



A PROLOG INTERPRETER WRITTEN IN OCAML

By:

Ram S - IMT2017521

Rathin Bhargava - IMT2017522

Brahma Kulkarni - IMT2017011



Basic Components of the Interpreter

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. Proof Searching - Manual Implementation

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.



Lexical Analyser

- Synonym for tokenizing
- Reading, ignoring whitespaces
- Tool used: Ocamllex
- Input format: string
- Output format: identified tokens

Regular Expression representing tokens

```
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 = [':']['-']
```

Tokens

```
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 }
```



Syntax Analyser

- Adherence to the grammatical constructs of Prolog.
- Types used (expr_type.ml)
- Tool used: Ocamlyacc
- Context Free Grammars (CFGs)
- Input format: tokens from lexer
- Output format: list of tokens

Types recognized by the Parser

```
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
```

The Context Free Grammar (CFG) of Syntax Analyser

```
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              }
```




Unification - Our Implementation

Our unifier is made up of 2 mutually recursive functions - `unify_expression` and `unify_list`. It returns a `boolean*Hashtable < Variable,Expression>`

- Constants unify with each other if they are equal
- Variables unify with anything. When we encounter such a scenario, we store it in a hash table.
- Functors can be unified if their names and arity is the same and their arguments unify with each other
 - We recursively check if the arguments unify other by calling the `unify_list` function
 - `Unify_list` calls `unify_expression` for each element. It returns true if all its arguments are unified.



```
?- 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
```



Proof Searching - Definitions

- Query - Expression which is iterated through the knowledge base
- HEAD - Head of a rule, or a fact
- Assumption that tail has only AND operators
- Goal List - Tail of the rule with each clause as an element in the list



Proof Searching

- Each query has a goal list
- Once a query is unified with a rule, the variables in the unification which are present in the goal list are substituted with their real value. The goal list is then sent for computation
 - If the goal list is satisfied, the present unification is also correct. Otherwise, the next rule is chosen
 - The variables in the head are substituted with their values from the goal list
 - The unification of the head and the query is returned
- A query is unified with a rule if it unifies with the head and all the corresponding clauses in the tail are true



Modes of Interaction

Listed below are some of the main modes of interaction with our interpreter:

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

```
> [expt]
Successfully loaded expt.pl
?- f(b)
Query: f(b) Goals:
?- f(a) = f(b)
false

Query: f(b) Goals:
?- f(b) = f(b)
true

true
```



Observations

Some of the key limitations present in our interpreter are:

1. 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. It also implies that any arithmetic computation also hasn't been implemented
2. The use of ; to backtrack hasn't been done. Our proof searching algorithm searches and gets the first solution it can.
3. Functionalities like cuts, lists and other standard operators like = and \= haven't been implemented
4. As of now, we aren't using full stop as a End of Query recognizer, but instead use '\n'. This doesn't allow for multi-line queries, but we felt it wasn't important for demonstration purposes.



Observations

- Another major limitation we have in our code is that we haven't implemented the OR or NOT Logical Operators yet. This is mainly because, when we unify for a particular expression, when all the other expressions are AND, it becomes easier to check if it returns TRUE for every other expression as well.
- Consider the case `(expr 1 || expr 2) AND (expr 3 || expr4)` as a rule.



THANK YOU