Assignment 2: Writing and Verifying JML Specifications

Overview

In this assignment, you will be asked to write Java Modelling Language (JML) method specifications for the TokenStream class introduced in Assignment 1. Once again, you will use Eclipse to modify, build, debug, and run a simplified JavaScript lexer. However, this time, you can use an Eclipse plugin known as ESC/Java2 to statically check your specifications.

Extensions to TokenStream

The JavaScript lexer you will use in this assignment supports a couple of additional features that were absent in the previous assignment. These features are as follows:

1. Lexing of octal and hexadecimal numbers
2. Lexing of keyword identifiers

In JavaScript, octal numbers (i.e., base 8) are written with a '0' prefix. For example, the decimal number 26 is written as 032 in octal. Octal digits range from 0 to 8. Hexadecimal numbers (i.e., base 16), on the other hand, are written with the prefix '0x' or '0X'. For example, the decimal number 177 is written as 0xB1. Hexadecimal digits range from 0 to 9 and A to F (the letters could also be lower case). The function stringToNumber() converts the string representation of a decimal, octal, or hexadecimal number into a decimal value of type double.

Keywords are a special type of identifiers that are part of the JavaScript language's syntax. These keywords are reserved and may not be used as identifiers of variable names, object names, function names, etc. TokenStream uses the stringToKeyword() method to determine if a certain string corresponds to a keyword. In this assignment, we are only interested in the following keywords: FUNCTION, IF, ELSE, TRUE, FALSE, and RETURN.

When you download the code for this assignment, you will see that the specifications for certain methods of TokenStream - including stringToNumber() and stringToKeyword() - have already been provided. Your task will consist of both identifying **two** bugs introduced in the extended TokenStream code and writing method specifications for **five** functions.

Instructions

1. Read the tutorial from Lab 2, which tells you how to download, configure, and use ESC/Java2. This tutorial also explains some of the basics of JML which you will need for this assignment.

2. Download the file eece310_assn2.zip from the course website. ***IMPORTANT*: Make sure you use the code from eece310_assn2.zip. Do NOT reuse code from Assignment 1, as most of the source files have been modified.**
3. Once you've unzipped the above file, add the project to Eclipse (check Lab 1 if you've forgotten how to do this) and switch to the Verification perspective (see Lab 2 for instructions on how to do this)
4. The project files are modified versions of the files from Assignment 1 (so once again, use the code provided for this assignment, not the code from Assignment 1). In particular, TokenStream now supports the features previously described, and methods have been annotated with JML specifications.
5. Run the Simplify checker of ESC/Java2. You will notice that ESC/Java2 will identify two errors in TokenStream.java. The reason is that TokenStream.java contains two bugs. Your task is to find and fix these two bugs. Once you've fixed these bugs, the ESC/Java2 errors should disappear (ESC/Java2 may still produce errors related to library code, which may not have been annotated with JML and which you will have no control over. You can ignore these errors and focus only on ESC/Java2 errors in TokenStream). In addition, the program should generate correct tokens for the files JavaScriptCode1.js, JavaScriptCode2.js, and JavaScriptCode3.js. You are advised to complete this step before moving on to writing specifications. **NOTE: The bugs are found exclusively in TokenStream.java. Do not modify any of the other source files.**
6. Write specifications for the following methods in TokenStream. You should write your specifications based on the descriptions of the methods given below, as well as the implementation of the methods. (search EECE310_TODO in TokenStream.java to find out which specifications you have to write).

ungetCharIgnoreLineEnd(): Moves back one character from the source buffer (containing the source code) and places this character in the "unget" buffer. The ungetBuffer must not be null. By the end of this method's execution, the buffer cursor should be decremented by one, and the unget cursor should be incremented by one.

getTokenLength(): Returns the length of the most recently scanned token. This method assumes the index of the end position of the token (tokenEnd) is greater than the index of the beginning position of the token (tokenBeg). The value returned must be positive.

getOffset(): Determines the offset into the current line (i.e., how far along the cursor is positioned relative to the very first character of the current line in the source code). If the current character is an end of line character (which means the value of lineEndChar is non-negative), the method assumes the index of the current source cursor is greater than the index at the start of the line. Otherwise, if the current character is not an end of line character (which means the value of lineEndChar is -1), the method assumes that the index of the current source cursor is greater than or equal to the index of the start of the line. The offset returned must be non-negative.

fillSourceBuffer(): Reads characters from the source code (via the source reader) into the source buffer until the source buffer becomes completely filled, and returns true if

the read is successful (returns false otherwise). It is assumed that both the source buffer and the source reader are not null.

addToString(): Adds a character into the string buffer, which is filled when the token is a number or an identifier. This method assumes that an array has already been allocated for string buffer. By the end, the marker for the top of the string buffer must be incremented.

If you wish, you may use ESC/Java2 to verify the correctness of your specifications, although you are not required to use this tool. (Note, however, that adding an assignable clause to the specification for addToString() may cause the verification counter for the getToken() method to become too large, as the latter calls the former many times. You may take out the assignable clause for addToString() while verifying your specifications, but make sure this clause is present when you hand in your code).

Also, please make sure you **do not** delete the “signals_only” and “assume” clauses. These prevent ESC/Java2 from complaining about exceptions (which you will learn later in the course).

Deliverables

You will submit only your modified version of TokenStream.java (complete with the bug fixes and the specifications for the five methods). We will test your code on the following inputs (note that these inputs are **not** the same as the ones from Assignment 1).

- JavaScriptCode1.js - correct tokens are found in JavaScriptCode1_CorrectTokens.txt
- JavaScriptCode2.js - correct tokens are found in JavaScriptCode2_CorrectTokens.txt
- JavaScriptCode3.js

The first two inputs (and their corresponding outputs) are made available to you, but the third input is not. If you have identified and corrected the bugs properly, your code will generate the correct tokens for all three inputs.

Evaluation

Your mark will be based on the correctness of your code and the correctness of your specifications. The breakdown is as follows:

Correct output for JavaScriptCode1.js - 5%
Correct output for JavaScriptCode2.js - 5%
Correct output for JavaScriptCode3.js - 10%
Correct specifications (for each method) - 16% x 5 = 80%

Partial marks may be given for semi-correct specifications.

Submission Guidelines

Create a **zip file (with extension .zip)** containing **only** your modified code for TokenStream.java (i.e., the zip file should not contain code for other classes). Submit the zip file as mentioned in the email guidelines.