

# Logics for Computer Science

## A (mini)Prolog implementation in OCaml

Marek Doniec      Stéphane Lescuyer

April 2005



Figure 1: A taste of Prolog...

# Contents

<b>1</b>	<b>Our (not so)mini-Prolog</b>	<b>3</b>
1.1	The core . . . . .	3
1.2	Useful extensions . . . . .	4
1.2.1	Negation as failure . . . . .	4
1.2.2	Lists . . . . .	5
1.2.3	Numerical constants, arithmetic expressions and the <i>is</i> operator . . . . .	5
1.2.4	Further goodies . . . . .	6
1.3	What it can't do yet . . . . .	6
<b>2</b>	<b>The OCaml implementation</b>	<b>7</b>
2.1	OCamllex - OCaml yacc . . . . .	7
2.1.1	Translating an input stream into tokens with ocamllex . . . . .	7
2.1.2	Parsing a context-free grammar with ocaml yacc . . . . .	8
2.2	Using streams for a finite representation of possibly infinite proof objects . . . . .	9
2.3	Unification and the search for proofs . . . . .	10
<b>3</b>	<b>Appendice A : Our implementation's grammar</b>	<b>12</b>
<b>4</b>	<b>Appendice B : The prompt commands' grammar</b>	<b>15</b>

# Introduction

Prolog is an unavoidable tool when studying logics for computer science. Therefore, it is of great advantage to fully understand its structure and its behavior. To do so, reimplementing Prolog by yourself helps very much, as was one of the main goals of this project. OCaml being the language of choice here, figures well because it is a strongly-typed functional language, where functions are first-order values, which made the manipulations of the data structures easier and safer.

## 1 Our (not so)mini-Prolog

In this section, we describe the features of our implementation of Prolog.

### 1.1 The core

Most of the modern Prolog implementations advertise with having more than three hundred built-in predicates and contain far more than is needed for first-order logic proofs. Also, while useful for writing programs, implementing most of these functions would have not helped to improve our general understanding of Prolog.

Thus, our goal was to implement the basic logical rules needed for first-order logic and the adequate proof search engine.

In the source files for our Prolog interpreter, the user should be able to define two different kinds of terms : *facts* and *clauses*.

- Facts are predicates on symbolic constants or variables, *e.g.*

```
weather(today,good).  
german(marek).  
french(stephane).  
equal(X,X).
```

- Clauses are Horn clauses, which means that given a number of proven predicates, another predicate can be proved. One can see these clauses as a disjunction of a finite number of formulae, of which all but one are negations of atomic formulae, *e.g.*

```

weather(X,good) :- sunshines(X), warm(X).
project(X,Y,good) :- german(X), french(Y).
project(X,Y,good) :- french(X), german(Y).
brother(X, Y) :- mother(X, Z), mother(Y, Z), notthesame(X,Y).

```

The user should then be able to load these terms in the prompt and ask Prolog to prove some things about them. We call *goals* a conjunction of terms that are to be proved. Goals have the same structure as facts (and therefore shall not be clauses), but they can be viewed as questions that must be answered using the facts and clauses defined earlier. These goals can be entered in the prompt after the traditional `-?`.

These are examples of queries that the user should be able to type in our implementation :

```

-? german(X).
...
-? french(X), german(X).
...
-? project(X,Y,good), brother(X,Y).
...
-? brother(marek, stephane).

```

## 1.2 Useful extensions

### 1.2.1 Negation as failure

The above described system allows us to describe a logical world and search proofs for certain goals in it. The problem is, that describing certain relations and data structures can be very difficult or even impossible. If the user wishes to write a program where all letters but *a* are provable, he will have to put 25 facts

```

nota(b).
...
nota(z).

```

in the program. Writing a program that lets us prove every constant except *a* becomes impossible, because there are too many constants (theoretical an infinite number).

It is evident that we miss negation here. The solution we implemented is called *negation as failure*, sometimes also called negation for failure. It means

that prolog will try to prove the negated term and only continue should this proof fail. If Prolog finds a proof for the negated term, the proof of the current term fails and Prolog backtracks.

