Prolog Documentation

Release 1.0.0

Adam J. Stewart and Zaid Qureshi

Jan 18, 2018

Contents

CHAPTER 1

Overview

We implement a simple version of Prolog in OCaml using the material we learned in the course.

1.1 Motivation

Prolog is a logic programming language used in fields like artificial intelligence.

Facts and rules (i.e. Horn Clauses) are used to express the program logic in Prolog. Prolog computation involves querying a database of facts and rules. If a given query can be proven for a given database, Prolog outputs the answers for the query and a message like “Yes” or “true” to tell the user that the query was proven. If the query can’t be proven, either a message like “false” or an error is outputted to the user. A query can have comma separated subgoals and the evaluation of the query is the evaluation of the conjunction between all of the subgoals.

In the Prolog program below, `cat(tom)` and `animal(X)` are both compound terms, `tom` is an atom, `X` is a variable, `cat(tom).` is a fact, `animal(X) :- cat(X).` is a rule (or Horn Clause), and `?- animal(X).` is a query. In this program, the query `?- animal(X).` evaluates to true where the one and only mapping for the variable `X` is `X = tom`.

```
cat(tom).
animal(X) :- cat(X).

?- animal(X).
```

The unification algorithm, similar to the one that was presented in lecture 16 and we implemented for ML4 during the semester, is at the center of query evaluation in Prolog. The lexer and parser for Prolog can be implemented in a similar way to how we implemented a lexer and parser for PicoML in MP4 and ML5, respectively, during the semester. Thus, this project should thoroughly test our understanding of some of the most important topics from the course.

1.2 Goals

When we looked at different Prolog implementations, we were surprised by how different they were from one another. We therefore proposed that our implementation would fully support Simon Krenger’s Prolog grammar. Based on this grammar, we planned on supporting string, integer, and float literals, variables, atoms, facts, rules based on conjunction, and queries based on conjunction of the subgoals in the query. We also proposed that our implementation would be able to interpret from both files and a command-line interpreter.

As we were not able to finish this project during the semester, we had to ask for an extension. Professor Gunter suggested we modify our original project proposal and leave string, integer, and float literals out of our implementation unless we had enough time.

1.3 Accomplishments

Our final implementation implements all of the goals we proposed in our initial project proposal except for the ability to interpret from files. We had enough time to support string, integer, and float literals as well.

Our implementation supports moderately complicated queries based on conjunction of the subgoals in the query.

We have also implemented a detailed testing framework with test suites for each one of the major components in our implementation.

CHAPTER 2

Implementation

2.1 Capabilities

We implemented an abstract syntax, parser, lexer, evaluator, and interpreter for a simple version of Prolog in OCaml. The interpreter is used to interact with the user so that the user can input Prolog facts, rules (Horn clauses), or queries and see the output. The user's input is tokenized by our lexer, the tokens are converted into our abstract syntax by our parser, and the abstract syntax tree is then evaluated by our evaluator.

Our components fully support the grammar we mentioned in our project proposal. We support the string, integer, and float Prolog literals. Prolog variables, atoms, and compound terms are fully supported. Prolog facts and rules based on conjunction are also fully supported. Prolog queries consisting of the components mentioned previously are also fully supported.

We also implemented a build system, test suites and framework, and continuous integration to thoroughly test our implementation.

2.2 Components

2.2.1 Build System

In order to build the project, we used the OCamlbuild build system. OCamlbuild allowed us to write a greatly simplified Makefile, as OCamlbuild performs a static analysis of the code to determine the correct order in which to build and link every file. Additionally, OCamlbuild uses OCamlfind to locate external dependencies like Menhir.

Our Makefile has both `native` and `byte` targets to build either a native or bytecode executable, respectively. By default, `make` builds a `main.byte` executable. When executed, this program provides an interactive interpreter for entering Prolog clauses and queries.

To view our OCamlbuild configuration, see `_tags`.

2.2.2 Abstract Syntax Tree

`ast.ml` contains the types needed to represent an abstract syntax tree of a Prolog program. Each line of the program is either a *clause* or a *query*. There are two types of clauses: *rules* and *facts*.

Rules

A *rule* in Prolog takes the form:

```
Head :- Body.
```

For example, a complex rule of the form:

```
sibling(X, Y) :- parent_child(Z, X), parent_child(Z, Y).
```

can be represented with the following abstract syntax tree:

```
Clause (
    TermExp ("sibling", [VarExp "X"; VarExp "Y"]),
    [
        TermExp ("parent_child", [VarExp "Z"; VarExp "X"]),
        TermExp ("parent_child", [VarExp "Z"; VarExp "Y"])
    ]
)
```

Facts

In Prolog, a rule with no body is called a *fact*. As an example, the fact:

```
cat(tom).
```

is syntactic sugar for the rule:

```
cat(tom) :- true.
```

and is represented as an abstract syntax tree in a way that reflects this:

```
Clause (
    TermExp ("cat", [TermExp ("tom", [])]),
    [
        TermExp ("true", [])
    ]
)
```

Queries

A query is an inquiry into the state of the database, and takes the form:

```
?- Body.
```

For example, a query of the form:

```
?- sibling(sally, erica).
```

can be represented with the following abstract syntax tree:

```
Query ([
    TermExp ("sibling", [
        TermExp ("sally", []);
        TermExp ("erica", [])
    ])
])
```

Terms

In Prolog, there is only a single data type, the *term*, which can either be an *atom*, *number*, *variable*, or *compound term*. Compound terms take the form:

```
functor(arg1, arg2, ...)
```

In order to simplify the language, we treat atoms as compound terms with arity zero.

2.2.3 Lexer

For lexing, our token list was largely based off of ECLiPSe Prolog. Additional inspiration was taken from Amzi! Prolog and SWI-Prolog.

Atoms

Atoms are identified by alphanumerical tokens starting with a lowercase letter, or any sequence of characters surrounded by single quotes.

Numbers

Our lexer supports tokenization of both positive and negative integers, floats, scientific notation, and infinity.

Strings

Strings are identified by any sequence of characters surrounded by double quotes. In addition, consecutive strings are automatically concatenated into a single string.

Variables

Variables are identified by alphanumerical tokens starting with a capital letter or underscore.

Comments

Our lexer supports line comments (identified by `%`) and multi-line comments (identified by `/*` and `*/`). Although not all Prolog implementations agree on nesting, our lexer supports nested multi-line comments.

Rules

Our lexer requires five lexing rules: one for general tokens, one for comments, one for atoms, one for strings, and one for escaped character sequences. Since both atoms and strings can contain escaped characters, the rule for handling escape sequences takes a callback rule as a parameter. This callback tells the lexer which rule to return to after the escaped character sequence has been evaluated. Our lexer handles both octal and hexadecimal characters in escape sequences.

