First Laboratory Assignment

1. Overview

The goal of this lab is to implement a lexical analyser for a simple programming language called PL/3007. In later labs, we will more fully explore the syntax and semantics of this programming language, and by the last lab you will have a full, working compiler from PL/3007 to Java bytecode.

In every lab, you will be provided with a code template (downloadable as a zip file from the course website) to get you started. You will then need to fill in the template to implement the relevant functionality of the compiler. The code template contains comments telling you which parts of the code you need to change, and which ones you should not touch.

Every template also comes with a unit test suite containing one or a few tests (suggestion: google search for "unit test" if you are not very sure what it means). At first, the test suite will fail, since the functionalities tested are not yet implemented. You should add your own tests and run the test suite frequently to make sure that your implementation behaves correctly, and that your changes do not lead to regressions.

Your assignment will be graded by the instructors using automated unit tests, so it is imperative that the code you hand in compiles and behaves correctly.

2. Setup Instructions

Download the file `lab1.zip` from the course webpage. Unzip it somewhere on your hard drive to reveal a folder **lab1**. Start Eclipse (the Eclipse J2EE Mars edition) and choose this folder **lab1** as your workspace. It contains a single project **Lab1**, which contains all the code you need to complete the project.

The code contained in this project is structured as follows:

- folder **src** contains the source code:
  - package **frontend** contains the code for the lexer and some helper code:
    - file `Token.java` contains a class defining the token type used for this lab; you do not need to modify it, but you should read and understand it
    - file `lexer.flex` contains the source code for the lexer; to complete this lab assignment, you will need to complete the code of the lexer as indicated by the comments in `lexer.flex` and the problem description below
  - package **test** contains the unit test suite in file `LexerTests.java;` to test your token specifications in `lexer.flex`, you should add more tests here
- folder **lib** contains third-party libraries
- `build.xml` is an Apache Ant build file that we will use to generate the lexer from `lexer.flex`

At first, Eclipse will display a build error about project **Lab1** missing a required source folder. This is expected, since we have not yet generated the lexer from `lexer.flex`.

To do so, click on the downward black triangle next to the "External Tools" button (a green circle with a white triangle in it and a red toolbox next to it). Choose "Lab1 build.xml" from the menu. This will run JFlex on `lexer.flex` and generate a file `Lexer.java` in package **lexer** in folder **gen**. You need to perform this step every time you have changed `lexer.flex`. Now, Eclipse should not report any build errors any more.

Next, we run the unit test suite. To do this, click on the downward black triangle next to the "Run Configurations" button (to the left of the "External Tools" button mentioned above), and choose "LexerTests" from the menu. This will bring up the JUnit pane on the left, where the results of the unit tests are displayed. Currently, there is only one test, and it fails: not too surprising, since we have not implemented any real lexer functionality yet.

3. Problem Description

3.1. Tokens for PL/3007

We will now give precise English descriptions of the kinds of tokens that may appear in a PL/3007 program. It is your task to complete the JFlex specification in `lexer.flex` to recognise these tokens, and return appropriate instances of type Token that encodes the **type**, **lexeme** and **position information** of every token.

As with many programming languages, there are five kinds of tokens in PL/3007: keywords, punctuation symbols, operators, identifiers, and literals.

PL/3007 has fourteen keywords: boolean, break, else, false, if, import, int, module, public, return, true, type, void and while. Note that keywords are case sensitive; e.g., Boolean is not a keyword, but an identifier.

There are eight punctuation symbols: the comma (“,”), the left bracket (“[”), the left curly brace (“{”), the left parenthesis (“(”), the right bracket (“]”), the right curly brace (“}”), the right parenthesis (“)”), and the semicolon (“;”).

Furthermore, PL/3007 has eleven operators (a subset of the operators of Java): the division operator (“/”), the equals operator (“==”), the assignment operator (“=”), the greater-or-equal operator (“>=”), the greater-than operator (“>”), the less-or-equal operator (“<=”), the less-than operator (“<”), the minus operator (“-”), the not-equals operator (“!=”), the plus operator (“+”), and the times operator (“*”).

An identifier in PL/3007 is a sequence of one or more characters. The first character must be a letter. Each subsequent character in the sequence must be a letter, a digit or an underscore.

Finally, there are two kinds of literals: integer literals and string literals.

Integer literals consist of a sequence of one or more decimal digits. Note that integer literals are unsigned: positive or negative sign are considered to be unary operators. Also note that superfluous leading zeros are allowed.

String literals in PL/3007 are enclosed by double quotes, and may contain zero or more characters (except the double quote and the newline character). For example, “a\”” is not allowed because it contains a double quote character. You are advised not to introduce other lexical states to process string literals in your `lexer.flex` because our test suite assumes you do not use special state for string literals (if you do not know what lexical states are, don’t worry – we did not introduce it in our lectures).

Apart from the above tokens, PL/3007 source code may contain whitespace characters (blank, tab, newline, return), which are skipped by the lexer. For the moment, we do not specify a syntax for comments, and your lexer does not need to handle them.

There is also a pseudo-token EOF indicating the end of input. The template code already handles this, so you do not need to worry about it.

3.2. Token.java

The enum type Type in class Token contains one enum constant for every token type in PL/3007 as described above. Your lexer should return, for every token, an instance of class Token that has the correct token type and lexeme, together with the source position of the token. This is done by calling the constructor of the Token class. For example, when keyword “module” or identifier “average” is recognized by the lexer, the corresponding token may be created by calling the constructor Token():

```
new Token(MODULE, yyline, yycolumn, yytext());
new Token(ID, yyline, yycolumn, yytext());
```

yyline and yycolumn together specify the text position of the token in the source file. yytext() gives the matched text.

