

ITI 1121. Introduction to Computing II

Winter 2014

Assignment 3 [[PDF](#)]

(Last modified on March 18, 2014)

Deadline: Monday March 17, 2013, 18:00

Solution

- [a3-solution.jar](#)

Learning objectives

- Implementing a stack whose capacity increases dynamically
- Describing the interpretation of computer programs
- Extending a stack-based interpreter (virtual machine)
- Defining, throwing and catching Java exceptions

Background information

With this assignment, we explore embedding a simple programming language into an application. The programming language is named **Luka** after the Polish mathematician Jan Łukasiewicz, who, in the 1920s introduced the (Reverse) Polish Notation (RPN). **Luka** is a stack-based programming language inspired by **PostScript**¹. The interpreter scans a program from left to right, pops and pushes operands from and to a stack during the evaluation of the program. The programming language **Luka** comprises several operations for drawing onto a two dimensional surface. The result of the execution of a **Luka** program is generally a drawing. Herein, the language is embedded into a spreadsheet application.

The class **Interpreter** implements a **Virtual Machine** for the programming language **Luka**. The method **execute** implements the “read-eval” loop of the interpreter. At each iteration of the **while** loop, the interpreter fetches the next element (**Token**) from the input program. Numbers are simply pushed onto the operands stack. Symbols cause the interpreter to execute specific actions. Here is an outline of the “read-eval” loop of the interpreter.

```
public void execute(String program) {
    Reader r = new Reader(program);
    while (r.hasMoreTokens()) {
        Token t = r.nextToken();
        if (t.isNumber()) {
            operands.push(t);
        } else if (t.getSymbol().equals("add")) {
            execute_add();
        } else if (...) {
            ...
        }
    }
}
```

```
private void execute_add() {
    Token op2 = operands.pop();
    Token op1 = operands.pop();
    Token res = new Token(op1.getNumber() + op2.getNumber());
    operands.push(res);
}
```

¹PDF (Portable Document Format) is largely based on PostScript.

Here is a small **Luka** program for adding two numbers:

```
15 10 add
```

At the first iteration, the interpreter reads and pushes 15 onto the operands stack. At the second iteration, the interpreter reads and pushes 10 onto the operands stack. Finally, at the third iteration, the interpreter reads the symbol **add** and calls the method **execute_add()**. The latter pops the two operands from the stack, adds them up, and stores the result onto the operands stack. The program terminates with the operands stack containing only one element, the value 25. See Appendix [sectionB](#) for a complete description of the virtual machine.

1 Rules and regulation (10 marks)

Follow all the directives available on the [assignment directives web page](#), and submit your assignment through the on-line submission system [uottawa.blackboard.com](#). You must preferably do the assignment in teams of two, but you can also do the assignment individually. Pay attention to the directives and answer all the following questions.

2 DynamicArrayStack (10 marks)

The class **DynamicArrayStack** implements the interface **Stack** and must use the technique presented in class and called dynamic arrays. This technique allows a data structure (here a stack) to increase its physical size according to the needs of the application. Modify the implementation of the class **DynamicArrayStack** so that the physical size of the stack increases by a fixed **increment** when needed, and decreases by a fixed **increment** when needed.

- Declare a constant specifying the default increment of the stack, its value should be 25;
- Modify the existing constructor, **DynamicArrayStack(int increment)**, so that it uses the value of specified parameter **increment** 1) as the initial size of the stack and 2) as the value of the increment to be used when increasing or decreasing the size of the array. In this implementation, a fixed number of cells are added or removed, when the physical size increases or decreases. The value of the parameter must be greater than zero;
- Add a constructor with no parameter. 1) It will initialize the stack to have an array whose size is the value of the increment constant defined above. 2) It will also memorise that value as the value of the increment;
- Make all the necessary changes to ensure that the capacity of the stack increases or decreases when needed. The size needs to be increased when there is no space for adding a new element. The size needs to be decreased, by **increment**, when there are $2 \times \text{increment}$ free cells.

The implementation of the class **DynamicArrayStack** has a parameter type, and therefore creates a type-safe implementation of the interface **Stack**.

3 Exceptions (10 marks)

3.1 New type of exception (4 marks)

Create a new run-time exception type called **EmptyStackException**.

