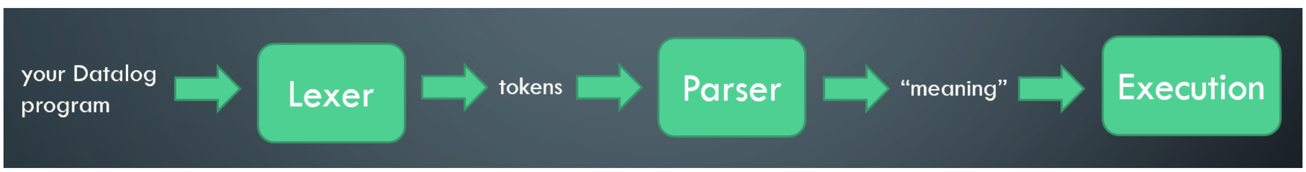
# Project 1 Guide

CS236 - Discrete Structures
Instructor: Brett Decker
Spring 2022

## 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 lexer is a program that takes as input a program file and scans the file for tokens. The output to the lexer is a collection of tokens. These tokens can then be read by the parser. The parser 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.

## Why Tokens?

You might be questioning the reason we are converting programming files into tokens. Programming languages are created to allow each programmer to use variable names, method names, comments, strings, etc. that are specific to him or her or the application domain. These items are very important for readability and organization to the programmer, but they are not essential for the computer to execute the code. A lexer turns programmer-specific "items" into programmer-independent structures called "tokens." Each token contains the information the computer needs, namely three items: 1) a token name, which is selected from a set of predefined names, e.g. "STRING"; 2) the specific "item" for that token, e.g. `"Hello World"`; and 3) a line number. This information allows the computer to get one step closer to execution.

# Project 1: Lexer

Building a lexer will help us apply our study of finite-state machines and automata. The lexer is either an implementation – a finite-state machine or several finite-state automata – that takes the Datalog program file as its input and outputs a collection of tokens. Good software design is about breaking out the functionality into smaller pieces, which we call decomposition. All projects are required to be completed in C++ and we *strongly* recommend using sound Object Oriented (OO) design practices. These guides are meant to help you learn and use such practices.

This guide contains the approach to implementing a lexer for Project 1. It is an algorithm called 'Parallel and Max' and its OO design closely matches finite-state automata theory and uses inheritance. Once you have the algorithm implemented, it is easy to build and test the needed automata classes incrementally. You are *required* to implement this algorithm. Remember, this project is the start of your interpreter; it will be used in *every* other project in this course. Develop your code in an iterative manner: create a small piece of functionality and test it as you go. This will save you time in the long run – nothing takes more time than coding by the "Hack and Pray" philosophy.

# Approach: Parallel and Max

**First, make sure you watch the Project 1 video in its entirety before reading this.**
This approach will take upfront effort to learn the algorithm, but it will make the overall coding much easier. It also encourages incremental development and will reduce your time spent debugging.

You will create starter code by completing Lab 1. This guide describes some of the code and what you need to do with it.

In this approach we take advantage of FSA theory and string recognition. Note that each valid `Token` is a specific string that could be generated by a regular expression (reminder, you cannot use a regular expression library for this project). That means that each type of token could define a language accepted by a finite-state automaton. We can use this to decompose the problem of lexing into multiple finite-state automata, instead of one, monolithic finite-state machine.

Your entry point is `main.cpp`, which should do the following:

- Your main file should contain only the `main` function. No global variables. Your `main` function should be small and serve the following purposes: verify command-line arguments, instantiate an instance of the `Lexer` class, and pass the input to `Lexer` and let it run. For this project, your main will then need to get the Tokens from the `Lexer` and print them out in a specified format (so the pass-off driver can check it for correctness). At the end, your `main` function should perform any clean up (e.g. deallocation of memory).

The `Lexer` class (.h file) should store a collection of finite-state automata. It also needs to store all the generated tokens. You will get started on this class in Lab 1.

