# CSC254/454 Assignment 2: Tokenizer

September 2018 (9/15 Update)

## 1 Introduction

This assignment is lexical scanning — the first step in compilation. Scanning identifies tokens from literals. You are required to build your own tokenizer and use it to process programs in a subset of the C language. (Note: do **NOT** use lex to generate your tokenizer automatically and do **NOT** use regular expressions.) In particular, it should rename all user defined identifiers, i.e. variables and function names, to end with our course number (_csc254). The tokenizer should report an error if a token does not conform to the token-language definition (below). If the input program is correct, the output should also be a correct program with the same behavior. (E.g., if the input to your tokenizer is a program that can be compiled by gcc, then the output better be too!). Your program should be one that takes a path to a valid C program as an argument and outputs a C program transformed as described above or halts and reports an error. See the following examples:

Input file: int main() { int i; i=0; printf("%d\n", i);}
Output file: int main() { int i_csc254; i_csc254=0; printf("%d\n", i_csc254);}

Input file: int main() { int 2010i; printf("%d\n", 2010i);}
Abort scanning and report illegal token "2010i".

## 2 The Token Language

For this and later projects on compilation, we use a subset of the C language. There are only integer variables and arrays and no external library calls except for I/O. There are no pointers except for reading the input. In this project, we are concerned with the token definition of the language. See the token specifications below. Note that tokens are separated either by white spaces (including tabs and line returns) or by symbols such as operators and semicolons.

### 2.1 Token Language Specification

The tokens include ⟨identifier⟩, ⟨number⟩, ⟨reserved word⟩, ⟨symbol⟩, ⟨string⟩, and ⟨meta statement⟩. They are defined as regular expressions as follows:

⟨letter⟩ → a | b | ... | y | z | A | B | ... | Z | underscore

⟨identifier⟩ → ⟨letter⟩ (⟨letter⟩ | ⟨digit⟩)*

⟨digit⟩ → 0 | 1 | ... | 9

⟨number⟩ → ⟨digit⟩ +

⟨reserved word⟩ → int | void | if | else | while | return | continue | break | scanf | printf | main | read | write

⟨symbol⟩ → left_parenthesis | right_parenthesis | left_brace | right_brace | left_bracket | right_bracket | comma | semicolon | plus_sign | minus_sign | star_sign | forward_slash | == | != | < | <= | > | >= | equal_sign | double_and_sign | double_or_sign | and_sign | or_sign (The symbols are: ( ) { } [ ] , ; + - * / == != < <= > >= = && || & | )

⟨string⟩ → any string between (and including) the closest pair of quotation marks.

⟨meta statement⟩ → any string begins with '#' or '//' and ends with the end of line ('\n').

Note: letters and digits should be part of the same token unless they are separated by a while space character or a symbol.

# 3 Tokenizer Design

For this project you must use Ruby or Python (Ruby is preferred). Note that your grade will be based on correctness. Although you may organize your code however you like, your tokenizer should be executable on cycle1 as such: "./tokenizer.extension input_file.c". Also, note that the tokenizer is the first step in a series of projects through which you will develop a calculator and a compiler. In later projects, you will reuse your tokenizer (if your tokenizer is not working well when the next assignment is given, please refer to the solution that will be provided once this assignment is complete).

A modular implementation can use two classes—the token and the scanner classes. A token in the token class has information such as type and name (for identifiers) or value (for numbers). The constructor of the scanner class takes the input file name as an argument. It provides an enumerator method that yields the next token when called. The following shows such an interface for the scanner class in Ruby syntax:

```
class Scanner
    def init( file_name )
        ... ...
    end

    # return the next token (as a Token object) or EOF
    def next_token
        ... ...
    end
end
```

# 4 Division of Labor and Writeup

This assignment should be done **alone** meaning you may not pair up for this assignment. Every student in the class is required to submit his/her own source code and report (See Below).

Write a brief README in no more than **two** pages (plain text is preferred). In this README provide a brief but comprehensive explanation of how your program works. If your program does not work, explain what you attempted, and what you think needed to be done in order to get a working project.

Be sure your write-up (README file) describes any features of your code that the TAs might not immediately notice.

# 5 Turn In and Due Date

On a csug machine, put your write-up in a README.txt or README.pdf file in the same directory as your code, and (while still in that directory) run the script ~cs254/bin/TURN_IN to turn in. The script will package the contents of the directory (and any subdirectories) into a bundle and send it to the TAs for grading (so clean up any mess you might have in the directory first).

The due date for this assignment is: **Sunday September 16, at 11:59pm; no extensions.**