3.2 Preconditions (6 marks)

Make all the necessary changes to the classes **DynamicArrayStack** and **LinkedStack** so that the necessary exceptions are thrown when needed. In particular, the value **null** is not a valid argument for the method **push**.

4 Interpreter (65 marks)

For this assignment, the signature of the method **execute** is as follows:

```
execute(String program, Graphics g, JTextArea output, Sheet sheet)
```

Notice that in addition to the **Luka** program and the reference of type **Graphics**, the interpreter receives a reference to the **JTextArea** of the **Viewer**, as well as the object representing the spreadsheet, a reference of type **Sheet**.

Make all the necessary changes to the class **Interpreter** to implement the following instructions for the programming language **Luka**.

pstack : displays the content of the stack in the Viewer's console (JTextArea object designated by output). The representation starts with the symbol "[" representing the bottom of the stack. The content of the stack follows, starting with the bottom of the stack up to the top of the stack. The elements are separated by a space. The execution of the following **Luka** program:

```
10 20 30 40 50 60 pstack
```

produces the following result in the Viewer's console:

```
[10 20 30 40 50 60
```

clear : removes all the elements from the operands stack. The execution of the following **Luka** program:

```
10 20 30 40 50 60 pstack clear pstack
```

produces the following result in the Viewer's console:

```
[10 20 30 40 50 60  
[
```

dup : pushes a second copy the top element on the top element on the operands stack. Since **Token** objects are immutable, it is safe to simply push a second copy of the same object (**shallow copy**). The execution of the following **Luka** program:

```
10 20 30 40 50 60 pstack dup pstack
```

produces the following result in the Viewer's console:

```
[10 20 30 40 50 60  
[10 20 30 40 50 60 60
```

count : counts the total of elements on the operands stack, pushes that information onto the top of the operands stack. The execution of the following **Luka** program:

```
10 20 30 40 50 60 pstack count pstack
```

produces the following result in the Viewer's console:

```
[10 20 30 40 50 60  
[10 20 30 40 50 60 6
```

sumtomark : sums all the elements following the top-most mark of the operands stack, and pushes the sum onto the operands stack.

The execution of the following **Luka** program:

```
mark 10 20 30 40 50 60 pstack sumtomark pstack
```

produces the following result in the Viewer's console:

```
[mark 10 20 30 40 50 60  
[mark 10 20 30 40 50 60 210
```

n j roll : performs a circular rotation of the top **n** elements of the stack by **j** positions. You must use stacks to store temporary data. There will be a five marks penalty for using arrays. The execution of the following **Luka** program:

```
10 20 30 40 50 60 pstack 4 2 roll pstack
```

produces the following result in the Viewer's console:

```
[10 20 30 40 50 60  
[10 20 50 60 30 40
```

c r cell : pushes onto the operands stack the content of the cell at column **c** and row **r** of the spreadsheet. That value must be a number.

Assuming that the cell **A 0** of the spreadsheet contains the value 10, the execution of the following **Luka** program:

```
/A 0 cell pstack
```

produces the following result in the Viewer's console:

```
[10]
```

The notation **/A** is a quoted symbol, the value of the symbol following the forward slash is simply pushed to the operands stack by the interpreter, without looking for a value associated with the symbol.

r pushrow : pushes onto the operands stack the value of all the cells of row **r** (of the spreadsheet) that can be interpreted as a number. Values that are not a number are simply ignored. The leftmost value of the row must be the top of the operands stack.

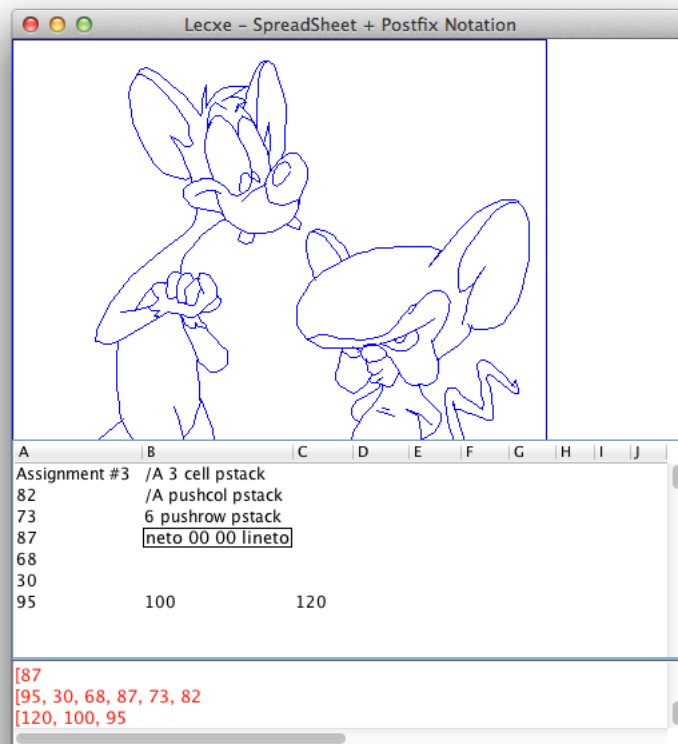
c pushcol : pushes onto the operands stack the value of all the cells of column **c** (of the spreadsheet) that can be interpreted as a number. Values that are not a number are simply ignored. The topmost value of the column must be the top of the operands stack.

