Debugger

Minju Kang & Kristen Behrakis

What Are we implementing?

The Debugger: Main Features

Debug Mode

Our debugger lets user to enter debug mode or regular run mode. In debug mode, the user can evaluate an expression step by step or block by block.

Breakpoint

Having a breakpoint in the middle of user's expression lets user to check the status of the environment.

Stepping

Step Into > Evaluates line-by-line

(+ 3 **(* 4 (+ 5 5))**)

-- Step into --

(+ 3 (* 4 **(+ 5 5)**))

Step Over > Skips over current block

(+ 3 **(* 4 (+ 5 5))**)

-- Step into --

(+ 3 **40**)

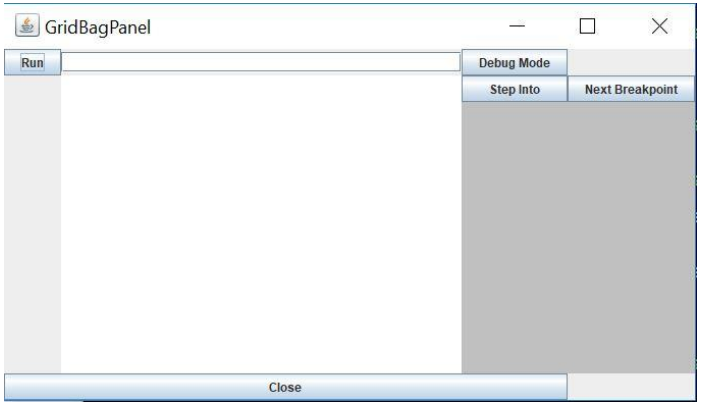
How did we do it?

An Overview: 2 Versions

Command Line

```
Debug mode (yes or no)?
TFUNC>
into or over for expression (+ 4 5):

into or over to simplify (+ 4 5):
Value: 9
9 : int
Elapsed time: 5125ms
```



GUI

Command Line

- Printing evaluation steps line-by-line
- Prompts during the evaluation function
- toString() methods
- Breakpoint expression and parser
- Environment extraction

