Read Eval Print Loop and User Interface for the DEX Security Policy Configuration Language

A Major Qualifying Project
Submitted to the Faculty of Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree in Bachelor of Science
in
Computer Science

Submitted By:
Marcus Chalmers
Felix Chen
Timothy Goon

December 11, 2021

Sponsoring Organization: F5 | Shape Security
Project Advisor: Mark Claypool
F5 Advisors: Gilad Bracha, Michael Ficarra, & Daniel Szmulewicz

Abstract

Shape Security is a division of F5 that focuses on online fraud and malicious attack prevention. To configure some of their security policies, Shape uses a functional programming language known as DEX. However, DEX lacked a live programming environment which supported all the features of DEX. To address this, we sought to develop a Read Eval Print Loop, or REPL, for DEX that would provide interactive programming for developers. We built off of an existing DEX tool to create both a command line REPL and a web-based REPL with a UI. These REPLs exceeded the expectations for the project and provided a new development environment that improves upon existing DEX development tools. This new REPL should make development with DEX more streamlined and allow for the testing of some DEX features that were not previously possible.

Table of Contents

Table of Figures

1. Introduction

F5 incorporated, based out of Seattle Washington, is a company that specializes in several services that grow and improve web applications, such as application delivery, availability, performance, and security. F5's mission statement is as follows: "F5's portfolio of automation, security, performance, and insight capabilities empowers our customers to create, secure, and operate adaptive applications that reduce costs, improve operations, and better protect users ("*About F5"*, n.d.)". Acquired by F5 in early 2020, Shape Security focuses on online fraud and malicious attack prevention (Gruening et. al., 2020). Using artificial intelligence and machine learning, to help protect mobile application sessions in real time (*"Enterprise security solutions overview: Shape security",* 2020).

To configure some of their defense capabilities, Shape utilizes a specific, purely functional, decidable, and statically typed language known as DEX (Bracha, 2021). Shape Security uses DEX to configure various security policies. DEX is used because of its reliability, as it will never diverge or crash. DEX is purely functional, but by design not Turing-complete because of limitations over looping constructs. This keeps the field engineers and security experts who need to directly interact with the model in a reliable environment. DEX is run on the Java Virtual Machine, and while DEX itself contains many intentional limitations, DEX does allow foreign function calls to Java which, if not used carefully, can undermine the semantic properties in place. DEX currently protects many major websites in the US and internationally (Bracha, 2021).

Development in DEX has some shortcomings. While the DEX language is complete, development within the language is rather slow as there is no ability for live code testing through a Read-Eval-Print Loop (REPL). Currently, DEX does not support a REPL or any similar tool for interactive programming; it is only currently executable within its primary embedding, an HTTP reverse proxy.

There are three applications which could be used as a DEX development tool. The first is an IntelliJ plugin for DEX. This plugin provides basic support for running and evaluating basic DEX code. The second tool is a DEX tour website. This website was designed as a way to showcase and teach the core features of the DEX language in an interactive environment. It includes a window where users can type DEX code and immediately see the results of the code they type in a text box below. However, there is no documentation on how the tool was made or how to maintain or extend it so it is running with a deprecated version of the DEX compiler. The last tool able to run DEX code is a DEX CLI. It is a program which takes a DEX source file and a JSON file to resolve DEX data bindings as inputs, then it compiles and runs the DEX code producing a report in the terminal. Although all of the tools listed can compile and run DEX code they do not support the more advanced features of the language such as the DEX FFI, and DEX module system.

The purpose of this project is to create a command line interface (CLI) REPL for interactive DEX development. In particular, a DEX developer should be able to redefine declarations and have those redefinitions affect all dependent definitions. This specific REPL behavior is not typical of functional languages, rather it tends to be the way imperative languages' REPLs are implemented. Moreover, if time permits the CLI REPL is to be expanded upon to include features such as a GUI to improve the user experience and further increase developer efficiency.