2.2.4 Parser

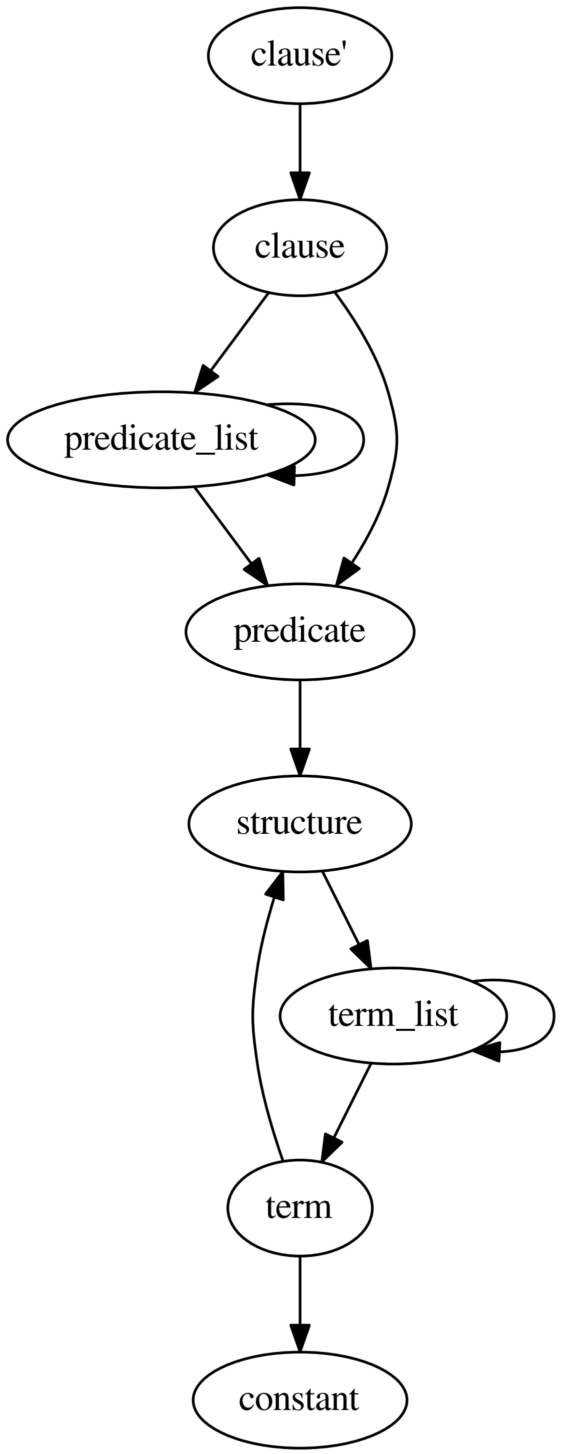
For parsing, our grammar was largely based off of Simon Krenger's Prolog parser. Additional inspiration was taken from ECLiPSe Prolog and SICStus Prolog, although we do not support the full range of syntaxes that those implementations do.

The full BNF grammar we support is listed here:

```
clause          ::=   <predicate> . |
                      <predicate> :- <predicate_list> . |
                      ?- <predicate_list> .
predicate_list  ::=   <predicate> |
                      <predicate> , <predicate_list> |
predicate       ::=   atom |
                      <structure>
structure       ::=   atom ( ) |
                      atom ( <term_list> )
term_list       ::=   <term> |
                      <term> , <term_list>
term            ::=   <constant> |
                      atom |
                      var |
                      <structure>
constant        ::=   int |
                      float |
                      string
```

Instead of OCamlyacc, we decided to use Menhir as our parser generator. Menhir offers several benefits over OCamlyacc, including more readable error messages and the ability to name semantic values instead of using the traditional keywords: `$1`, `$2`, etc.

The following graph represents the connections between each non-terminal in our grammar, and was generated using `menhir --graph` and Graphviz:


clause'
clause
predicate_list
predicate
structure
term_list
term
constant

2.2.5 Evaluator

The top level function of the evaluator is `eval_dec` in `evaluator.ml`. The function takes in a declaration to evaluate and a database. The database is a list of declarations, more specifically `ClauseExp`, representing the facts and rules the user has entered so far. The declaration to evaluate can be either a `ClauseExp`, representing a new fact or rule to add to the database, or a `QueryExp`, representing a query to answer.

Evaluating a Clause

To evaluate a declaration `d` that is a `ClauseExp` with a database `db`, the evaluator returns a new database with `d` prepended to `db`. The one exception to this is if `d` is giving meaning to the `true` atom. We consider `true` to be a built-in predicate used only to define facts and thus users are not allowed to redefine it. In the case the user tries to add a clause for the `true` atom, a message is printed telling the user that this is not possible and `db`, the original database, is returned.

Evaluating a Query

To evaluate a declaration `d` that is a `QueryExp` (a goal) with a database `db`, the evaluator has to use the facts and rules in `db` to prove all of the subgoals in the query `d`. A subgoal is an element of the list of `exp` that defines a `QueryExp`. A query asks to prove all of (i.e. the conjunction of) the subgoals. After evaluating all possible results, the evaluator prints each result including the binding of all the variables in the query, if there were any, then prints `"true"` if there was at least one result and `"false"` otherwise, and returns `db`, the database passed in.

The Query Evaluation Algorithm

Our query, or goal, evaluation algorithm was adapted from an algorithm presented by Dr. Hrafn Loftsson of Reykjavik University in one of his video lectures. We used the behavior of query evaluation in SWI-Prolog as the example for our evaluator to follow. This includes things like the order in which subgoals are evaluated and the order in which the database is walked to find rules and facts to prove a subgoal.

The pseudocode for our implementation of the algorithm to evaluate a query `G` with a database `db` is listed here:

```
eval_query (G, db, subs):
  if G is empty:
     return [subs]
  else if G = (g1 :: g):
     results = []
     G' = g
     if g1 = true:
        results = results @ eval_query(G', db, subs)
     else:
        foreach ClauseExp(h,[b1 .. bn]) in db:
           if unify(g1, h) = σ1:
              if n = 1 and b1 = true:
                 G' = σ1(g)
              else:
                 G' = σ1([b1 .. bn] @ g)
              if unify(σ1 @ subs) = σ2:
                 results = results @ eval_query(G', db,  σ2)
              else:
                 continue

           else:
              continue
```

```
    return results
```

The first thing the `eval_query` function does is check if `G` is empty, meaning that there are no subgoals in `G` to prove and that the substitutions in `subs` provide one solution for the query. Since there is nothing left to prove for `G` the function returns the substitutions inside of a list. This is necessary because at the end `eval_query` returns a list of list of substitutions, where each element is a set of substitutions that proved the query.

