CS 3240 : Languages and Computation
Course Project (Phase I)

Due date: November 14, 11:59 pm (No extensions!)

Guidelines:

1. You can program in C, C++ or Java. Do **not** use tools (like lex and yacc) -- such solutions will get ZERO points.
2. Code should be properly documented with meaningful variable and function names. Short elegant code is preferred.
3. You will find the course slides on DFA/NFA/scanner useful along with the relevant chapters from the Louden book.
4. Provide instructions about how to compile and execute your files along with test inputs.
5. Any re-use of code (from internet or any other source) is prohibited and will constitute act of plagiarism as per Georgia Tech honor code.
6. **If you run into road-blocks or have any questions, seek help from the TAs (please use the Forums to clarify the doubts– we check those round the clock and will respond expediently). Please get started on this project as soon as possible.**

Goal for main project:

We would like to automatically generate a scanner given lexical specifications – thus, we will be **implementing a scanner generator**… The lexical specification of our input language will be given to us in a certain file format (which is given below) – we will write a tool to first read this specification and then convert it through a series of steps to a scanner based on the concept of a table driven DFA. Thus, the scanner will be implementing the lexical specification given in terms of the regular expressions using a table driven DFA. The program which converts any such lexical specification to its table driven DFA implementation is called **scanner generator** which is the deliverable of this assignment.

Designing a scanner generator:

First the scanner generator should be able to read the lexical specification as per the input file format. The scanner generator will use the operator hierarchy in the regex and first generate the primitive NFAs for each of the parts of the regex. It will then generate one giant NFA by combining these primitive NFAs and then convert this giant NFA into a DFA table. You will then write (not generate) a program called **table-walker** that will read input characters one at a time, consult the DFA table and take actions as per the table and eventually generate a token or an error. The table-walker should be capable of skipping white spaces, tabs, newlines, eof characters in the input file. The table walker + DFA table is together called the scanner. This scanner is repeatedly called from a driver program to which it will return the generated tokens. The stream of tokens then will be stored and processed by the driver program. The generated scanner and the driver program which calls it and stores the returned token is also a part of the deliverable of your assignment.

Examples as to how to create a DFA table can be found here.

The following is the input file format to be used by the scanner generator is as follows (this file will define the lexical specification of the language we are designing scanner for):

%% Definitions for character classes will be contained in this section – examples below

$DIGIT [0-9]

$NON-ZERO [^0] IN $DIGIT // Comment: All the characters except 0 from DIGIT

$SMALLCASE [a-z]

$LETTER [A-Za-z]

%% Token definitions will be contained in this section using regexes – examples below

$IDENTIFIER $LETTER ($LETTER| $DIGIT)* // Example regex for IDENTIFIER

Detailed specification for regex grammar is given in the document **Project1-Regex.doc**

How to do it: First you will parse the input file and regular expression by writing a recursive descent parser as per the given grammar in **Project1-Regex.doc** Then you will generate primitive NFAs, combine them and generate a big NFA and then convert it to DFA

Testing and Deliverable:

Deliverables:

- Scanner Generator
- Test input files : lexical specifications that are input to scanner generator
- Output files : generated scanner tables (DFA) and the table-walker which will walk the DFA as per the table reading the input to generate tokens.

The scanner generator should be able to take any lexical specification (as per the above file format) as input and generate the scanner out of it. You can test it using some of your own lexical specifications and using lexical specs of C/C++/Java or any language of your choice available on the web. You can test the generated scanner using available C/C++/Java programs. You will also be given some sample test inputs.

The delivered version will be evaluated using our own lexical specs and using sample inputs based on the spec.

You will deliver full code (source + make files etc) along with instructions on compiling and testing it.

Submission Guidelines:

You need to submit the following with your project submission:

1) A report containing:

   a) A brief description of how you have implemented each part of the project

b) Any assumptions you have made.

c) Any problems you faced, and any module you were not able to implement (if at all) and why.

d) Test Cases

2) A tar ball containing:

a) Source code of all the files along with the makefile or equivalent compilation/execution instructions.

b) Test cases - input files tested on.

c) Output for the test cases used.

d) Some documentation of the organization of the code and its functionality with respect to different modules etc.

Bonus parts (7 % extra for this phase)

Write an NFA simulator for your NFA; the simulator works as follows: it takes the NFA table and the string to be processed as inputs. It then implements the actions of the NFA table; when necessary it clones the NFAs and when they reach a dead end kills them. When one or more of the clones accepts the input strings, it generates the corresponding token and returns it to the calling program.

You will use your generated NFAs from the base phase above for testing the NFA along with the token strings. **Please report:** the total number of transitions made by the NFA vs the ones made by the DFA for a given input. The **deliverable** for this part is:

- NFA simulator
- Inputs: NFA Table and File of strings
- Output: Tokens
- Output: The total number of transitions performed by the simulator (for all the clones), the maximum number of clones encountered during simulation, comparison (graph) of NFA vs DFA transitions for the same inputs.