Exercise 7 – (Simplified) Java Compiler

OOP 2014

1 Goals

To exercise:

1. Implementing concepts learned in class (Regular Expressions)
2. Design & implement a complex system

2 Submission Details

- Submission Deadline: **Thursday, 20/06/2013, 23:55**
- This exercise will be done **in pairs**.
- Note: In this exercise you may use the following classes in standard java 1.7 distribution: Classes and interfaces from the *java.util*, *java.text*, *java.io* and *java.lang* packages (including sub-package). Specifically, you are required to use *java.util.regexp.Matcher* & *java.util.regexp.Pattern* (for working with regular expressions). You may also use **any class you want** under the **java 1.7 distribution** (this **does not** include code that requires installation of other jar files).

  If you choose to use tools not learned in class, you must explain in your README file why you chose to use them, and what are the downsides/upsides of using them compared to other alternatives (if such exist). Using an advanced tool without understanding and explaining the full effect of using it will result in point reduction.

3 Introduction

One of the main reasons for using regular expressions is to analyze text by assessing whether a given text string matches a pre-defined pattern. A common setting (which you are now ~~very~~ familiar with) that involves the evaluation of text against a given set of rules is the compilation of programming code. This process involves getting a text file as input, and examining each of its lines to see whether the file is a legal code file, as defined by the language specification.

In this exercise you will implement a java compiler. As java's syntax is a rather complicated, writing a java compiler is a highly challenging task. As a result, to make this task feasible for you, you will in fact implement an *s*-java compiler (*s* stands for *simplified*), which only supports a limited set of java features (see details below). In addition, your compiler will not have to run any code, it will only output whether or not the input file is a legal *s*-java file or not.

4 Input/Output

Your program will receive a single parameter (the *s*-java source file) and will run as follows:

java oop.ex7.main.Sjavac source_file_name

The output of your program is a single digit:

- 0 – if the code is legal
- 1 – if the code is illegal
- 2 – in case of IO errors (see Section 6.1).

Printing should be done using *System.out.println(output)*.

Other than the output digit described above, you are also required to print an informative error to the screen using *System.err(error_message)*, generally describing what went wrong. There is no specific format you have to follow here, as it won't be tested automatically. However, very uninformative error messages might result in point reduction. See examples below.

5 *s*-java specifications

s-java is a (very) simplified version of java. In the following, we describe its features.

5.1 General Description

- An *s*-java file does not interact with other files. That is, it cannot import data from or export data to other files. Each file stands on its own.
- Each *s*-java file is composed of two components: *member variables* and *methods*. *Members* are general variables or arrays (similar to java class members) and *methods* are general functions (similar to java class methods). Each method is composed of a list of code lines: defining local variables, giving new values to variables (local or members), calling methods, defining *if/while* blocks and returning (see more details below).
- Each *s*-java line ends with one of the following suffices: ';' (for defining variables, changing variable values, calling methods and returning) '{' (for opening method descriptions or *if/while* blocks) and '}' (for closing '{' blocks). In addition, empty lines (containing only white spaces or tabs) may appear in the code, and are to be ignored.
- *s*-java supports the single line java comments style (// ...). The single-line comment signs (//) may only appear in the beginning of a code line and this line should be ignored. Other comments style like multi-line comments (/* ...*/), javadoc comments (/** ...*/) and single-line comments appearing in the middle of a line are not supported by *s*-java. Any appearance of such comments is illegal and should result in printing the value 1.

5.2 Variables

s-java comes with a strict set of primitive variable types and arrays of such types (introduced in section 5.2.1); It does not support the creation of other types (e.g., classes, interfaces, enums).

The language supports the definition of two kinds of variables: members variables and local variables (defined inside a method – see below). Both types of variables are defined in the same way as in java:

type name = value;

where:

- Table 1 shows the legal *s*-java *types*.
- *name* is any sequence (length > 0) of letters (capital or minuscule), digits and the underscore characters (_). *name* may **not** start with a digit. *name* may start with underscore, but in such case it must contain at least one more character (i.e., '_' is not a legal name).
- *value* can be one of the following:
  - a legal value for *type* (see Table 1)
  - another (existing and initialized) variable of the *same type*

Type	*Description*	*Value Format*	*Examples*
int	an integer number	a number (positive, 0 or negative)	int number12 = 5; int num2 = -3;
double	a double number	an integer or a floating number (positive, 0 or negative)	double b = 5.2124; double c_3 = 2;
String	a string of characters	a string of any characters (inside double quotation marks)	String s = "hello"; String s2 = "a%#";
boolean	a boolean variable	*true*, *false*	boolean a = true;
char	a single character	any character (inside single quotation marks)	char my_char = 'a'; char g = '@';

Table 1: *s*-java types.

