Introduction

A lexical analyzer groups characters in an input stream $S$ into tokens. Parsing determines if an input stream of tokens is a member of a language $L$ as defined by a grammar $G$. (i.e., is $S \in L(G)$?)

Project Description

Using your LexicalAnalyzer from Project 1, write a parser for the Datalog grammar defined below. You will want to add a few methods to your LexicalAnalyzer to make it easy for you to access the tokens and use the tokens in order to determine if the given input is an instance of a valid DatalogProgram.

If the parse is successful (i.e., if $S \in L(G)$), return the string: “Success!” followed by all the schemes, facts, rules, queries, and the domain values (i.e., all the strings that appear in the facts, rules, and queries). Include the number of items in each list. Note that the domain is a set (no duplicates) of strings.

If the parse is unsuccessful (i.e. if $S \not\in L(G)$), output “Failure!” followed by the offending token, (i.e., its triple including its token-ID name, string value, and line number). Note that the parser stops after encountering the first offending token.

Requirements (Get the Design Right)

The following are required for passoff (and will help you succeed in this project):

- A recursive descent parser with a parse method for every non-terminal in the grammar. The parser can either be placed in a separate parser class that would have methods for each non-terminal (i.e., methods called parseDatalogProgram, parseSchemeList, parseSchemeListTail, etc.) or separated into the constructors of or parse methods in the individual classes. Note that whether the parsing happens in a parser class or in the individual classes, the actual parsing should be almost identical.

- A class for each type of object in the parse tree. You must create classes for at least the following parts of the language: DatalogProgram, SchemeList, Scheme, FactList, Fact, RuleList, Rule, QueryList, Query, Domain, Predicate, and Parameter.

- Each class in the parse tree must have a toString method, and the output of the program must be formed by these toString methods (not by the parse methods).

The Datalog Grammar

In the grammar below, we use the following meta-symbols: “<” and “>” enclose non-terminal names, “->” means “is defined as,” “|” means “or,” and “ε” represents the empty string.
Note that comments do not appear anywhere in the grammar because comments should be ignored. In terms of parsing, this means that you should be able to skip any number of comments showing up at any place in the DatalogProgram.
### Examples

Your program must have output formatted exactly like the example outputs below. Note that these are not sufficient to completely test your program.

#### Example 20 Input: `ex20.txt`

| Schemes: | employee(N,I,A,J)  
|          | WhoToBlame(N,J)   |
| Facts:   | employee('Dilbert','51','10 Main','EE').  
|          | employee('Dilbert','51','10 Main','Marketing').  
|          | employee('Dogbert','52','10 Main','EE').  
|          | employee('PHB','53','Hades','Pain Management').  |
| Rules:   | WhoToBlame(N,J) :- employee(N,'51',A,J),expr((A+A)).  
|          | WhoToBlame(N,I) :- NotEquals(!=('Dilbert','PHB')).  |
| Queries: | WhoToBlame('Dilbert',J)?  
|          | WhoToBlame(N,'EE')?    |

#### Example 20 Output: `out20.txt`

| Success! | Schemes(2):  
|          | employee(N,I,A,J)  
|          | WhoToBlame(N,J)   |
| Facts(4):| employee('Dilbert','51','10 Main','EE').  
|          | employee('Dilbert','51','10 Main','Marketing').  
|          | employee('Dogbert','52','10 Main','EE').  
|          | employee('PHB','53','Hades','Pain Management').  |
| Rules(2):| WhoToBlame(N,J) :- employee(N,'51',A,J),expr((A+A)).  
|          | WhoToBlame(N,I) :- NotEquals(!=('Dilbert','PHB')).  |
| Queries(2):| WhoToBlame('Dilbert',J)?  
|          | WhoToBlame(N,'EE')?    |

#### Example 21 Input: `ex21.txt`

| Schemes: | snap(S,N,A,P)  
| #comment | HasSameAddress(X,Y)   |
| Facts:   | snap('12345','C. Brown','12 Apple','555-1234').  
|          | snap('33333','Snoopy','12 Apple','555-1234').  |
| Rules:   | HasSameAddress(X,Y) :- snap(A,X,B,C),snap(D,Y,B,E).  |
| #comment | HasSameAddress('Snoopy',Who)?  |

#### Example 21 Output: `out21.txt`

| Success! | Schemes(2):  
|          | snap(S,N,A,P)  
|          | HasSameAddress(X,Y)   |
| Facts(2):| snap('12345','C. Brown','12 Apple','555-1234').  
|          | snap('33333','Snoopy','12 Apple','555-1234').  |
| Rules(1):| HasSameAddress(X,Y) :- snap(A,X,B,C),snap(D,Y,B,E).  |
| Queries(1):| HasSameAddress('Snoopy',Who)?  |

#### Example 22 Input: `ex22.txt`

| Schemes: | snap(S,N,A,P)  
|          | NameHasID(N,S)   |
| Facts:   | snap('12345','C. Brown','12 Apple','555-1234').  
|          | snap('67890','L. Van Pelt','34 Pear','555-5678').  |
| Rules:   | NameHasID(N,S) :- snap(S,N,A,P).  |
| Queries: | NameHasID('Snoopy',Id)?  |

#### Example 22 Output: `out22.txt`

| Failure! | (Q_MARK,"?",10) |
Notes

Build your project using the cs236Bundle. Your source code goes in the src/project2 directory. All classes must be package project2 and import project1. You must implement the body method in the Project2 class, which receives a filename from the command line arguments. Its only task is to construct a DatalogProgram and return “Success!\n” + datalogProgram.toString()'.

To format the output for pass-off, you must make it formatted exactly like the examples, including capitalization, punctuation, and whitespace. Note in particular that the list of domain values is sorted.

The parseDatalogProgram method returns a DatalogProgram object. The output of the program primarily consists of the string returned by “datalogProgram.toString()” (which in turn calls the toString methods of its children in the parse tree).

The input for Project 4 will be the SchemeList, FactList, RuleList, and QueryList produced by your DatalogProgram.