INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY, BANGALORE

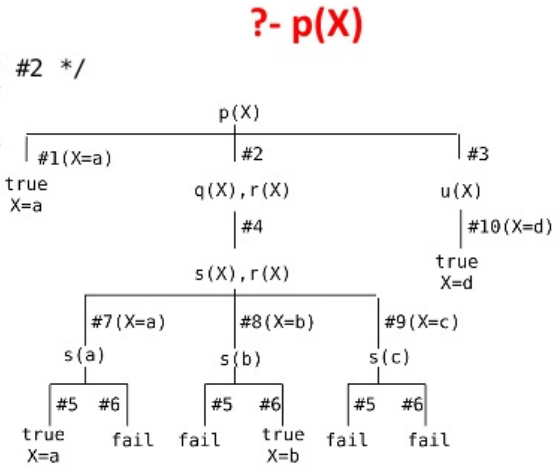
Programming Languages Project
Tiny-Prolog Interpreter

Mahidhar Bandaru | IMT2016070
Srinivasan Vijayraghavan | IMT2016083
Satvik Ramaprasad | IMT2016008

April 30, 2019

Prolog Search Tree

```
/* program P clause # */

p(a). /* #1 */
p(X) :- q(X), r(X). /* #2 */
p(X) :- u(X). /* #3 */

q(X) :- s(X). /* #4 */

r(a). /* #5 */
r(b). /* #6 */

s(a). /* #7 */
s(b). /* #8 */
s(c). /* #9 */

u(d). /* #10 */
```



CONTENTS

1 PROLOG - KEY TERMS AND FUNCTIONALITY

A Prolog program consists of a number of clauses. Each clause is either a fact or a rule. After a Prolog program is loaded in a Prolog interpreter, users can submit goals or queries, and the Prolog intepreter will give results (answers) according to the facts and rules.

1.1 FACTS

A fact must start with a predicate and end with a fullstop. The predicate may be followed by one or more arguments which are enclosed by parentheses. Arguments are separated by commas. In a Prolog program, a presence of a fact indicates a statement that is true. An absence of a fact indicates a statement that is not true

```
father(john, peter).                    /* Fact  1 */
father(john, mary).                     /* Fact  2 */
mother(susan, peter).                   /* Fact  3 */
```

1.2 RULES

A rule can be viewed as an extension of a fact with added conditions that also have to be satisfied for it to be true. It consists of two parts. The first part is similar to a fact. The second part consists of other clauses which must all be true for the rule itself to be true. These two parts are separated by ":-".

```
father(jack, susan).                    /* Fact  1 */
father(jack, ray).                      /* Fact  2 */
mother(karen, susan).                   /* Fact  3 */
mother(karen, ray).                     /* Fact  4 */

parent(X, Y) :- father(X, Y).           /* Rule  1 */
parent(X, Y) :- mother(X, Y).           /* Rule  2 */
```

1.3 GOALS

A goal is a statement starting with a predicate and probably followed by its arguments. In a valid goal, the predicate must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the goal must be the same as that appears in the consulted program. Also, all the arguemnts (if any) are constants.

```
parent(jack, susan).
```

1.4 Queries

A query is a statement starting with a predicate and followed by its arguments, some of which are variables. Similar to goals, the predicate of a valid query must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the query must be the same as that appears in the consulted program.

The purpose of submitting a query is to find values to substitute into the variables in the query such that the query is satisfied.

1.5 Summary of Prolog Artifacts

Facts and Rules form the prolog program (database of information and rules). Rules can be considered as a generalization of a fact. Similarly, goals and queries are questions we ask the prolog program. Queries can be considered as a generalization of goals.

1.6 Built in Predicates

Prolog has certain built in predicates

- Arithmetic predicates (arithmetic operators): +, -, *, / (These operators are the same as those found in other programming languages. They only operates on numbers and variables.)
- Comparison predicates (comparison operators):
    1. < (less than) - operates on numbers and variables only
    2. > (greater than) - operates on numbers and variables only
    3. =< (less than or equal to) - operates on numbers and variables only
    4. >= (greater than or equal to) - operates on numbers and variables only
    5. is - the two operands have the same values
    6. = - the two operands are identical
    7. =\= the two operands do not have the same values

2 PROLOG SEARCH ALGORITHM

The following things happen during the life of a prolog program.