We implemented this by first expanding on the code in the existing DEX CLI, extending it to behave like a REPL and allow line-by-line input. Moreover, we added features of the DEX language to the CLI REPL that were not supported in the DEX CLI such as DEX FFI and its module system. Lastly, we incorporate the code for the CLI REPL into a full stack web app which allows us to create a UI for the web-based REPL.

In Chapter 2 of this paper we provide background on topics and technologies used in development of the REPL. Chapter 3 and 4 include our initial designs for the REPL and actual implementation details, respectively. Chapter 5 explains how we evaluate the correctness and effectiveness of our REPL. Finally, Chapters 6 and 7 conclude and present future work that can be done with the REPL.

2. Background

In this chapter we first give information on some basic DEX features we mention in later chapters. Next, we define what a REPL is and its features along with an exploration of live programming. Lastly, we cover the technologies we used to implement our project.

2-1. Basic DEX

Specific constructs and features of the DEX language that are referenced later in the document are “let” bindings, “data” bindings, the “report” statement, DEX FFI, and the DEX module system. An example of all of these DEX constructs being used in a program is shown in Figures 1, 2, 3, and 4 which contain DEX code, JSON data input, Java FFI code, and the DEX program output, respectively.

“let” Bindings

“let” bindings are variables in the DEX language. Its syntax is “let <name> = <expr>” where <name> is an identifier name and <expr> is any valid DEX expression to serve as the value of the identifier. The types of “let” bindings are inferred by the compiler but it is also possible to explicitly declare their type of the binding like so: “let <name> : <type> = <expr>”.

“data” Bindings

“data” bindings are DEX’s way of retrieving input values from its runtime environment. “data” bindings are similar to “let” bindings in that they are also variables in the DEX language.

The syntax is “data <name> : <type>” where <name> is an identifier name for the binding and <type> is its data type. The difference from “let” bindings is the way in which a value is assigned to the “data” bindings. The values come from a JSON object that is given to the compiler by the current execution environment. The compiler maps the values of fields in the JSON to data bindings of the same name.

“report” Statement

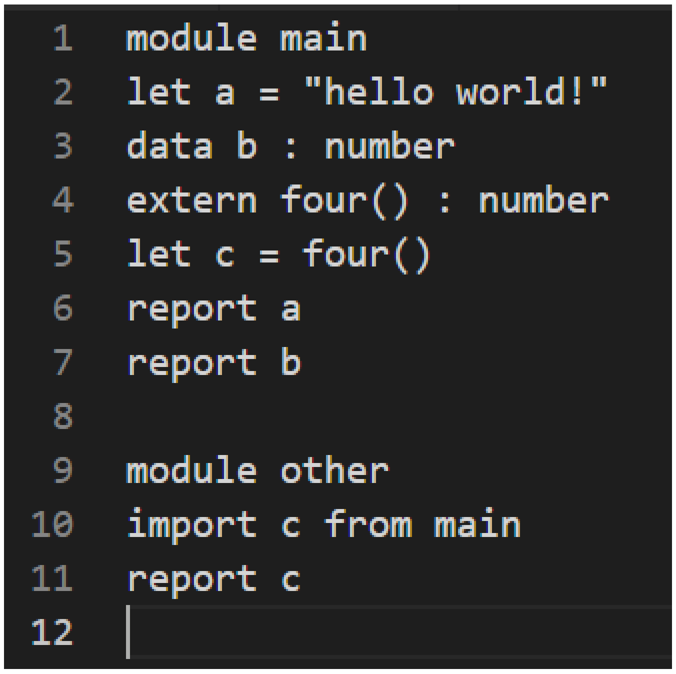
The “report” statement in DEX allows for the computation and printing of “let” and “data” bindings to the console. The syntax for it is “report <name>” where <name> is the name of the identifier that one would want to compute and print the value for.

DEX FFI

FFI stands for “**F**oreign **F**unction **I**nterface”. This feature of the DEX language allows developers to call functions that were written in other JVM languages in a DEX program. The DEX execution environment would instantiate an instance of a compiled JVM class file and give it to the DEX compiler. Then the DEX compiler is able to map the names of the methods in the passed-in object to DEX functions declared with “extern” bindings. Those functions can then be used in the DEX code just as any other function would..

