(LINFO2335 — Assignment Module 1) Logic Programming Mission: Implementing an SQL Interpreter in Prolog

K. Mens, J. Liénard, O. Goletti

February 2024

Abstract

For this mission, as an exercice on logic programming, you will have to implement an SQL-like query language, using Prolog as your implementation language. As you undoubtedly know, SQL is a standard query language for accessing and manipulating relational databases. Since both SQL and Prolog are query languages, it doesn't require too much of a paradigm shift to implement SQL in Prolog. If your knowledge of SQL is a bit rusty or non-existent, don't worry, we will briefly revisit the language and precisely specify the operations that you need to implement.

Introduction to SQL

Before getting started, this section provides a brief introduction to SQL. For a more detailed introduction as well as additional examples you can consult this SQL Tutorial. Or you can use this alternative introduction to SQL with simple, interactive exercises.

SQL, which stands for "Structured Query Language" is used to define and manipulate relational databases. In a way, relational databases and SQL queries are somewhat similar to Prolog (even though Prolog is a general-purpose language and much more powerful). You could implement the data stored in a database as logic facts, and SQL queries which act on that data as predicates in Prolog which act on those facts.

To help you during this project, we will give you some hints on how to use SQL. If you already know the basics of the SQL language, you can directly pass to the next section, even though reading this section as a gentle reminder won't hurt you either. Here are some examples of basic SQL queries that you will need to implement or simulate in Prolog.

CREATE TABLE

`Persons`

Id	FirstName	LastName	Age	City

The query below will `create` a table `Persons` with five columns.

```
CREATE TABLE Persons (
    Id int,
    FirstName varchar(255),
    LastName varchar(255),
    Age int,
    City varchar(255)
);
```

`Id` and `Age` will store an integer and `FirstName`, `LastName` and `City` will contain strings with a maximum size of 255 characters.

In Prolog, we could represent such a table as a set of facts with five arguments and to simplify things we will not specify the type for each column. And to avoid confusion with Prolog variables that start with a capital we will only use lowercase for the names of tables and columns.

```
persons(id,firstname,lastName,age,city)
```

INSERT INTO

Persons

Id	FirstName	LastName	Age	City
1	"John"	"Wick"	42	"London"

Once the table has been created, the `insert` query can be used to populate the database, i.e. to add rows to an existing table with corresponding values for each of its columns. It requires the name of the table and the values that we want to insert.

```
INSERT INTO table_name VALUES (value1, value2, ...);
```

For example, if we want to add something to the table named `Persons`, we could write:

```
INSERT INTO Persons VALUES (1,"John","Wick",42,"London");
```

This would add a new row to the table corresponding to a person with `Id` 1, `Age` 42, `FirstName` “John”, `LastName` “Wick” and `City` “London”. Note that when inserting the values we don’t need to specify the column names. The values are associated to their corresponding columns according to the order of declaration of the columns.

We can also select only a few columns to fill when inserting a new row. (For the other columns, a null value will then be inserted as default value.) In that case the insert query would look like:

```
INSERT INTO table_name (column1, column2) VALUES (value1,
     value2);
```

For example, the following query

```
INSERT INTO Persons (FirstName, LastName, City) VALUES ("
    Carla", "Bruni", "Paris");
```

would add a new row to the table `Persons` corresponding to a person with first name “Carla”, last name “Bruni” and city “Paris”; for the unspecified columns `Id` and `Age`, the default value `null` will be inserted.

Persons

Id	FirstName	LastName	Age	City
1	"John"	"Wick"	42	"London"
null	"Carla"	"Bruni"	null	"Paris"

SELECT

The `select` keyword is by far the most important one in SQL as it is used to retrieve data from tables in the database. Its most common form is:

```
SELECT selectors FROM tables WHERE conditions;
```

Where:

- `selectors` is the name of the columns for which we want to retrieve their values;
- `tables` are the names of the tables in which we want to look up these columns;
- `conditions` are some filtering conditions that can be applied on the selected values.

For example, if we want to find the `LastName` and `FirstName` of each entry in the table `Persons` that is older than 20 years, we have:

```
SELECT LastName, FirstName FROM Persons WHERE Age>20;
```

This query returns a result table of the form

LastName	FirstName
"Wick"	"John"

Mission Statement

Now that you know a bit more about SQL and how to make queries, we will describe the different predicates that we ask you to define, in order to implement an SQL-like language in Prolog. In particular, we ask you to implement the predicates that follow. This will also entail using Definite Clause Grammars (DCGs).

