HERE I AM!

EECS665

Compiler Construction

Drew Davidson
Ruturaj Vaidya

Lecture: LEEP2 G415
MWF 3:00-3:50

Lab: Eaton 1005B

ANOUNCEMENTS | LAB | SCHEDULE | MATERIALS | ASSIGNMENTS

Project 1

Due on ~~8/31~~ 9/01 @ 11:59 PM

Accepted for 90% credit or 1 late day on ~~9/01~~ 9/02 @ 11:59 PM
Accepted for 80% credit or 2 late days on ~~9/02~~ 9/03 @ 11:59 PM
Accepted for 70% credit or 3 late days on ~~9/03~~ 9/04 @ 11:59 PM

Updates

- **Update 1 (8/20/2018 - 10:00 PM) -** The complete description of the project has been included.
- **Update 2 (8/21/2018 - 6:09 PM) -** An error was found in the example input (see Piazza: link). In light of the error, I've extended the P1 deadline by 1 day (not that I think you'll really need it).
- **Update 3 (8/22/2018 - 5:25 PM) -** You may use the command line g++ -std=c++11 dfa.cpp to compile your project. In fact, if you use one of the cycle servers, you currently MUST use

the flag to get my code to compile.

Overview

In this project, you will implement a simple Deterministic Finite Automaton (DFA) engine from a subset of the GraphViz Dot lanugage of directed graphs.

Each student must do this assignment **alone**. You may talk about the assignment, at a high level, with other students but you may not share code.

The purpose of this project is to remind you of how DFAs work, what they do, and how they are implemented. The goal of the project is to show you an implementation of an automaton of a type used in Scanners.

Each student must do this assignment **alone**. You may talk about the assignment, at a high level, with other students but you may not share code.

Description

The project will be graded based on your program's ability to pass test cases that we have written. The program takes two arguments:

1. A path to a file written in a subset of dot (described below)
2. An input string

If the DFA represented by the dot file matches the input string, your program should output the string "1" followed by a newline ('\n') character.

DFA Input Format

Note that the parser for the fragment of the dot language used to specify DFAs will be provided to you as part of the assignment. Nevertheless, if you'd like to test your program by providing

custom DFAs, they should be formatted as follows:

```
digraph {
        rankdir = LR;
        node [ shape = doublecircle ]; ACCEPTSTATES ;
        node [ shape = circle ];
        EDGES
}
```

where

- *ACCEPTSTATES* is a space-delimited list of the accepting states of the DFA
- *EDGES* is a newline-delimited list of *EDGE*

  where

  - *EDGE* is formatted as follows:

    FROM `->` *TO* `[ label = "`*CHAR*`" ];`

    where

    - *FROM* is the state that is the tail of the edge
    - *TO* is the state that is the head of the edge
    - *CHAR* is a single character of the transition

    (note that *EDGE* may begin with a tab character)

States may be any combination of upper- and lower-case "non-special" characters, *i.e.,* white space is not allowed in a state name, nor are the characters `-,).>"(`. A good policy is to look at the sample test files provided for you and augment them by adding and subtracting conformant states and edges.

For this assignment, a number of additional constraints will be put on the input (though keep in mind that these are NOT necessarily constraints on DFAs):

- Every state must have either one incoming or outgoing edge
- One special state S must always exist in the DFA. This state will serve as the start state
- Every DFA must have at least one accepting state

Resources

You will be provided with this tarball (link) containing three files:

- io.txt - lists input/output examples
- dfa.cpp - the parser code and file where you should implement your dfa engine
- ex1.dot - an example dot file for you to test your DFA engine
- ex2.dot - a second example dot file for you to test your DFA engine

dfa.cpp implements a DotParser class to parse the dot fragment file. The DotParser member functions `getAllEdges`, `getAllNodes`, `getAllAcceptingNode` are exposed publicly for your use, and should be self-explanatory.

The reason we went through all the awkwardness of using the dot format is so you can easily visualize the DFA. For an online DFA viewer, try viz-js.

Advice

A good way to implement the DFA matcher is using a state-transition table, as discussed in the course readings. You will need to take the edge list, node list, and list of nodes representing

accepting states and use them to derive the state-transition table yourself.

Imperfect specifications are a fact of life. Make an honest effort to interpret the specification and if you have questions feel free to post about it on Piazza.

Start early and seek clarification early. As described in the policies section, you should report issues in time for them to be addressed before the deadline. Also, keep in mind the old aphorism "Sleep is the best debugger". The closer to the deadline you submit, statistically speaking, the greater the chance that you made a mistake.

Deliverables

Upload a gzipped tarball p1.tgz containing the following to the P1 submission box... thing on

Blackboard:

- A single directory named p1
- A single file named dfa.cpp inside of p1

Policies

No additional files should be included in the tarball. Notably, no build files are required or allowed for this project.

Should any confusion or bugs arise from the demonstration code, it is your responsibility to report these issues either via email to drewdavidson@ku.edu or on the course Piazza page. Issues should be reported with enough time for them to be addressed prior to the deadline.

Your code need not handle ill-formed DFAs nor DFAs that do not meet the above spec. Your code will not be tested against such DFAs. Indeed, the parser will likely break on them, perhaps in colorful and unexpected ways. The purpose of this project is to get you implementing the matcher.

Your code should be able to compile and run using g++ on one of the EECS cycle server machines (cycle1.eecs.ku.edu, *etc.*) without requiring any runtime or compile-time flags except that you may use the -std=c++-11 flag. Your program should not require the use of external programs (i.e., you can't just call out to some pre-existing DFA engine as part of your code, though it would be admittedly hilarious).

Your code should be written in C/C++

You're allowed to submit as many times as you'd like before the deadline.

Most project scores are substantially or entirely assigned points based on automated tests. If you feel that your project grade does reflect the quality of your submission, please let us know as soon as possible to initiate a regrade request.

While your code is not expected to be optimized, it should complete in a reasonable amount of time with no obvious delays.

Code that you turn in should mostly consist of code that you have written. Code from sources outside of the class should be referenced, credited, and used sparingly. Please keep in mind that the collaboration policy prevents you from sharing your code with other students, or placing it in a public place where it can be easily accessed by other students. In particular, which you may avail yourself of BitBucket or GitHub, you must not place your code in a public repository or share credentials to a private repository with others.

Instructor KU EECS