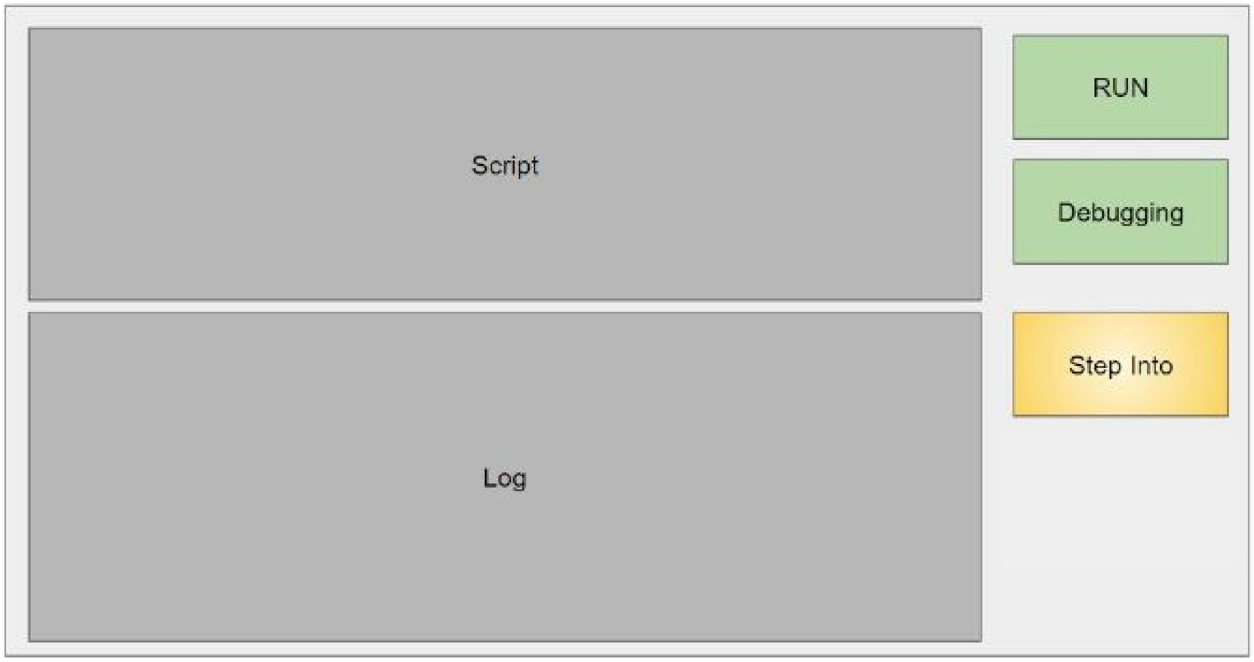
What? (what did we do)

Our goal for this project was to create a visual debugger using the functional language we developed throughout Programming Languages. Our MVP was step-by-step evaluation and breakpoints through the use of a Graphical User Interface. We began with a more recent version of the in-class code. This version includes a type system and uses tail-recursion.

We had two main components of this project. One was a version that works exclusively on the command line and another that includes a GUI. The command line version allows user to step over, step into, or see the environment at a breakpoint. Ideally, the GUI version would have the same functionality, but it would prompt the user with buttons instead of prompts on the command line. We developed the GUI using Scala Swing, which is similar to Java Swing.



Command line version:

The purpose of the command line version was to ensure we could successfully print evaluation steps line-by-line in the existing Shell. We achieved this by prompting users for input (i.e. would you like to step into or step over?) during the evaluation function itself. We also changed all of the existing toString methods, added a new breakpoint expression and parser, and created a function to extract the current environment.

1. **Debug Mode**

- Although we're developing a debugger, we designed it to be a "mode." It would sit atop of the existing interpreter instead of being only a "debugging interpreter."
- To achieve this, we prompt the user and ask if they'd like to enter debug mode before they enter in their expression. The user's response is then taken in as input to the evaluation function.

2. **Step Into vs Step Over**

- Step into allows users to see all the step-by-step evaluations. Step over skips over the given evaluation. Ultimately, debug mode is “shut off” for that evaluation step. To achieve this, we added two new inputs to the evaluation function. It is important to remember that the evaluation function recursively calls itself.
- One of the inputs, called origDebug, was kept constant throughout all recursion. If true, it means the user wants to be in debug mode.
- The other input was called isDebug. It was toggled between recursive calls, depending on if the user wants to step into or step over. If the user wants to step over, isDebug is set to false and any further recursive calls would not print out the evaluation steps. If the user wants to step into, isDebug is set to true and recursive calls would print the evaluation steps.

3. **Breakpoint**

- One of the main goals of breakpoint is to view the environment at the current step. The parser was updated such that, when a user surrounds an expression with “bp,” a breakpoint expression would be called.
- If there’s a breakpoint detected, the user is asked if they’d like to see the environment. If so, the environment is printed legibly on the command line.

GUI version:

Our original goal was to take our command line code and just redirect the println function to our GUI output. After some advice from Riccardo, we decided against this because it would be too significant an undertaking. Instead, we saved our evaluation steps to a text file and printed the lines of the text file in our GUI. Depending on the button pressed by the user, the text file will be parsed differently. Ultimately, we would want to redirect the println function instead of using a text file, but using the text file at least helps us make sure the GUI and our language are compatible.

1. **Adding button functionality**

- We have 4 main buttons on our GUI: Debug Mode, Run, Step Into, and Next Breakpoint
- The debug button is a toggle switch. Debug mode is either on or off.
- The run button calls the Shell and indicates whether or not the user is in debug mode.
- If in debug mode, the Step Into and Next Breakpoint buttons become available. If not in debug mode, these buttons are not operational.
- When the run button is pressed, the text file is populated with all of the evaluation steps, regardless of if the user wants to step into or go to the next breakpoint. Therefore, the step into or next breakpoint buttons must parse the text file. This is definitely not the optimal situation, but it does allow us to test out the functionality of our GUI.

2. **Printing the environment**

- When a breakpoint is reached, it is saved in another text file and printed onto the GUI.

Given that we now know our language is compatible with our GUI, the next main step is to remove the text file stand-in. If we redirect println and user input to our GUI, we will be left with our fully operational visual debugger.

Why Not? (other approaches we considered/tried)

1. Why didn't we have a GUI that could interrupt the eval?
    - Redirecting println to the GUI was not recommended

2. Halting evaluation through threading
    - One option we explored quite a bit was halting the evaluation steps through threading. Scala has threading operations called wait() and notify(). The idea was that we would call our evaluation function and have it wait() before each recursive call. Once the user pressed step into or next breakpoint, it would notify() the thread of the action.
    - Unfortunately, this action ended up freezing our GUI. When the Run button on the GUI is pressed, the evaluation function runs. If the evaluation function is in a wait(), then the GUI still thinks the run button is being pressed. The user is unable to select step into or breakpoint because the GUI can only handle one button being pressed at a time.
    - It appears that the only way to use threading with the GUI is to separate the run button from the rest of the GUI.

4. Continuation
    - Another really interesting option, but we learned about it too late in the course to actually implement it.

What Now? (future steps)

- Redirect println to the GUI
- Add colors for easier interpretation of out evaluation steps