We chose the common symbol `/+` for negation as failure. So to prove all but *a*, the user can now simply write

```
isa(a).
nota(X) :- \+ isa(X).
```

### 1.2.2 Lists

Another problem the user will encounter with the basic version of Prolog is the representation of data. Lists for example can be represented as nested funtions, where `cons(a,L)` means that *a* is attached to the list *L* and `nil` is the empty list. Thus `cons(a,cons(b,cons(c,nil)))` is a perfect representation for the OCaml list `[a;b;c]`. While being logically equivalent, this representation is not very userfriendly. It is much more desirable to simply write `[a,b,c]` than the above mentioned structure. Therefore we have implemented in the parser corresponding rules, that translate the representation of lists using brackets and commas into their functional representation. Since the user wants to be able to match lists, we have implemented the operator `|` that allows to split a list into its head and tail.

### 1.2.3 Numerical constants, arithmetic expressions and the *is* operator

We encountered with numbers the same problem as for lists. Surely, these can be written and manipulated as Church numbers, but this is rather of theoretical meaning and interest and does not suit a general application of Prolog. Unfortunately, simply adding numerical constants and arithmetic expressions to Prolog does not solve our problem. Since Prolog represents all data as structural trees, the comparison of `2 + 1` with `3` will yield the result `No`<sup>1</sup>.

For evaluating and simultaneously allowing a way of assigning values to variables, we have implemented the `is` operator. `M is exp` will first evaluate the arithmetic expression on the right. To do so, all used variables have to be already assigned due to prior substitutions in the proof search. Should a variable during evaluation not already be assigned or if there is a recursive assignement, then the proof search will terminate, notifying the user about

---

<sup>1</sup>for the comparison operator see section 1.2.4

the error. After evaluation, Prolog will substitute all occurrences of `M` in the current proof state with the result of the calculation.

Given these nice new features, we can now compute the length of lists :

```
length([],0).
length([_|L],N) :- length(L,M), N is M + 1.
```

`-? length([a,b,c],X).` will thus return `Yes, X = 3` as a proof and we have successfully retrieved the length of our list `[a,b,c]`.

### 1.2.4 Further goodies

In addition to the features mentioned above, we found it useful to add some additional internal operators. Structural equality for example can be very easily tested by adding the fact `equal(X,X).` to the program and thus defining the needed operator. But this is a rather slow solution and assuming that, in OCaml, all data structures are already represented as trees, we added the binary operator `=` that will not only test more efficiently for structural equality, but also allows for the much more intuitive notation `X = X.` rather than `equal(X,X).`

We also implemented the operator `==` which is semantical equality, meaning that it will first evaluate both sides and then compare the numerical values. Again, should one of the used variables not be instantiated, then the proof will terminate with an error notification. In addition we implemented the semantical operators `<` and `<=` which operate accordingly with `==`.

The last addition was the n-ary predicate `write(a,b,...)` which will print all n terms given as arguments. Since we did not add strings as a data structure, it prints the structure of the given terms and separates them with whitespaces. At the end a newline is also printed.

## 1.3 What it can't do yet

Of course, our implementation lacks some important features that one can find in professional Prolog implementations like GNUProlog.

First of all, we did not implement the so-called *cut rule*, that amongst other things, allows one to implement a sort of `if .. then .. else ..` control sequence. We believe this would be the very next thing to add to our implementation if we had some more time. Moreover, this implementation completely lacks user input, file I/O, other data types than integers, and basically all connectivity to the working environment.

## 2 The OCaml implementation

In this section, we describe some details of the actual OCaml implementation.

### 2.1 OCamllex - OCamllyacc

Unlike the Standard ML version of the program that we translated, we chose to use the `ocamllex` and `ocamllyacc` utilities to achieve the tokenization and the parsing of both the source files and the input in the prompt.