The Parallel and Max algorithm is implemented in the public method `Run`. This algorithm does the following:

- Initialize two variables: maximum read value to 0 and corresponding maximum finite-state automaton (denoted as the "max automaton") to the first finite-state automaton in the collection
- For each finite-state automaton
  - Give it the input string and have it return whether or not it accepts the string (first $n$-symbols of the string)
  - Get the number of symbols (characters) read by it
  - Update the maximum read value if more than the current value and then update max automaton to reference the current finite-state automaton
- Now that we've found the finite-state automaton that will read the maximum amount of characters, tell it to generate a `Token` given the input it could accept
- Store new `Token` and update the input string (remove the characters read – remember the number of characters in stored in our variable
- While the input has more characters repeat

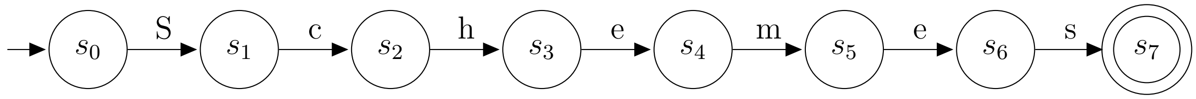
You will work on this code as part of Lab 1. Note that you must keep track of line numbers for each token. The code and algorithm rely on inheritance. There must be some base class for all automata, e.g. `Automaton`, that contains the following virtual method: `s0`, and public methods `getNewLines`, and `getType`. Your `Automaton` class definition is found in Lab 1. Look at that class definition. Note that `s0` is a pure-virtual method and thus must be defined in every subclass that inherits from `Automaton`.

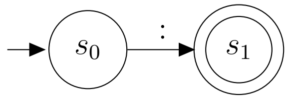
In general, you need to create a class that inherits from `Automaton` for each type of `Token` (some designs could potentially use one subclass for multiple `Token`). Somewhat counter-intuitive, you also need to create a subclass for undefined types of `Token`: multi-block comments that never end and strings that are never closed, as well as unrecognized single characters. IMPORTANT: The order of the automata in your collection *does* matter. Your automata that match to keywords should come first (this is the case because of the way the Parallel and Max algorithm works). This will make sure your automaton for a keywords is chosen as the "max automaton" instead of your identifier automaton (see below for why).

You may be wondering how this algorithm works (or if it even does). Let's consider the syntax of a C++ program (you should be able to relate this to the syntax of the Datalog program you are parsing). How does a compiler tokenize the following line: `int intx = 5;`? These are the corresponding tokens (while the whitespace is important for separating tokens, it is just consumed): `<integer><identifier="intx"><assignmentoperator>`

<numberliteral="5"><semicolon>. The compiler has to be able to tokenize “int” differently from “intx.” Using our algorithm, we’d have an automaton to check for the keyword “int.” We’d also have an automaton for identifiers. When the input is “intx = 5;” (after tokenizing the first two tokens) the keyword automaton will say that it can accept up to 3 characters: ‘i’, ‘n’, ‘t’, but the identifier will say that is can accept up to 4 characters, so the identifier automaton will be chosen to create the next token, instead of the keyword. Now let’s consider the first “int.” As long as your keyword automaton for `int` is checked first, when the identifier automaton says it also can read 3 characters, it will not be selected, since there is already an automaton (your `int` one) that said it can read 3 characters.

The first place to start is by create FSA diagrams for each of the automaton you will need. Doing this upfront will reduce coding time and help you work out your understanding of the token types. Learning how to create diagrams and then turn them into code is an important software engineering skill. Let’s show the FSA diagrams for the automata that will recognize the ‘Schemes’ token and the ‘:’ token.




Note that this approach is very easy to unit test. Each `Automaton` subclass can be tested separately. The overall algorithm in the `Lexer` class can also be tested separately. You can incrementally build new automata subclasses and add them to the `Lexer` for testing, thus increasing the functionality a step at a time.

# 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 1 on the course website for requirements and specifications and Lab 1 to get started.