We will use the SWI Prolog notation for describing predicates (`https://www.swi-prolog.org/pldoc/man?section=preddesc`). In particular, parameters that must be instantiated to a term upon calling the predicate are preceded by a +, while parameters that can still be unbound upon calling are preceded with -. Of course, the idea is that the predicate then tries to bind these unbound parameters to actual values.

Do note that this notation is somewhat loose, since, usually, it is possible to pass (partially/totally) instantiated values to - parameters in order to perform a check rather than to bind the variables. Nevertheless, your implementation should support at least the basic case (+ parameters fully instantiated, and - parameters unbound). Additional possibilities are at your own discretion.

Indications in bold below are valid for all predicates to be implemented.

Usage examples for the various predicates can be found in the test file that we supply you (tests.pl). In case of doubt, contact the course assistant.

Required predicate

tables.

Prints the names of all existing tables, one per line (use `writeln/1`).

A table name is always an atom.

---

tables(-Tables).

Unify Tables with a *list* of the names of all existing tables.

---

create_table(+Table, +Cols).

When this predicate is executed, the effect will be the creation of a new table with the specified list of column names (order matters!).

A column name is always an atom.

If a table with the given name already exists, the predicate must throw a descriptive exception (use `throw/1`).

All exceptions must have a descriptive error message.

---

cols(+Table, -Cols).

Unifies Cols with the list of columns for the specified table (in the same order as they were supplied to `create_table/2`).

If the given table does not exist, the predicate must throw a descriptive exception (use `throw/1`).

---

row(+Table, -Row).

Unifies Row, one result at a time, with each row in the given Table.

If the given table does not exist, the predicate should fail.

---

rows(+Table).

Displays all rows in the given table, one per line (use `writeln/1`).

If the given table does not exist, the predicate must throw a descriptive exception.

---

insert(+Table, +Row).

When this predicate is executed, the effect will be the addition of a given row in the given table. The given row is a list of values for each of the corresponding columns in the table (in the order in which the columns were supplied to `create_table/2`).

If the given table does not exist, the predicate must throw a descriptive exception.

If the row does not have as many elements as the number of columns in the table, the predicate must throw a descriptive exception.

---

drop(+Table).

When this predicate is executed, the effect will be the deletion of the given table.

Do make sure that all of its rows are deleted as well, so that they don't magically reappear again if you would recreate a table with the same name and signature later on.

If the given table does not exist, the predicate must throw a descriptive exception.

---

delete(+Table).

When this predicate is executed, the effect will be the deletion of all rows in the given table. The table itself should still exist after, but with no more rows.

If the given table does not exist, the predicate must throw a descriptive exception.

---

delete(+Table, +Conds).

When this predicate is executed, the effect will be the deletion of all rows from the given table that match all of the given conditions. The table must still exist after.

If the given table does not exist, the predicate must throw a descriptive exception.

A condition is any Prolog predicate that could have been typed at the prompt, but which may include selectors. Selectors are terms of the form +<`column`> where <`column`> should be replaced by a column name.

(See tests.pl for some concrete usage examples.)

---

selec(+Table, +Selectors, +Conds, -Projection)

Note that the name of this predicate is `selec` (without t) for the simple reason that `select/4` is already a built-in Prolog predicate.

`Table` is the name of a single table.

`Selectors` is either `*` or a list of selectors. These define the resulting projection. `*` means: select all column names from the table. Other selectors explicitly specify which columns to pick. (See above for what selectors look like.) For example, `+name` would select the column named `name`.

`Conds` has the same form as in `delete/2` and works the same way: only rows that match all conditions are selected.

Finally, `Projection` unifies with <`selectors`>/<`projection`>, where:

- <`selectors`> is the list of requested selectors.
- <`projection`> is a list of values coming from a single row from the given table that matches the conditions. This mean this predicate should be able to backtrack to generate all projections that match the query.

For example, `selec(persons,[+id,+first],[],P)` returns as first result `P = [+id, +first]/[0, "Jeffrey"]`.

To obtain all projections that match the query, one could use the Prolog query `findall(X, selec(Table, Selectors, Conds, X), Projections)`. For example: `findall(X, selec(persons,[+last],[],X), Projections)` returns `Projections = [[+last]/["Bowman"],[+last]/["Michaels"],...]`

Or if you only want the rows (since the selectors are repeated):
`findall(X, selec(Table, Selectors, Conds, _/X), Projections).`

For example:
`findall(Values, selec(persons,[+id,+first],[],Values), Projections)`
returns: `Projections = [[0, "Jeffrey"], [1, "Lorena"], [2, "Joseph"],`
...

---

selec(+TableOrTables, +Selectors, -Projection)

Simplified variant of the `selec/4` predicate when there are no conditions to be checked.