#### 2.1.1 Translating an input stream into tokens with `ocamllex`

Translating an input stream into a list of tokens is the first step for parsing the user input. The most common and low-level way to do so is to read the characters from the input buffer one after the other, grouping them together into the desired tokens. This can be very difficult sometimes, especially when reading the first character of an input does not give any accurate information on the nature of the token, and that one has to read more and more characters forward. Not only is it very error-prone, but the code you get can be very tricky to understand, to debug, and last but not the least, to modify if you want to slightly change the parsed language.

The `ocamllex` utility is an easy way to achieve lexical analysis in OCaml without these obstacles. In a `.mll` source file, one can define rules for transforming the input stream into tokens, using regular expressions on characters and strings. For example, here is the `goallexer.mll` source for the parsing in the prompt :

```
% open Goalparser
% exception LexError of int * int

rule token = parse
  [' ' '\t' '\n' '\r']
    { token lexbuf }      (* skip blanks *)
| '"' ((['A'-'Z','a'-'z','.','0'-'9']* as filename) '"'
    { FILE filename }
| (['A'-'Z'] (['A'-'Z','a'-'z','0'-'9']* as mot)
    { VSYM mot }
| (['a'-'z'] (['A'-'Z','a'-'z','0'-'9']* as mot)
    { check_keyword mot }
| (['0'-'9']+ as entier)
    { CONST(int_of_string entier) }
```

```

| "."          { DOT }
| ","         { COMMA }
| "|"         { BAR }
...
| ']'         { RBRACKET }
| "\\+"      { NOT }
| ""         { raise (LexError ...) }

```

This rules transform the given regular expressions into tokens. To separate the keywords of the language from the user-defined variable names, we use a hashtable<sup>2</sup> containing the reserved keywords, and we call the function `check_keyword` to return either a keyword symbolic token or a variable name token :

```

let keywords = Hashtbl.create 17
let check_keyword s =
  try Hashtbl.find keywords s with Not_found -> CSYM s
and add_keyword s k = Hashtbl.add keywords s k
;;

add_keyword "help" HELP ;
add_keyword "reload" RELOAD ;
add_keyword "open" OPEN ;
add_keyword "quit" QUIT ;
add_keyword "nil" NIL ;
add_keyword "cons" CONS ;
add_keyword "is" IS ; ()
;;

```

### 2.1.2 Parsing a context-free grammar with `ocamlyacc`

In the previous paragraph, the first line of our `goallexer.mll` is the following

```
% open Goalparser
```

because the definition of the token abstract type is not in the lexer file, but in the parser. The parser is `ocamlyacc` and given the description of a LALR(1) grammar in a `.mly` file, it computes a finite state machine to parse the language described by the grammar. The grammar must be described

---

<sup>2</sup>because in a real implementation, there would be a lot of keywords and hashtables are more efficient in that case

with rules for non-terminal symbols, and the terminal symbols are precisely the symbolic constants used as tokens.

Each rule of the grammar is also associated to an action, which describes how to construct an object, given the objects associated to the non-terminal and terminal symbols used in the rule. For example, the following is an excerpt from the `goalparser.mly` showing the rules for the axiom of the grammar (they roughly correspond to the different kind of commands you can type in the prompt, and the rest of the grammar is basically the actual parsing of the goals) :

```
/* the entry point */
%start main
%type <Terms.command > main
%%
main:
  HELP helpkeyword DOT { Help($2)}
| QUIT { Quit }
| RELOAD { Reload }
| OPEN FILE {Open($2)}
| terme COMMA main { match $3 with Goals(v) -> Goals($1::v)
                      | _ -> assert false }
| terme DOT {Goals([$1])}
;

terme:
...

```

*The full syntax of our Prolog can be found in the appendices*

## 2.2 Using streams for a finite representation of possibly infinite proof objects

A problem one has to face when searching for proofs in a Prolog program is that there might be an infinite number of proofs or that there is one proof at the beginning of the search and after that the search goes on for an infinite time with no results. In each case we wish to find the first proof and then continue to search if the user desires to do so, but we do not wish to lose the state of our search. OCaml allows for a very neat solution with the fact that functions can be passed as parameters. We construct a stream that is capable of holding substitution data (which might be a proof) and the function call for continuing the search. To do so, our stream contains two types of cells:

1. `ForcedCell` is used to hold an actual substitution.
2. `DelayedCell` contains only a function that will, when being called, create the content of this cell and initialize the next cell.

