CS306: Programming Languages
IIIT-Bangalore
Ram S - IMT2017521
Rathin Bhargava - IMT2017522
Brahma Kulkarni - IMT2017011

Project Report
A Prolog Interpreter in Ocaml

June 7, 2020

Contents

1. Project

1.1 Abstract

This report documents our team's implementation of a Prolog Interpreter in the Ocaml 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 constrasting with the actual Prolog Interpreter.

1.2 Project Description

The project is to build a basic Prolog interpreter in the Ocaml 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 progtamming 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.

Ocaml is a general-purpose, multi-paradigm programming language which extends the *Caml* dialect of *ML* ("Meta-Language", a general-purpose funcitonal programming language) with object-oriented features. The *ML* derived languages are best known for their static type systems and type-inferring compilers. Ocaml unifies functional, imperative and object-oriented programming under an *ML* like type system.

The implementation of a Prolog Interpreter in Ocaml gives us a unique opportunity ot deepen our understanding in both funcitonal and logic programming in a unique, yet practical way.

To achieve this implementation we built various modules that simulate the behaviour of the Prolog Interpreter and these are explained in detail in Section 3.

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

2. Solution Description

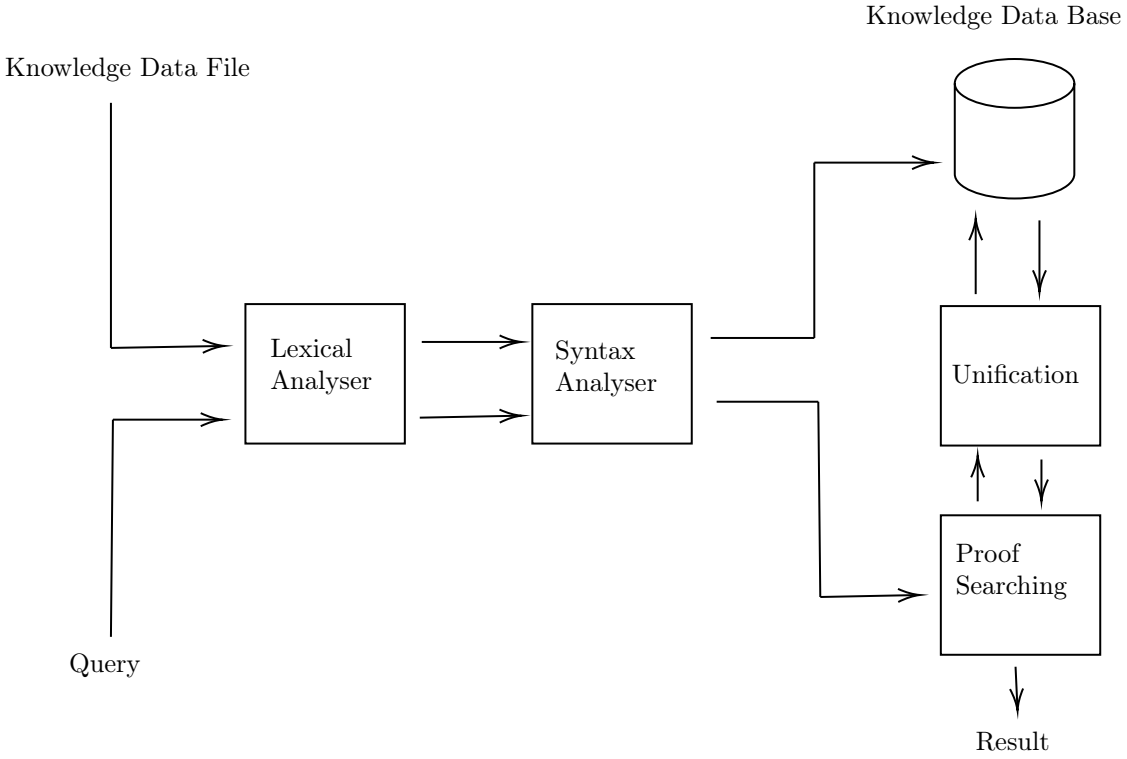
Given below is the complete description of the implementation of our interpreter.