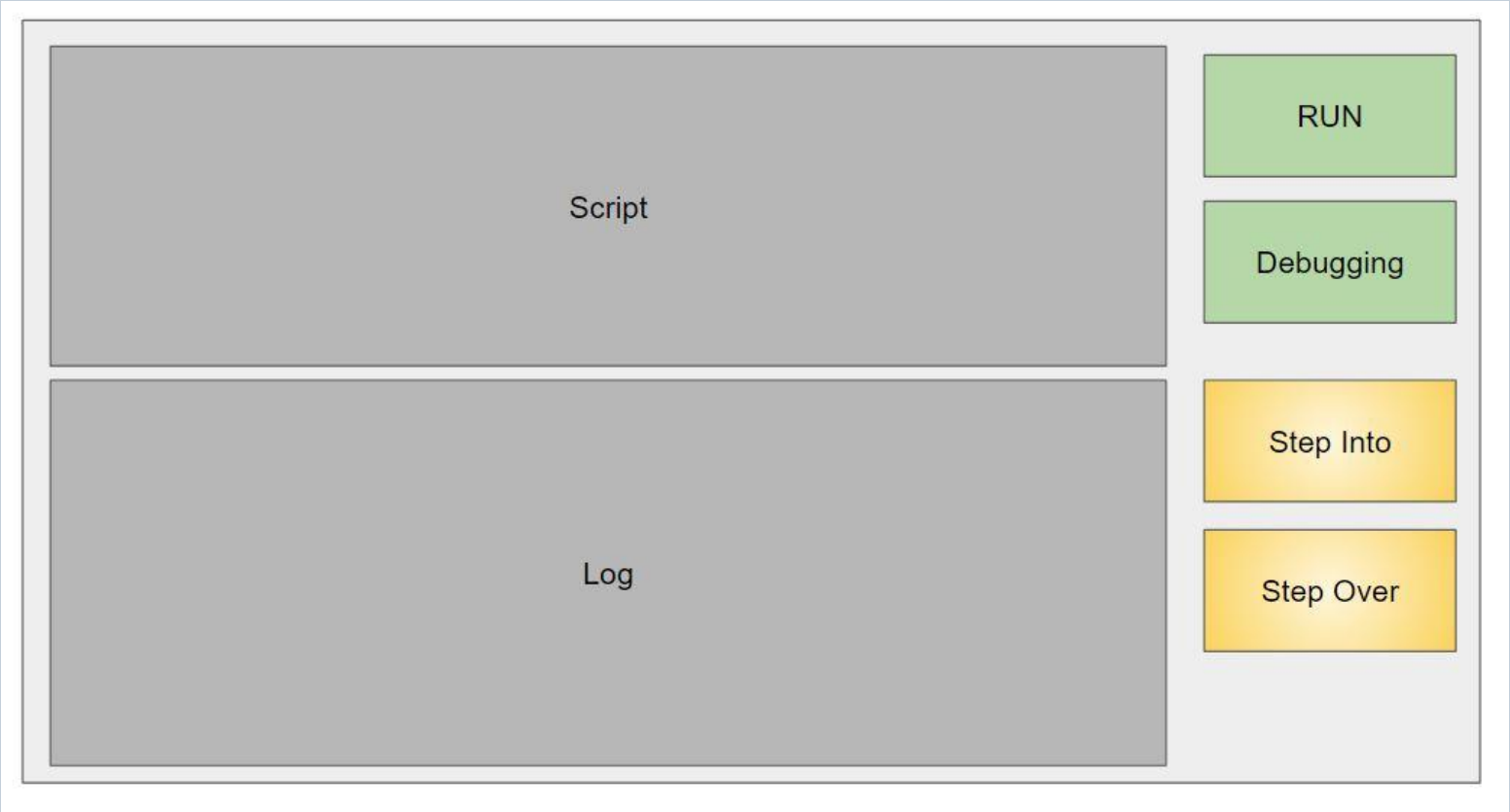
GUI

- Redirect println to GUI? NO
- Text file
- Parsing the text file
- Button functionality

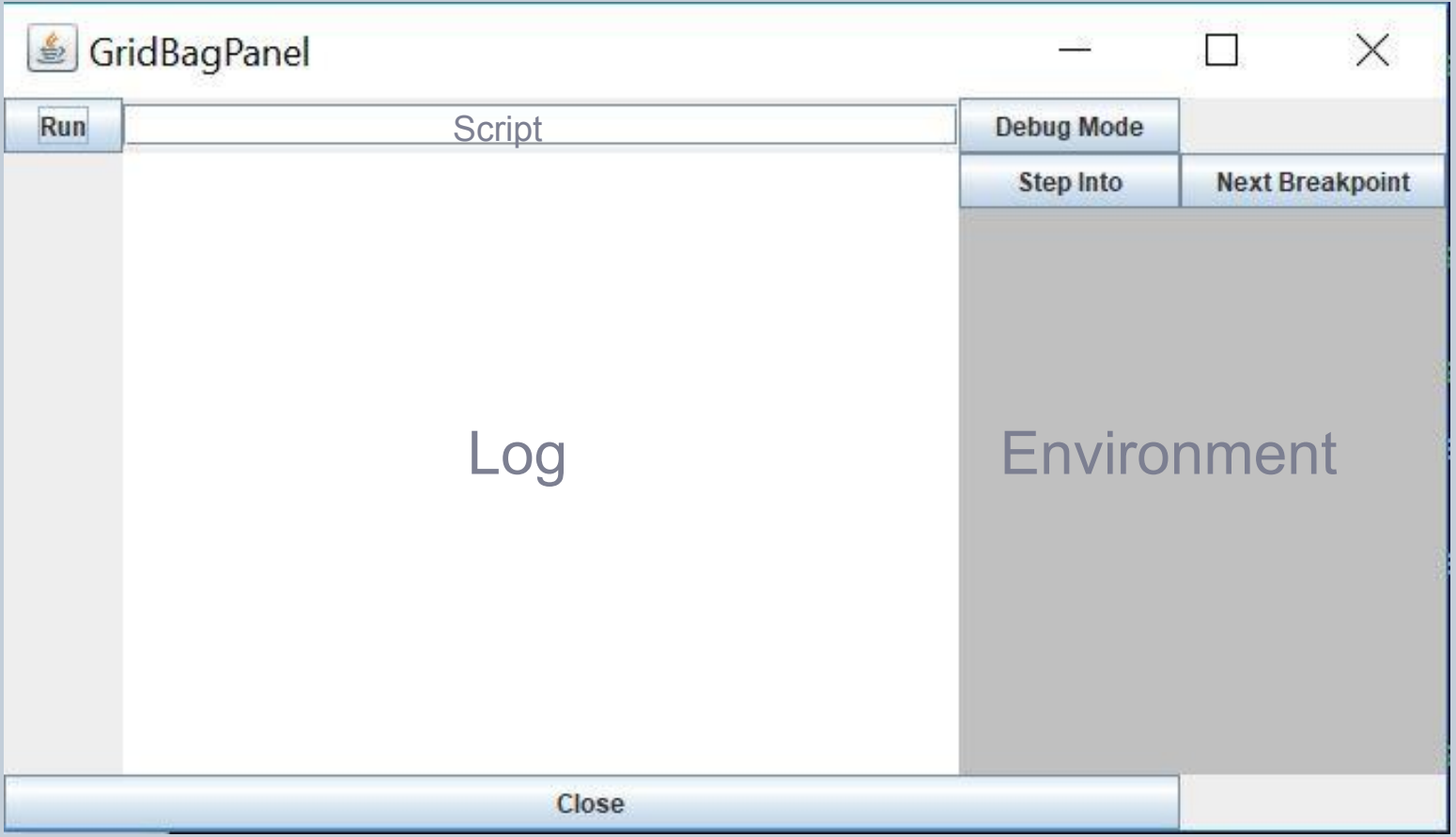
GUI

What actually happens...

