Exercise 4 – (Simplified) Java Compiler

CLIDS 2013

May 17^{th}, 2013

1 Goals

To exercise:

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

2 Submission Details

- Submission Deadline: **Monday, 03/06/2013, 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 (as introduced in previous exercises): 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 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 deduction.

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 clids.ex4.main.Sjavac source_file_name

Your program should use the exit value mechanism to indicate whether the code is legal or not: 0 means a legal code, while 1 means an illegal code. 2 should be the exit value in case of IO errors (see Section 6.1). Use *System.exit(return_value)* to set the exit value of your program. In order to get the program's exit value from the terminal, enter the command "echo $?" immediately after running your program. In eclipse, the exit value can be queried in debug mode: after your program has terminated (either in debug or in regular java mode), the top-left

window should contain a line such as *<terminated,* **exit value: 0***>/usr/local/bin/java (Apr 22, 2013 15:21:03 PM).*

Your program's standard output will not be tested automatically. However, in case the input code didn't compile, you are required to print an informative message to the screen, generally describing what went wrong. There is no specific format you have to follow here, as you will only be tested automatically on the exit values of your program. However, very uninformative messages might result in point reduction.

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, you 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: *members* and *methods*. *Members* are general variables (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. The rest of such lines 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 exit value 1.

5.2 Variables

s-java comes with a strict set of variable types. It does not support the creation of new 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* or any number (int or double)	boolean a = true; boolean b = 5.2;
char	a single character	any character (inside single quotation marks)	char my_char = 'a'; char g = '@';

Table 1: *s*-java types.

Notice the following:

- A variable may be declared with or without an initialization. That is, both *int a;* and *int b2 = 5;* are legal *s*-java lines.
- Any variable may be declared using the modifier *final*, which makes it a constant. Such variables must be initialized with some value at declaration time: **final** *int a = 5;*
- Other java modifiers (e.g., *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 may be define in a single line, separated by a comma. For example, the followings are **legal**:
  - double a, b;
  - int a , b = 6;
  - char a = 'c' , b;
  - String a = "hello" , b = "goodbye";
  - boolean a, b ,c , d = true, e, f = 5;
- 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 (non final) 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 local variable b of the same type, but only after b was declared and initialized in the same scope or in the scope above it (see test052.sjava, test054.sjava, test257.sjava in the automatic tests).
- A local 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 (see test251.sjava).
- You may assume '\' characters will not appear in String or char values.
- For simplicity, operators are not supported in s-java (*int a = 3 + 5;*, *String b = "CLIDS" + "is fun"* are not part of the language's syntax).
- Similarly, arrays are not supported either in s-java.

5.3 Methods

s-java methods are a simple version of java's method definition:

$$void\ method\_name\ (\ parameter_1, parameter_2, \ldots, parameter_n)\ \{$$

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.
- Only **void** methods are supported.

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 initialization lines (e.g., $a = 5$; where a is either a member or a local variable). Such lines are legal only if the variable has been declared (and is not final), 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 its location in the file (i.e., before or after the definition of the current method). The syntax of calling is the java syntax:

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

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 return statement should follow the method's declaration and shouldn't return any value. The format is:

$$return;$$

Note the following:

- Method parameters may be final. That is:
  - **legal**: void foo(final int a) {
- You may change the value of a non-final parameter inside the method's code.
- 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 (see test255.sjava). 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. See test474.sjava).
- The order in which methods appear in the code is meaningless. Methods may also appear before/after/between member declarations.

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 or number [int or double] variable). Also, multiple conditions separated by AND/OR may appear (e.g., *if (a || b || c) {*).

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 conditioning containing brackets.
  - *if ((a || b) && c ...) {*

  - ...
  - }

- 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.
- Two local variable with the same name (regardless of their type) cannot be defined inside the same block (see test408-9.sjava). This also applies to variables with the same name as a method parameter (see test407.sjava).
- However, two local variable with the same name can be defined inside different blocks, even if they are one-within-the-other (see test410.sjava).
- Accessing variables declared in inner scope is illegal (see test475-6.sjava).
- Naming variable with reserved words (e.g 'int', 'if'...) is not part of the *s*-java specification and you can handle it as you wish. for example: int double = 6;
- A variable may be defined as with the same name as a method (see test472-3.sjava).
- Last, as mentioned earlier, a local variable can be defined with the same name as a member (see test404-5.sjava).
- 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**)
  - *final* modifier and variable type (e.g., "finalint a;" 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 (test442.sjava).
- 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 *ex2*, 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 exit using the exit 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 lecture 6, where we introduced *ex2* design. While the task here is very different than the one in *ex2*, 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.
  - For loops.
  - 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

A file called ex4_files.zip is found in the course website. The zip contains a list of *s*-java files, on which your program will be tested. You can run your code against the input files and compare the exit values against the school solution. As in previous exercises, when submitting your exercise, we will run your code against all input files, and you will receive an email with the tests you failed on. Error messages will contain the file number. For example, you may receive the following error:

exit value error, when ran with arguments test011.sjava :
boolean member test no value

This means that your program returned a different exit value than the school solution when tested with the file *test011.sjava*.

As in previous exercises, you can run the automatic tests via your terminal by the following command (ex4.jar is your jar file):

~clids/bin/ex4AutoTests.sh ex4.jar

7.3 Submission Guidelines

As in previous exersices, you should submit a file named ex4.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 you 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 ex4.jar README clids/ex4/main/*java . . .*

8 Misc.

8.1 School Solution

School solution may be found under *~clids/bin/ex4SchoolSolution*. 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.

8.2 Grading

- Correct usage of regular expressions – 10%
- Working according to Coding style guidelines – 10%
- Design and README file – 30%
- Correct implementation – 50%

8.3 Forums and Questions

As in previous exercises.

Good Luck!