1. A Prolog program is consulted and the facts and rules in the program are loaded into the knowledge database.
2. When a query or goal is given, the Prolog interpreter seaches the knowledge database from **top to bottom** to find any facts or rules match with the query or goal.
3. If a match is found, the search for answers to the query succeeds or the goal is true. The values of any variables found are given.
4. The user can choose to see multiple solutions if available.

The Prolog interpreter starts with the goal/query and works by identifying the facts and the rules from which the query can be derived. This can be done by the expansion by applying a rule and the reduction by applying a fact.

2.1 UNIFICATION

Unification is the derivation of a new rule from a given rule through the binding of variables. In order to match a goal or a query, any variable encountered is substituted with the value of an appropriate constant.

2.2 BACKTRACKING

Sometimes there are more than one fact/rule which can be applied to a goal/query. In this case, the fact/rule which appears first in the knowledge database will be applied first. The other applicable facts or rules will be applied later if more than one solution is required or if the previous solution didn't work. This is done in a way that all possible solutions are found. This process is known as backtracking.

2.3 BASIC ALGORITHM - RULE RESOLUTION

Consider the Rule R1 to have sub-goals G1, G2 and G3. Two stacks are maintained as follows. One stack containing all the pending sub-goals called S1 and another containing all the successful sub-goals called S2.

When a successful unification happens to a goal in S1, it is popped out from S1 and pushed into S2. If there are no goals left in S1, then return success. If goal in S1 has no solution and S2 is empty, then return failure. If goal in S1 has no solution and S2 is not empty, then pop last goal from S2 and backtrack. This essentially involves asking the more recent successful sub-goal for a new solution.

```
1. Initialize stack S1 with sub-goals of R1, create empty stack S2
2. if S1 is empty then return success
3. G = Pop S1
4. soln = NextSoln(G)
5. if status(soln) = success then
     4.1  Unify soln, push G to S2 and go to Step 2
   else
     4.2 if S2 is not empty then
        4.2.1 Backtrack - Pop G from S2 and push to S1 and go to Step 2
        else
        4.2.2 Return failure
```

2.4 Caveat with Implementation of Backtracking

Implementation of backtracking can be tricky as all the computations, bindings etc has to be undone when backtracking takes place. More importantly, it is important to maintain the state/path of the current solution, so that a new solution can be generated.

3 Our implementation of Prolog in OCaml

Our implementation of prolog uses `ocamllex` which produces a lexical analyser from regular expressions and `ocamlyacc` which produces a parser from a grammar.

3.1 Lexing and Parsing

3.1.1 Lexing

The following is the OCaml lex specification of regular expressions. This will generate the lexical analyser.

```
let integer = ['0'-'9']['0'-'9']*
let id = ['a'-'z''A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9']*
let string = ['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9']*
let variable = ['A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9']*

rule scan = parse
  | [' ' '\t' '\n']+  { scan lexbuf                       }
  | '('               { Parser.LPAREN                     }
  | ')'               { Parser.RPAREN                     }
  | ":-"              { Parser.COLON                      }
  | string as s       { Parser.STRING (s)                 }
  | integer as s      { Parser.INTEGER((int_of_string s)) }
  | variable as s     { Parser.VARIABLE (s)               }
  | '.'               { Parser.DOT                        }
  | ','               { Parser.COMMA                      }
  | eof               { Parser.EOF                        }
  | '['               { Parser.L_SQUARE                   }
  | ']'               { Parser.R_SQUARE                   }
  | "'"               { Parser.APOSTROPHE                 }
```