1. Flushes the output in the form of .txt file
2. Reads the line from the saved .txt file
3. Display the imported line @ gui



Plan for our GUI



Our actual GUI

IT'S DEMO TIME!

See what we've got!!!

Let's Dive In

Command Line : Debug Mode

```
while (true) {
    try {
        var isDebug = false

        print("\nDebug mode (yes or no)? ")
        val debugInput = scala.io.StdIn.readLine()
        if (debugInput == "yes"){
            isDebug = true
        }

        print("\nTFUNC> ")
        val input = scala.io.StdIn.readLine()
        val se = parse(input)
        val result = time { se.processEntry(env,symt, isDebug, isDebug) }
        env = result._1
        symt = result._2
    } catch {
```

Command Line : Step Into and Step Over

```
while (true) {
    try {
        var isDebug = false

        print("\nDebug mode (yes or no)? ")
        val debugInput = scala.io.StdIn.readLine()
        if (debugInput == "yes"){
            isDebug = true
        }

        print("\nTFUNC> ")
        val input = scala.io.StdIn.readLine()
        val se = parse(input)
        val result = time { se.processEntry(env,symt, isDebug, isDebug) }
        env = result._1
        symt = result._2
    } catch {
```

GUI: Button functionality

Debug Mode

```
// IF DEBUG BUTTON WAS PRESSED
case ButtonClicked(component) if component == toggle =>
  if (toggle.selected){
    display.text = "**** ENTERED DEBUG MODE ****\n"
    inDebugMode = true

    // Only listen to step buttons when in debug mode
    listenTo(stepInto)
    listenTo(breakPoint)
  }
  else{
    display.text = "**** EXITED DEBUG MODE ****\n"
    inDebugMode = false

    // Step buttons should not work outside debug mode
    deafTo(stepInto)
    deafTo(breakPoint)
  }
```

GUI: Button functionality

RUN

```
// IF RUNNING THE CODE
case ButtonClicked(component) if component == runButton =>
  // Make sure file is clear before beginning the new evaluation
  val writer = new PrintWriter("steps.txt")
  counter = 0
  numLines = 0

  display.append("\n ------ Evaluating ------ \n")

  shell()
  for(line <- Source.fromFile("steps.txt").getLines()){
    numLines = numLines + 1
  }
  if(!inDebugMode){
    val wholeFile = Source.fromFile("steps.txt").getLines()
    val resultLine = wholeFile drop((numLines-2)) next()
    display.append(resultLine)
    counter = 0
    numLines = 0
  }
```

GUI: Button functionality

Step Into

```
// IF STEPPING INTO
 case ButtonClicked(component) if component == stepInto =>
   into = true

   val currentFile = Source.fromFile("steps.txt").getLines()
   val currentLine = currentFile drop(counter) next()

   if(counter < (numLines-1)){
     display.append(currentLine)
     display.append("\n")
     counter = counter + 1
   }
   else{
     counter = 0
     numLines = 0
   }
```

GUI: Button functionality

Breakpoint

```
// IF BREAKPOINT
 case ButtonClicked(component) if component == breakPoint =>
   val currentFile = Source.fromFile("steps.txt").getLines()
   var finalCounter = -1

   // When the line is not equal to the breakpoint yet
   for(line <- currentFile){
     if (!(line == "~ BREAKPOINT ~")){
       // Keeps track of how many lines we've seen
       counter = counter + 1
     }
     else{
       // Keep track of the line number where the breakpoint was hit
       finalCounter = counter
     }
   }

   // If you reached the end of the file without hitting a breakpoint
   if(finalCounter == -1){
     counter = 0
     display.append("No breakpoints found.\n")
   }
   // If you hit a breakpoint, display the next line
   else{
```

Next Steps

Redirect println to GUI

Make our command line code fully integrated with the GUI

Restructuring the Code

Make GUI and evaluation part of the same class to avoid println redirect

Text Highlighting

Highlight the block of expressions that are evaluated by users for better visualization purpose


THANK YOU!!!