If `G` is not empty, then there is at least one subgoal, `g1`, to prove and `g` is the possibly empty list of other subgoals. Since `g1` is the head of the list, it will be the leftmost subgoal in the goal. So we always try to prove the leftmost subgoal, just like how SWI-Prolog does it. If `g1` is the `true` predicate then we do not need to prove it and can move on to the other subgoals in `g`. Otherwise, to prove `g1`, we iterate over the database `db` in the order in which the entries in the database were entered and find each rule or fact in the database that matches with `g1`. A rule or fact matching `g1` implies that the rule or fact can be used to prove `g1`. Since both facts and rules are represented as a `ClauseExp` with a head (`h`) and body (`[b1 .. bn]`) component, to match `g1` with a rule or fact we use unification on the constraint `[{g1, h}]`. If unification succeeds and a substitution $\sigma 1$ is returned, we can use that rule or fact to prove `g1`. If the entry from the db was a fact, the only subgoals left to prove are in `g`, so our new goal `G'` gets assigned to the result of applying the substitution $\sigma 1$ to `g`. If the entry from the db that matched `g1` was a rule, then we have more subgoals to prove, more specifically the subgoals from the body of the rule, `[b1 .. bn]`, along with the other remaining subgoals from `g`. In this case, `G'` is set to the substitution $\sigma 1$ applied to the result of prepending the body of the rule to `g`. Then the substitution $\sigma 1$ is appended to the substitutions passed into `eval_query`, `subs`, and the result is unified to give us a new substitution $\sigma 2$ for proving this answer. We add to our list of results thus far (`results`) the result of recursively calling `eval_query` with `G'` as the new goal and $\sigma 2$ as the new `subs`. For a subgoal `g1`, this process happens for each item in the database.

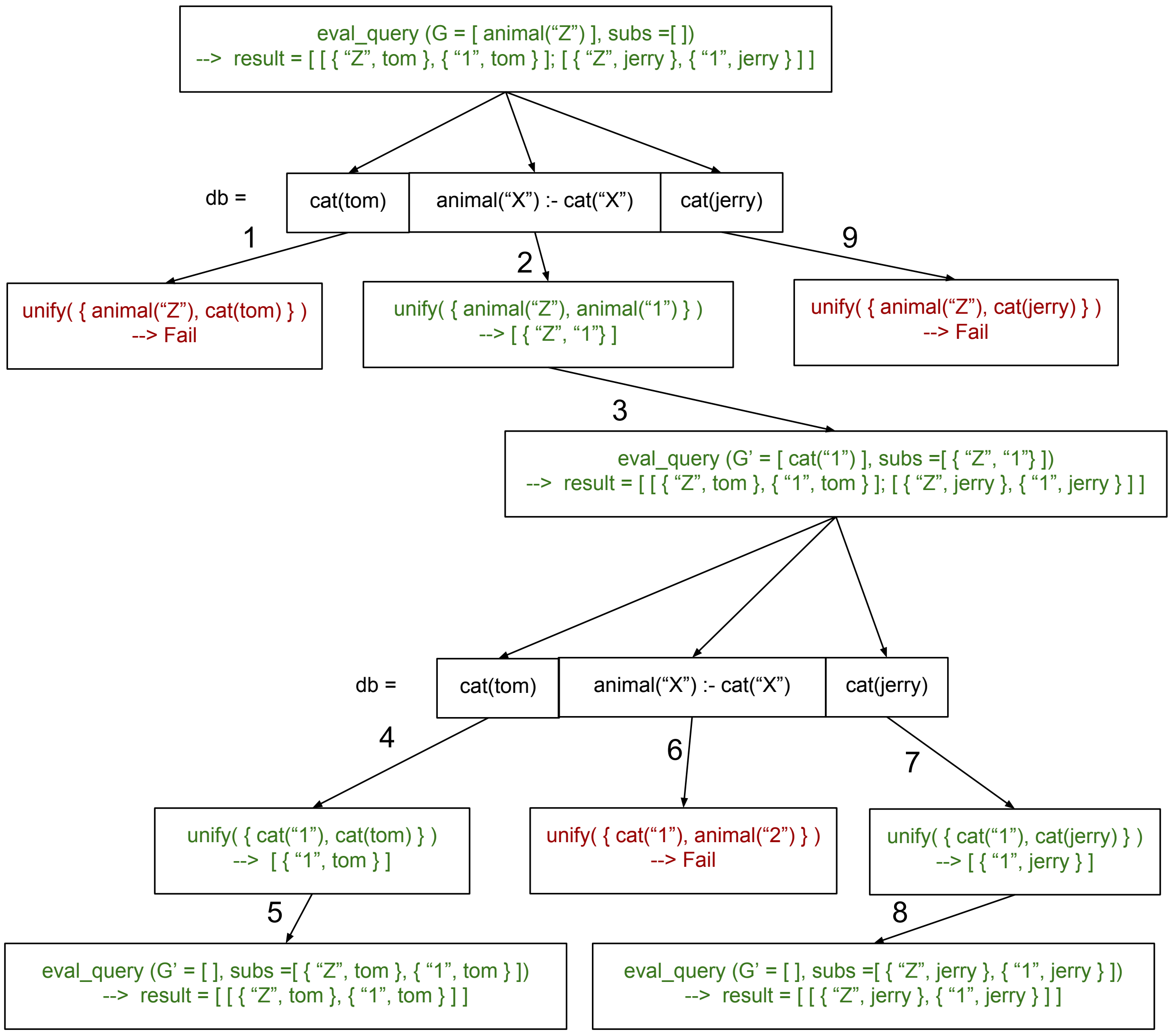
The `eval_query` function finds answers to queries in a depth-first fashion as it always recurses after a fact or a rule matches the current leftmost subgoal `g1`. When that call returns because either `G'` was proven or disproven then it continues on to the next fact or rule in the database. Backtracking is inherently handled as the leftmost subgoal `g1` is always matched against all rules and facts in the database and if, after checking against each element of the database, the subgoal `g1` can not be proven that partial candidate is abandoned. When the iteration over the database is done, only the possible results for a goal `G` will be present in `results`.

Although not shown in the pseudocode, when we pick a clause out of the database, we rename all variables occurring in the clause to fresh variable names. This avoids a mess with variable bindings when the same clause is possibly picked again for evaluating the query.