3.1.2 PARSING

The following is the OCaml yacc specification of the parsing grammar. This will generate the parser.

```
%{
%}

%token          NEWLINE APOSTROPHE WS COMMA EOF LPAREN RPAREN
                COLON DOT L_SQUARE R_SQUARE
%token <int>    INTEGER
%token <string> STRING
%token <string> VARIABLE

%start database
%start filename
%start interpreter_query
%type <Expression.predicate> predicate
%type <Expression.value> value
%type <Expression.argument> argument
%type <Expression.query> query
%type <Expression.query> interpreter_query
%type <Expression.predicateName> predicateName
%type <Expression.predicate list> database
%type <string> filename

%left ADD SUBTRACT
%left AND OR
%right NOT

%% /* Grammar rules and actions follow */

database :
 database_r EOF {$1}
;

database_r:
| predicate                                          {[$1]}
| predicate database                          {$1 :: $2}
;

predicate :
 | predicateName LPAREN argumentlist RPAREN DOT {Expression.Rule($1, $3, [])}
;
```

```
predicateName:
 | STRING {Expression.PredicateName($1)}
;

argument:
 | VARIABLE {Expression.Variable($1)}
 | value    {Expression.Constant($1)}
;

value :
 | STRING    {Expression.Atom($1)}
 | INTEGER   {Expression.Integer($1)}
;

rulelist :
 | query                                    {[$1]}
 | query COMMA rulelist                   {$1 :: $3}
;

interpreter_query:
 query DOT {$1}
;

query:
 | predicateName LPAREN argumentlist RPAREN {Expression.Query($1, $3)}
;

filename:
  | L_SQUARE APOSTROPHE STRING DOT STRING APOSTROPHE R_SQUARE DOT  { $3 ^ "." ^ $5 }
;
argumentlist :
 | argument {[$1]}
 | argument COMMA argumentlist                {$1 :: $3}
;
%%
```

3.2 Abstract Syntax Tree Type Specification

Variant type `value` currently supports only integers and atoms. Variant type `argument` can either be a Variable or a Constant. Variant type `predicate` can either be built in predicates or facts and rules.

```
type value =
    | Integer of int
    | Atom of string;;

type argument =
    | Constant of value
    | Variable of string;;

type predicateName = PredicateName of string;;

type query = Query of predicateName * argument list;;

type predicate =
    | Rule of predicateName * argument list * query list
    | LessThan of argument * argument
    | GreaterThan of argument * argument
    | LessThanOrEqualTo of argument * argument
    | GreaterThanOrEqualTo of argument * argument
    | Is of argument * argument
    | Equal of argument * argument
    | IsNot of argument * argument;;

type database = predicate list
```

3.3 Features Implemented

Our current query resolution implementation only supports atom type even though our lexer parses integers as well. The backtracking framework proved to be tricky to implement in OCaml. We have implemented both goals as well as queries. The interpreter implemented is very similar to swi-prolog and gives multiple solutions if available.

3.4 Backtracking Implementation

We implemented backtracking using the algorithm described in section 2.3 in OCaml. The implementation was done as follows. There are two functions `resolveQuery()` and `resolveRule()`. `resolveQuery()` attempts to find a solution match to a query or goal. `resolveRule()` is

used when a query or goal is matched to a rule. Note that in a single query, `resolveQuery()` and `resolveRule()` can call each other several times.

However in order to implement backtracking algorithm, we need a way to generate next solution. We need to have a way to maintain solution state as described in section 2.4.

Therefore both `resolveQuery()` and `resolveRule()` returns a generator. Internally they maintain solution state of the dynamic solution tree. A call on either of the generators, returns a new solution. If there is no solution available, it will raise an exception. This is implemented recursively, that is `resolveRule()` `generator` maintains reference to `resolveQuery()` `generators` and vice versa.

4 Conclusion

We have successfully explored and used OCamllex and Ocamlyacc to implement a lexer and a parser respectively. We have done a study on how prolog query resolution works internally and seen algorithms to implement the same. We have successfully implemented back tracking algorithm on Rules and Queries. The following is an example output of a multiple solution query.

```
| ?- ['database.pl'].
| ?- fact4(X,Y).

        = X = a, Y = b
        = X = b, Y = c
        = X = c, Y = d
        = X = d, Y = e
        = X = e, Y = f
        = X = a, Y = c
        = X = a, Y = d
        = X = a, Y = e
        = X = a, Y = f
        = X = b, Y = d
        = X = b, Y = e
        = X = b, Y = f
        = X = c, Y = e
        = X = c, Y = f
        = X = d, Y = f
```

5 FUTURE SCOPE

The current interpreter can be extended to implement built-in predicates as described in Section 1.6. With this, more complex queries like factorial etc can be performed. Further, features like cuts can be implemented. Other constructs like Structures, Lists etc can be explored as well.

REFERENCES

[1] Ocaml lex and ocaml yacc. https://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html.

[2] Prolog backtracking approach. https://people.eng.unimelb.edu.au/adrianrp/COMP90054/lectures/Prolog_How_Works.pdf.

[3] Prolog concepts. http://www.ablmcc.edu.hk/~scy/prolog/pro02.htm.

[4] Prolog interpreter. https://github.com/swapnil96/Toy-Prolog-Interpreter.