3.3. lexer.flex

`lexer.flex` specification consists of three parts, divided by %%:

- usercode,
- options and declarations
- lexical rules.

You do not have to change anything in the usercode section. This section is copied verbatim into the beginning of the source file of the generated lexer before the scanner class is declared. This is the place to put package declarations and import statements.

In the options and declarations section, we see these options:

```
%public
%final
%class Lexer
```

```
%function nextToken
%type Token
%unicode
%line
%column
```

`%public` and `%final` make the generated class public and final. `%class Lexer` tells JFlex to give the generated class the name "Lexer" and to write the generated code to a file "Lexer.java". `%function nextToken` causes the scanning method to get the name "nextToken". `%type Token` causes the scanning method to be declared as returning values of the type "Token". Actions in the specification can then return values of Token as tokens. `%unicode` defines the set of characters the scanner will work on. `%line` switches line counting on (the current line number can be accessed via the variable yyline) and `%column` switches column counting on (current column is accessed via yycolumn).

The code included in `%{ ... %}` is copied verbatim into the generated lexer class source. Here you can declare member variables and functions that are used inside scanner actions. Fill in the code in the two methods

```
private Token token(Token.Type type)
private Token token(Token.Type type, String text)
```

which can be called in the lexer rule actions to create a token without providing all 4 parameter values as in Section 3.2. For example, with these two methods, you only need to use

```
return token(MODULE);
return token(ID, yytext());
```

in the lexer rule actions to return the two tokens respectively. If you do not use these two methods, then remove them and do something like

```
return new Token(MODULE, yyline, yycolumn, yytext());
return new Token(ID, yyline, yycolumn, /*here put the string with " and " removed from yytext()*/);
```

in the lexer rule actions. You will also need to change "return token(EOF)" in the last line of lexer.flex accordingly.

Note that for string literals, yytext() includes the beginning and ending double quotes but the lexeme of the token should exclude them. So the first and last characters of yytext() should be excluded. You may want to use the substring() method in Java to do this.

3.4. Test Suite

As mentioned above, it is very important that you write your own unit tests to test your lexer. Class LexerTests contains a utility method runtest that makes it very easy to do so: this method accepts as its first argument a string, and as its remaining arguments a list of tokens that the lexer should split this string into. When invoked from inside a unit test, runtests invokes the lexer on the string, and then checks that it returns the right sequence of tokens. If any of the tokens do not match, an assertion failure is raised, which is displayed as a failed test by Eclipse.

For example, here shows the invocation of runtest for three unit tests in LexerTests:

```
runtest("module false\n\treturn while",
            new Token(MODULE, 0, 0, "module"),
            new Token(FALSE, 0, 7, "false"),
            new Token(RETURN, 1, 1, "return"),
            new Token(WHILE, 1, 8, "while"),
            new Token(EOF, 1, 13, ""));

runtest("\"\"\"",
            new Token(STRING_LITERAL, 0, 0, ""),
            (Token)null,
            new Token(EOF, 0, 3, ""));

runtest("\"\\n\"",
            new Token(STRING_LITERAL, 0, 0, "\\n"),
            new Token(EOF, 0, 4, ""));
```

The first call to runtest() states that when the lexer analyses the string "module false\n\treturn while", it should produce five tokens: a token of type MODULE, followed by a token of type FALSE, then a token of type RETURN, a token of type WHILE, and finally a token of type EOF, each with the appropriate line and column information (which are given as the second and the third arguments to the Token constructor), and lexeme (the fourth argument to the constructor). The line number and the column number in the source both start from 0. Each character occupies one column in the line. Note that escaped characters like "\n' or "\t" also occupy one column each. After each "\n", the line number increases by 1 and column number restarts from 0.

The second call to runtest() tests a string with 3 double quote characters. The first two double quotes are recognized as an empty string literal. The lexer is expected not to recognize the third double quote as a valid token so it is expected to return a null token. Note that whenever the lexer is expected not to recognize something as a valid token, it is expected to return the null token.

The third call to runtest() tests a string literal of two characters which is a backslash followed by 'n'. What the lexer sees is a character stream: a double quote, a backslash, a character n and a double quote. This is recognized as a string literal.

4. Resources

The JFlex user's manual is available online at http://jflex.de/manual.html. It contains documentation on the JFlex specification syntax and some examples of its use, which you are welcome to study. You do not need to understand lexer states for this lab.

You also do not need to learn about how to invoke JFlex from the command line. This is taken care of by the Ant build script provided to you.

Table 1 provides a short summary of the syntax JFlex uses for regular expressions.

5. Administrativa

You should upload your **`lexer.flex`** to the Lab group site within 7 days after your lab session. Please upload only this file, not any of the other files in your workspace. **Also please do not submit a zipped file**.

Each team should only hand in *one* copy of the file.

Late submissions will be penalised. If you submit before the deadline and later find a mistake in your submission, you can resubmit, but only before the deadline (otherwise, it is a late submission).

Table 1: JFlex regular expression syntax

Syntax	**Meaning**
"foo"	the literal string foo
[fox]	a character class: either the character f, or the character o, or the character x; equivalent to "f" | "o" | "x"
[^fo]	an inverted character class: any character except f and o
[0-9]	a character range: any character between 0 and 9, both inclusive
\"	the double quote character
\\	the backslash character
{A}	use of a named regular expression
R*	zero or more occurrences of R
R+	one or more occurrences of R
R | S	R or S
(R)	same as R (used to enforce precedence)
.	any character