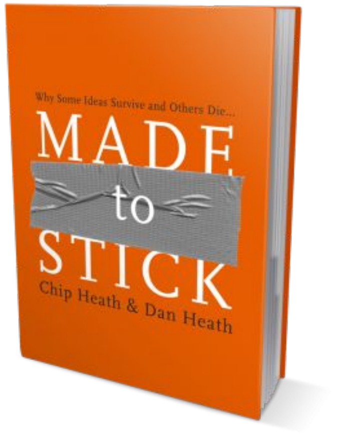
A Prolog Interpreter in Java

Nazar Kashif- IMT2018042
Nipun Goel - IMT2018052
Vinayak Agarwal - IMT2018086

What is Prolog?

- “Prolog” is short for “Programming with Logic”.
- At the heart of Prolog lies a beautiful idea: don’t tell the computer what to do, simply describe situations of interest.
- When we ask questions. Prolog enables the computer to logically deduce new facts about the situations we describe, and gives its deductions back to us as answers.



Major components

- ➔ Parsing and Lexical Analysis
- ➔ Unification
- ➔ Bindings Substitution
- ➔ Proof Search
- ➔ Interface

Implementation in Java

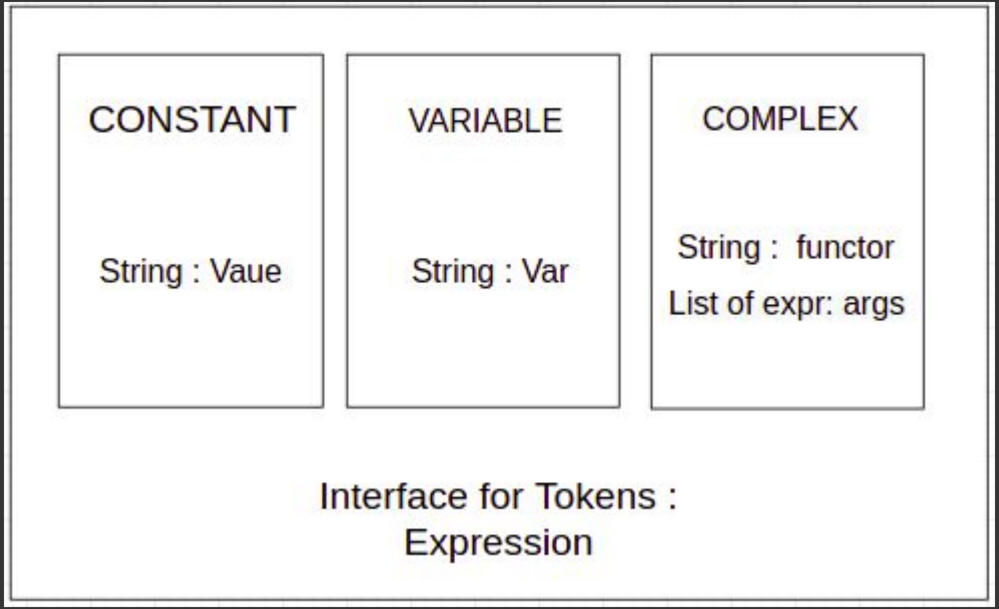
- ➔ We have 3 types of tokens in Prolog and so we have 3 classes each for one token : Constant.java , Variable.java and Complex.java .
  All these 3 classes implement a common interface called Expressions which contains the methods common to all tokens.

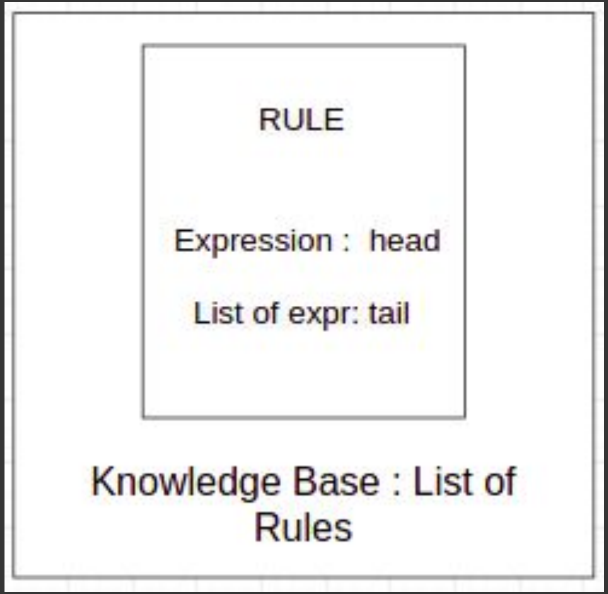
- ➔ For parsing, we have Parser.java . It has appropriate methods to parse a query, rule or fact.