Below is an example Prolog program and its resulting query evaluation tree. The only unification shown is the one used to match the subgoal against rules and facts from the database. Variables are represented in between double quotes (i.e. `"Z"`, `"1"`, `"X"`, `"2"`). Variable renaming is shown in the two cases when the rule for `animal` is selected from the database for unification and `"X"` is renamed to `"1"` and `"2"`. The result for each `eval_query` node in the tree contains all the results from all subtrees of that node. In the black font is the database, in red font are the calls that failed, and in the green font are the calls that were successful. The numbers on the edges represent the order in which the nodes are visited.

```
cat(tom).
animal(X) :- cat(X).
cat(jerry).

?- animal(Z).
```


eval_query (G = [animal("Z")], subs =[ ])
--> result = [[{ "Z", tom }, { "1", tom }]; [{ "Z", jerry }, { "1", jerry }]]
db =
cat(tom)
animal("X") :- cat("X")
cat(jerry)
1
2
9
unify({ animal("Z"), cat(tom) })
--> Fail
unify({ animal("Z"), animal("1") })
--> [{ "Z", "1"}]
unify({ animal("Z"), cat(jerry) })
--> Fail
3
eval_query (G' = [cat("1")], subs =[{ "Z", "1"}])
--> result = [[{ "Z", tom }, { "1", tom }]; [{ "Z", jerry }, { "1", jerry }]]
db =
cat(tom)
animal("X") :- cat("X")
cat(jerry)
4
6
7
unify({ cat("1"), cat(tom) })
--> [{ "1", tom }]
unify({ cat("1"), animal("2") })
--> Fail
unify({ cat("1"), cat(jerry) })
--> [{ "1", jerry }]
5
8
eval_query (G' = [ ], subs =[{ "Z", tom }, { "1", tom }])
--> result = [[{ "Z", tom }, { "1", tom }]]
eval_query (G' = [ ], subs =[{ "Z", jerry }, { "1", jerry }])
--> result = [[{ "Z", jerry }, { "1", jerry }]]

Printing Query Results

Since the evaluator returns a database to the interpreter, the evaluator needs to print the results of the query before returning. If the results are empty the evaluator prints `false` for the user and returns. If there is at least 1 item in the results the evaluator prints all of the bindings for the variables from the user's query and then prints `true` and returns. For each result in `result`, for each variable in the user's query, the result is checked for a binding for that variable. If the binding is to another variable or there is no binding then that variable is free and the user gets told that the variable is free. Otherwise, the binding is printed.

The Unification Algorithm

Unification is at the center of the query evaluation algorithm. It is used to match a rule or fact from the database to a subgoal to see if that rule or fact can be used to prove the subgoal. It is also used to update the substitutions to use for the `eval_query` recursive call when a rule or fact from the database has matched the subgoal. The algorithm is mostly the same as the one that was presented in lecture 16 and we implemented for ML4 during the semester, except for a few differences.

In our case `VarExp` represents a variable, `TermExp` represents a functor or atom, and `ConstExp` represents a constant integer, float, or string. As such we needed to add an orient case for the situation when there is a constraint `(s, t)` where `s` is a `ConstExp` and `t` is a `VarExp`, a fail case for the situation when there is a constraint `(s, t)` where `s` is a `TermExp` and `t` is a `ConstExp`, and a fail case for the situation when there is a constraint `(s, t)` where `s` is a `ConstExp` and `t` is a `ConstExp` and `s != t`.

The modified unification algorithm psuedocode is listed here (inspired by the algorithm that was presented in lecture 16):

```
let S = {(s1, t1), (s2, t2), ... , (sn, tn)} be a set of constraints

case S = {}; unify(S) = []

case S = {(s, t)} ∪ S':
  Delete
     if s = t
     then unify(S) = unify(S')
     else Fail
  Decompose
     if s = TermExp(f, [q1, ... , qm]) and t = TermExp(f, [r1, ... , rm])
     then unify(S) = unify({(q1, r1), ... , (qm, rm)} ∪ S')
     else
        if s = TermExp(f, [q1, ... , qm]) and t = ConstExp(c)
        then Fail
  Orient
     if t = VarExp(x) and (s = TermExp(f, [q1, ... , qm]) or s = ConstExp(c))
     then unify(S) = unify({(t, s)} ∪ S')
  Eliminate
     if s = VarExp(x) and s does not occur in t
     then
       let sub = {s -> t};
       let S'' = sub(S');
       let phi = unify(S'');
       unify(S) = {s -> phi(t)} o phi
  Extra Fail Case
     if s = ConstExp(c) and t = ConstExp(d) and s != t
     then Fail

  All other cases cause unify to Fail.
```

2.2.6 Interpreter

The interpreter, the front-end program in `main.ml`, is derived from the `picomlInterp.ml` file given to us for MP5 during the semester. It essentially loops until the lexer reaches `EOF` and raises the `Lexer.EndInput` exception. The loop function takes in a list of declarations which is the database that will be used to evaluate whatever declaration the user inputs. The database starts out empty at the start of the interpreter. For each iteration of the loop, a lexbuf is created from user input to standard input, which is then passed into the parser to get the AST representation of the input. The AST representation of the input and the database are passed into the evaluator to evaluate the input and return a, possibly updated, database which is passed into a recursive call of the loop. If there are any exceptions raised by the lexer, parser, or evaluator, a message is printed for the user and the loop is recursively called with the same database that was passed in. The loop ends only when the lexer sees `EOF`.

2.3 Status

After thorough testing, we believe our components like the lexer, parser, and evaluator fully implement all of the grammar we mentioned in our proposal.

Although we mentioned in our project proposal that we wanted to support interpretation from both files and a command-line interpreter, our implementation does not support files. We decided to focus on the interpreter for this project.

When we asked for an extension, Professor Gunter suggested we leave strings and numbers out of the implementation unless we had enough time. We implemented string, int, and float constants as well in all components, although we do not support binary operations on these types. Our proposed grammar did not include binary operations on these types.

Major Prolog implementations implement disjunction between subgoals along with conjunction. Implementing disjunction would have significantly complicated our implementation so we did not implement it. Also, the grammar we proposed did not include disjunction between subgoals.

Also, we do not implement Prolog's unification operator = as well as any other built-in predicate besides the `true` predicate. Prolog list types are not implemented either. Again, our proposed grammar did not include these elements. In our implementation, strings and atoms can't be unified, but in major Prolog implementations they can be if they are the same sequence of characters.

One feature we had implemented in the evaluator but later took out was prompting the user after finding a result in query evaluation to see if the user wanted more results. We had implemented this but as this feature requires user interaction, it became very difficult to write unit tests for. This feature is present in all of the major Prolog implementations as it can help avoid a lot of evaluation if the user already got the answer they were looking for. We decided it was better to be able to test the evaluator thoroughly with unit tests than to have this feature so we removed it.

CHAPTER 3

Tests

When building a complex project like an interpreter, stability is crucial. Our Prolog implementation comes with a wide variety of tests to detect and prevent bugs.

3.1 Unit Tests

In order to write unit tests, we used the OUnit unit test framework for OCaml. OUnit provides several convenience functions for writing simple test suites. It also provides useful command-line arguments to enable more specific testing requests.

All unit tests can be found in the `tests` directory. `test.ml` is the main testing file, which ties together all of the other testing files. Each `<file>` in source has a corresponding testing suite: `<file>_test.ml`. For example, tests for the lexer can be found in `lexer_test.ml`.

Not only do our unit tests ensure that each component of the project is working correctly, they also ensure that inputs that are supposed to fail do in fact fail. In `lexer_test.ml` and `parser_test.ml`, there are two test suites: one for test cases that are expected to pass and another for test cases that are expected to fail. This is important to ensure that things like nested comments and term lists work correctly. The `evaluator_test.ml` file has only one test suite that tests each function in `evaluator.ml` with inputs that are supposed to succeed and fail even if those inputs will not be generated by the parser.