When we try to read a cell that is in the state `DelayedCell` it will automatically compute its value and set itself to `DelayedCell`, eventually initializing the next cell. When the `prove` function is called the first time, it will return such a stream object. Of course it will only return if a first proof is found. Such a structure does not help to prevent that there might be no proof at all or that the search may fall into an infinite searchpath without a proof on it. This stream will contain the substitutions for the first proof and the second cell contains the call to continue the search and return the second proof. Thus once this stream is empty we know that there is no further proof.

### 2.3 Unification and the search for proofs

To actually prove a list of goals entered by the user, our implementation uses the unification rules that we saw in class.

More precisely, the `unify` function in `unify.ml` takes a list of term couples and returns the *most general unifier* (mgu) of the unification problem. This function is use by the function `body_and_sub` in `toplevel.ml` to unify a goal and one of the facts or clauses loaded from the program<sup>3</sup>. When it succeeds, `body_and_sub` applies this substitution to the body of the clause and then returns a stream of all the mgus and modified bodies that it obtains this way when iterating through the program.

This stream is basically all the ways one can use to prove a given goal according to a given program. Therefore, the function `prove` is reapplied to each element of this stream, considering the modified bodies as new goals, and appends (through the `squash` function) the adequate substitutions it finds into a stream.

Given a unique goal, `prove` gives namely the stream of substitutions that each make this goal true. When called upon a list of goals, `prove` begins by computing this stream for the head of the list, and for each substitution in this stream, applies it to all the other goals and tries to prove them afterwards. That way, it finds all the substitutions that each make all the goals true, according to the program.

---

<sup>3</sup>if it is a clause, then it unifies the goal with the head of the clause

## Conclusion

In the end, we can say that we not only translated the Prolog implementation we had been given from one language to another, but we used new tools as `ocamllex` and `ocamlyacc` and added new build-in predicates and new features to the code. Still, the main experience was to learn how Prolog really works, how proofs are searched, and thus how it can be used.

### 3 Appendice A : Our implementation's grammar

```
%{
open Terms
open Clauses
%}
%token <int> CONST
%token <string> CSYM
%token <string> VSYM
%token PLUS MINUS TIMES DIV
%token DOT COMMA BAR LPAREN RPAREN LBRACKET RBRACKET COLONHYPHEN
%token NOT IS CONS NIL
%token EOF

%left PLUS MINUS
%left TIMES DIV
%left UMINUS
%nonassoc DOT COMMA BAR LPAREN RPAREN LBRACKET RBRACKET COLONHYPHEN

/* the entry point */
%start main
%type <Clauses.clause list> main
%%
main:
  clause main { $1::$2 }
| EOF {[]}
;

clause:
  terme DOT { elabClause $1 }
| predicate COLONHYPHEN termelist DOT
  { elabClause (App(Fun(":-", List.length $3 +1), $1::$3)) }

termelist:
  terme {[$1]}
| terme COMMA termelist { $1::$3 }

terme:
  NOT terme { App(Fun("\\+",1),[$2]) }
```

```

| LPAREN terme RPAREN { $2 }
| VSYM IS arithmexp { App(Fun("is",2),[Var(VName($1));$3]) }
| predicate { $1 }

predicate:
  funname LPAREN args RPAREN {
    let lArgs = $3 in
    App(Fun($1,List.length lArgs),lArgs)
  }

args:
  expression { [$1] }
| expression COMMA args { $1::$3 }

expression:
  terme { $1 }
| arithmexp { $1 }
| expression_non_terme { $1 }

expression_non_terme:
  LPAREN expression_non_terme RPAREN { $2 }
| LBRACKET list RBRACKET { $2 }

arithmexp:
  ident {$1}
| LPAREN arithmexp RPAREN { $2 }
| MINUS arithmexp %prec UMINUS { App(Fun("-",1),[$2]) }
| arithmexp PLUS arithmexp { App(Fun("+",2),$1::$3::[]) }
| arithmexp MINUS arithmexp { App(Fun("-",2),$1::$3::[]) }
| arithmexp TIMES arithmexp { App(Fun("*",2),$1::$3::[]) }
| arithmexp DIV arithmexp { App(Fun("/",2),$1::$3::[]) }

list:
  { Con(CName "nil") }
| expression { App(Fun("cons",2),[$1; Con(CName "nil")]) }
| expression BAR ident { App(Fun("cons",2),[$1; $3]) }
| expression COMMA list { App(Fun("cons",2),[$1; $3]) }

ident:
  CSYM { Con(CName($1)) }
| CONST { Con(CInt($1)) }

