UE: **MIF14**

Name : **CECILLON Enzo**

Student Number : **11805901**

Git repository

Sujet

Language and Library

Language : *Python*

Parser : *ANTLR4*

Requirement : *Docker Engine*, *Docker Composer*

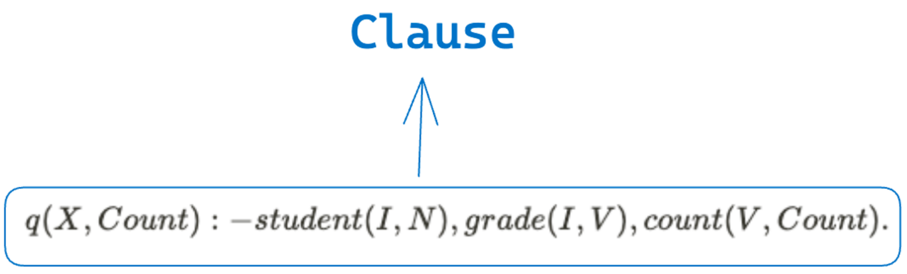
You can run this entire project using only Docker that will handle image pulling and deps download.

Check out the README.md for more information.

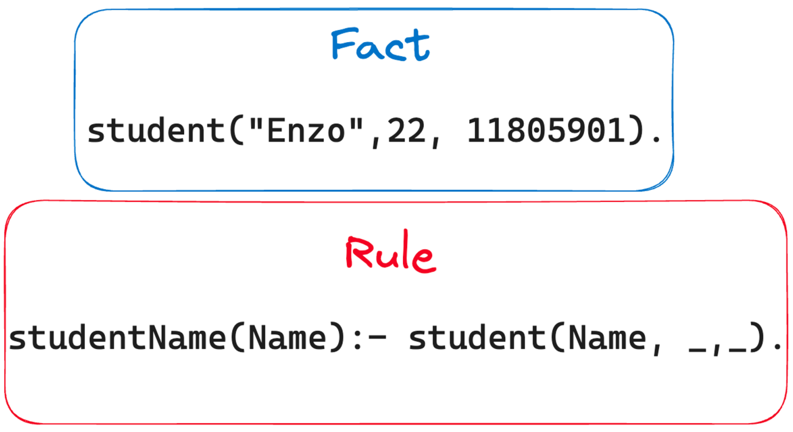
Building the Datalog parser

Using what I learned in **MIF08** (Compilation) and using **ANTLR4** I decided to define a simple grammar that will parse any input file and create their corresponding object.

Clause


Clause
$q(X, Count) :- student(I, N), grade(I, V), count(V, Count).$

- **Clause** : Head ***litteral*** followed by an **optionnal** body.
    - A Clause without a **body** is called a **Fact**, and a **Rule** if it has one.
    - `:-` **seperates** the **head** from its **body** (viewed as a neck).
    - **A Clause is called safe** if, **every variable** in its **head occurs** in some **litteral in its body**.

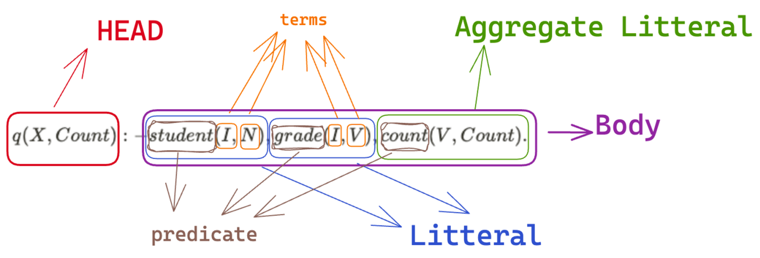


```
student("Enzo",22, 11805901).
```

```
studentName(Name):- student(Name, _,_).
```

Clause Details

Datalog Parser



$$q(X, Count) :- student(I, N), grade(I, V), count(V, Count).$$

EDB(entities)	IDB(rules)
student	q
grade	

Head

The head is an assertion we can conclude if the body is true.

Body

The body is a series of predicates (conditions) that must be satisfied for the rule to be applicable.

Litteral

A literal is a predicate symbol followed by an optional parenthesised list of comma separated terms, or it is an external query as described below.