- ➔ For solving a query, we have Query.java . It has the implementation of 2 major algorithms unification and proof search.

- ➔ Finally, we have Main.java which is our interface that gives real Prolog like feel.

Program Token Structure

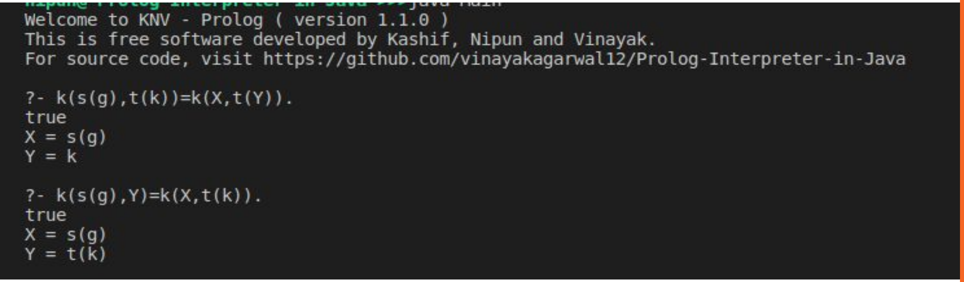




Lexical Analysis and Parsing

- Lexical analysis is the first phase of the interpreter.
- Various predefined tokens are identified.
- Expression interface is implemented by token classes.
- Three token classes namely Constant, Variable and Complex class.
- Rule Class which has three attributes - Head , Body(tail) and list of operations.
- Parsing make sure that the tokens obtained from the lexical analysis follow grammar rules described by Prolog.
- Parser class implemented to accomplish this task.
- Parser class has three methods namely parse rule, parse term and parse query.

Unification



```
Welcome to KNV - Prolog ( version 1.1.0 )
This is free software developed by Kashif, Nipun and Vinayak.
For source code, visit https://github.com/vinayakagarwal12/Prolog-Interpreter-in-Java

?- k(s(g),t(k))=k(X,t(Y)).
true
X = s(g)
Y = k

?- k(s(g),Y)=k(X,t(k)).
true
X = s(g)
Y = t(k)
```

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

Challenges in unification(solved)

- A variable can be binded to a constant or complex and another variable which in turn may be binded to a group of variables. And in these group of variables, each variable may be binded to a different constant/complex term.
- Our interpreter needs to ensure that these terms also unify among themselves.
- For that, we validated our map. We treated each expression(term) as a node in graph and there is an edge between 2 nodes if and only if there is some sort of binding present between 2 nodes.
- We collected all such disjoint sets, and applied unification between all pair of terms possible. We get a valid binding iff all such pairs unify.

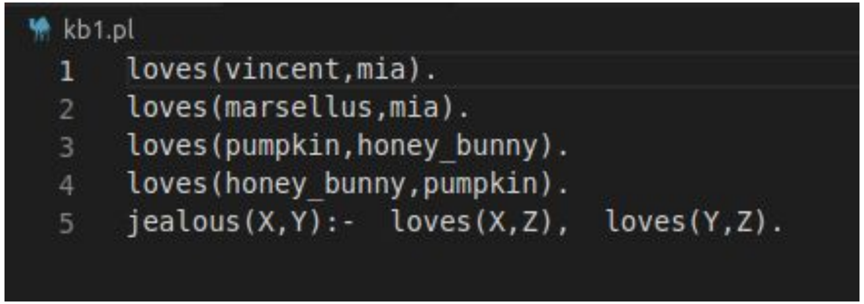
Substituting Bindings

- This method returns the expression, after substituting the variable bindings.
- We get bindings as a map from unification, and pass it to this function.
- Defined in the Expression interface as an abstract method , to induce Dynamic Polymorphism.
- Has a structure analogous to DFS, keeps a set of visited expressions.
- Constant returns a copy itself.
- Variable :
  - If binds to constant, return it.
  - If binds to variable, get binding of that.
  - If binds to complex, call method on complex.
- Complex:
  - Calls method recursively on arguments.
- Had to deal with infinite recursion and shallow copy issues.