```

```
| VSYM { Var(VName($1)) }  
| NIL { Con(CName( "nil")) }
```

funname:

```
  CSYM { $1 }  
| COLONHYPHEN { ":-" }  
| CONS { "cons" }
```

## 4 Appendice B : The prompt commands' grammar

```
%{
open Terms
open Clauses
%}
%token <int> CONST
%token <string> CSYM
%token <string> VSYM
%token <string> FILE
%token PLUS MINUS TIMES DIV
%token DOT COMMA BAR LPAREN RPAREN LBRACKET RBRACKET
%token NOT IS CONS NIL
%token QUIT OPEN RELOAD
%token HELP

%left PLUS MINUS
%left TIMES DIV
%left UMINUS
%nonassoc DOT COMMA BAR LPAREN RPAREN LBRACKET RBRACKET

/* the entry point */
%start main
%type <Terms.command > main
%%
main:
  HELP helpkeyword DOT { Help($2)}
| QUIT { Quit }
| RELOAD { Reload }
| OPEN FILE {Open($2)}
| terme COMMA main
  { match $3 with Goals(v) -> Goals($1::v) | _ -> assert false }
| terme DOT {Goals([$1])}
;

terme:
  NOT terme { App(Fun("\\+",1),[$2]) }
| LPAREN terme RPAREN { $2 }
| funname LPAREN args RPAREN {
```

```

        let lArgs = $3 in
          App(Fun($1,List.length lArgs),lArgs)
    }
| VSYM IS arithmexp { App(Fun("is",2),[Var(VName($1));$3]) }

args:
  expression { [$1] }
| expression COMMA args { $1::$3 }

expression:
  terme { $1 }
| arithmexp { $1 }
| expression_non_terme { $1 }

expression_non_terme:
  LPAREN expression_non_terme RPAREN { $2 }
| LBRACKET list RBRACKET { $2 }

arithmexp:
  ident {$1}
| LPAREN arithmexp RPAREN { $2 }
| arithmexp PLUS arithmexp { App(Fun("+",2),$1::$3::[]) }
| arithmexp MINUS arithmexp { App(Fun("-",2),$1::$3::[]) }
| arithmexp TIMES arithmexp { App(Fun("*",2),$1::$3::[]) }
| arithmexp DIV arithmexp { App(Fun("/",2),$1::$3::[]) }
| MINUS arithmexp %prec UMINUS { App(Fun("-",1),[$2]) }

list:
  { Con(CName "nil") }
| expression { App(Fun("cons",2),[$1; Con(CName "nil")]) }
| expression BAR ident { App(Fun("cons",2),[$1; $3]) }
| expression COMMA list { App(Fun("cons",2),[$1; $3]) }

ident:
  CSYM { Con(CName($1)) }
| CONST { Con(CInt($1)) }
| VSYM { Var(VName($1)) }
| NIL { Con(CName("nil")) }

funname:
  CSYM { $1 }

```

```
| CONS { "cons" }
```

```
helpkeyword:
```

```
    { "open\nhelp\nquit\nType help <command> for more information.\n" }  
| OPEN { "open <filename> consults the code contained in <filename>.\n" }  
| QUIT { "quit closes this application.\n" }  
| VSYM { "The command you typed is unknown.\n" }  
| CSYM { "The command you typed is unknown.\n" }
```