---

query(+Query, -Result)

query(+Query)

where `Query` is a string whose syntax is defined by the following grammar:

⟨*query*⟩	::=	⟨*select*⟩ | ⟨*insert*⟩
⟨*select*⟩	::=	SELECT ⟨*selectors*⟩ FROM ⟨*table*⟩ [⟨*where*⟩];
⟨*selectors*⟩	::=	* | ⟨*cols*⟩
⟨*cols*⟩	::=	⟨*col*⟩ [, ⟨*cols*⟩]
⟨*where*⟩	::=	WHERE ⟨*cond*⟩
⟨*insert*⟩	::=	INSERT INTO ⟨*table*⟩ [(⟨*cols*⟩)] VALUES (⟨*values*⟩);
⟨*values*⟩	::=	⟨*value*⟩ [, ⟨*values*⟩]

About the notation: the pipe (`|`) and square brackets (`[]`) symbols in the production rules above denote choice and optionality, respectively.

<`table`> and <`col`> denote table and column names (respectively). Table names are atoms, while column names should follow the format outlined before (+<`atom`>). <`cond`> denotes a Prolog goal, in Prolog syntax — following the same format as the possible values of items of the `Conds` list passed to `selec` and `delete`.

<`value`> denotes a prolog value that can be stored into a row. You do not need to handle parsing Prolog (<`cond`> and <`value`>) by yourself, we'll show how to do it below.

The semantics of this predicate is that of the SQL-predicate contained in the `Query` string.

A "SELECT" query maps to the `selec` predicate, while an "INSERT" query maps to the `insert` predicate. For "SELECT", all instances of <`cond`> (appearing in <`where`>) are mapped to the `Conds` parameter. <`selectors`> map to the `Selectors` parameter. Tables names (or *) are mapped to the `Table` parameter.

For "INSERT", if the <`cols`> part is absent, the mapping to `insert/2` is straightforward. If <`cols`> is present, you will have to:

1. reorder the values according to the columns that are present;
2. fill in the missing columns (if any) with a null default values.

Example:

```
query("INSERT INTO cities (+name, +state)
VALUES (\"Tempa\", \"Florida\");").
```

maps to (for instance):

```
insert(cities, ["Tempa", "Florida"]).
```

When used with “SELECT”:

- `query/1` must display the results a bit like the `rows` predicate would. You have some flexibility here (displaying the selected column names is a nice touch for instance).
- `query/2` must pass the `Result` parameter as last parameter to the `selec/4` predicate.

For “INSERT”, nothing needs to be printed by `query/1` or in case of success, and if `query/2` is used the `Result` parameter can be ignored.

Getting Started

We supply you with one file called `input.pl` containing a sample database which you can use to test your solution. You are allowed to change the predicates used to store the data (on the other hand, there is no need to — and it's like this for a good reason).

If you want to be able to modify the database, do not forget the proper `:- dynamic` declarations at the top of the input file. The input file already includes these for the predicates we defined. You should also repeat these declarations at the top of your solution file.

We also give you a file called `tests.pl` with a series of unit tests. As we will repeat in the modalities section, your implementation **must** pass all these tests for you to get a passing grade (>= 10) for this mission.

As a starting point, set up the tests. Copy the signatures you must implement in your solution file `solution.pl`, remove the +/- annotations, and make sure all tests fail but that no error is produced. Then, start implementing the predicates. The order in which they are specified in this document is an appropriate implementation order.

A few predicates that could help you:

- `maplist/N` allows you to apply a partial predicate over lists of parameters. Much nicer than iterating explicitly each time you have to handle a list.
- Dynamic predicates (manipulated using assertz, retract, …): necessary to modify the database. To find out more about this, type apropos(database).

- Predicates for analysing and constructing terms, in particular the =.. operator and `functor/3`. To find out more about this, type apropos("Analysing and Constructing Terms").
- `forall/2`, `findall/3`, (and maybe `bagof/3`, `setof/3`). As discussed in the lab sessions.
- Meta-call predicates: `call/N`, `apply/2`.
- `throw/1` to throw exceptions. Beware of the behaviour of this predicate when backtracking ! It is usually recommended to cut a working alternative before the fail case.
  For example: `mypredicate(X, Y, Z), !  ; throw("BOOM").`
- `string_concat/3` to build nice error messages (note: you can concatenate strings with atoms using this predicate).
- The biggest difficulty in the project is the handling of selection conditions. Think carefully about your plan for them before starting to implement.