Proof Search

- Core of how Prolog is able to perform high level tasks.
- Validates whether a given query maps to a certain rule(s) or fact(s) in our knowledge base
- Returns various Variable bindings in tree.
- Steps:
    - Unifies query with head.
    - Substitute binding in sub-rules, and call Proof Search on sub-rules.
    - Handle “and” and “or” operations through a flag and list of ops.
- Recursion tree analogous to Search tree in Prolog.

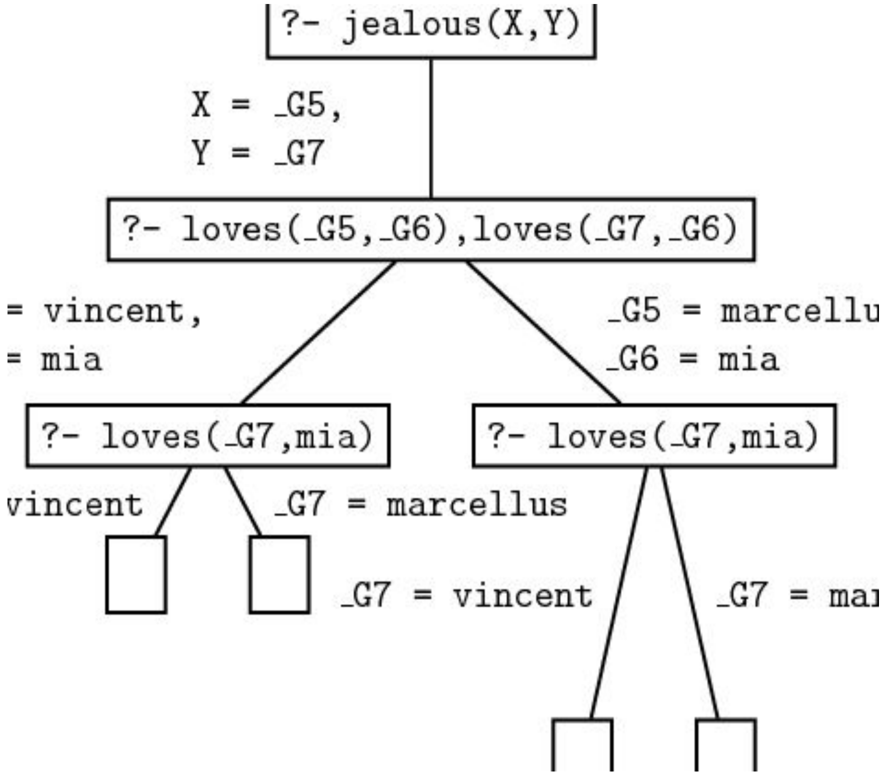
An Example:



kb1.pl

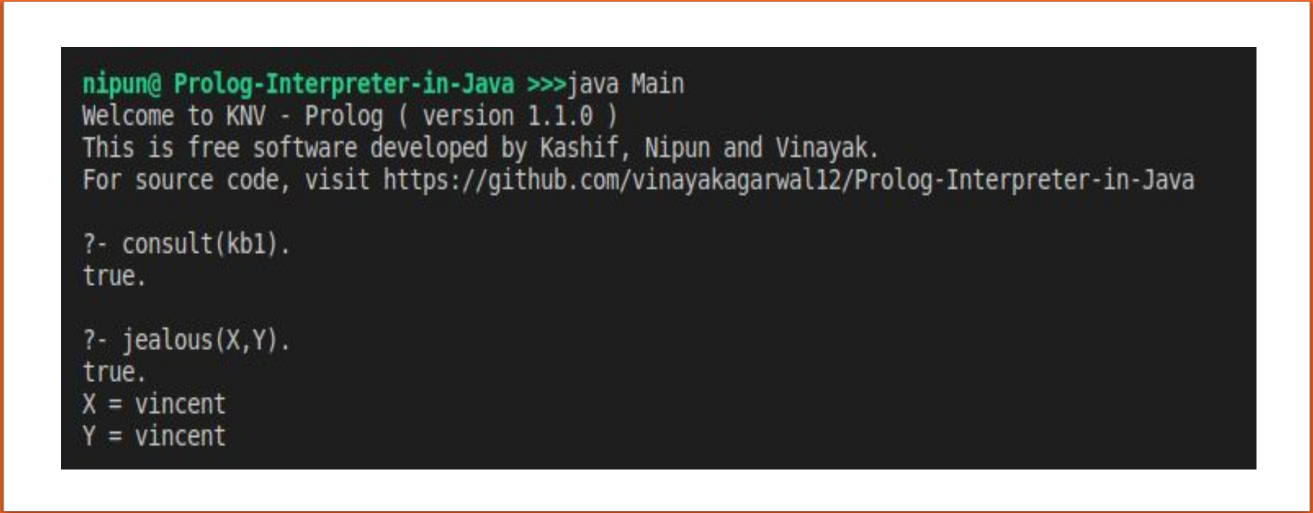
```
loves(vincent,mia).
loves(marsellus,mia).
loves(pumpkin,honey_bunny).
loves(honey_bunny,pumpkin).
jealous(X,Y):-  loves(X,Z),  loves(Y,Z).
```

The Knowledge Base



Proof search Tree for jealous(X,Y)

In our Prolog interpreter.....