The screen capture below shows the results of executing the following three **Luka** programs:

```
/A 3 cell pstack
```

```
/A pushcol pstack
```

```
6 pushrow pstack
```



The following **Luka** program calculates the average of all the values of column **A** of the spreadsheet.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
```

Given that column **A** contains the elements 10, 20, 30, and 40, where 10 is the topmost element. Here is a detailed description of the above algorithm. (the top row is the algorithm, the second is current token read, and the final row is the content of stack)

1. First, make sure that the stack is empty by executing **clear**.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[
```

2. Push the token **mark** onto the stack.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[mark
```

3. Push the the symbol **A** onto the stack

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[mark A
```

4. Push the elements from column **A** onto the stack

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[mark 40 30 20 10
```

5. Push onto the stack the sum of all the elements from the top of the stack up to mark.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[mark 40 30 20 10 100
```

6. Push onto the stack the number of elements currently onto the stack.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[mark 40 30 20 10 100 6
```

7. Push 1 onto the stack.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[mark 40 30 20 10 100 6 1
```

8. Rotate the top 6 elements of the stack by one position.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[100 mark 40 30 20 10
```

9. Push onto the stack the number of elements from the top up to mark.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[100 mark 40 30 20 10 4
```

10. Push onto the stack the total number of elements currently onto the stack.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[100 mark 40 30 20 10 4 7
```

11. Push 1 onto the stack the stack.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[100 mark 40 30 20 10 4 7 1
```

12. Subtract

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[100 mark 40 30 20 10 4 6
```

13. Push 1.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[100 mark 40 30 20 10 4 6 1
```

14. Rotate the top 6 elements of the stack by one position.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[100 4 mark 40 30 20 10
```

15. Remove from the stack all the elements up to mark, these were elements from column **A**.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[100 4
```

16. Divide the sum by the number of elements.

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[25
```

17. Print the content of the stack

