

Simple Prolog Interpretor

Aditya Sheth - IMT2018003

Dev Patel - IMT2018021

Prateek Kamboj - IMT2018057

May 19, 2021

Contents

1	Project	2
1.1	Abstract	2
1.2	Project Description	2
2	Solution Description	3
2.1	Simple Parser	3
2.2	Unification	3
2.3	Recursive Query Processor - Proof search	4
3	Code in action	5
3.1	Unification	5
3.2	Proof-search	6
4	Future Improvements	7
5	Conclusion	7

1 Project

1.1 Abstract

This report documents our team's implementation of a Prolog Interpreter in C++ language and a description of the various components that make up this interpreter and how we went about implementing it. It also contains some of our observations regarding this interpreter while comparing and contrasting with actual Prolog interpreter.

1.2 Project Description

The project is to build basic prolog interpreter in the C++ language.

Prolog is a logic programming language has it's roots in first order logic, a formal logic, and is intended primarily as a declarative programming language. The program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over the relations.

C++ is a powerful general-purpose programming language. It can be used to develop operating systems, browsers, games, and so on. C++ supports different ways of programming like procedural, object-oriented, functional, and so on. This makes C++ powerful as well as flexible.

The implementation of Prolog interpreter in C++ gives us a unique opportunity to deepen our understanding in both general purpose as well as Logic programming languages in a unique, yet practical way.

To achieve this implementation we built various functions and data structs in C++ that simulate the behaviour of the Prolog interpreter back-end and these are explained in detail in later sections.

In section 4, we compare and contrast results with actual prolog interpreter and discuss the capability and limitations of our implementation. Then we conclude with a note and where we could direct our efforts in the future.

2 Solution Description

2.1 Simple Parser

An input string is converted to Term data structure using Simple parsing technique. Recall that there are 3 types of terms in Prolog:

1. **Constants:** Can be either string or integers.
2. **Variables**
3. **Complex terms:** $functor(term_1, term_2, \dots, term_n)$

Now, we first parse the string and check for some condition to decide which category does this term lie in:

1. If last character is ')', this term is complex and split this string in functor and internal arguments(which are indeed terms again). we recursively parse each of the internal strings and make them term.
2. Else if the first character is Capital letter: then this term is variable.
3. Else this term is constant: Iterate through it and decide if it's a string or an integer.

2.2 Unification

Now we give brief idea to solve Unification problem:

- We maintain left term and right term at all recursive states during solving unification.
- There can be cases when any two arbitrary variables should hold the same value. To keep that information, we use an additional data structure : Disjoint Set union to keep track of equality relations between any 2 variables. If the variables X and Y are in same set in DSU struct, they must hold the same value for a valid unification.
- Now, we describe cases for solving unification instance case by case:
 1. **Left term and right term both are variable:** Update the disjoint set union by merging the sets of both variables.
 2. **One term is variable and other term is not variable:** Check previous assignment of the variable term and it must unify with the non-variable term.
 3. **Both terms are constant:** Check constant equality.
 4. **Both terms are complex:** Compare the functors of both terms, and all the corresponding arguments should unify for the whole complex term to unify.

2.3 Recursive Query Processor - Proof search

Proof searching is the process of searching through a knowledge base to satisfy a given query. Every fact in the knowledge base has either a head or both, head and a tail. A fact which has a head is the subject of the predicate and the tails is the object.

We've solved the problem where tail is set of conjunction.

Firstly, a fact F is only evaluated if the input query Q unifies with the head of the query F. In that case, we first proof-search and find all valid variable assignment solutions to each Term in the tail of Fact.

Our main aim is to find all the variable assignment for the input query Q. Now, a variable assignment for Q is valid with respect to Fact F only if this variable assignment satisfies atleast one proof-search result for every Term in tail of F. Therefore, we recursively try to solve this problem for each F with Q.

