A6 :- ~~Toy~~ ~~Logic~~ PROLOG INTERPRETER

file : lexer.mll

```
{ type token = VAR of string | ID of string
             | LPAREN | RPAREN | PERIOD | COMMA | IFF
  exception EOF
}

rule token = parse
    {' ' '\t' '\n'}      { token lexbuf }
  | ['.']                { PERIOD }
  | [',']                { COMMA }
  | ['(']                { LPAREN }
  | [')']                { RPAREN }
  | ":-"                 { IFF }
  | ['A'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm   { VAR (lxm) }
  | ['a'-'z']['a'-'z' 'A'-'z' '0'-'9' '_']* as lxm   { ID (lxm) }
  | eof                  { raise EOF }

{ let main () = begin
    let outbuf = Buffer.create 255 in
    try let filename = Sys.argv.(1) in
    let file_handle = open_in filename in
    while true do
      let result = token lexbuf in match result with
          | VAR(x) -> Printf.bprintf outbuf "VAR \n%s\n" x
          | ID(x)  -> Printf.bprintf outbuf "ID \n%s\n" x
          | IFF    -> Printf.bprintf outbuf "IFF \n"
          | PERIOD -> Printf.bprintf outbuf "PERIOD\n"
          | LPAREN -> Printf.bprintf outbuf "LPAREN\n"
          | RPAREN -> Printf.bprintf outbuf "RPAREN\n"
          | COMMA  -> Printf.bprintf outbuf "COMMA\n"
    done with | EOF -> begin
      let filename = Sys.argv.(1) ^ "-lex" in
      let file_handle = open_out filename in
      Printf.fprintf file_handle "%s" (Buffer.contents outbuf);
      close_out file_handle
    end
    | _ -> begin Printf.printf "Error while printing to file";
      exit 1
    end
  in () ;;
}
```

file: parsermly :- ①

```
%{ open Backend ;; %}

%token <string> ID VAR
%token LPAREN RPAREN COMMA IFF PERIOD
%token EOF

%start parser_program
%type <Backend.clause list> parser_program

%%
parser_program : clause_list      { $1 } ;

clause_list : /*empty*/    { [] }
            | clause PERIOD clause_list      { $1 :: $3 } ;

clause : term      { Clause ($1, []) }
       | term IFF term_list    { Clause ($1, $3) } ;

term_list : term      { [$1] }
          | term COMMA term_list    { $1 :: $3 } ;

term : ID LPAREN symbol_list RPAREN   {Term (Id $1, $3)} ;

symbol_list : symbol      { [$1] }
            | symbol COMMA symbol_list   { $1 :: $3 } ;

symbol : ID     { Id $1 }
       | VAR    { Var $1 }
       | term   { $1 } ;

%%
```

file: parser.mly ②

```
let rec read_tokens token_filename =                    (* returns the queue of
                                                           tokens based on the
                                                           input lex file *)
    let fin = open_in token_filename in
    let tokens_queue = Queue.create() in
    let get_line () = String.trim (input_line fin) in
    ( try while true do
        let token_type = get_line () in
        let token = match token_type with
            | "ID" -> ID(get_line ())
            | "VAR" -> VAR (get_line ())
            | "LPAREN" -> LPAREN
            | "RPAREN" -> RPAREN
            | "COMMA" -> COMMA
            | "IFF" -> IFF
            | "PERIOD" -> PERIOD
            | _ -> begin Printf.printf "Unexpected Token\n"
                   exit 1
                   end in
        Queue.add token tokens_queue
      done
      with _ -> () );
    close_in fin ;
    tokens_queue ;;

let main () = begin                                     (* the parser functio
                                                           *)
    let token_filename = Sys.argv.(1) in
    let tokens_queue = read_tokens token_filename in
    let lexbuf = Lexing.from_string "" in
    let lexer_token lb =
        if Queue.is_empty tokens_queue then EOF
        else begin
            let next_token = Queue.take tokens_queue in
            next_token
        end
    in
    let ~~xs~~ claus_list = parser_program lexer_token lexbuf in
    ~~resolve_query claus_list~~
    let query = read_query () in
    resolve_query claus_list query [] 0 [] ;
    Printf.printf "Parsed Successfully \n" ;
end ;;

main () ;;
```

file : backend.ml ①

```
type symbol = Id of string | Var of string | Term of symbol * (symbol list) ;;
type clause = Clause of symbol * (symbol list) ;;
exception NOT_UNIFIABLE ;;

let rec advance_by offset lst =                    (* returns lst[offset:] *)
    if (offset <= 0) then lst
    else match lst with
        | [] -> []
        | hd::tl -> advance_by (offset-1) tl
    ;;

let rec delete_front lst =                         (* deletes the front element of list lst *)
    match lst with
        | [] -> []
        | hd :: tl -> tl
    ;;

let rec subst_var v s =                            (* substitutes Var 'v' as per the substitution list 's'.
    match s with                                      substitution is symbol*symbol list *)
    | [] -> Var v
    | (Var v1, Var t1) :: tl -> if v=v1 then Var t1 else (subst_var v tl)
    | (Var v1, Id sym) :: tl -> if v=v1 then Id sym else (subst_var v tl)
    | (Var v1, Term (p,l))::tl -> if v=v1 then Term (p,l) else (subst_var v tl)
    ;;

let rec subst trm_lst s =                          (* substitutes the term list trm_lst with substitution s *)
    match trm_lst with
        | [] -> []
        | (Var v)::tl -> (subst_var v s) :: (subst tl s)
        | (Term (p,l)) :: tl -> (Term (p, (subst l s))) :: (subst tl s)
        ;;
```