2.1 Basic Overview

The basic components of the Prolog Interpreter are as follows:

1. Lexical Analyser - Using Ocamllex
2. Syntax Analyser - Using Ocamlyacc
3. Unification Component - Manual Implementation
4. Recursive Query Computation - Manual Implementation

The structure of the interpreter is illustrated in the following diagram:



The important point to note here is that we have simulated some of the key features that are present in Prolog (Logic Programming). We came up with the logic of implementation ourselves, so it does not represent the actual methods used by the Prolog interpreter.

2.2 Lexical Analysis

Lexical analysis deals with tokenizing a given input of characters. The input sequence is read, white spaces are ignored and the various predefined tokens from them are identified. In our case, we used the ocamllex tool to provide the specification of our lexer. Ocamllex provides us the convenience of defining regular expressions and using them to identify tokens. It throws an error if an invalid token is encountered. The lexer basically takes the input of characters and returns a list of tokens. A comprehensive list of all the regular expressions and lexer tokens that we used are shown below:

```
let digit = ['0'-'9']
let integer = ['0'-'9']['0'-'9']*
let upperCase = ['A'-'Z']
let lowerCase = ['a'-'z']
let underScore = '_'

let alphaNumeric = upperCase | lowerCase | underScore | digit

let variable = upperCase alphaNumeric*
let iden = (lowerCase alphaNumeric*)
let lparen = '('
let rparen = ')'
let comma = ',' (* also used for 'and' *)
let or_op = ';'
let not_op = '!'
let implies = [':']['-']
```

Figure 2.1: Regular expressions

```
rule scan = parse
  | [' ' '\t']      { scan lexbuf                 } (*skips blanks*)
  | ['\n']          { Parser.EOL                  }
  | comma           { Parser.COMMA                }
  | or_op           { Parser.OR                   }
  | not_op          { Parser.NOT                  }
  | implies         { Parser.IMPLIES              }
  | empty           { Parser.EMPTY                }
  | variable as v   { Parser.VAR(v)               }
  | integer as i    { Parser.NUM(int_of_string i) }
  | iden as c       { Parser.IDEN(c)              }
  | lparen          { Parser.LPAREN               }
  | rparen          { Parser.RPAREN               }
  | eof             {Parser.EOF                 }
  | _               {scan lexbuf                }
```

Figure 2.2: Tokens

More on Ocamllex: Ocamllex[2] is a tool for generating scanners: programs which recognize lexical patterns in text. Ocamllex reads the given input files, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and Ocaml code, called rules. Ocamllex generates as output an Ocaml source file which defines lexical analyzer functions. This file is compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding Ocaml code.

2.3 Syntactic Analysis

Syntactic analysis follows lexical analysis. Here, we make sure that the tokens obtained from the lexer follow certain grammar constructs that prolog, as a language, follows. Let's call the program that accomplishes this the parser. If a valid grammar construct isn't followed, an error is thrown. To accomplish this we used the ocamlyacc tool to provide the specification of our parser. Ocamlyacc provides us the convenience of defining Context Free Grammars (CFGs) to define the grammar constructs that prolog follows.

In our implementation, we have a separate program to store the expression types that we use (expr_type.ml and expr_type.mli). We have designed our parser to return those expression types when it identifies a valid grammatical construct. In the images below you can see a list of all types used and the CFG that we are using:

```
type constant =
  | NUM of int
  | IDEN of string

type expr =
  | VAR of string
  | CONST of constant
  | FUNC of string * expr list

type right =
  | LEAF of expr
  | OR of right * right
  | AND of right * right
  | NOT of right

type rule =
  | HEAD of expr
  | NODE of expr * right
```

Figure 2.3: Types

```
rule:
    | expr                              { Expr_type.HEAD($1)          }
    | expr IMPLIES right                { Expr_type.NODE($1,$3)       }


const:
    | NUM                               { Expr_type.NUM($1)           }
    | IDEN                              { Expr_type.IDEN($1)          }

right:
    | expr                              { Expr_type.LEAF($1)          }
    | right OR right                    { Expr_type.OR($1, $3)        }
    | right COMMA right                 { Expr_type.AND($1, $3)       }
    | NOT right                         { Expr_type.NOT($2)           }

expr:
    | VAR                               { Expr_type.VAR($1)           }
    | const                             { Expr_type.CONST($1)         }
    | IDEN LPAREN expr_list RPAREN      { Expr_type.FUNC($1, $3)      }

expr_list:
    | expr                              { [$1]                        }
    | expr COMMA expr_list              { $1 :: $3                    }
```

