PA1 -- Lexical Analysis

---

1. Description

For this programming assignment, you will implement the first phase of the compiler -- lexical analyzer (scanner/lexer). The lexer you build should be able to scan through the source code, recognize and return the meaningful **Token,** which will be used in the later stages of the compiler.

You will build the lexer using **flex**(The Fast Lexical Analyzer)**.** flex is a handy tool to generate lexical analyzer. Before you start this assignment, please go through the document, ***Flex_In_A_Nutshell***, that explains how to use flex with some nice examples. The document is really helpful, and after you try and play with flex following the document, you should be ready to start working on this assignment.

2. Starter Code

Makefile	used to build the lexer
errors.h / .cc	class that defines error reporting methods
location.h	class that defines type for token location
lexer.h	type definitions and prototype declarations for lexer
lexer.l	**flex input file**
samples	directory contains test input files

When you are going through the starter code, please read the comment as well. The comments are really helpful in explaining the structure of the code.

In this assignment you only need to modify **lexer.cc**. You need to implement the pattern and action for all the token types which are defined in lexer.h. The token types will be defined in bison input file(for parser) for the next assignment. Because we do not have bison input file for this assignment, we will temporarily define them in lexer.h. For most of the toke types, it is easy to know what do they mean by looking at their names, except for the following tokens:

T_EQ	Stands for “==”
T_Equal	Stands for “=”
T_Identifier	Stands for all the valid java variable or function names

Also there is another method that is quite useful: **DoBeforeEachAction()**

This method will be executed before each action is executed. Take a look at its implementation and try to understand how the location of each token is updated. You can add print statement in this method to help you debug your implementation.

3. Example

There are 2 rules which have been implemented for **T_Void** and **T_BoolConstant.** You can use it as an example on how to implement rules for other tokens. You can make and execute against the given test input file: **test1.java** by doing:

```
%  make
% ./lexer < samples/test1.java
```

And then you should get the following output:

```
%true         line 1 cols 1-4 is T_BoolConstant (value = true)
% false        line 1 cols 6-10 is T_BoolConstant (value = false)
%
%void         line 1 cols 12-15 is T_Void
```

Two mistakes of the output you might notice:

1. The extra space before *false* and the extra newline before void. This will happen because flex will directly output the source code if it cannot match it to any rule defined in lexer.l
2. The line number and column number of void is wrong. This is also because of missing rules for whitespace and newline. You need to implement rules for them in order to get the correct line number and column number for void.

The correct output can be found in **test1.out**:

```
%true         line 1 cols 1-4 is T_BoolConstant (value = true)
%false        line 1 cols 6-10 is T_BoolConstant (value = false)
%void         line 2 cols 1-4 is T_Void
```

4. Testing your work

We will provide all of the test cases and the corresponding output which you can find in the **samples** folder. You can also come up with your own test cases. You can test your work as the following:

```
% ./lexer < samples/test1.java > samples/test1.myoutput
% diff samples/test1.out samples/test1.myoutput
```

5. Grading

You can go to any teaching staff's lab hours and ask for a check-off when you are done with your assignment. You will be asked to run your assignment against the provided test cases, and we will check the output of your assignment. We will also look at your code, and you need to describe how did you approach the assignment to the teaching staff.