**COP 5556 // Project #2 // Fall 2018**

|  |  |
| --- | --- |
| **Date Assigned:** | September 7, 2018 |
| **Date Due:** | September 17, 2018 |

# Submission Format

You will submit a soft copy of your solution using e-Learning ( <http://elearning.ufl.edu> ) by the end of the day ( 23:59 / 11:59 PM ) on the assigned date ( September 17 ). Save your solution as a **jar** file and name the file **p2** ( p2.jar ).

# Assignment

At the top of every solution file you submit this semester include: your name, the assignment number, and the date due. You will implement your solution within the PLPScanner.java class file. Any additional testing you would like to perform can be included by implementing additional method test cases [see the final three comments / two methods in PLPSCannerTest.java].

# Description

Using the lexical analysis given here [this analysis is very similar to your P1 solution with some additions and course-wide consistency], implement a Java solution that will tokenize a string.

Note, you are not parsing in the project, you are identifying valid tokens and constructing a representation of the tokens found.

# Lexical Analysis

InputCharacter any 7-bit ASCII character.

LineTerminator LF | CR | CR LF

*LF is the ASCII character also known as “newline”. The Java character literal is ‘\n’.*

*CR is the ASII character also known as “return”, The Java character literal is ‘\r’.  
CR immediately followed by LF counts as one line terminator, not two.*

Input (WhisteSpace | Comment | Token)\*

WhiteSpace SP | HT | FF | LineTerminator

*SP is the ASCII character also known as “space”. The Java char literal is ‘ ‘.*

*HT is the ASCII character also known as “horizontal tab”. The Java char literal is ‘\t’.*

*FF is the ASCII character also known as “form feed”. The Java char literal is ‘\f’.*

Comment %{ ( (% NOT({) ) | NOT(%) )\* %+}

Comments will be identified and discarded [you do not need to keep them as a token].

Token Identifier | Keyword | Literal | Separator | Operator

Identifier IdentifierChar but not a keyword or a Boolean literal

IdentifierChar IdentifierStart IdentifierPart\*

IdentifierStart UnderScoreStart IdentifierStart |A..Z IdentifierPart | a..z IdentifierPart

UnderScoreStart \_ UnderScoreStart | \_

IdentifierPart A..Z IdentifierPart | a..z IdentifierPart | Digit IdentifierPart | \_ IdentifierPart |

Literal IntegerLiteral | FloatingPointLiteral | BooleanLiteral | StringLiteral | CharLiteral

IntegerLiteral 0 | NonZeroDigit Digit\*

FloatingPointLiteral IntegerLiteral . Digit Digit\*

StringLiteral " ASCII\* "

CharLiteral ' ASCII '

ASCII ASCII\_CHAR |

NoneZeroDigit 1 .. 9

Digit NonZeroDigit | 0

BooleanLiteral true | false

Separators ( | ) | [ | ] | ; | , | { | }

Operators < | > | <= | >= | - | + | \* | / | % | ! | \*\* | == | =

Keywords print | int | float | boolean | char | string | sleep | if

* If an illegal character is encountered, your scanner should throw a LexicalException. The message should contain useful information about the error. The contents of the message will not be graded, but you will appreciate it later if it is descriptive.
* If a numeric literal is provided that is out of the range of the Java equivalent of that type, then your scanner should throw a Lexical exception. The contents of the error message will not be graded, but you will appreciate it later if it is descriptive.
* Use the provided PLPScanner.java and PLPScannerTest.java as starting points. The scanner will be part of all the subsequent assignments. Errors may cause failures in subsequent assignments. A careful job now, including a complete test suite developed now will help you later.

# Evaluation

* This code must remain in package cop5556fa18 (case sensitive): do not create additional packages.
* Names (of classes, method, variables, etc.) in starter code must not be changed. You may, of course, add additional variables, methods, enums, etc.
* Your code should not import any classes other than those from the standard Java distribution.
* **Make sure that sources are included in the jar file**​. Many IDEs (including Eclipse) do not do this by default.