```
clear mark /A pushcol sumtomark count 1 roll counttomark count 1 sub 1 roll cleartomark div pstack
[25
```

5 LukaSyntaxException (5 marks)

- Create a specialized type of **IllegalArgumentException** called **LukaSyntaxException**.
- Modify the **Interpreter** so that it throws such an exception if the next token read is not a reserved word from the **Luka** programming language.
- Modify the method **execute** of the **Viewer** to catch such exception and display an error message on the console.

Files

You must hand in the following files.

- **Stack.java**
- **DynamicArrayStack.java**
- **LinkedStack.java**
- **EmptyStackException.java**
- **Interpreter.java**

- [Sheet.java](#)
- [Reader.java](#)
- [Token.java](#)
- [LukaSyntaxException.java](#)
- [README.txt](#)
- [Display.java](#)
- [Viewer.java](#)
- [StudentInfo.java](#)
- [Run.java](#)
- [prog-01.luka](#)
- [prog-02.luka](#)
- [prog-03.luka](#)
- [prog-04.luka](#)

Here is jar containing all of the above files: [a3.jar](#).

A Frequently Asked Questions (FAQ)

1. I see that the assignment was last modified on March 17, what modification was made?

I simply corrected a typographical error, where the name of the sample Luka programs should have had a .luka extension, and not .java.

2. Where can I find information regarding the methods for drawing onto the canvas?

Consult the documentation of the class **Graphis**

- java.sun.com/javase/6/docs/api/java/awt/Graphics.html

B Description of the Luka Virtual Machine (LVM)

Here is a detailed description of the class **Interpreter**:

- **Instance variables.** A **Reader** object is used to for the lexical analysis of the input program. The interpreter uses a stack to store the operands during the execution of a program. The interpreter has a graphics state, which here consists of a pair of coordinates, x and y , as well as a default color (of type `java.awt.Color`). The pair (x, y) represents the current position of the pen.

- Method **public void execute(String program, Graphics g)**.

The execution of a **Luka** program always starts with an empty stack. The pen is moved to the location $(0, 0)$, and the default color is set to blue (`Color.BLUE`).

A **Reader** object is used read the program one **Token** at a time.

The essential of the method consists of a loop to read and execute the input program; this is sometimes called the “read-eval” loop of the interpreter.

This loop terminates when the whole program has been read (**hasMoreTokens()** returns **false**). At each iteration, a **Token** is read. Inside the loop, the method must implement each of the operations of the programming language **Luka**. Numbers are simply pushed onto the operands stack.

For each of the following operations there is a corresponding method in the interpreter named **execute_operation**, for instance **execute_add** in the case of the operation **add**.

add pops off the top two elements from the operands stack, adds them together and pushes back the result onto the stack.

sub: pops off the top two elements from the operands stack, subtracts them together and pushes back the result onto the stack. E.g.: $(3 - 1)$ would be represented as “**3 1 sub**”.

mul: pops off the top two elements from the operands stack, multiplies them together and pushes back the result onto the stack.

div: pops off the top two elements from the operands stack, divides them and pushes back the result onto the stack. E.g.: $(8/4)$ would be represented as “**8 4 div**”.

/: is used to introduce a quoted symbol. This prevents the evaluation of the symbol following the “/”. below.

exch: exchanges the order of the two elements on the top of the stack.

pop: removes the top element of the stack.

mark: push the token **mark** onto the operands stack. All the tokens **mark** are identical. The operands stack may contain an arbitrary number of tokens **mark** at any given time.

cleartomark: removes from the operands stack all the elements between the top of the stack and the first token **mark**, which is also removed.

counttomark: counts the number of elements between the top of the stack and the first token **mark**, the token **mark** is excluded from the count. Following the execution of **counttomark** the content of the operands stack remains unchanged, except that the count has been pushed onto the top of the stack. For example, if the operands stack has the following content before the execution of **counttomark**:

```
[30 50 100 mark 400 40 225 90]
```

The operands stack will have the following content after the execution of **counttomark**:

```
[30 50 100 mark 400 40 225 90 4]
```

moveto: sets the position of the pen to (x', y') , where (x', y') are read from the stack. For example, if the current location of the pen is $(10, 30)$ and the content of the stack is as follows.

```
[30 50 100 400 20 80]
```

the operation **moveto** sets the position of the pen to $(20, 80)$. After the execution of the operation, the stack contains the following elements:

```
[30 50 100 400]
```

lineto: draws a line from (x, y) to (x', y') , where (x, y) is the current location of the pen (which is part of the graphics state of the interpreter), and (x', y') are read from the stack. Once the line has been drawn, the position of the pen, (x, y) , is set to (x', y') . For example, if the current location of the pen is $(10, 30)$ and the content of the stack is as follows.

```
[30 50 100 400 20 80]
```

the operation **lineto** draws a line from $(10, 30)$ to $(20, 80)$. After the execution of the operation, the stack contains the following elements:

```
[30 50 100 400]
```

and the new position of the pen is $(20, 80)$. For drawing a line onto a **Graphics** object, the method uses the method **drawLine(x1,y1,x2,y2)**.

arc: draws an arc onto the **Graphics** object using the method **drawArc(x, y, radius, radius, startAngle, angle)**, where (x, y) is the current location of the pen (which is part of the graphics state of the interpreter), and the **radius**, **startAngle** and **angle** are read from the stack. For example, if the current location of the pen is $(100, 100)$ and the content of the stack is as follows.