Parsing Prolog

To parse Prolog, we supply you with the following predicates:

```
:- use_module(library(dcg/basics)).
% DCG rule that checks if the remainder of the input
% starts with Delim (a list), but does not consume
% any input.
lookahead(Delim, L, L) :-
    prefix(Delim, L).

% Checks if the remainder of the input starts with one
% of the items in the supplied list (as per lookahead/3).
lookaheads([H|T]) -->
    lookahead(H), ! ; lookaheads(T).

% Reads text until one of the delimiters in Delims is
% encountered, then unifies R with the Prolog term parsed
% from the text.
% ! Will crash the parse if a delimiter is encountered
% but the intervening text is not a Prolog term (you
% don't have this handle this case).
prolog_term(R, Delims) -->
    string(S), lookaheads(Delims), !, {
    read_term_from_atom(S, R, []) }.
```

So for instance:

```
?- phrase(prolog_term(Term, [`;`]),
    `+kitchen(sink, 42, example(youhou/lol));`, Rest).
Term = +kitchen(sink, 42, example(youhou/lol)).
Rest = [59]. % `;`
```

Note that the + in front of kitchen is merely a functor, i.e. +x is the same as +(x) and a+b is the same as +(a, b).

Modalities

First off, half of your grades (10/20) are determined by your score on the tests we supply you in tests.pl. Hardcoding expected answers so that the tests pass will be considered as cheating.

Here is how we will evaluate your code:

- With `swipl` in your solutions directory we will run
  `[solution].`
  then
  `[input].`
  and then
  `[tests].`
  Finally, we run the tests using
  `run_tests(sql).`
- We will additionally run a few secret tests.
- We will also carefully look at your code.

In particular, here are a few guidelines:

Code style. Write the signature of each rule on a separate line, then one clause per line (some exceptions are admissible, use your judgement). Try not to go over 80 characters per line (the occasional exception is admissible). Most of the rest is common sense: use descriptive variable names, split complex predicates when (and if) it makes sense, extract common idioms etc.

Make your code idiomatic. Getting it to work is only half the challenge, we want you to show that you've integrated how to think in Prolog. After you have a solution, look at your code and ask yourself what might be improved.

Don't embarrass yourself: read the Getting Started section above.

Comments. All predicates you implement should be commented as necessary (which is to say most predicates should probably be commented unless they are so obvious that they can be understood at a glance).
It's also useful to include a signature with the +/- notations in a comment, especially when you use pattern matching inside the actual signature (e.g. `mypredicate([H|T])` could use a comment to explain what the the list parameter being pattern-matched is!).
If some part of your logic bears further explanations / clarifications, don't hesitate to include a small comment paragraph. Treat your code as a document.

Readability. This goes with the other points, but it bears repeating. Since your code will be read by human beings (your dear teaching assistant and professor) you want to make it as clear as possible to make them happy.

Terseness. The priority is style and readability. But given that all else is equal, the shortest code is the better code.

If you have any further questions, please ask them to your course assistant.

Deliverables

Follow these instructions to the letter, or you will LOSE POINTS.

The project deadline is Friday 15 March. (That is, the last day of week 6 of the second quadrimester.) We will use the week after that to run your tests and look at your code. If based on that we have sufficient information to give you a passing grade, we will grade you based on the code received and the results of the tests (i.e., the tests which we gave you as well as some additional tests).

If, however, we need some clarifications on how to understand or run your code, we may invite you for an oral discussion.

You are strongly requested to work in pairs. If you work in pairs, of course each of the students is supposed to understand in detail all of the code that was produced by both students. If you work in pairs, please indicate the names of both partners clearly at the top of your solution file.

At the project deadline you must post on Moodle (section: Mission 1 submission page) the following files (and only those, with exactly these names):

- `solution.pl` - The full implementation of your SQL framework.
- `input.pl` - Your own version of the input database. This must contain exactly the same data as the database you were given, but you may use different predicates to store the data.

We do not request a separate report for this mission. As stated above, your code is the report. So make sure it is *readable* and well *commented.* Treat your code as a document. Make it readable. Don't hesitate to include small commented paragraphs with explanations, assumptions or implementation choices, where needed.

Final warning: We will be intransigent towards plagiarism. We are well aware of existing solutions, and will cross check students' submissions, both manually and automatically. You may discuss the assignment between yourself, but are not allowed to share any code, nor to borrow any code you found online or elsewhere. We will find out. We always do. And we won't be happy if we do.

Additional warning: The use of artificial intelligence tools and large language models like ChatGPT for this mission will not be allowed. If you have questions about Prolog, ask the professor or the course assistant, use StackOverflow, or check the webpages or community of SWI-Prolog. They will likely provide you more correct questions to your answers than ChatGPT too. The goal is to learn the language, not to learn to use a tool than can solve the problem for you.