Assignment 1: Getting to Know Eclipse

Overview

The main purpose of this assignment is to introduce you to Eclipse, an integrated development environment (IDE) used mostly to write Java applications. The secondary purpose is to give you some hands-on experience with debugging applications.

All subsequent assignments in this course will require you to read, write, and modify Java code; thus, it is imperative that you learn right from the start how to properly organize Java files and how to debug erroneous Java code.

Preliminaries: JavaScript

JavaScript is an interpreted, weakly typed programming language most commonly used in web application development (i.e., client-side JavaScript), where its main purpose is to provide interactivity. This programming language includes a rich set of features, designed in part to simplify its interaction with the Document Object Model (DOM), which is a tree-like structure representing the elements found in a webpage.

The JavaScript programming language follows what's called the ECMAScript standard, defining the syntax of the language. Throughout this course, you will be developing a simplified version of a JavaScript parser.

For more information on JavaScript, you can visit the following links: link, link

Preliminaries: Background on Lexing

Lexing is the process of generating tokens when compiling or interpreting code. These tokens must be generated in order for code to be parsed and, eventually, compiled/executed. Tokens, in essence, are numerical identifiers representing symbols that are meaningful to the syntax of the programming language. For example, consider the following JavaScript (JS) expression:

```
x = foo(a,b);
```

When the above JS code is lexed, the tokens generated will be as follows:

```
NAME: x
ASSIGN
NAME: foo
LP
NAME: a
```

COMMA
NAME: b
RP
SEMI

Note that ASSIGN refers to the "=" (i.e., assignment operation), LP refers to the left parenthesis, RP refers to the right parenthesis, and SEMI refers to the semicolon. In addition, variables and function names are *identifiers*, and are assigned the token NAME.

Your task

In this assignment, we provide you with a token stream generator for JavaScript code. This program, which generates tokens as described above, will contain **four bugs**. Your task is to identify these four bugs using Eclipse debugging techniques and fix the token stream generator code.

Instructions

1. Download and install Eclipse at http://www.eclipse.org/downloads/
2. Read the Eclipse tutorial (Lab 1). The tutorial can be found in the course website. This tutorial will teach you the basics of organizing Java packages in Eclipse, as well as debugging in Eclipse.
3. Download the file eece310_assn1.zip. Once you've unzipped the file, add the folder eece310_assn1 as a project to Eclipse. (see "How to Add Existing Projects to Eclipse" in the tutorial)
4. Once you have set up the Eclipse project browse the code. To this end you can click on the triangle beside the project name under the "Package explorer" pane to expand the project's contents. Show the contents of the folder src and the package org.mozilla.javascript. You will notice that this package contains six files, described below:

Token.java: This class contains an enumeration of all JavaScript tokens. Recall that tokens are nothing more than integers. It is, however, easier to refer to these tokens by name (e.g., Token.ADD, Token.ASSIGN, etc.) rather than by number.

Kit.java: For the purpose of this assignment, this is a helper class containing functions that will assist in the lexing.

ObjToIntMap.java: For the purpose of this assignment, this class helps us associate the NAME token with its corresponding identifier name (i.e., variable name or function name).

UniqueTag.java: This is a helper class to ObjToIntMap containing tags for special object values.

TokenStream.java (*Note: this is the only class you'll have to modify for this assignment)*. This class is where the lexing takes place. When a new TokenStream object is created, a file reader containing the JavaScript code to be lexed is passed to its constructor (the file reader passed to the constructor must be stored in the sourceReader member of the TokenStream class). Once the TokenStream object is created, each call to the getToken() method generates the next token. In the JavaScript example given above, the first call to getToken() generates the token NAME, the second call generates the token ASSIGN, the third call generates NAME, the fourth call generates LP, etc.

How does getToken() know what the next token is? The answer is that it parses the JavaScript code character by character (using the method getChar()), and determines if any combination of characters forms a syntactically meaningful symbol. For instance, if the character '+' is encountered, getToken() first peeks at the next character. If the next character is another '+', then an increment operator (i.e., ++), represented by token INC, is returned. If the next character is a '=', then an addition assignment operator (i.e., +=), represented by token ASSIGN_ADD, is returned. Otherwise, no meaningful symbol goes with the '+' character, so an addition operator, represented by token ADD, is returned.

Identifiers and numbers are also identified character by character; however, they are a little trickier to lex. It is your task to read through the TokenStream.java code and try to understand how these are lexed.

GenerateTokenStream.java: This class contains the main function, which creates a TokenStream object, calls getToken() repeatedly in a loop to generate tokens one by one, and prints the generated tokens to standard output. When the main function is run, you will be prompted to enter the name of the JavaScript file to parse. Have a look at the files JavaScriptCode1.js, JavaScriptCode2.js, and JavaScriptCode3.js; these are the JavaScript files for which you need to generate correct tokens in this assignment.

When you try running the lexer with the above JavaScript files, you will notice that a null exception will be thrown. This is caused by one of the bugs introduced in TokenStream.java which you must identify. The correct tokens for the JavaScript files are found in JavaScriptCode<num>_CorrectTokens.txt (where <num> is 1, 2, or 3).

Please note that only TokenStream.java contains bugs. As a result, **only TokenStream.java should be modified, and all other files should remain unchanged.**

Deliverables

You will submit only your corrected version of TokenStream.java. We will test your code on the following inputs:

- JavaScriptCode1.js - correct tokens are found in JavaScriptCode1_CorrectTokens.txt
- JavaScriptCode2.js - correct tokens are found in JavaScriptCode2_CorrectTokens.txt
- JavaScriptCode3.js - correct tokens are found in JavaScriptCode3_CorrectTokens.txt
- JavaScriptCode4.js

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

HINT: JavaScript allows unicode characters in variable names. This might help you find one of the bugs.

Evaluation

Your mark will be based on the correctness of your code, as follows:

Correct output for JavaScriptCode1.js - 20%
Correct output for JavaScriptCode2.js - 20%
Correct output for JavaScriptCode3.js - 20%
Correct output for JavaScriptCode4.js - 40%

Submission guidelines

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