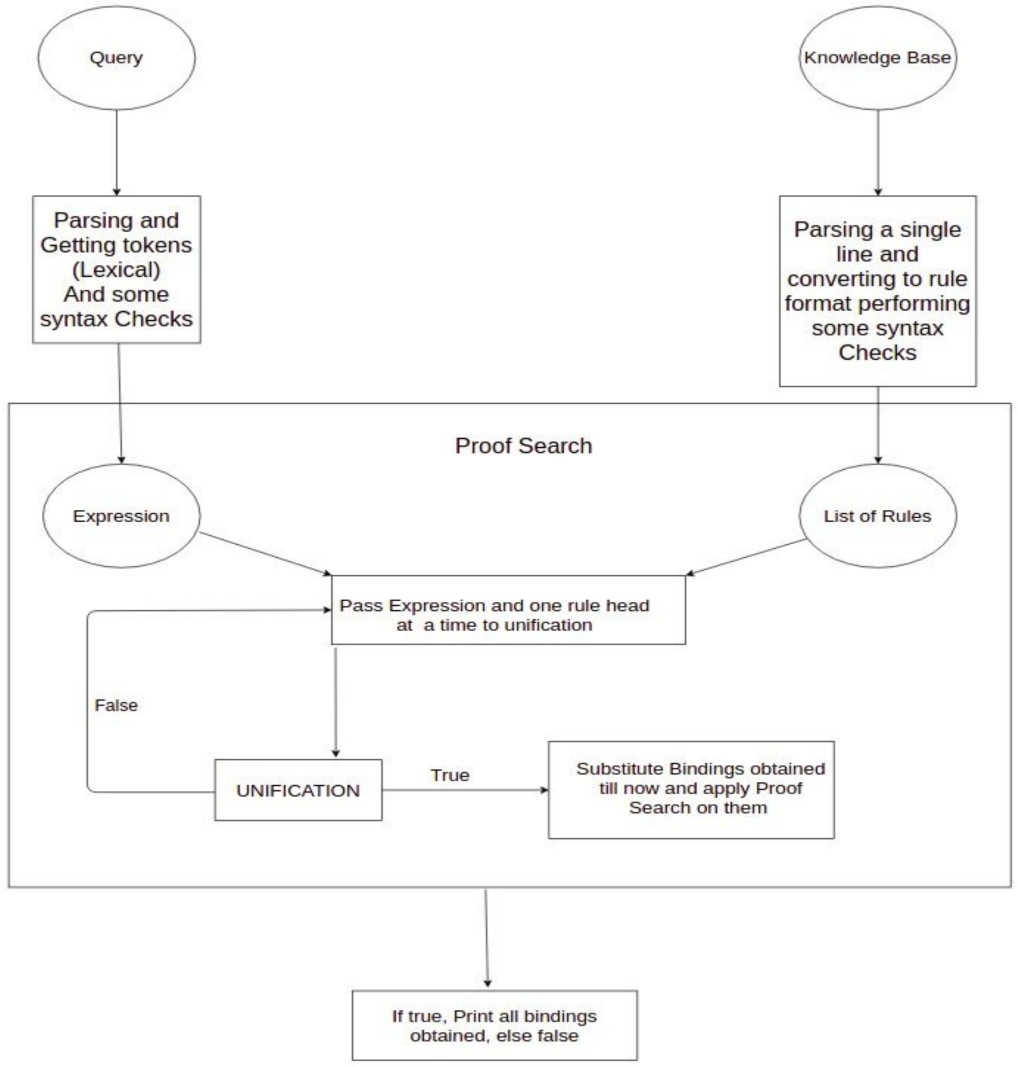


```
nipun@ Prolog-Interpreter-in-Java >>>java Main
Welcome to KNV - Prolog ( version 1.1.0 )
This is free software developed by Kashif, Nipun and Vinayak.
For source code, visit https://github.com/vinayakagarwal12/Prolog-Interpreter-in-Java

?- consult(kb1).
true.

?- jealous(X,Y).
true.
X = vincent
Y = vincent
```

