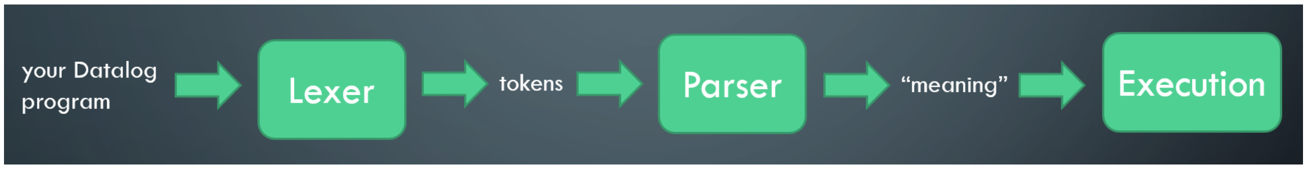
Project 3 Guide

CS236 - Discrete Structures
Instructor: Brett Decker
Spring 2021

CS236 Projects Overview

There are five projects for CS236 that are each a piece of creating an interpreter for Datalog programs (see `https://www.geeksforgeeks.org/compiler-vs-interpreter-2/` for details about interpreters). An interpreter is comprised of three components: the Lexer, the Parser, and the Execution engine. In Project 1 you will build the Lexer. In Project 2 you will create the Parser. In Projects 3, 4, and 5 you will develop the Execution engine.
Here is a graphical representation:



An execution engine is a program that takes as input the "meaning" of source code (from a parser). It executes the meaning of the program defined by the source code. For this project, you will implement the beginning of a Datalog execution engine, the last piece you need for the Datalog interpreter.

Project 3: Execution Engine, Part I

Building an execution engine for Datalog will help us apply our study of logic and relations. First, go read the entire specification for Project 3. Good software design is about breaking out the functionality into smaller pieces, which we call decomposition. The project specification requires a specific decomposition of classes used to store the meaning of program. It is required.

Two Step Process

It is strongly encouraged to approach this project in two steps:
(1) create the database classes,
(2) create the database interpreter.
During and after each step, test your code for correctness and perform clean-up before moving to the next step (do this after step 2 before trying to pass off).

Step 1: Create the database classes

Required Classes

- `Tuple` - contains `vector` of values (`string`)
- `Header` - contains `vector` of attributes (`string`)
- `Relation` - contains a `set` of `Tuple`s associated with a name and `Header`
- `Database` - contains a `map` from name (`string`) to `Relation`

The standard library `set` requires that its templated type have a less-than operator. Since you will want a `set` of `Tuple`, you need to add the less-than operator function to the class `Tuple` to make `std::set<Tuple>` compatible with `std::set<T>`. In order to compare two tuples, you must compare the values inside of each to decide which is less than the other. You may not already know this, but this comparison is already defined for `std::vector<std::string>`. As such, you can borrow this already defined comparison to implement your custom tuple comparison. Essentially: "If the vector within this tuple is less than the vector within the other tuple, then we'll say that this tuple is less than the other tuple." This process is called 'delegation', which is a pattern that you can learn more about in future CS classes. The implementation can be generated with CLion using `Code` → `Generate` → `Relational Operators`. The following is the implementation necessary for this lab:

```
class Tuple {
  ...
  bool operator< (const Tuple &rhs) const {
    return values < rhs.values;
  }
  ...
};
```

Another way is to have Tuple extend a vector of strings, but this is not recommended, because it can lead to issues with deleting a Tuple pointer (see `https://stackoverflow.com/questions/4353203/thou-shalt-not-inherit-from-stdvector`).

The `Relation` class should have the following operations (methods):

- `select`: parameters are index and value (relation column and value to select)
- `select`: parameters are index and index (relation columns to select on same values)
- `project`: parameter is list of indices (the columns to keep)
- `rename`: parameter is list of attributes (defines the new header)

Note that the list of indices given to `project` specifies the new order of the columns in the new relation it returns.
The methods `select`, `project`, `rename` all return a new `Relation` object.

Step 2: Create the interpreter

Required Classes

- `Interpreter` - should contain or be given the objects of class `Datalog` and `Database`

There should be no coupling between `Datalog` and `Database` classes. Either the interpreter needs these two object at construction, or pass them by pointer or reference to the method that will run the interpretation.

Interpreter Algorithm

Here is the algorithm for running the interpreter (pseudo code thanks to Dr. Barker). Note that constants are Datalog `STRING`s and variables are Datalog `ID`s:

```
for each scheme ‘s’
   create a relation using name and parameter values from ‘s’
for each fact ‘f’
   make a tuple ‘t’ using the values from ‘f’
   add ‘t’ to relation with the same name as ‘f’
for each query ‘q’
   get the relation ‘r’ with the same name as the query ‘q’
   select for each constant in the query ‘q’
   select for each pair of matching variables in ‘q’
   project using the positions of the variables in ‘q’
   rename to match the names of variables in ‘q’
   print the resulting relation
```

Note

We strongly encourage you to create a method with the following signature:
`Relation* Interpreter::evaluatePredicate(const Predicate& p)`

If you don't feel comfortable with pointers, you could return just a `Relation`, but recognize that you will be then copying the object onto the stack. Having a function that receives just one Predicate and returns a corresponding Relation is extremely useful for Project 4; In Project 3, this function is only used to evaluate each query, but in Project 4 it will be used to evaluate each predicate in the body of each rule.

Conclusion

Start this project as early as possible. You will code better when not rushed, and you will be more inclined to test as you go (which will reduce overall coding time). See Project 3 on the course website for requirements and specifications.