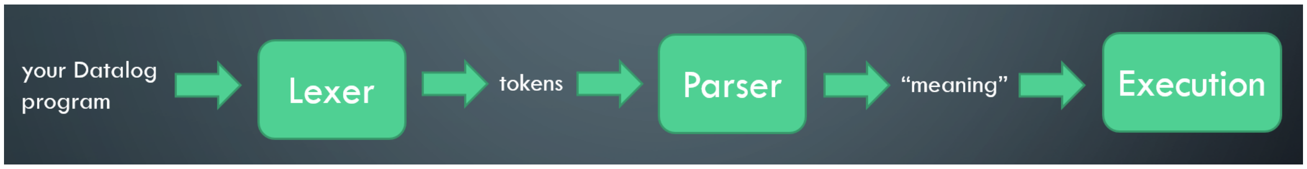
Project 5 Guide

CS236 - Discrete Structures
Instructor: Brett Decker
Fall 2020

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 improve the algorithm that implements the generation of new Datalog facts using the rules to complete the Datalog engine, which completes the Datalog interpreter.

Project 5: Execution Engine, Part III

Building this next part of the execution engine for Datalog will help you apply your study of graphs and algorithms, especially depth-first search. First, go read the entire specification for Project 5. Next, make sure you understand the depth-first search forest algorithm and the algorithm on finding strongly connected components in directed graphs. Good software design is about breaking out the functionality into smaller pieces, which we call decomposition. If you have a good design from your previous projects you should be able to add in the new functionality with relative easy. If your design is a mess, adding the new functionality will be a nightmare.

Six Step Process

It is strongly encouraged to approach this project in six steps:
(1) create the `Graph` class (just the data structure),
(2) create the function to build the original and reverse `Graphs` from your Datalog rules,

(3) create two depth-first search function to operate on an object of `Graph`: 1) returns the post-ordering; 2) returns the search tree,

(4) create two depth-first search forest functions: 1) returns the post-ordering; 2) returns the forest,

(5) integrate all of the above code to implement the function for the algorithm for finding strongly connected components,

(6) integrate all of the above code to modify your rule evaluation based on the order of the strongly connected components.

During and after each step, test your code for correctness and perform clean-up before moving to the next step (do this after step 6 before trying to pass off). You *could* write all the code and then just try to pass off. But then you will have no idea where errors in your code are. Testing at each step helps you localize your error detection and correction. Consider having your `Graph` class have a map from integer to set, `map<int, set<int>>`, that contains your adjacency list (the integers in your map should correlate to the indices of your set of vertices. See the project specification and lecture notes (on Depth-First Search Forest) for details on how to implement the steps 2-5 (the last step is dependent on your code).

Pseudo-code

There is pseudo-code in the reading Depth-First Search Forest: Finding Strongly Connected Components for both depth-first search and depth-first search forest. You will need to add code to keep track of the postorder. Consider using this pseudo-code as a starting point. Get your depth-first search code implemented and testing, then implement and test depth-first search forest. Then add in the postorder code. Decomposition and incremental development will reduce the number of errors in your code and the time you spend debugging.

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 5 on the course website for requirements and specifications.