In order to build and run the unit tests, simply run:

```
$ make test
```

This will run the full test suite. If you only want to run a subset of the tests, you can see a list of available tests by running:

```
$ ./test.byte -list-test
```

If you only want to run the lexer tests, for example, you would run:

```
$ ./test.byte -only-test suite:1:Lexer
```

For a full list of options, run:

```
$ ./test.byte -help
```

3.2 Documentation Tests

Our documentation is built using Sphinx and hosted on Read the Docs. In order to ensure that our documentation doesn't contain any broken links, we run regular tests on our documentation. `sphinx-build` has a `-W` flag that turns warnings into errors. The documentation can be built using this flag like so:

```
$ make docs
```

`sphinx-build` will crash and report an error message if it encounters any problems.

Note: Sphinx, sphinx_rtd_theme, and Graphviz are required in order to build the documentation. If you want to build a PDF, LaTeX is also required.

3.3 Continuous Integration

Each commit to the GitHub repository can introduce new features, but it can also introduce bugs. In order to prevent these bugs from creeping in, we use Travis CI for continuous integration testing. Travis runs our unit and documentation tests after each commit, and reports any problems to the developers when the build crashes. It runs these tests using the five latest versions of OCaml (4.02 - 4.06) to make sure that our Prolog interpreter works with any modern version of OCaml.

Travis does not have official OCaml support, so we had to manually install each version of OCaml in our CI script. Our Travis configuration was partially inspired by the ocaml-ci-scripts repository.

To view the results of the latest build, see https://travis-ci.org/adamjstewart/prolog. For our Travis configuration, see `.travis.yml`.

3.4 Coverage

Unit tests are useless if they don't actually test the function containing a bug. We used bisect_ppx to get an accurate measurement of what percentage of our code base was actually covered by unit tests. By integrating our unit tests with bisect_ppx, we can generate coverage reports that can be viewed through a web browser to see exactly which lines were hit. This was extremely beneficial when testing the lexer and parser, as it told us exactly which match cases were being missed. Before we started using bisect_ppx, we were getting around 65% coverage. With the help of bisect_ppx, we were able to attain 93% coverage. The remaining 7% is really obscure corner cases that can arise in the lexer and parser, causing errors to occur.

After each successful build, Travis uploads our coverage reports to Coveralls. To view our coverage reports, including which lines are not yet covered by unit tests, see https://coveralls.io/github/adamjstewart/prolog?branch=master.

The reports sent to Coveralls are helpful, but it isn't possible to view coverage for generated files like `lexer.ml` and `parser.ml`. In order to view coverage for these files, you can generate coverage reports locally like so:

```
$ make coverage
```

This will automatically open up the coverage reports in your default web browser.

CHAPTER 4

Listing

Here we provide a listing of our code.

4.1 ast.ml

This file contains the types for representing our Abstract Syntax Tree.

```
(* Constants *)
type const =
    | IntConst of int          (* integers *)
    | FloatConst of float      (* floats   *)
    | StringConst of string    (* strings  *)

(* Expressions *)
type exp =
    | VarExp of string                (* variables                 *)
    | ConstExp of const               (* constants                 *)
    | TermExp of string * exp list    (* functor(arg1, arg2, ...) *)

(* Declarations *)
type dec =
    | Clause of exp * (exp list)   (* Head :- Body. *)
    | Query of (exp list)          (* ?- Body.      *)
```

4.2 common.ml

This file contains common functions needed throughout the program.

```
open Ast
open Parser
```

```
(* Takes a string s and returns the abstract syntax tree generated by lexing and␣
↪parsing s *)
let parse s =
    let lexbuf = Lexing.from_string s in
        let ast = clause Lexer.token lexbuf in
            ast

(* Takes a string s and returns Some of an abstract syntax tree or None *)
let try_parse s =
    try Some (parse s) with
    | Error -> None

(* String conversion functions *)
let string_of_token t =
    match t with
    | INT    i -> "INT "       ^ string_of_int i
    | FLOAT  f -> "FLOAT "     ^ string_of_float f
    | STRING s -> "STRING \"" ^ String.escaped s ^ "\""
    | ATOM   a -> "ATOM \""   ^ String.escaped a ^ "\""
    | VAR    v -> "VAR \""    ^ v ^ "\""
    | RULE     -> "RULE"
    | QUERY    -> "QUERY"
    | PERIOD   -> "PERIOD"
    | LPAREN   -> "LPAREN"
    | RPAREN   -> "RPAREN"
    | COMMA    -> "COMMA"
    | EOF      -> "EOF"

let string_of_token_list tl =
    "[" ^ (String.concat "; " (List.map string_of_token tl)) ^ "]"

let string_of_const c =
    match c with
    | IntConst    i -> "IntConst "       ^ string_of_int i
    | FloatConst  f -> "FloatConst "     ^ string_of_float f
    | StringConst s -> "StringConst \"" ^ String.escaped s ^ "\""

let rec string_of_exp e =
    match e with
    | VarExp v   -> "VarExp \"" ^ v ^ "\""
    | ConstExp c -> "ConstExp (" ^ (string_of_const c) ^ ")"
    | TermExp (f, args) ->
        let func = String.escaped f in
            "TermExp (\"" ^ func ^ "\", [" ^
                (String.concat "; " (List.map string_of_exp args)) ^ "])"


let string_of_exp_list g =
    "[" ^ (String.concat "; " (List.map string_of_exp g)) ^ "]"

let string_of_dec d =
    match d with
    | Clause (e1, g) ->
        "Clause (" ^ (string_of_exp e1) ^ ", " ^
            (string_of_exp_list g) ^ ")"
    | Query g -> "Query (" ^ (string_of_exp_list g) ^ ")"
```

```
let string_of_db db =
    "[" ^ (String.concat "; " (List.map string_of_dec db)) ^ "]"


let string_of_subs s =
    "[" ^ (
        String.concat "; " (
            List.map (
                fun (x,y) ->
                    "(" ^ (string_of_exp x) ^ ", " ^ (string_of_exp y) ^ ")"
            )
            s
        )
    ) ^ "]"

let string_of_unify_res s =
    match s with
    | None    -> "None"
    | Some l -> string_of_subs l

(* Convert ConstExp to a readable string *)
let readable_string_of_const c =
    match c with
    | IntConst    i -> string_of_int i
    | FloatConst  f -> string_of_float f
    | StringConst s -> "\"" ^ String.escaped s ^ "\""

(* Convert exp to a readable string *)
let rec readable_string_of_exp e =
    match e with
    | VarExp   v      -> v
    | ConstExp c      -> readable_string_of_const c
    | TermExp (s, l)  ->
        s ^ (
            if List.length l > 0
            then "(" ^ (
                String.concat ", " (
                    List.map readable_string_of_exp l
                )
            ) ^ ")"
            else ""
        )

(* Print a db *)
let print_db db =
    print_endline (string_of_db db)
```