```
[30 50 100 400 40 225 90]
```

the operation **arc** draws an arc of radius 40, with start angle value 225, angle is 90, at position $(x = 100, y = 100)$. After the execution of the operation, the position of the pen has not changed, and the stack contains the following elements:

```
[30 50 100 400]
```

quit: exits the application (calls **System.exit(0)**).

Token

The lexical analyzer (**Reader**) converts the input **Luka** program into a sequence of tokens (objects from the class **Token**). A **Token** represents either a number or a symbol.

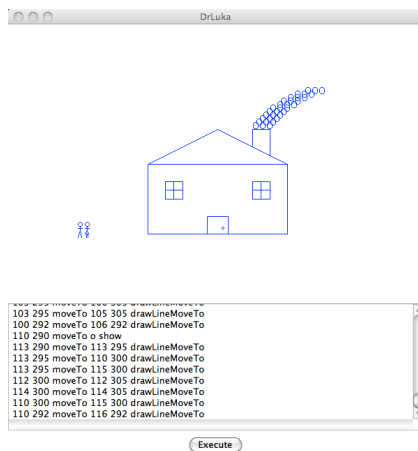
- **Token(int number)**: this constructor is used to initialize a token to represent a number.
- **Token(String symbol)**: this constructor is used to initialize a token to represent a symbol.
- **boolean isNumber()**: returns **true** if the object was created using the constructor **Token(int number)**.
- **boolean isSymbol()**: returns **true** if the object was created using the constructor **Token(String symbol)**.
