**Score: \_\_\_\_\_**

**Lab05 – Writing a *SILLI* Interpreter**

COMP256 – Computing Abstractions

Dickinson College

Spring 2023

Prof. Grant Braught

**Name(s):**

**Introduction:**

The topic on language abstractions began by drawing a distinction between interpreted and translated (or compiled) languages. In class, we have gotten a little feel for language translation by learning how high-level language instructions can be expressed in assembly language and how those instructions relate to machine language. In this lab you will gain hands-on experience with interpretation by writing an interpreter for a small programming language named SILLI (**S**imple **I**nstruction **L**anguage for **L**earning **I**nterpreters) that was invented just for this lab.

**An Example:**

The following is a short program in the programming language *Silli* – **S**imple **I**nstruction **L**anguage for **L**earning **I**nterpreters, which was invented just for this lab.

# Make some variables.

LET I=0

LET J=2

LET K=0

# Read an int and store in K

READ K

LOOP: PRINT I

I=I+J

IF I<K GOTO LOOP

PRINT I

🔑 1. Study the above Silli program. Imagine the user provides the value 5 for the variable K. What output would this program print?

**The Silli Language Syntax:**

Every high-level language has a grammar that defines the statements that may be used in a program written in that language. Statements in the Silli language will be made up of the following *elements*:



🔑 2. Identify the *type* of each of the following *elements*.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Element** | **Element Type** |  |
|  | a. > |  |  |
|  | b. \* |  |  |
|  | c. START |  |  |
|  | d. 8730 |  |  |
|  | e. X |  |  |
|  |  |  |  |

The above *elements* are used to create *statements* in the grammar for the Silli language. The Silli language has a total of seven different types of statements, each of which is described in the table below:



🔑 3. Identify the *type* of each of the following *statements*:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Statement** | **Statement Type** |  |
|  | a. READ X |  |  |
|  | b. Z=M+R |  |  |
|  | c. IF A=B GOTO START |  |  |
|  | d. PRINT Z |  |  |
|  | e. LET K=7 |  |  |
|  |  |  |  |

Notice that Silly is a little peculiar in that arithmetic statements may contain only variables. So, the statement X=Y+Z is valid assuming that X, Y and Z have been defined. **But X=Y+2 or Z=3+5 are not valid statements.** This complicates programming in Silli a little bit, but it makes writing an interpreter for the language easier. However, because we can always put a value into a variable, it doesn’t actually limit what we can do with the language. At the end of the lab there are some extensions that will allow arithmetic statements with literal values like X=Y+2 if you are interested.

🔑 4. What is the only type of statement in Silli that is allowed to have an integer value in it?

There are a few other things you need to know about the Silli language before we can begin creating an interpreter for it.

* Silli is *case insensitive*. Thus, the statement LeT R=2 has the same meaning as the statement LET r=2.
* Blank lines and comments are ignored.
* Any leading spaces or tabs (i.e. indentation) are ignored.
* Within statements the ***exact spacing*** shown in the table above must be used. Thus, the statement X=Y+Z is valid, but the statements X = Y + Z or X=Y + Z or X= Y+ Z, ect. are not valid.

🔑 5. Consider the following statements. Rewrite each statement with proper spacing if possible. If a statement cannot be rewritten so that it is valid in Silli, just write *invalid*.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Statement** | **Rewritten Statement** |  |
|  | a. LET X = 7 |  |  |
|  | b. C = X+ Q |  |  |
|  | c. IF X > R GOTO END |  |  |
|  | d. Y=X + 9 |  |  |
|  |  |  |  |

The last piece of Silli syntax that we need to be precise about are the *line labels*. **A line label is defined by starting a line with a <LABEL> element, followed by a colon followed by a space, followed by a valid Silli statement.**  Some examples of lines that define labels are:

INPUT: READ K

LINE1: PRINT I

MATH: I=I+J

COND: IF M=N GOTO LOOP

🔑 6. In the program at the top of the lab, which line has a label? What is the label that is defined on that line?

🔑 7. Notice that only one type of Silli statement makes use of a label. What type of statement uses a label? Given that observation, what is the purpose of a label?

**Getting Setup:**

You’ll be writing your interpreter for Silli in Java. To make this easier and to ensure that everyone has a unform environment we will be using a Docker container.