[A quick reference for how to export a jar file from Eclipse](https://www.cs.utexas.edu/~scottm/cs307/handouts/Eclipse%20Help/jarInEclipse.htm)

If you are not using Eclipse, check ​[Creating a JAR file](http://docs.oracle.com/javase/tutorial/deployment/jar/build.html)

* To ensure that we will be able to compile and run your submission: upload your jar file
* to one of the UF CISE servers, e.g. storm.cise.ufl.edu, uncompress it, and run it from the command line. (See ​<https://www.cise.ufl.edu/help/access/remote>​ for information about remote access to CISE machines) Instructions:
  + - Copy/upload your file to a CISE server. Suppose your CISE ID is ​*username*​, the following instruction will upload the ​*HW1.jar* ​to your home folder on cise server:

scp /my/path/to/HW1.jar [username@storm.cise.ufl.edu:~/](mailto:username@storm.cise.ufl.edu:~/)

* + - Uncompress jar file:

jar xf HW1.jar

* + - Compile

javac -cp .:/usr/share/java/junit4.jar:/usr/share/java/hamcrest-core.jar cop5556fa18/\*.java

* + - Run Junit test from command line

java -cp .:/usr/share/java/junit4.jar​:/usr/share/java/hamcrest-core.jar org.junit.runner.JUnitCore cop5556fa18.PLPScannerTest

* **Make sure that your jar file has the same directory structure as the original one that you downloaded from Canvas. If it doesn’t, the grading script will fail resulting in a grade of 0.**
* Note that you can try this upload before you are finished with the assignment, giving you time to figure out what is wrong if you have problems.
* **No matter how your program runs on your own machine, if it fails to compile/run on the CISE server (storm or thunder) with the aforementioned instructions, your submission will get a zero grade, and there will be no regrade. So double check before your submission.**
* If you are non-CISE student who does not have a CISE account, see https://www.cise.ufl.edu/help/account​ for instructions to get one.
* If you use Eclipse, we suggest creating a project and importing the jar files (eg. HW1.jar) provided by each assignment into the project. After completing your work on the source files (keep all source files within the package cop5556fa18), you can export the package cop5556fa18 as a jar file for submission (remember to select the option of including source files in the jar package), so that it will have the same directory structure as the original jar file.

**[From P1, the following content is still relevant and useful for evaluation]**

In the language grammar we have defined, the structure of a standard programming language such as Java or C++ has been followed with few modifications. For example, we are constructing if statements with a conditional expression within following ()s and a statement block contained within {}s. We have used the standard data types (noting the lower ‘s’ in string) and their corresponding literals.

Here are some test cases that will help you evaluate your grammar’s ability to parse the language described above. Note, within the valid cases given no compilation is required. Recall, you are just creating the context free grammar and are not implementing compilation. Therefore, parsing individual lines or statement blocks is required. However, identifying consistency within or between lines is not. For example, if a variable is not declared it can still be parsed by your grammar because we have not begun considering how to manage known variables and a symbol table. Another example is given below where a string variable is compared to an integer value. Clearly, this will not be something our programming language will allow. However, at this time, we are merely parsing the statements and not evaluating program correctness.

In addition, you can consider error cases, however you know these cases cannot be parsed successfully by your grammar. For example, if an opening curly bracket/brace is listed, but a closing one is not, your grammar will not be able to parse the statement. Similarly, single quotes and double quotes will need to have their corresponding quote. Another example is the usage of operators. All of the operators we have described are binary and expect two operands. Therefore, placing two operators in sequence without an operand on each side will not be something you can parse.

# Valid [Parse-able] Cases

boolean a;

int b, x, y;

char  c;

double d, t;

string e;

a = true;

B = 10;

c = ‘a’;

d = 23.2;

e = “Hello, World!”;

a = 1+2;

d = 2.12 – 1;

a==3;

a  = 1 + 2 \*4.5;

t = (1+2) \* 4.5;

t = (((4-2)\*5.6)/3)+2;

t = 4 - 2 \* 5.6 / 3;

int score = 100;

if ( a==100 ){

print (“Value of a is 100”);

}

if ( score > 100 ) {

print( a );

print(B);

print(score);

}