4.3 evaluator.ml

This file contains the code needed to evaluate our expressions.

```
open Ast
open Common
```

```
(*
   fresh:
      * takes in:
          unit
      * returns a string of the increment of the counter
   reset:
      * takes in:
          unit
      * returns unit and the counter is reset
*)
let (fresh, reset) =
    let nxt = ref 0 in
    let f () = (nxt := !nxt + 1; string_of_int (!nxt)) in
    let r () = nxt := 0 in
    (f, r)

(*
   find_vars:
     * takes in:
         q - a list of exp
     * returns a list of all VarExp in the list
*)
let rec find_vars q  =
    match q with
    | [] -> []
    | (x :: xs) -> (
        match x with
        | VarExp v -> x :: (find_vars xs)
        | ConstExp c -> (find_vars xs)
        | TermExp (s, el) -> (find_vars el) @ (find_vars xs)
    )

(*
   uniq:
     * takes in:
         l - a list
     * returns the list reversed with only one copy of each element
  adapted from https://rosettacode.org/wiki/Remove_duplicate_elements#OCaml
*)
let uniq l =
    let rec tail_uniq a l =
        match l with
        | [] -> a
        | hd :: tl ->
            tail_uniq (hd :: a) (List.filter (fun x -> (x <> hd) ) tl) in
            tail_uniq [] l

(*
   sub_lift_goal:
     * takes in:
         sub - a list of substitutions for variables
         g - a goal of type exp
     * returns the goal with the substitutions applied
*)
let rec sub_lift_goal sub g =
    match g with
    | VarExp v -> (
        (* if this variable has a substitution do the substitution *)
        try let i = List.assoc g sub in i
        with Not_found -> VarExp v
    )
    | TermExp (s, el) ->
        TermExp (s, List.map (fun g1 -> sub_lift_goal sub g1) el)
    | _  -> g

(*
   sub_lift_goal:
     * takes in:
         sub - a list of substitutions for variables
         gl - a list of goals each of type exp
     * returns the list of goals with the substitutions applied to each goal
*)
let sub_lift_goals sub gl =
    List.map (fun g1 -> sub_lift_goal sub g1) gl

(*
   rename_vars_in_dec:
     * takes in:
         d - a dec type
     * returns a dec with all the variables in d renamed to fresh variable names
*)
let rec rename_vars_in_dec d =
    match d with
    | Clause (h, b) ->
        let head_vars = find_vars [h] in
        let body_vars = find_vars b in
        (* find uniq vars from both head and body *)
        let vars = uniq (head_vars @ body_vars) in
        (* get fresh variable mappings *)
        let sub = List.map (fun x -> (x, VarExp (fresh()))) vars in
        (* substitute new names for variables *)
        Clause (sub_lift_goal sub h, sub_lift_goals sub b)
    | Query (b) ->
        (* find uniq vars in query *)
        let body_vars = find_vars b in
        (* get fresh variable mappings *)
        let vars = uniq (body_vars) in
        let sub = List.map (fun x -> (x, VarExp (fresh()))) vars in
        (* substitute new names for variables *)
        Query (sub_lift_goals sub b)

(*
   pairandcat:
     * takes in:
         sargs - a list of exps
         targs - a list of exps
         c - a list of constraints where each constraint is of type (exp * exp)
     * returns a new list of constraints where c is prepended with each entry
       from sargs is paired with a corresponding entry from targs
       to make a new constraint
     * used for implementing decompose for unification
     * sargs and targs must be the same length, otherwise an exception is thrown
*)
let rec pairandcat sargs targs c =
    match sargs with
    | [] -> (
        if (List.length targs = 0)
        then c
        else raise (Failure "sargs and targs should be the same length")
    )
    | (s :: ss) -> (
        match targs with
        | (t :: ts) -> pairandcat ss ts ((s, t) :: c)
        |  _ -> raise (Failure "sargs and targs should be the same length")
    )

(*
   replace:
     * takes in:
         c - a list of constraints where each constraint is of type (exp * exp)
         sub - a list of substitutions
     * returns a new list of constraints where the substitutions are applied to
       both sides of each constraint
*)
let rec replace c sub =
    match c with
    | [] -> []
    | ((s, t) :: xs) ->
        (sub_lift_goal sub s, sub_lift_goal sub t) :: (replace xs sub)

(*
   occurs:
     * takes in:
         n - a string
         t - an exp
     * returns true if n matches any variable names in t and false otherwise
*)
let rec occurs n t =
     match t with
     | VarExp m -> n = m
     | TermExp (st, el) ->
       List.fold_left (fun acc v -> acc || (occurs n v)) false el
     | _ -> false

(*
   unify:
     * takes in:
         constraints - a list of constraints where each constraint
         is of type (exp * exp)
     * returns None if the constraints can't be unified,
       otherwise returns Some(i) where i is a list of substitutions
       that unify the constraints
*)
let rec unify constraints =
    match constraints with
    | [] -> Some []
    | ((s, t) :: c') ->
        if s = t
        then unify c'  (* Delete *)
        else (
            match s with
            | VarExp(n) ->
                (* Eliminate *)
                if (occurs n t)
                then None
                else let sub = [(s,t)] in
                     let c'' = replace c' sub in
                     let phi = unify c'' in (
                        match phi with
                        | None -> None
                        | Some l -> Some ((s, sub_lift_goal l t) :: l)
                     )
            | TermExp (sname, sargs) -> (
                match t with
                (* Orient *)
                | VarExp k -> unify ((t, s) :: c')
                (* Decompose *)
                | TermExp (tname, targs) ->
                    if (tname = sname && List.length targs = List.length sargs)
                    then unify (pairandcat sargs targs c')
                    else None
                | _ -> None
            )
            | _ -> (
                match t with
                (* Orient *)
                | VarExp k -> unify ((t, s) :: c')
                | _ -> None
            )
        )

(*
   eval_query:
     * takes in (all in a triple):
         q - a list of exp
         db - a list of dec
         env - a list of substitutions
               where each substitution is of type (exp * exp)
     * returns a list of lists of substitutions where each
       substitution is of type (exp * exp)
         - if the returned list is empty then no solutions were found for the