8. Run the Docker Desktop application (find it and double click its icon). Give Docker Desktop a little time to start.

9. Open a Terminal on your machine:

* For Windows: Run the WSL app from the Start Menu.
* For Mac: Run the Terminal app from the Utilities folder in the Applications folder.

10. Copy and paste the following command into the terminal and press Enter or Return:

docker create --name Comp256Silli --publish 5901:5901 --publish 6901:6901 braughtg/comp256-silli:1.0.0

11. Using the Docker Desktop application, find the “Containers” tab. Find the entry for the Comp256Silli container. On MacOS this entry looks as follows:  
  


12. Click the “Play” button () to start the container. The container should begin running:  
  


13. Open the TigerVNC application that you installed earlier and fill in the address of the “VNC server” with localhost:5901 as shown here:

Graphical user interface

Description automatically generated

14. Click the “Connect” button. When you do you should see a window like the following:

A screenshot of a computer

Description automatically generated

The window above is a virtual machine running the Linux operating system and you will use it for the activities and labs in the Language Abstractions unit, including this lab.

15. Open the VSCodium application by clicking on its icon in the “Launcher”.

16. Choose “Open Folder” from the “File” menu. Then select the “Silli” directory and click the “Open” button in the upper right. You will be asked if you “Trust the Authors” of the code in the “Silli” directory. Confirm that you do. You should then see the following files in the “Explorer” pane in the top left of the VSCodium window.

Text

Description automatically generated

**Compiling and Running the Interperter:**

17. To compile the Silli Interpreter open a Terminal window by clicking it icon in the “Launcher.” Then use the following commands in the Terminal window:

cd /home/student/Silli

javac \*.java

The first command, cd Silli, changes into the directory with all of the Silli starer code. The second command, javac \*.java, tells the Java Compiler to compile all of the files that end with the extension .java.

When the compilation is complete you will now see a file with the .class extension corresponding to each of the files with a .java extension. The .class files contain the Java Byte Code translation of the source code from the corresponding .java files.

18. Before running the Silli interpreter we need a Silli program for it to interpret. The starter code comes with the program named FirstProg.si. Find this file and open it in VSCodium.

a. What three types of Silli statements are contained in this program?

b. What three variables does this program declare? What are their initial values?

19. To run the FirstProg.si program use the following command in the terminal:

java Interpreter FirstProg.si

Does the FirstProg.si program generate any output when it is run? Why or why not?

As you saw the FirstProg.si program does not generate any output when it is run. That is because the starter code you were given does not yet support the output (i.e. PRINT) statement. In fact, the starter code supports only the three types of statements that are in the FirstProg.si program (Comment, Blank and Declaration).

Because the Interpreter does not yet support Output statements it is hard to know if the FirstProg.si program actually worked. A feature of the interpreter can help us with this.

20. Run the Interpreter using the following command line:

java Interpreter -debug FirstProg.si

Notice that the Interpreter now generates some debugging output. What information in this debugging output shows that the instructions in the FirstProg.si were executed properly?

Keep the -debug flag in mind. It can be a helpful tool for debugging as you add and test the features that you will be adding to the Silli interpreter.

**Getting Familiar with the Provided Code:**

The provided starter code includes the Interpreter, ProgramStatement, Blank, Comment and Declaration classes. The Interpreter class contains the main method and handles reading the source code and coordinates the work of the other classes. It is these other classes that do the real work if interpreting instructions and it is those that we will be focusing on.

21. The Blank, Comment and Declaration classes each represent one of the types of statements in the Silli language. Use the source code for these classes to answer the following questions:

a. All of these classes are subclasses of which super class?

b. What line of code is contained in the constructor for all three of these classes?

c. What does the line of code you found in part c do?

d. In addition to their constructor, each of these classes contain two methods. These methods have the same signature in all three classes. What two methods are contained in all three of these classes?

e. Read the JavaDoc for the methods that you found in part d in the Blank class. Briefly describe the purpose of each of these methods.

22. Use the source code for the ProgramStatement super class that you identified in #21 part a to answer the following questions:

a. What fields does the ProgramStatement super class have?

b. Will code in the subclasses of ProgramStatement (e.g. Blank) be able to access these fields? How can you tell?

c. In which method within the ProgramStatement are the subclasses of ProgramStatement used?

d. What is the main job of the method that you found in part c?

e. Which method in the subclasses does the method you found in part c use?