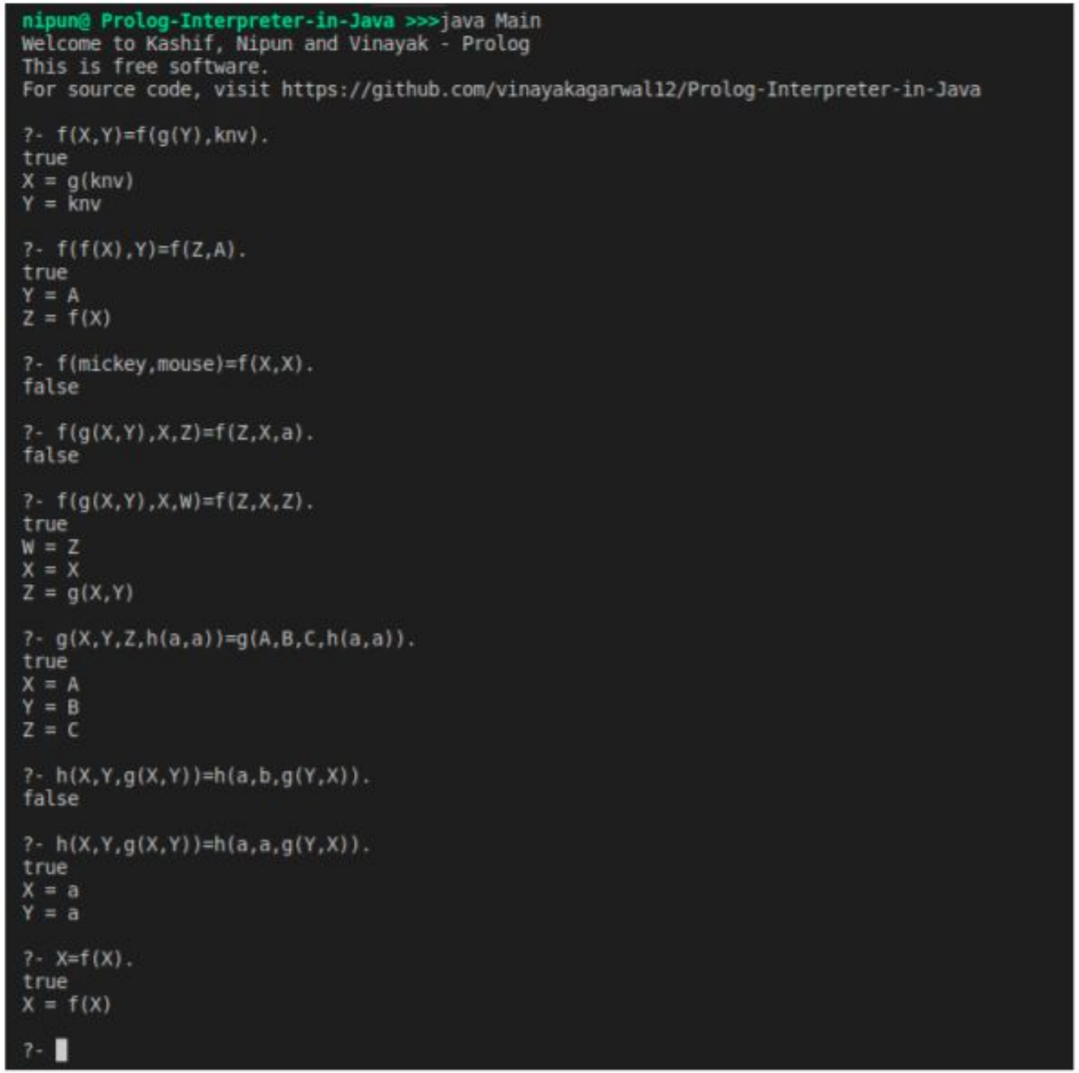
Interface

- Interface is the final component of this interpreter.
- Handles two types of inputs : - query and consult(knowledge base name)
- For input of type consult(knowledge base name) knowledge base is loaded by parsing each rule in knowledge base line by line and adding into the rules list.
- For input of type query, object of query class is created which takes two parameter input string and list of rules and finally output of the query is obtained by calling the solve method of query.