- **int getNumber()**: returns the number stored in this object. The method can only be called if the object was created using the constructor **Token(int number)**, throws **IllegalStateException** otherwise.
- **String getSymbol()**: returns the symbol stored in this object. The method can only be called if the object was created using the constructor **Token(String symbol)**, throws **IllegalStateException** otherwise.

Sheet

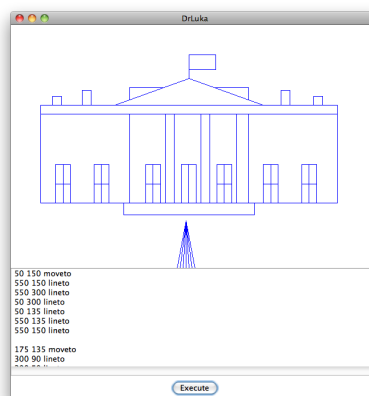
An object of the class **Sheet** stores the data of the spreadsheet application. As required by **JTable**, this is a sub-class of **AbstractTableModel**.

Hall of Fame

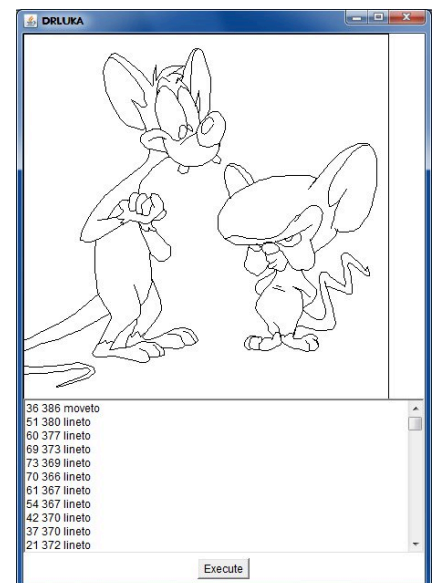
Optional (i.e. no marks): you can add further operations to the interpreter, to fill geometric forms, or to change the color of the pen. For changing the color of the pen, I would suggest using constants, say 1 for black, 2 for blue, 3 for red, and so on, then adding a command such as **3 setcolor**, which would set the default color in the graphics state to be red (i.e. **Color.RED**). For