**Adding the Output Statement:**

Now that you have an idea of how the ProgramStatement and its subclasses (e.g. Blank, Comment, Declaration) work together to find code that can interpret a Silli language statement we can start expanding the capabilities of our Silli Interpreter.

The next logical step is to add support for Output statements so that programs can print the values of variables. The questions in this section will walk you through the process for adding support for Output statements.

You will first create a new sub-class of ProgramStatement that can recognize and interpret Output statements. You will then add code to the getStatement method in the ProgramStatement super class that recognizes Output statements and creates an instance of your subclass to interpret them when they are encountered in a Silli program.

23. Create a new subclass of ProgramStatement named Output in the file Output.java. Your Output class should have a constructor and the same two methods as the other ProgramStatement classes. You might copy and paste these methods from the Blank classes and then modify them. In particular, you will need to:

* Change the match(…) method so that it recognizes Output statements (i.e. those that start with PRINT). Adapt the match(…) method from the Blank class to do this.
* Change the execute() method so that it gets the value to be printed from the varMap and prints it.
* Your method should also report errors using the printError method if:
  + the PRINT statement is not properly formatted.
  + the variable being printed has not been declared.

See the execute() method in the Declaration class for examples.

***You will turn in all of the Java code that you write at the end of the lab, so no answer is required here for this question.***

24. Add code to the getStatement method in the ProgramStatement class that recognizes an Output statement and returns an object that can interpret it.

***You will turn in all of the Java code that you write at the end of the lab, so no answer is required here for this question.***

25. Create a small Silli program named TestOutput.si with the following code:

# Test Output statements

LET A=1

LET B=10

LET C=100

PRINT A

PRINT B

PRINT C

Run the TestOutput.si program with the Interpreter to test that your implementation of Output statements works correctly.

***There is no answer required here, but the Interpreter must correctly process Output statements before going on.***

**Adding the Input Statement:**

26. Add to the ProgramStatement class so that it can recognize and interpret Input statements (i.e. start with READ). The execution of these statements will require that you prompt the user for input, read that input, check that it is valid (i.e. an integer) and then update the values in the varMap. Your method should also display appropriate error messages using the printError method if the input statement contains errors (e.g. is not properly formatted, or the variable being read into has not been declared).

You can use a Scanner object to read input from the user. See the link below for an example of how to read an integer using the Scanner class:

* <http://www.learningaboutelectronics.com/Articles/How-to-read-input-with-the-Scanner-class-in-Java.php>

***You will turn in all of the Java code that you write at the end of the lab, so no answer is required here for this question.***

27. Write a small Silli program named TestInput.si that reads values for at least 2 variables from the user and then prints them. Run your TestInput.si program with the Interpreter to test that your implementation of Input statements works correctly.

***You will turn in all of the Silli code that you write at the end of the lab, so no answer is required here for this question. Just be sure to save the source code for your TestInput.si program in the same directory as your .java files.***

28. **Optional Extra Challenge:** If your processing of Input statements does not already do this, have it display an error message and prompt again if the user enters a value that is not an integer.

***You will turn in all of the Java code that you write at the end of the lab, so no answer is required here for this question.***

**Adding Arithmetic Statements:**

29. Add a ProgramStatement subclass that can recognize and interpret Arithmetic statements. The execution of these statements will require that you determine which operation to perform and then use and update the values in the varMap. Your method should also display appropriate error messages using printError if the Arithmetic statement is not properly formatted.

Hints:

* The String.contains and String.substring methods will be useful here. You can find documentation for them at the following links:
  + <https://www.w3schools.com/java/ref_string_contains.asp>
  + <https://www.w3schools.com/jsref/jsref_substring.asp>
* Remember to use .equals(…) when comparing Strings!
  + <https://www.w3schools.com/java/ref_string_equals.asp>

***You will turn in all of the Java code that you write at the end of the lab, so no answer is required here for this question.***

30. Write a small Silli program named TestArithmetic.si that reads values from the user, **performs each of the arithmetic operations** **at least once** and prints output that allows you to check that they have worked correctly.