Notice the following:

- As described in the table above, a double is also an int therefore, this is legal:

  int a = 9; double b = a;

- Boolean may not be assigned with nor int or double values (or variables), only the reserved words "true" or "false".
- A variable may be declared with or without an initialization. That is, both *int a;* and *int b2 = 5;* are legal *s*-java lines.
- Other java modifiers (e.g., *final, static, public/private*) are **not** allowed (they are not part of the language syntax).
- Variable declaration must be encapsulated in a single line. For example, the following is **illegal**:
  - int a
    = 5;
- Multiple variables of the same type **are not** allowed to be define in a single line. For example, the following are **illegal**:

  - double a, b;
  - int a , b = 6;
  - char a = ‘c’ , b;

- There may not be two members with the same name (regardless of their types). For example, in the following, given the first line, the second line is **illegal**:

  - int a = 5;
  - String a = “hello”;

  However, a local variable can be defined with the same name as a member (regardless of their types). That is, in the previous example, had the first line been a declaration of a member, the second would have been legal if it was a declaration of a local variable.

- A variable can be assigned with a value after it is created.

  - int a = 5;
  - ...
  - a = 7;

  This applies both to members and local variables. Value assignment can only be done inside a method (both for members and local variables).

- A variable (member or local) *a* can be assigned with another variable *b* of the same type, but only after *b* was declared and initialized in the same scope or in any scope above it.

- A variable *a* can be assigned with an (initialized) member *c* of the same type, regardless of whether *c* was declared before or after *a*’s containing method.

- You may assume ‘\’ characters will not appear in String or char values.

- Operators are supported in *s*-java in the following manner:

  - Only +,-,*,/ operators are supported.
  - *s*-java supports the use of operators only on int and double variables.
  - Operators may appear in the right hand-side of any int or double assignment. (e.g., int a = -1+6;).
  - *s*-java supports only one operator in a single phrase (e.g., double a = 8.2+9;, int a = 8+9+0;).
  - Operators can handle numbers and initialized variables in the same manner (e.g., int a = b+5;).

5.2.1 Arrays

Arrays are an extension of the primitive types. All variables from type array should follow the same rules as the standard variables declared above (may be declared with or without an initialization, cannot conflict with other variable names, can be assigned after it’s created, etc.).

Defining an array in *s*-java is identical to the way it’s done in java:

$$\textit{type[] name} = \{value_1, value_2, \ldots, value_n\}; \text{ (initialized)}$$

$$\textit{type[] name}; \text{ (uninitialized)}$$

- *type*, *name* and $value_i$ should follow the rules mentioned above.
- Array with empty initialization is also supported by *s*-java (e.g., int a[] = {};).
- All values in an array should be of the same type and follow the same rules described for assigning a primitive values (can be an initialized variable, can be a result of an operator etc.)

double[] a = {8.2+9,0,1+2};

- s-java does not support any other array initialization such as: int[] a = new int[8];

Assigning into an array in *s*-java is also identical to the way it's done in java:

name[j] = value;

where :

- *j* must be a non-negative integer. It may also be any initialized integer variable or a result of an operator. (e.g., int[9+1] = 6;). There is no need to check the array boundaries, the following example is legal:

int[] a = 1,3,4; a[6] = -1;

- *value* should follow the same rules as in array initialization.

5.3 Methods

s-java methods are defined similarly to their definitions in java:

$$return\_value\_type\ method\_name\ (\ parameter_1, parameter_2, \ldots, parameter_n)\ \{$$

where:

- *method_name* is defined similarly to variable names (i.e., a sequence of length > 0, containing letters (capital or minuscule), digits and underscore). Method names must start with a letter (i.e., they may not start neither with a digit nor with underscore).
- *parameters* is a comma-separated list of parameters. Each *parameter* is a pair of a valid type and a valid variable name, but **not** a value.
  - A parameter's value can be defined with the same name as a member.