file : backend.ml ②

```
let rec occurs v l =                    (* occurs-check for mgu
                                           - checks if the Var 'v' occurs
                                             in term list 'l'  *)
    match l with                        (* true if 'v' occurs in 'l' *)
    | [] -> false
    | (Var v1)::tl -> if v=v1 then true else (occurs v tl)
    | (Term (pred, l1))::tl -> (occurs v l1) || (occurs v tl)
    ;;

let rec mgu l1 l2 =                     (* Returns the mgu of two term-
                                           lists l1 and l2 (if unifiable) *)
    match (l1, l2) with
(* (Var, Var) *)
    |([], []) -> []
    |((Var v1)::tl1 , (Var v2)::tl2) -> (Var v1, Var v2)::(mgu tl1 , tl2)
(* (Id, Id) *)
    |( (Id sym1)::tl1, (Id sym2)::tl2) -> if (sym1 = sym2)
                                          then [(mgu tl1 tl2)]
                                          else raise NOT_UNIFIABLE
(* (Var, Id) *)
    |( (Var v1)::tl1, (Id sym2)::tl2) -> (Var v1, Id sym2)::(mgu tl1, tl2)
(* (Id, Var) *)
    |((Id sym1)::tl1, (Var v2)::tl2) -> (Var v2, Id sym1)::(mgu tl1 tl2)
(* (Var, Term) *)
    |( (Var v1)::tl1 , (Term (p2,l2))::tl2) -> if (occurs v1 l2)
                                               then raise NOT_UNIFIABLE
                                               else (Var v1, Term (p2,l2))::(mgu tl1 tl2)
(* (Term, Var *)
    |((Term (p1,l1))::tl1 , (Var v2)::tl2) -> if (occurs v2 l1)
                                              then raise NOT_UNIFIABLE
                                              else (Var v2, Term(p1,l1))::(mgu tl1 tl2)
(* (Term, Term) *)
    |( (Id pred1 , sym_list1)::tl1 , (Id pred2, sym_list2)::tl2)
                -> if (pred1 = pred2) then
                   List.append (mgu sym_list1 sym_list2) (mgu tl1 tl2)
    | _ -> raise NOT_UNIFIABLE
    ;;
```

file : backend.ml ③

```
let rec find_mgu claus_lst index hd =          (* finds the mgu of the
  match claus_lst with                          head_term of clause_list
  | [] -> ([], index)                           with the query term 'hd'
  | (hd_trm, tl_trms_lst) :: tl_clauses -> try (mgu [hd_trm] [hd])   Also returns the index at which a
                                                                     unifiable clause-head term is found *)
                                          with
                                          | NOT_UNIFIABLE -> find_mgu tl_clauses (index+1) hd
(* Query resolution function *)            | _ -> ((mgu [hd_trm] [hd]), index)

let rec resolve_query kb_clause_list goal prev_goals_list tree_level offset_list =
    match (goal, tree_level) with
| ([], x) -> if (x=0) && (prev_goals_list = []) then true.
        else resolve_query (advance_by ((List.nth offset_list 0)+1) kb_clause_list)
                           (List.nth prev_goals_list 0)
                           (delete_front prev_goals_list)
                           (tree_level - 1)
                           (delete_front offset_list)
| (hd::tl, x) -> let (s, temp_offset) = find_mgu kb_clause_list
                                        (List.nth offset_list 0)
    in if (s=[]) && (tree_level=0) then false
    else if (s=[]) && (tree_level <> 0)
    then resolve_query (advance_by ((List.nth offset_list 0)+1) kb_clause_list)
                       (List.nth prev_goals_list 0)
                       (delete_front prev_goals_list)
                       (tree_level - 1)
                       (delete_front offset_list)
    else resolve_query (kb_clause_list)
                       (subst (append tl_trms_lst (delete_front goal)) s)
                       goal
                       (tree_level + 1)
                       (List.append [temp_offset] offset_list)
;;
```

file : README :

Compilation Steps and Execution steps :

1. Generate 'lexer' from 'lexer.mll'

```
ocamllex lexer.mll
ocamlopt -o lexer lexer.ml
```

2. Generate the lexer tokens file from the knowledge base file 'kb'. This step creates the file 'kb-lex' which contains the tokens from lexed 'kb'.

```
./lexer kb
```

3. Generate 'parser' from file 'parser.mly'

```
ocamlyacc parser.mly
rm parser.mli
ocamlopt -c backend.ml
ocamlopt -c parser.ml
ocamlopt -o parser str.cmxa backend.ml parser.ml
```

4. Parse the lex file 'kb-lex'

```
./parser kb-lex
```

5. Enter query on STDIN

Remarks :-

A. Query resolution occurs through the function 'resolve-query' in file 'backend.ml' which does search with backtracking for query resolution.

Example 'kb' file :-

```
male(arjun).
male(pandu).
male(yudhishthir).
father(pandu, arjun).
father(pandu, yudhishthir).

son(X, Y) :- male(X), father(Y, X).
```