Predicate

A predicate symbol is either an identifier or a string.

Term

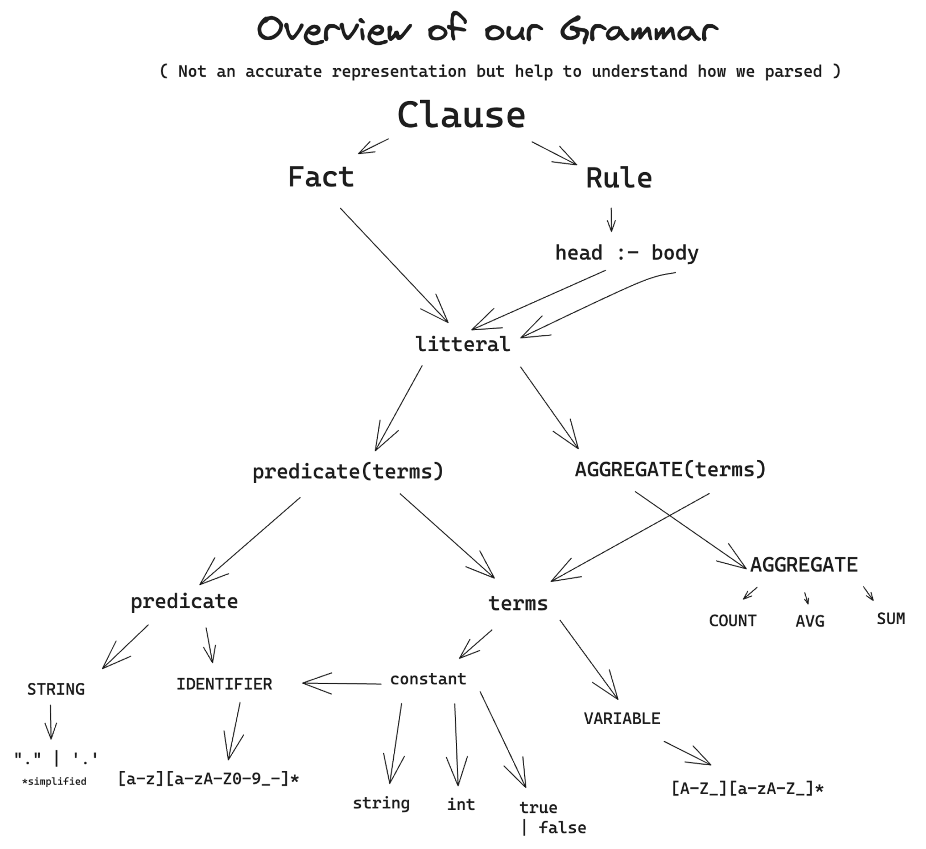
A term is either a variable or a constant.

Constant

- A constant is an identifier, string, integer, or boolean, where booleans are written the same as the identifiers true and false, and integers are written the same as identifiers 0 or those with a nonempty sequence of digits, no leading zero, and optionally prefixed with -.
- As a special case, two terms separated by = (!=) is a literal for the equality (inequality) predicate.

Variable

- A variable is just a sequence of letter starting with an Upper Case and give us the possibility to cast some informations so it can be used through an evaluation.