Figure 2.4: CFG

More on Ocamlyacc: Ocamlyacc[2] is a general-purpose parser generator that converts a grammar description for context-free grammars into an Ocaml program to parse that grammar. In order for Ocamlyacc to parse a language, it must be described by a context-free grammar. This means that you specify one or more syntactic groupings and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an 'expression'. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

2.4 Unification

Recall that there are 3 types of terms in Prolog:

1. Constants: Can either be atoms or numbers

2. Variables

3. Complex terms of the form: $functor(term_1, \ldots, term_n)$.

Now, the general idea behind unification is that, two terms unify if they are the same term or if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal. A more precise definition taken from book [1] is given below:
Given two terms to unify:

1. If $term_1$ and $term_2$ are constants, then term1 and term2 will unify if and only if they are the same atom/number

2. If term1 is a variable and $term_2$ is any type of term, then term1 and term2 unify and term1 is instantiated to term2. Similarly, if $term_2$ is a variable and term1 is any type of term, then $term_1$ and $term_2$ unify and $term_2$ is instantiated to $term_1$. So if they both are variables, the're both instantiated to each other, and we say that they share values.

3. If $term_1$ and $term_2$ are complex terms, then they unify if and only if:

   (a) They have the same functor and arity and

   (b) All their corresponding arguments unify and

   (c) The variable instantiations are compatible. (the unification for variables is consistent across the arguments)

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

This can be found in the Learn Prolog Now! book by Patrick, Johan and Kristina.[**?**]

2.5 Recursive Query Processor - Proof Searching Step

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

We've described the unification process when the tail is a set of disjunctions.

Firstly, a rule is only evaluated if the query unifies with the head of the query. We define a goal list to be the set of clauses in the tail of the query clause. By definition, the goal list is empty for the user-defined query. Every clause in the tail has to evaluate to *true* for the object of the predicate to be *true*. We implemented this by following 3 steps

1. **Instantiation of the goal list**
   Let's assume that the query has been successfully unified with a rule and the variables in the query have been instantiated. This instantiation is stored in the form of a hash table and is applied to every goal in the query goal list.

2. **Unification with head** The query is unified with the head. If the unification succeeds, evaluate the rule. Otherwise, try unifying with the next rule.

3. **Evaluating a rule** Rules can be of 2 types

(a) **Head** - The unification for the head and the query instantiates a set of variables in the query. This instantiation is stored in a hash table which is applied to the rest of the goals in the query goal list. The query goal list is then evaluated. If that returns true, the present instantiation works. Otherwise, we search for the next rule.

(b) **Head-Tail** We have 3 types of variables present while evaluating a Head-Tail rule. We have variables from the query, head and tail. We know that the query and the head can be unified. Note that the variables used in the tail are different from the ones used in the head. Hence, after evaluating the tail, there's a 'dictionary shift', where we substitute the values of the variables in the head before unifying the head and the query again to transfer the values of the variables back to the query.

3. Observations and Results

3.1 Test Case Evaluation Results

```
?- X = Y
X = Y
true

?- X = mia
X = mia
true

?- f(X, Y) = X
X = f(X, Y)
true

?- vertical(line(point(1, 1), point(1, 3))) = vertical(line(point(X, Y), point(X, Z)))
Y = 1
Z = 3
X = 1
true

?- vertical(line(point(X, Y), point(X, Z))) = vertical(line(point(1, 1), point(2, 3)))
false

?- food(bread, X) = food(Y, sausage)
Y = bread
X = sausage
true

?- food(bread, X) = food(Y, bread)
Y = bread
X = bread
true

?- meal(food(f1(f2(f3(Y)))), X) = meal(X, food(f1(f2(f3(Y)))))
X = food(f1(f2(f3(Y))))
true

?- meal(food(f1(f2(f3(100)))), X) = meal(X, food(f1(f2(f3(10)))))
false
```

Figure 3.1: Unification in action

3.2 Methods of Interaction

3.2.1 Running the Code

To run the code, simply run

```
$make parser
```

and to run the code, simply run

```
$./parser
```

Now that the application is running, the section below describes the way to interact with our interpreter.

3.2.2 Interaction

The main methods of interaction with our interpreter is as follows:

1. You can write down a set of facts and rules in a file (Database File) and feed it to our interpreter. The file will pass through the Lexical and Syntax Analyzer and be stored as the Knowledge Database. The filename alone should be provided in square brackets, as shown in the figure below.
2. Then, as the interpreter is waiting for input, you can specify a query, just like you would send queries in Prolog.