Program Flow Diagram

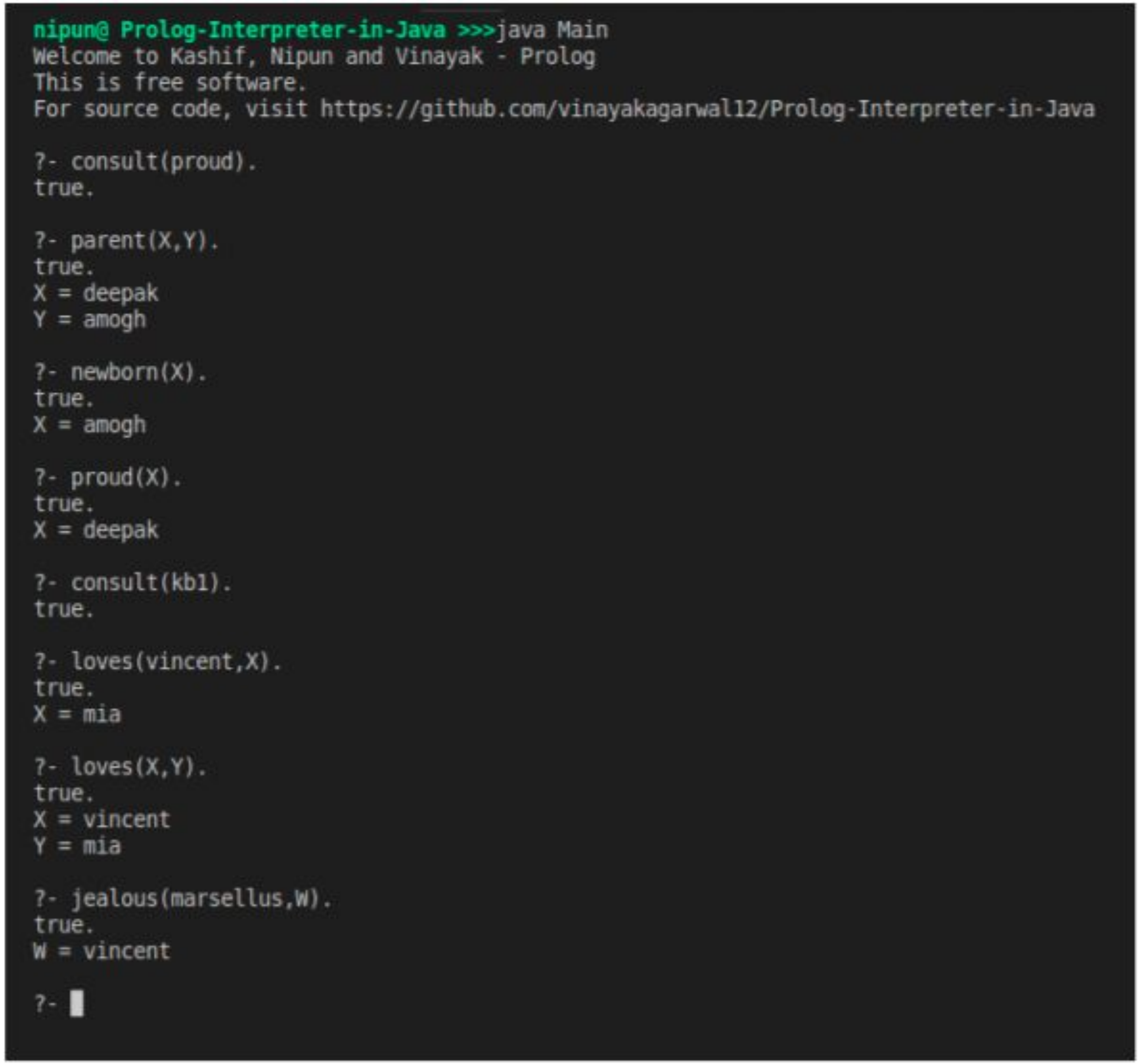

Query
Knowledge Base
Parsing and Getting tokens (Lexical) And some syntax Checks
Parsing a single line and converting to rule format performing some syntax Checks
Proof Search
Expression
List of Rules
Pass Expression and one rule head at a time to unification
False
UNIFICATION
True
Substitute Bindings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

RESULTS



```
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)

?- 
```

RESULTS



```
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

?- 
```

Future scope...

Displaying multiple results of a query by maintaining global state which tells us if a path is already exploited.

Handling operator precedence in the body of a rule in knowledge base in the same fashion as in the actual Prolog.

Implementing more advance features like lists and “NOT” operators will make the interpreter more robust.

THANK YOU!!