***You will turn in all of the Silli code that you write at the end of the lab, so no answer is required here for this question. Just be sure to save the source code for your TestArithmetic.si program in the same directory as your .java files.***

**Adding the Conditional Statement:**

Conditional statements allow Silli programs to branch. Thus, adding them will allow our Interpreter to run programs that do more interesting things. For example, Conditional statements will allow us to implement programs with if/else statements and loop. Implementing conditional statements will also give you a more complete understanding of how interpreters work.

31. Add a ProgramStatement subclass that it can recognize and interpret Conditional statements. You will need to determine which comparison is being performed, retrieve the values from the varMap and perform the comparison. Based on that comparison, you will need to decide which line should be executed next. If the branch condition is true you will need to use the labelMap to find the line to which to branch. Note that the Interpreter ensures that the labelMap contains all of the line labels defined in the program.

As with prior statement types, your implementation of Conditional statements should display appropriate error messages using printError if the conditional statement is not properly formatted or the label being branched to does not exist.

***You will turn in all of the Java code that you write at the end of the lab, so no answer is required here for this question.***

32. Write a small Silli program named TestConditional.si that reads 2 values from the user and prints out the larger of the two values that was read. Run this program with a variety

***You will turn in all of the Silli code that you write at the end of the lab, so no answer is required here for this question. Just be sure to save the source code for your TestConditional.si program in the same directory as your .java files.***

**Congratulations!!** You have just written your first programming language interpreter. What you have built can now run the FirstProg.si program from the start of the lab. Run it and be proud of what you have accomplished!

**Submitting:**

33. Submit this lab as follows:

a. Use the following commands in a terminal window in the container:

cd /home/student

tar -zcvf Silli.tar.gz Silli

b. Use the FireFox browser in the container to go to the course Moodle site and submit the Silli.tar.gz file to the LAB05 Code assignment.

c. Convert this activity sheet to a PDF and submit it to the LAB05 Activity Sheet assignment on Moodle.

**Optional Extensions:**

If you found working on the Silli interpreter fun and interesting, you may also find some of the following optional extension to be fun, challenging and rewarding.

**Programming in Silli:**

You invested a lot of time in creating an interpreter for Silli. It might be fun to write a program in this language that does something more substantial and then see it run with your interpreter.

34. Write a Silli program named Exponent.si that reads two values (e.g. X, Y) and computes XY. Be sure to run your program in your interpreter to make sure that it works.

**Improving the Interpreter:**

While our interpreter is pretty good and can run a wide variety of programs it is still limited in many way. All of the following are optional extensions that address these limitations and enhance the operation of the interpreter in interesting ways. You are free to do none of these, do one of them or do all of them. Each improvement is independent of the others and you may do them in any order that works for you.

35. Extend the interpretation of output statement so that they can print either a variable or a string literal. For example:

PRINT “The result is: “

PRINT X

36. Extend the syntax of all types of instructions to allow variable names to have more than one character. For example:

LET AGE=19

LET ONE=1

AGE=AGE+ONE

Hint: The String.split method will be useful here.

37. Extend the syntax of the conditional statement to allow for the additional relational operators: >=, <=, !=. For example:

IF X>=Y GOTO START

38. Extend the syntax of arithmetic statements so that they may contain both variables and integer literals. For example:

X=Y+1

Z=2\*3

Q=7-X

M=N+P

Hint: The Integer.parseint method will be useful here.

39. Extend the syntax of conditional statements to allow either variables or integer values in conditional statements. For example:

IF X>0 GOTO START

40. Extend the syntax of conditional statements to allow compound conditions using AND or OR. For example:

IF X > Y AND Y > Z GOTO XMAX).

41. Make the processing of the statements less picky about the spacing. For example:

LET X =2

LET Y = 3

X=Y + 2

42. **EXTRA SUPER CHALLENGE**: Extend the syntax of arithmetic expressions to allow for any infix expression on the right-hand side (e.g. accept expressions like X=R\*X\*(1-X)). Hint: You’ll need to use a stack data structure. Here is a resource to get you started if you are interested:

* <https://www.geeksforgeeks.org/expression-evaluation/>

Optional: To help me improve and scope these activities for future semesters please consider providing the following feedback.

a. Approximately how much time did you spend on this activity outside of class time?

b. Please comment on any particular challenges you faced in completing this activity.