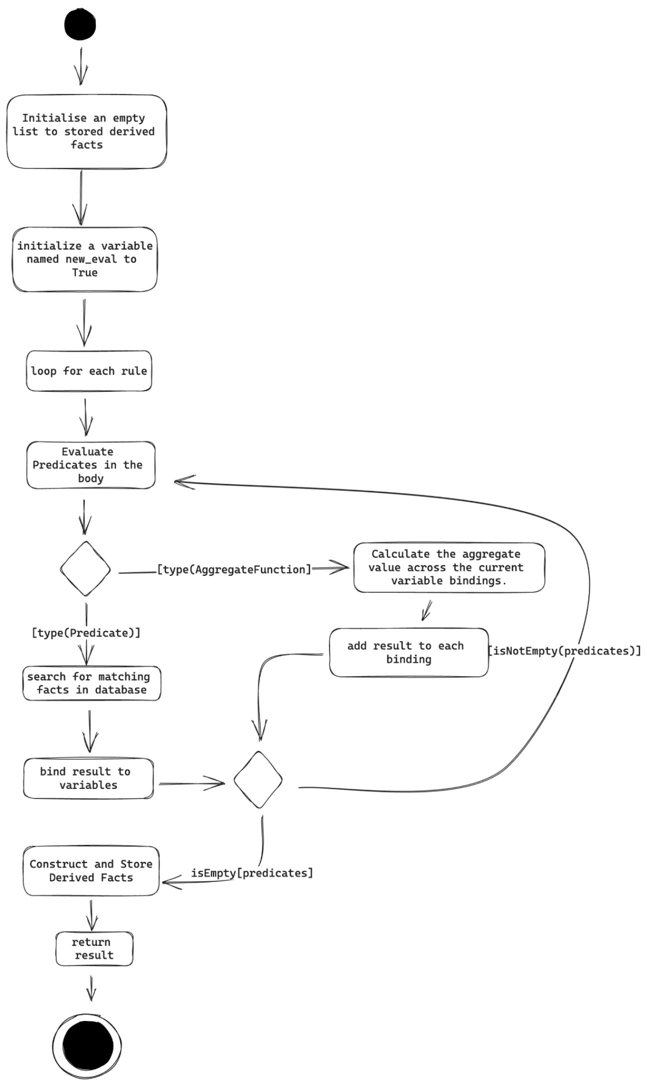


Assumptions

- **EDB** and **IDB** always start with a **Lower Case.**
- **Variables** start with an **UpperCase** and can be followed by **any symbol.**
- We consider the following **Constant** :
    - `bool` : true | false
    - `int`: [0-9]
        - **no float**
    - `string` : 'hello world', "hello", "yoolo dsq q" ….
- Available Aggregate functions:

  The input variable needs to contain a number or return an **Error message**.
  (in a production environment we would raise a custom exception)

    - **AVG**
    - **SUM**
    - **COUNT**


Initialise an empty list to stored derived facts
initialize a variable named new_eval to True
loop for each rule
Evaluate Predicates in the body
[type(AggregateFunction)]
Calculate the aggregate value across the current variable bindings.
add result to each binding
[type(Predicate)]
search for matching facts in database
bind result to variables
[isNotEmpty(predicates)]
isEmpty[predicates]
Construct and Store Derived Facts
return result

Bottom-up evaluation algorithm

Detailed Description

The primary goal is to find all possible derivations of the facts based on the given rules and facts until no new information can be inferred.

The algorithm receives a set of known facts that constitute our database.

Input:

- **`datalog_program`**: A list of Datalog rules.
- **`EDB`**: A list of Extensional Database (EDB) facts, representing the base facts known to the system.

Output:

- A list of derived facts inferred from the base facts and rules.

Algorithm Steps:

1. Initialize an empty list `derived_facts` to store the derived facts.
2. For each rule in the `datalog_program`:
    1. Extract the rule's head and body into variables `head` and `body`.
    2. Initialize a list `all_variable_bindings` with an empty dictionary to store variable bindings.
    3. For each predicate in the `body`:
        1. If the predicate is of type `Predicate`:
            1. Retrieve all matching bindings for this predicate from the EDB facts and store them in `predicate_bindings`.
            2. If the `all_variable_bindings` list only contains an empty dictionary and there is more than one `predicate_bindings`, replace `all_variable_bindings` with `predicate_bindings` and continue to the next predicate.
            3. Index all the bindings in a dictionary called `index`.
            4. Initialize an empty list `new_variable_bindings` to store updated variable bindings.
            5. For each combination of existing and new bindings, if they have common keys with the same values, merge them into a single binding and append to `new_variable_bindings`.
            6. Replace `all_variable_bindings` with `new_variable_bindings`.
        2. If the predicate is of type `AggregateFunction`:
            1. Apply the aggregate function to `all_variable_bindings` and store the result in `result`.
            2. Add the `result` to each individual binding in `all_variable_bindings`.
    4. For each variable binding in `all_variable_bindings`:
        1. Construct a derived fact using the `head` and the variable binding.
        2. If this derived fact is not already in the `derived_facts` list, append it.
3. Return the list `derived_facts` containing all the derived facts.