           query for the given db
         - otherwise, each element is a list of substitutions for one solution
           to the query with the given db
*)
let rec eval_query (q, db, env) =
    match q with
    | [] -> (
        (* no more subgoals to prove so finished *)
        [env]
    )
    | (g1 :: gl) -> (  (* have at least one more subgoal (g1) to prove *)
        match g1 with
        (* if goal is the true predicate *)
        | TermExp("true", []) -> (
          eval_query (
              gl,
              db,
              env
            )
        )
        (* if goal is some other predicate *)
        | TermExp(_,_) -> (
        (* iterate over the db *)
        List.fold_right (
            fun rule r -> (
                match (rename_vars_in_dec rule) with (* rename vars in rule *)
                | Clause (h, b) -> (
                    (* check if this rule can be used for this subgoal *)
                    match unify [(g1, h)] with
                    | Some s -> (
                        match unify (s@env) with
                        | Some env2 -> (
                            if (List.length b = 1)
                            then (
                                match b with
                                (* if the rule proved the subgoal (ie. rule was a
                                   fact) then recurse on remaining subgoals *)
                                | ((TermExp ("true", _)) :: ys) ->
                                    ((eval_query (
                                        sub_lift_goals s gl,
                                        db,
                                        env2
                                     )) @ r)
                                (* if rule wasn't a fact then we have more
                                   subgoals from the body of the rule
                                   to prove *)
                                | _ -> ((eval_query (
                                    (sub_lift_goals s b) @ (sub_lift_goals s gl),
                                    db,
                                    env2
                                )) @ r))
                            else
                                (* if rule wasn't a fact then we have more
                                   subgoals from the body of the rule
                                   to prove *)
                                ((eval_query (
                                    (sub_lift_goals s b) @ (sub_lift_goals s gl),
                                    db,
                                    env2
                                )) @ r)
                        )
                        (* the substitution from unify the rule head and subgoal
                           doesn't unify with the environment gathered so far *)
                        | _ -> r
                    )
                    (* this rule's head doesn't unify with the subgoal *)
                    | _ -> r
                )
                (* found a dec in the db that isn't a Clause *)
                |  _ -> r
        )) db [] )
        (* subgoal isn't a TermExp *)
        | _ -> eval_query (gl, db, env)
    )

(*
   string_of_res:
     * takes in:
        e - a list of lists of substitutions where each
```

```
             substitution is of type (exp * exp)
         orig_query_vars - a list of uniq VarExps that appeared
                           in the original query
         orig_vars_num - number of uniq VarExps that appeared
                         in the original query
     * returns a string consisting of all substitutions of variables
       appearing in the original query of all solutions found and the
       word true if solution(s) were found and false otherwise
*)
let string_of_res e orig_query_vars orig_vars_num =
   (* iterate over e for each solution *)
   List.fold_left (
        fun r2 env ->
        if orig_vars_num > 0
        then
          "====================\n" ^
            (* iterate over original query vars to find their substitution *)
            (List.fold_left (
                 fun r d -> (
                   match d with
                   | VarExp v -> (
                     (* find variable substitution in the solution *)
                     try let f = List.assoc (VarExp v) env in (
                             match f with
                             | VarExp v2 ->
                                (v ^ " is free\n") ^ r
                             | _ ->
                                (v ^ " = " ^ (
                                   readable_string_of_exp f) ^ "\n") ^ r
                         )
                     with Not_found -> (v ^ " is free\n") ^ r)
                   | _ -> r
                 )
             ) "" (orig_query_vars) ) ^
            "====================\n"  ^ r2
        else "" ^ r2
     )  (if List.length e > 0 (* if e is empty then there were no solutions *)
         then "true\n"
         else "false\n")
                  e

(*
   add_dec_to_db:
     * takes in (all in a tuple):
         dec - a dec type
         db - a list of dec types
     * returns db prepended with dec if dec is not
       of the pattern TermExp("true",_) as we don't want users to be able
       to redefine the "true" atom
       otherwise, db is returned
*)
let add_dec_to_db (dec, db) =
    match dec with
    | Clause (h, b) -> (
        match h with
        (* don't allow user to add a new definition of true *)
        | TermExp ("true", _) ->
            print_string "Can't reassign true predicate\n"; db
        | _ -> dec :: db
    )
    | Query (b) -> (
        dec :: db
    )

(*
   eval_dec:
     * takes in (all in a tuple):
         dec - a dec type
         db - a list of dec types
     * evaluated the dec with the given db
       returns the original db in the case
       dec is a Query type
       otherwise returns db prepended with dec
*)
let eval_dec (dec, db) =
    match dec with
    | Clause (h, b) -> add_dec_to_db (dec, db)
    | Query b -> (
        (* find all uniq VarExps in query *)
        let orig_vars = uniq (find_vars b) in
        (* find num of VarExps in query *)
        let orig_vars_num = List.length orig_vars in
        (* evaluate query *)
        let res = eval_query (b, db, []) in
        (* print the result *)
        print_string (string_of_res (res) orig_vars orig_vars_num);
        (* reset fresh variable counter *)
        reset ();
        db
    )
```