After the method declaration, comes the method's code. It may contain the following lines:

- Local variable declaration lines (as defined in Section 5.2)
- Variable assignment lines (e.g., $a = 5$; where a is either a member, a local variable or the method's parameter). Such lines are legal only if the variable has been declared, and the value is of legal type (that is, (a) as defined in Table 1 or (b) another (initialized) variable of the same type).
- A call to another existing method. Any method *foo()* may be called, regardless of its return type (void or other), and its location in the file (i.e., before or after the definition of the current method).

$$method\_name\ (param_1, param_2, \ldots, param_n);$$

where *method_ name* is a legal method and parameters are variable values that agree with the method definition. Calling a method with an incompatible number of arguments, values of the wrong type or uninitialized variables is **illegal**.

- An *if/while* block (see below).
- A return statement. Return statements may appear anywhere inside a method, but must also appear as the last line in the method's code. The format is:

  return value; (where *value* is a legal value of type *return_ value_ type*)

  or

  return; (for methods returning void)

Note the following:

- Methods must end with a line containing the single token '}' (which comes right after the *return* line as presented above).
- Recursive calls are allowed. I.e., a method may call itself. You are not required to check for infinite loops in the code.
- Two methods with the same name cannot be defined (regardless of their parameters.). However, a method may have the same name as any variable (member or local).
- The order in which methods appear in the code is meaningless. Methods may also appear before/after/between member declarations.
- Note that return values and parameters can be also arrays.
- A local variable can not be defined with the same name as its method's parameters.

5.4 *If* Blocks

If blocks are defined in the following way:

- *if (condition)* {
- ...
- }

where condition is a boolean value (i.e., *true* or *false*, an (initialized) boolean variable or a call to a method returning a boolean).

Notice the following:

- *If* blocks may contain the same type of lines as methods (that is, variable declaration, method calls, etc. See Section 5.3).
- You are not required to support complex conditions (e.g., &&, ||).
- Much like methods, *if* blocks must start with a '{' token and end with a '}' token.
- You are not required to support *else if* or *else* blocks.

5.5 *While* Blocks

While blocks are defined in the following way:

- *while (condition)* {
- ...
- }

where *condition* is defined similarly to *if* conditions.

The general comments regarding *if* blocks apply to *while* blocks as well. You are not required to support *do/while* loops, *for* loops or *switch* loops. That is, the words 'do', 'for' and 'switch' are not part of the language syntax.

5.6 General Comments

- A *main* method is not required in *s*-java. Of course a method called *main* may be defined, much like any other method name.
- Two local variable with the same name (regardless of their type) cannot be defined inside the same block. This also applies to variables with the same name as a method parameter.
- However, two local variable with the same name can be defined inside different blocks, even if they are one-within-the-other.
- Accessing variables declared in inner scope is illegal. However, you may access variables defined in outer scopes.
- Naming variable with reserved words (e.g 'int', 'if'...) is not allowed in *s*-java and should result in printing 1.
- A variable may be defined with the same name as a method.
- As mentioned earlier, a local variable can be defined with the same name as a member.
- White spaces (spaces and tabs) are allowed anywhere inside the code, and are to be ignored. This excludes white spaces inside variable names, values or reserved words (e.g., the following line is **illegal**: "i n t   a = 5;").
- On the other hand, white spaces are required to separate between a variable type and variable/method name (e.g., "inta;" is **illegal**). No space is required between any other input tokens (this includes before/after parentheses, '=', ';','{' and '}' signs, etc.).
- Each line of code must appear in a single line, and not broken into several lines.
- Packages, as well as exceptions, are not supported in *s*-java. That is, the words "package", "try", "catch" and "finally" are not part of *s*-java syntax.

6 Important Requirements

6.1 Error Handling

As in *ex6*, you are required to use the exceptions mechanism to handle general errors of your program (e.g., an IOException caused by an illegal file name). As noted earlier, in such cases you should print the value 2.