3 Code in action

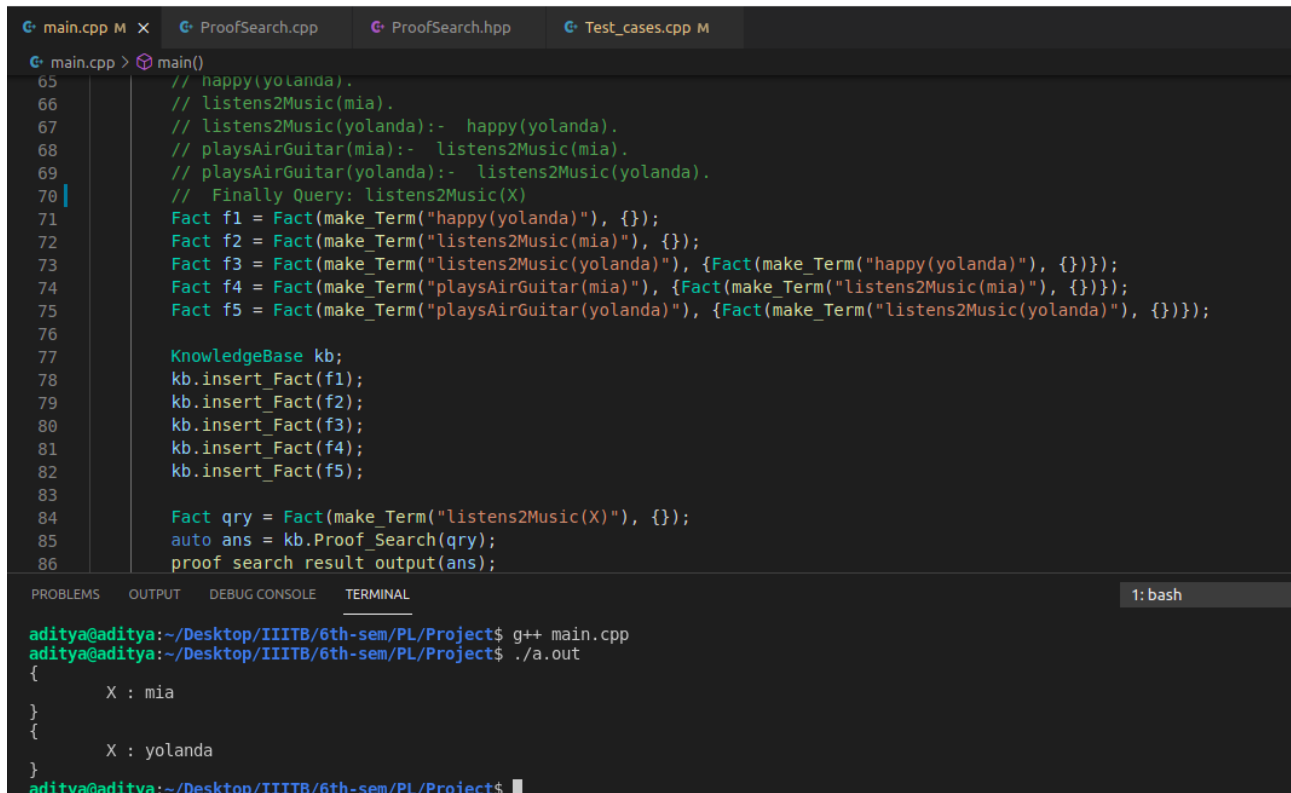
3.1 Unification

```
46
47  int main()
48  {
49
50      {
51          map<Variable, Term *> variable_assignment;
52          Term* lhs=make_Term("h(X,Y,g(X,Y))");
53          Term* rhs=make_Term("h(a,a,g(Y,X))");
54          lhs->print();
55          cout<<endl;
56          rhs->print();
57          cout<<endl;
58          bool f = unification(lhs,rhs, variable_assignment);
59          cout<<([f?"true":"false"])<<endl;
60          if (f)
61              print_variable_assignment(variable_assignment);
62      }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
aditya@aditya:~/Desktop/IIITB/6th-sem/PL/Project$ g++ main.cpp
./a.out
aditya@aditya:~/Desktop/IIITB/6th-sem/PL/Project$ ./a.out
h(X,Y,g(X,Y))
h(a,a,g(Y,X))
true
X = a; Y = a;
aditya@aditya:~/Desktop/IIITB/6th-sem/PL/Project$
```

3.2 Proof-search



```
main.cpp M x ProofSearch.cpp ProofSearch.hpp Test_cases.cpp M
main.cpp > main()
65 // happy(yolanda).
66 // listens2Music(mia).
67 // listens2Music(yolanda):- happy(yolanda).
68 // playsAirGuitar(mia):- listens2Music(mia).
69 // playsAirGuitar(yolanda):- listens2Music(yolanda).
70 // Finally Query: listens2Music(X)
71 Fact f1 = Fact(make_Term("happy(yolanda)", {}));
72 Fact f2 = Fact(make_Term("listens2Music(mia)", {}));
73 Fact f3 = Fact(make_Term("listens2Music(yolanda)", {Fact(make_Term("happy(yolanda)", {}))}));
74 Fact f4 = Fact(make_Term("playsAirGuitar(mia)", {Fact(make_Term("listens2Music(mia)", {}))}));
75 Fact f5 = Fact(make_Term("playsAirGuitar(yolanda)", {Fact(make_Term("listens2Music(yolanda)", {}))}));
76
77 KnowledgeBase kb;
78 kb.insert_Fact(f1);
79 kb.insert_Fact(f2);
80 kb.insert_Fact(f3);
81 kb.insert_Fact(f4);
82 kb.insert_Fact(f5);
83
84 Fact qry = Fact(make_Term("listens2Music(X)", {}));
85 auto ans = kb.Proof_Search(qry);
86 proof_search_result output(ans);

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash
aditya@aditya:~/Desktop/IIITB/6th-sem/PL/Project$ g++ main.cpp
aditya@aditya:~/Desktop/IIITB/6th-sem/PL/Project$ ./a.out
{
    X : mia
}
{
    X : yolanda
}
aditya@aditya:~/Desktop/IIITB/6th-sem/PL/Project$
```

4 Future Improvements

1. Code parsing is done using trivially implemented parser and it doesn't check for errors in syntax. Lexer libraries can be integrated to upgrade syntax checking.
2. The proof-search algorithm assumes a Directed acyclic structure for dependencies. To allow cyclic dependencies between facts, changes should be implemented.
3. In dependency of fact, only conjunction operation is used. It would be interesting problem to solve for any arbitrary use of (or,and) operations instead of just and operations.
4. Cuts, a way of keeping the backtracking algorithm efficient hasn't been implemented.
5. Lists, a fundamental data structure in Prolog hasn't been included in implementation. It can be added on top of the current functionality.

5 Conclusion

In this project, we have built a basic Prolog Interpreter in C++. We have implemented various components that make up this interpreter such as the Simple Lexer, Unification and recursive query processor(proof-search).

We have noted some of the key features of this interpreter as well as it's limitations. We have also documented various directions in which we can develop it further so that it can have the same functionality as a true Prolog Interpreter.