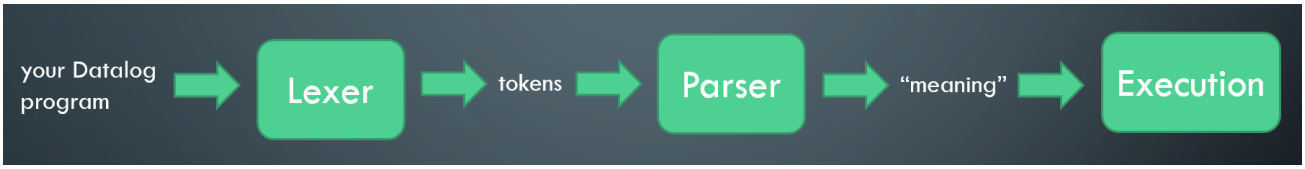
Project 2 Guide

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:



A parser is a program that takes as input a collection of tokens (from a lexer). The parser then tries to parse the tokens according to the programming language grammar to determine correctness. The parser also determines the meaning of the program based on the order and type of tokens. This meaning is then given to the execution engine, which executes the program.

Project 2: Parser

Building a parser will help us apply our study of grammars. First, go read the entire specification for Project 2. 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. But, before we get to the meaning we need to parse the tokens to ensure they follow the rules of the Datalog grammar. You will create a recursive-descent parser for this. See the lecture notes and the practice (on recursive-descent parsing) for pseudo-code. Consider creating the pseudo-code (maybe as comments in a C++ file) for the entire recursive-descent parser before writing any code. This guide will go into the three step process for the project and how to get your code correct.

Datalog Grammar

You should have already read the project specification. The Datalog grammar has to be understood before coding. The convention with the Datalog grammar is that terminals are in all caps like `LEFT_PAREN` and that non-terminals are in in lower-camelcase as in `scheme` – don't confuse the non-terminal `scheme` with the keyword `Schemes` which is represented as the terminal `SCHEMES`. It is important to understand that the Datalog grammar deals with tokens, not text. Each token created by your lexer is a single terminal. The collection of tokens creates the terminal string for the parser to check. Let's explore the grammar a little more by looking at the `scheme` non-terminal:

`scheme` $\rightarrow$ `ID LEFT_PAREN ID idList RIGHT_PAREN`
`idList` $\rightarrow$ `COMMA ID idList` | λ

Look at the production from the `idList` to λ. This is what makes the list work. It allows `idList` to produce zero or more terminal `ID`s (separated by `COMMA`s). Let's look at a concrete example of a scheme. This is the line in the text file:

```
snap(S,N,A,P)
```

Your lexer converts this into tokens:

```
ID LEFT_PAREN ID COMMA ID COMMA ID COMMA ID RIGHT_PAREN
```

The logic for parsing the non-terminal `scheme` will be in a function `ParseScheme` that takes as input the above tokens. Given the above input, this function will check and advance past the first three tokens (`ID LEFT_PAREN ID`) and then call the parse function for `idList` – let's call it `ParseIdList`. This function will check if the first token is a `COMMA` or not. If it is not, the function simply returns. This is how we encode taking the lambda production in code – do nothing. If the function finds the `COMMA` it will advance the input and check for an `ID`. It throws an exception if there is not an `ID`. If there is an `ID`, then the function calls itself, recursively. Given our above example, see if you can determine how many times the `ParseIdList` function will be called. Once this function finally returns back to `ParseScheme`, `ParseScheme` will check for the final `RIGHT_PAREN`. The answer is that `ParseIdList` gets called four times, because there are three `COMMA`s.

Three Step Process

The project specification gives a three step approach to implementing the project. It is worth mentioning a second time.

- First: write (and test) the recursive-descent parser to check the grammar (don't build any data structures)
- Second: write (and test) classes for data structures (Rule, Predicate, Parameter, etc)
- Third: add code to the parser to create data structures (if you have unit tests, you can re-run them easily to make sure nothing breaks. You can also test this code to make sure you create the right data structures with the right information)

Difficulties

Granted you understand the algorithm for recursive-descent parsing, this project is conceptually easier than Project 1 (if you don't feel confident with recursive-descent parsing yet, study the lecture notes again, then get help – once you feel comfortable with the algorithm, the project will go smoothly). The devil is in the details – if your parsing methods are not perfect, your parser will not work. The urge to copy and paste is irresistible (at least for me) because of the similarity of the parsing methods. Unit testing is the key to catching copy/paste errors (because who wants to do all that typing?).

How to Test

As previously mentioned, good software design is about decomposition and encapsulation. But these principles can sometimes be at odds with unit testing. Your first design might be to create `Parser` class that has a single entry (public) method, say `Parse(...)` that takes a collection of tokens. Then you might add all the non-terminal and terminal specific parsing methods as private methods. This is not a bad approach; in fact, it is a good one. But it is hard to test. The only method you can test is the entry one, `Parse(...)`. This means you'll have to run the entire parsing code for each test. How do we make testing easier without violating encapsulation and switching our private methods to be public? We create what's known as a "helper class." We move all the private non-terminal and terminal specific parsing methods into the helper class (`ParserHelper` is a perfectly acceptable name). We make these private methods public. Now we can unit test each non-terminal and terminal specific parsing method for correctness in a separate unit test.

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 2 on the course website for requirements and specifications.