4.4 lexer.mll

This file defines the lexer.

```
{
    open Parser


    exception EndInput
}

(* Refer to:

   https://www.cs.uni-potsdam.de/wv/lehre/Material/Prolog/Eclipse-Doc/userman/node139.
↪html
   http://www.amzi.com/manuals/amzi/pro/ref_terms.htm

   for a list of valid tokens *)

(* Character classes *)
let upper_case = ['A' - 'Z']
let underline = '_'
let lower_case = ['a' - 'z']
let digit = ['0' - '9']
let blank_space = [' ' '\t']
let end_of_line = '\n'
let atom_quote = '''
let string_quote = '"'
let line_comment = '%' [^ '\n'] *
let open_comment = "/*"
let close_comment = "*/"
let escape = '\\'

(* Groups of characters *)
let alphanumerical = upper_case | underline | lower_case | digit
let any_character = [' ' - '~']
let non_escape = any_character # ['\\']
let sign = ['+' '-']

(* Atoms *)
let atom = lower_case alphanumerical *
let octal = ['0' - '7']
let octals = octal octal octal
let hex = ['0' - '9' 'A' - 'F' 'a' - 'f']
let hexes = 'x' hex + escape

(* Numbers *)
let digits = digit +
let integers = sign ? digits
let floats =
      integers '.' digits (['e' 'E'] sign ? digits) ?
    | integers ['e' 'E'] sign ? digits
let inf = '+' ? digits '.' digits "Inf"
let neg_inf = '-' digits '.' digits "Inf"

(* Variables *)
let variable = (upper_case | underline) alphanumerical *

rule token = parse
    (* Meta-characters *)
    | [' ' '\t' '\n']    { token lexbuf }
    | eof                { EOF }

    (* Comments *)
    | line_comment       { token lexbuf }
    | open_comment       { comments 1 lexbuf }
    | close_comment      { raise (Failure "unmatched closed comment") }

    (* Atoms *)
    | atom as a          { ATOM a }
    | atom_quote         { atoms "" lexbuf }

    (* Numbers *)
    | integers as n      { INT   (int_of_string n)    }
    | floats   as f      { FLOAT (float_of_string f) }
    | inf                { FLOAT infinity            }
    | neg_inf            { FLOAT neg_infinity        }

    (* Strings *)
    | string_quote       { strings "" lexbuf }
    (* Variables *)
    | variable as v     { VAR v }

    (* Symbols *)
    | ":-"              { RULE      }
    | "?-"              { QUERY     }
    | '.'               { PERIOD    }
    | '('               { LPAREN    }
    | ')'               { RPAREN    }
    | ','               { COMMA     }

and comments count = parse
    | open_comment      { comments (1 + count) lexbuf }
    | close_comment     { match count with
                          | 1 -> token lexbuf
                          | n -> comments (n - 1) lexbuf
                        }
    | eof               { raise (Failure "unmatched open comment") }
    | _                 { comments count lexbuf }

and strings acc = parse
    (* Consecutive strings are concatenated into a single string *)
    | string_quote blank_space * string_quote   { strings acc lexbuf }
    | string_quote                              { STRING acc }
    | non_escape # ['"'] + as s                 { strings (acc ^ s) lexbuf }
    | escape                                    { escaped strings acc lexbuf }

and atoms acc = parse
    | atom_quote                    { ATOM acc }
    | non_escape # ['''] + as a     { atoms (acc ^ a) lexbuf }
    | escape                        { escaped atoms acc lexbuf }

and escaped callback acc = parse
    | 'a'           { callback (acc ^ (String.make 1 (char_of_int   7))) lexbuf }
    | 'b'           { callback (acc ^ (String.make 1 (char_of_int   8))) lexbuf }
    | 'f'           { callback (acc ^ (String.make 1 (char_of_int  12))) lexbuf }
    | 'n'           { callback (acc ^ (String.make 1 (char_of_int  10))) lexbuf }
    | 'r'           { callback (acc ^ (String.make 1 (char_of_int  13))) lexbuf }
    | 't'           { callback (acc ^ (String.make 1 (char_of_int   9))) lexbuf }
    | 'v'           { callback (acc ^ (String.make 1 (char_of_int  11))) lexbuf }
    | 'e'           { callback (acc ^ (String.make 1 (char_of_int  27))) lexbuf }
    | 'd'           { callback (acc ^ (String.make 1 (char_of_int 127))) lexbuf }
    | escape        { callback (acc ^ "\\") lexbuf }
    | atom_quote    { callback (acc ^  "'") lexbuf }
    | string_quote  { callback (acc ^ "\"") lexbuf }
    | end_of_line   { callback acc lexbuf }
    | 'c' (blank_space | end_of_line) *      { callback acc lexbuf }
    | octals as o   { callback (acc ^ (String.make 1 (char_of_int (
                        int_of_string ("0o" ^ o))))) lexbuf }
    | hexes as h    { callback (acc ^ (String.make 1 (char_of_int (
                        int_of_string ("0" ^ (String.sub h 0 (
                            (String.length h) - 1))))))) lexbuf }

{
    (* Takes a string s and returns a list of tokens generated by lexing s *)
    let get_all_tokens s =
        let b = Lexing.from_string (s ^ "\n") in
```

```
            let rec g () =
                match token b with
                | EOF -> []
                | t -> t :: g () in
                  g ()

    (* Takes a string s and returns Some of a list of tokens or None *)
    let try_get_all_tokens s =
        try Some (get_all_tokens s) with
        | Failure _ -> None
}
```

4.5 main.ml

This file contains the main interpreter program.

```
open Ast
open Common
open Lexer
open Parser
open Evaluator


(* Try to detect if something is getting piped in *)
let is_interactive = 0 = (Sys.command "[ -t 0 ]")

let _ =
    (if is_interactive
     then print_endline "\nWelcome to the Prolog Interpreter\n"
     else ()
    );
    let rec loop db = (
        try (
            let lexbuf = Lexing.from_channel stdin
            in (
                if is_interactive
                then (print_string "> "; flush stdout)
                else ()
            );
            let dec = clause (
                fun lb -> (
                    match Lexer.token lb with
                    | EOF -> raise Lexer.EndInput
                    | r -> r
                )
            ) lexbuf
            (* evaluate dec and get a new db *)
            in let newdb = eval_dec (dec,db)
            in loop newdb (* loop again with new db *)
        )
        (* exception raised *)
        with
        | Failure s -> ( (* in case of an error *)
            print_newline();
            print_endline s;
            print_newline();
            loop db
        )
        | Parser.Error -> ( (* failed to parse input *)
            print_string "\nDoes not parse\n";
            loop db
        )
        | Lexer.EndInput -> exit 0 (* EOF *)
        | _ -> ( (* Any other error *)
            print_string "\nUnrecognized error\n";
            loop db
        )
    )
    in (loop [])
```

4.6 parser.mly

This file defines the parser.

```
%{
    open Ast

    (* The fact:
           Head.
       is equivalent to the rule:
           Head :- true.
    *)
    let fact_sugar p =
        Clause (
            p,
            [TermExp ("true", [])]
        )

    (* An atom can be regarded as a compound term with arity zero *)
    let atom_sugar a =
        TermExp (
            a,
            []
        )
%}

/* Refer to:

   https://www.cs.uni-potsdam.de/wv/lehre/Material/Prolog/Eclipse-Doc/userman/node140.
↪html
   https://github.com/simonkrenger/ch.bfh.bti7064.w2013.PrologParser/blob/master/doc/
↪prolog-bnf-grammar.txt

   for a description of the grammar */

/* Tokens */

/* Constants */
%token <int> INT
%token <float> FLOAT
```

```
%token <string> STRING ATOM

/* Variables */
%token <string> VAR

/* Symbols */
%token RULE        /* :- */
%token QUERY       /* ?- */
%token PERIOD      /* .  */
%token LPAREN      /* (  */
%token RPAREN      /* )  */
%token COMMA       /* ,  */

/* Meta-characters */
%token EOF

/* Start symbols */
%start clause

/* Types */
%type <Ast.dec> clause
%type <Ast.exp> predicate term structure
%type <Ast.exp list> term_list predicate_list
%type <Ast.const> constant

%%

clause:
    | p = predicate; PERIOD                              { fact_sugar p }
    | p = predicate; RULE; pl = predicate_list; PERIOD  { Clause (p, pl) }
    | QUERY; pl = predicate_list; PERIOD                { Query pl }

predicate_list:
    | p = predicate                                      { [p] }
    | p = predicate; COMMA; pl = predicate_list          { p :: pl }

predicate:
    | a = ATOM                                           { atom_sugar a }
    | s = structure                                      { s }

structure:
    | a = ATOM; LPAREN; RPAREN                           { atom_sugar a }
    | a = ATOM; LPAREN; tl = term_list; RPAREN           { TermExp (a, tl) }

term_list:
    | t = term                                           { [t] }
    | t = term; COMMA; tl = term_list                    { t :: tl }

term:
    | c = constant                                       { ConstExp c }
    | a = ATOM                                           { atom_sugar a }
    | v = VAR                                            { VarExp v }
    | s = structure                                      { s }

constant:
    | i = INT                                            { IntConst i }
    | f = FLOAT                                          { FloatConst f }
    | s = STRING                                         { StringConst s }
```