-Java

?- f(X,Y)=f(g(Y),knv).
true
X = g(knv)
Y = knv

?- f(f(X),Y)=f(Z,A).
true
Y = A
Z = f(X)

?- f(mickey,mouse)=f(X,X).
false

?- f(g(X,Y),X,Z)=f(Z,X,a).
false

?- f(g(X,Y),X,W)=f(Z,X,Z).
true
W = Z
X = X
Z = g(X,Y)

?- g(X,Y,Z,h(a,a))=g(A,B,C,h(a,a)).
true
X = A
Y = B
Z = C

?- h(X,Y,g(X,Y))=h(a,b,g(Y,X)).
false

?- h(X,Y,g(X,Y))=h(a,a,g(Y,X)).
true
X = a
Y = a

?- X=f(X).
true
X = f(X)

?- 
```

Queries after loading knowledge base :



```
nipun@ Prolog-Interpreter-in-Java >>>java Main
Welcome to Kashif, Nipun and Vinayak - Prolog
This is free software.
For source code, visit https://github.com/vinayakagarwal12/Prolog-Interpreter-in-Java

?- consult(proud).
true.

?- parent(X,Y).
true.
X = deepak
Y = amogh

?- newborn(X).
true.
X = amogh

?- proud(X).
true.
X = deepak

?- consult(kb1).
true.

?- loves(vincent,X).
true.
X = mia

?- loves(X,Y).
true.
X = vincent
Y = mia

?- jealous(marsellus,W).
true.
W = vincent

?- █
```

3.3 Observations and Limitations

Following are the key observations and future scope of our Prolog interpreter :

1. Our Prolog interpreter works exactly like the actual Prolog when given two expressions to unify.
2. Our Prolog interpreter cannot display multiple results of a query. That is, in Prolog user has an option to enter ';' and Prolog starts searching for next matching bindings.
3. Future scope is to implement operation precedence using brackets in a rule/query. Our current version considers left to right precedence of 'AND' and 'OR' operations. We can also extend the Proof Search algorithm, to obtain all possible combination of variable bindings by fully traversing the proof search tree.

4 Conclusion

This project is our version of how the Prolog interpreter could have been implemented in Java. It consists of various key components namely Lexical analysis and parsing, Unification, Bindings substitution and proof searching step, and how they can be implemented with day to day Data Structures and Algorithmic techniques. We also noted the limitations of this interpreter which can be improved upon in future to resemble the fuctionality of the original Prolog interpreter. Not only this , we got to experience the depths of how Prolog works, and how Logic Programming does things. This effort taught us the basics of compiler design, or at least how it works at the core, which is a thing which will definitely help us be a better Computer Scientist.

Overall, it was a good learning experience. We learnt how one language can be used to built interpreter for another type of language and the challenges involved.

5 Acknowledgement

Only the theory behind a concept can't help us gain insight on how it takes effect in real life. At the same time blindly implementing a thing without studying it first will never help us in the long run. Projects bridge this gap and help us to understand how theory applies in reality.

We can't stress enough on the fact that how much this project helped us to understand the core concepts, and this was only possible because of our respected Professor Sujit Kumar Chakrabarti and all our TAs who helped us throughout the course and the project.