Regarding the main task (deciding whether or not the file is a legal code file), error handling is a bit tricky. Supposedly, all the information many of your methods have to provide in this exercise is a single bit – whether or not some part of the code is legal. In this case, it is tempting to define boolean methods that return true or false according to whether the code is legal. However, some of the benefits of the exceptions mechanism might come in handy here. We advise you to consider using this mechanism in this exercise. Regardless, report in your README file how you handled *s*-java code errors in this exercise, and why you chose to do so.

6.2 Object Oriented Design

As in previous exercises in this course, your program should follow the object oriented programming principles studied in class. When working on your design, we advise you to review Tirgul 10, where we introduced *ex6* design. While the task here is very different than the one in *ex6*, the working process should be similar: try to divide your program into small, independent units. Try to think of several design alternatives for each unit, consider the pros and cons of each alternative, and select the one you think is best. You are required to specify your design, as well as your thinking process and your ruled out alternatives, in your README file.
In addition, you should address the following points in your README file:

- How would you modify your code to add new types of variables (e.g., short).
- Below are two features your program currently does not support. Please select **one of them**, and describe which modifications/extensions would you have to make in your code in order to support it. Please briefly describe which classes would you add to your code, which methods would you add to existing classes, and which classes would you modify. You are **not required** to implement these features.
  - if-else block.
  - Compile a few files together and by that allow the import of new files (i.e., their methods and members) from one file to the other.

6.3 Regular Expressions

One of the main goals of this exercise is to practice using the regular expression mechanism. You are required to make extensive use of it in your program. However, you are allowed (and encouraged!) to use other text analyzing mechanisms in your code (e.g., *String* class methods such as *substring(), charAt()*, etc.). The general rule of thumb is that you should use regular expressions whenever it makes your life easier, and not when it makes it harder.

Be sure to follow the recommendations discussed in class about writing good regular expressions. In your README file, please describe the two main expressions you used in your code.

7 Submission Requirements

7.1 README

The README instructions from previous exercises apply here as well. Please describe your design and focus on non-trivial decisions you took. The README file should also include answers to the questions from Sections 6.1, 6.2 and 6.3.

7.2 Automatic Testing

In the course website you can find 3 files :

- tester.zip - a zip file containing JUNIT tester (including 4 files).
- tests.zip - a zip file including *s*-java files, on which your program will be tested.
- school_output.txt - a text file containing the desired output for each of the tests supplied in tests.zip.

You can run the testers using eclipse by:

- Copying the four JUNIT files to the default package of your project.
- Copying the tests (from tests.zip) to a directory named "tests" located next to the "src" directory.
- Copying the file sjavac_tests.txt to also be next to the "src" directory.

Ex7Runner.java reads sjavac_tests.txt line by line. for each line it runs your project with the specified s-java file and compares the output against the desired one. In case you failed some of the tests you will receive a list containing these tests, for example:

```
runTests[0](Ex7Tester): int problem in test number:001 expected:<[1]> but was:<[0]>

1 tests failed
```

Passing all the tests will result in output "Perfect!". As in previous exercises, when submitting your exercise via moodle you will receive an email with the tests you failed. Error messages will be the same as mentioned above. Please notice: passing the tests in your local environment on eclipse is not exactly the same as submitting via moodle, therefore, do not submit your project at the last minute.

We will also compare your program's output against unseen tests. their grade will be 10% of the automatic test part.

As in previous exercises, you can run the automatic tests via your terminal by using the following command (ex7.jar is your jar file):

```
~oop/bin/ex7 ex7.jar
```

7.3 Submission Guidelines

As in previous exercises, you should submit a file named ex7.jar containing all *.java* files of your program, as well as the README file. Files should be submitted along with their original packages. No *.class* files should be submitted. Remember that your program must compile without any errors or warnings, and that javadoc should accept your program and produce documentation. You may use the following unix command to create the jar files:

```
jar –cvf ex7.jar README oop/ex7/main/*java ...
```

8 Misc.

8.1 School Solution

School solution may be found under ∼*oop/bin/ex7SchoolSolution*. You are strongly advised to experiment with it before starting to work on your design, in order to get a feeling of how your program should behave on different inputs. Running the school solution against an s-java file will print 0, 1 or 2 but in case of 1 (a compilation error) we will also print an informative message that will help you to understand what is the problem. The error message is being printed using Systen.err(error_message) so it is not part of the program output.

Good Luck!