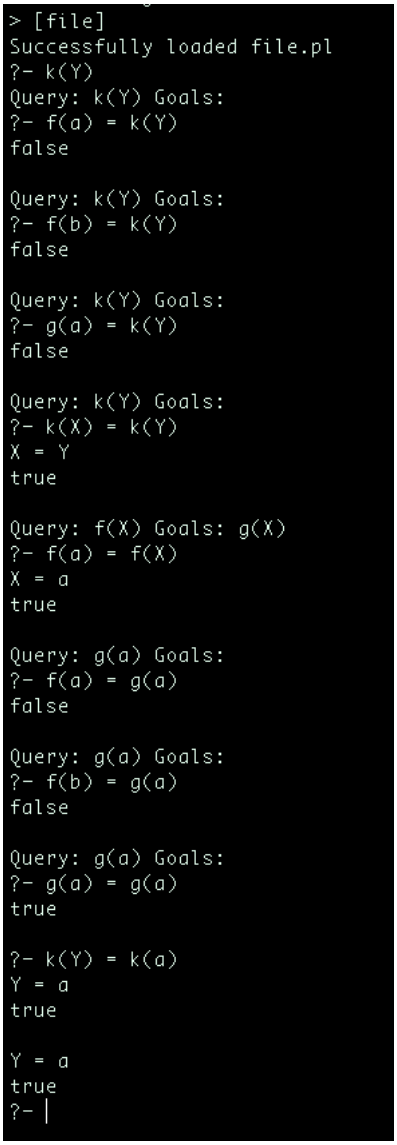


Figure 3.2: Code in action

3.3 Key Observations and Limitations

Some of the key limitations present in our interpreter are:

1. We aren't accepting multiple solutions for a query. That is, in Prolog, the user has the option of entering ; and Prolog backtracks, throws away it's current answer and continues searching for the next answer.

2. As of this implementation, every integer/floating point number is simply treated atomically. This is because we focused mainly on getting the logic programming right for atomic facts and relationships. This also implies that any arithmetic computation along with the keyword *is* hasn't been implemented.

This will probably the next important feature to implement, as no programming language is complete without it.

3. Standard functions/operators like = and \ = haven't been implemented either
4. Cuts, a way of keeping the backtracking algorithm efficient hasn't been implemented either.
5. Lists, a fundamental data structure in Prolog hasn't been implemented either. It will be an interesting task from a parsing standpoint to do so.
6. We haven't implemented the NOT logical Operator. We only handle boolean expressions in their disjunctive normal form. This is mainly because of our method of implementation of the interpreter.

   We would like to illustrate how the addition of the boolean expressions in their conjunctive form greatly increases the complexity of the Recursive Proof Searching Algorithm.
   Consider the case ($expr_1$ OR $expr_2$) AND ($expr_3$ OR $expr_4$) and other such similar expressions.

   By the structure of our implementation, we perform unification of variables for each expression within a particular depth of recursion. For simplification, we simply return back the unified variables to the previous level. Because we didn't maintain a global variable unification table, the variables unified for $expr_1$ or $expr_2$ are not visible to $expr_3$ and $expr_4$.

   If we were simply evaluating a clause in the disjunctive normal form of the boolean expression, then this wouldn't be a problem as all *expressions* must be true together for any instantiation of a variable. However, when we check for OR operators, multiple instantiations are possible since only one *expression* needs to be true for a group of expressions conjugated by OR operators. But the instantiations of a variable must carry across the entire rule.

   This necessitates the need for a global storage of unifications for each variable, which would need to be maintained at any level of the recursion and the algorithm itself should support a more sophistical level of backtracking so that for each variable we are able to keep track of (possible) instantiations for all the variables involved

4. Conclusion

In this project, we have built a Prolog Interpreter in Ocaml. We have implemented various components that make up this interpreter such as the Lexical Analyser, Syntactic Analyser, Unificaiton Module and Recursive Query Processor Module.
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.

Bibliography

[1] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!*. Texts in Computing Vol. 7, College Publications, 1-904987-17-6, 2006

[2] Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy and Jérôme Vouillon. *The OCaml system release 4.10*. Institut National de Recherche en Informatique et en Automatique, February 21st 2020