the time, this year **Luka** is embeded into a spreadsheet application. It would be interesting to see operations that are specific to a spreadsheet, for instance for plotting the values of a column or range of values. You can send me your interpreter, as well as a screen capture to have your name on the Hall of Fame:



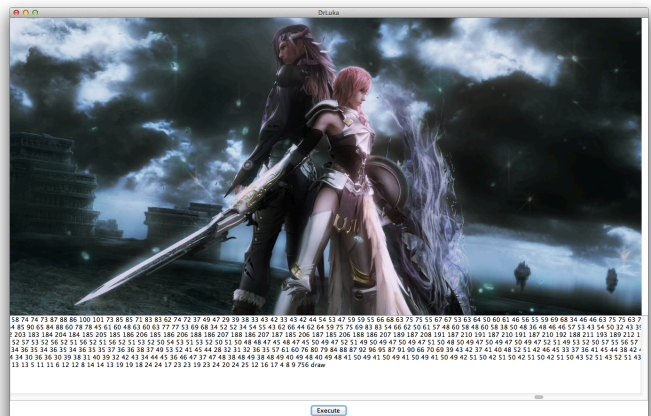
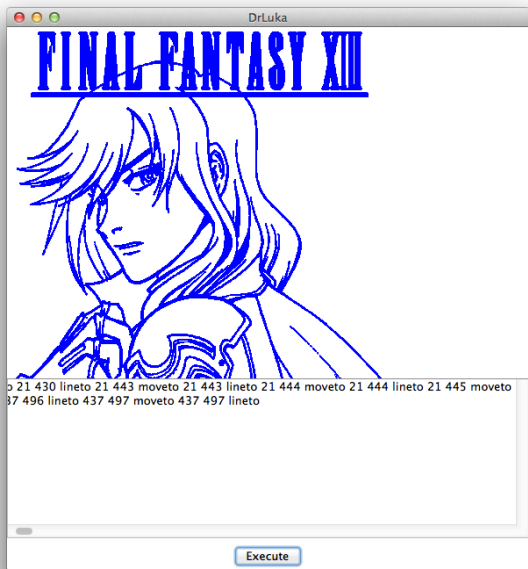
Samuel Bostock 2010



Liam Shea Williams 2010

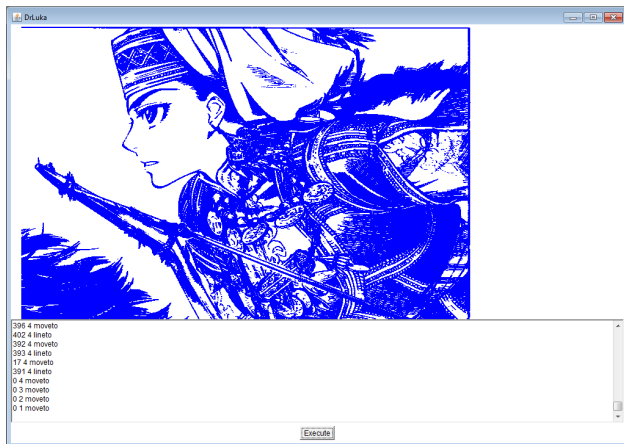


Quentin Smith 2011

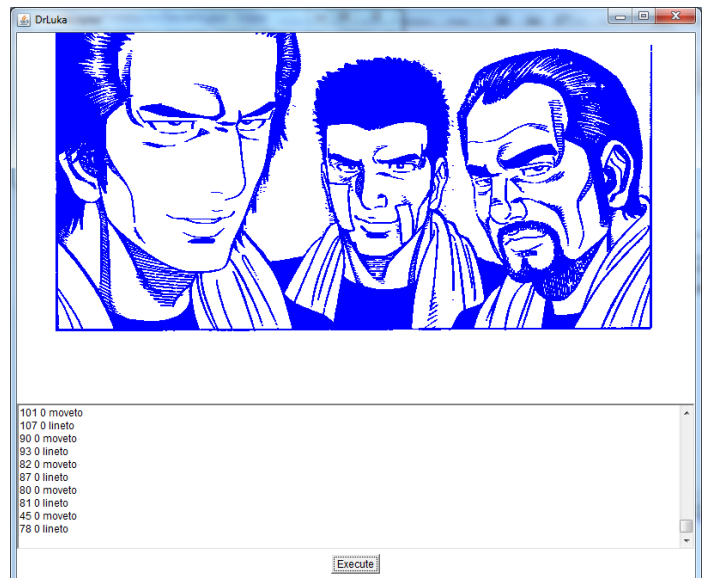


Olivier Gagnon 2012 (LVM Modifie)

Olivier Gagnon 2012



Liban Osman 2012



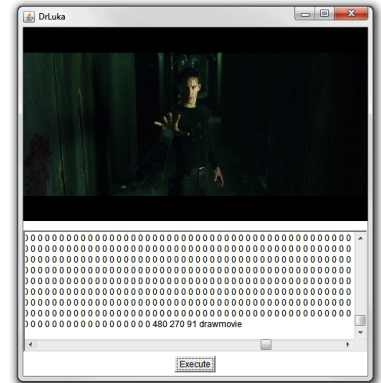
Liban Osman 2012



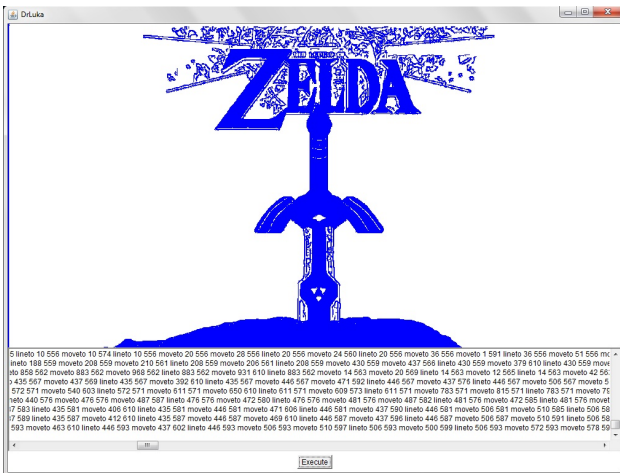
Matthew Horton 2012



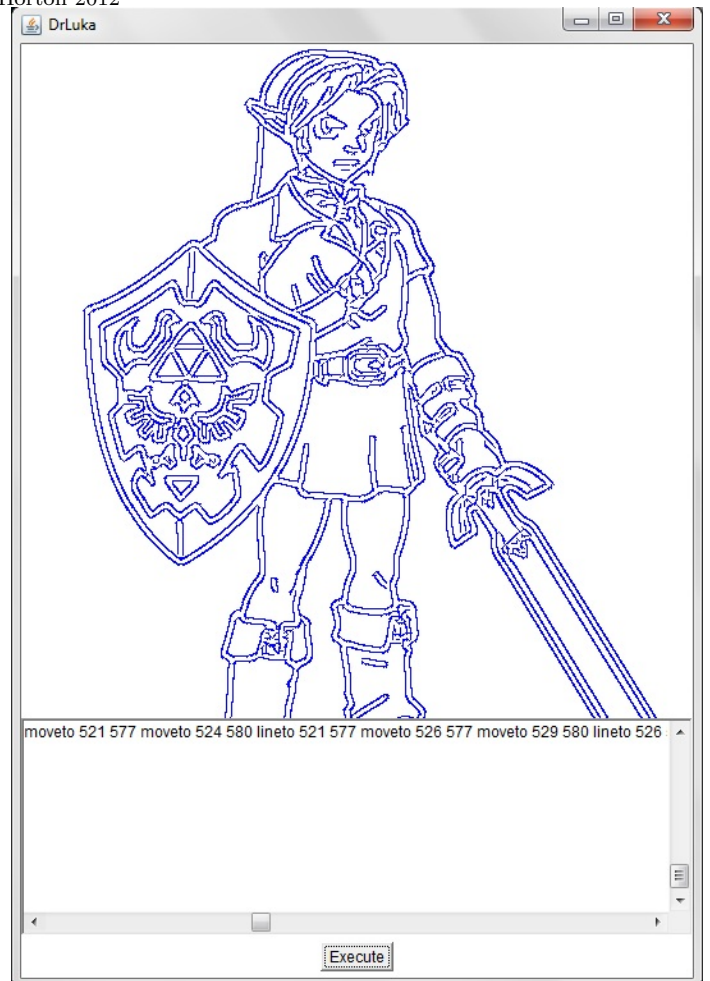
Matthew Horton 2012



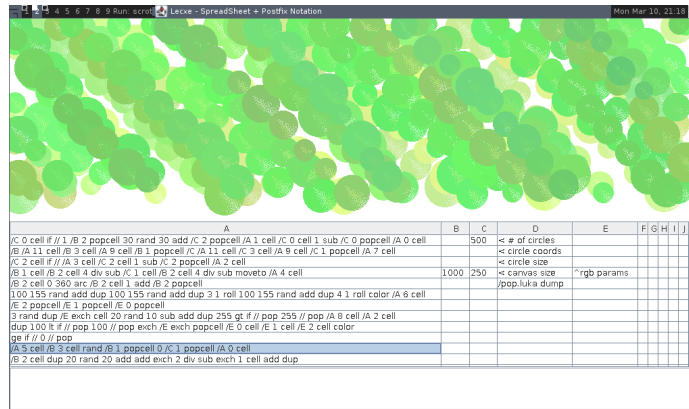
Matthew Horton 2012



Jonathan Ermel 2012



Jonathan Ermel 2012



William Pearson 2014 — modified LVM with **if** and **execute_cell** statements

Last Modified: March 18, 2014