DEX Module System

In DEX the largest building block of a program is called a module. A DEX program may have one or many of these modules. Within any one module one is able to import other modules in part or in whole. DEX also has support for partial modules. This feature allows for many partial modules to be declared with the same name and when passed to the compiler, the compiler will piece together the code in all the partial modules of the same name to form complete modules. This potentially allows for multiple developers to separately write parts of a single module which are all considered valid DEX at compile time.

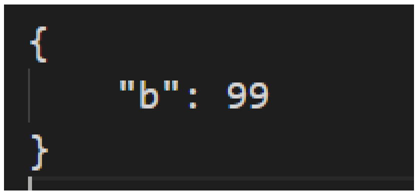


```
module main
let a = "hello world!"
data b : number
extern four() : number
let c = four()
report a
report b

module other
import c from main
report c
```

Figure 1:

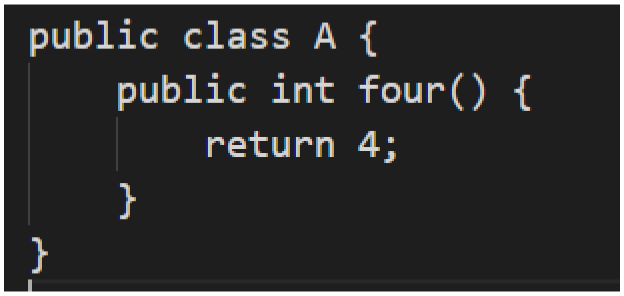
Example DEX Program



```json
{
    "b": 99
}
```

Figure 2:

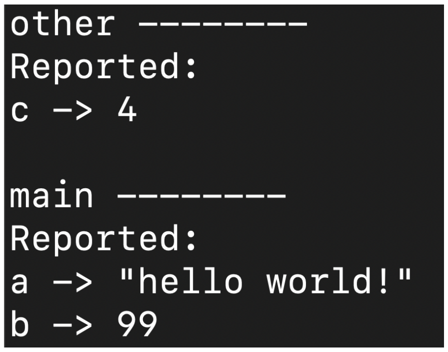
JSON Input for DEX Program in Figure 1



```java
public class A {
    public int four() {
        return 4;
    }
}
```

Figure 3:

Java for DEX FFI for DEX Program in Figure 1



```
other --------
Reported:
c -> 4

main --------
Reported:
a -> "hello world!"
b -> 99
```

Figure 4:

Output from DEX Program in Figure 1

2-2. REPLs

A Read-Eval-Print Loop or REPL is a command line interface that allows programers to run groups and individual lines of code, evaluating the result in real time. Because of their ease of use and access, REPLs have become a popular tool in software development, allowing for fast testing and debugging. Because full size projects or files take longer to edit, compile, and run, the benefit of REPLs, especially when the user only wants to evaluate a single line, allows for faster development (Thomas Van Binsbergen et. al., n.d.).

While learning about the functions of REPLs, our group came across several different examples, along with a research paper dedicated to classifying many of them. Freely available REPLs can be found for many of the most popular languages including CLing (C/C++), Jshell (Java), Python, Node.js (JavaScript), and SQLite. To classify as a REPL, the interpreter has to have "the ability to execute multiple code snippets across multiple interactions in a single session (Thomas Van Binsbergen et. al., n.d.)." Because users often want to execute several lines in tandem, most of the REPLs investigated allowed for snippets to be evaluated incrementally, building on the last. Alternatively, some REPLs allow users to compile several lines into a small program and evaluate all the lines together. Most of the features found were optional additions to improve the user experience but were not directly necessary for the Read-Eval-Print Loop. Minor improvements included auto-completion and help commands which assist users' ability to learn and write code in the given language. Saving state was a large area of improvement where some REPLs would save state and allow users to view command history, or switch between different states or sessions. These features have advantages as they clearly improve many users' experiences, however, the core definition for what constitutes a REPL is fairly simple (Thomas Van Binsbergen et. al., n.d.).

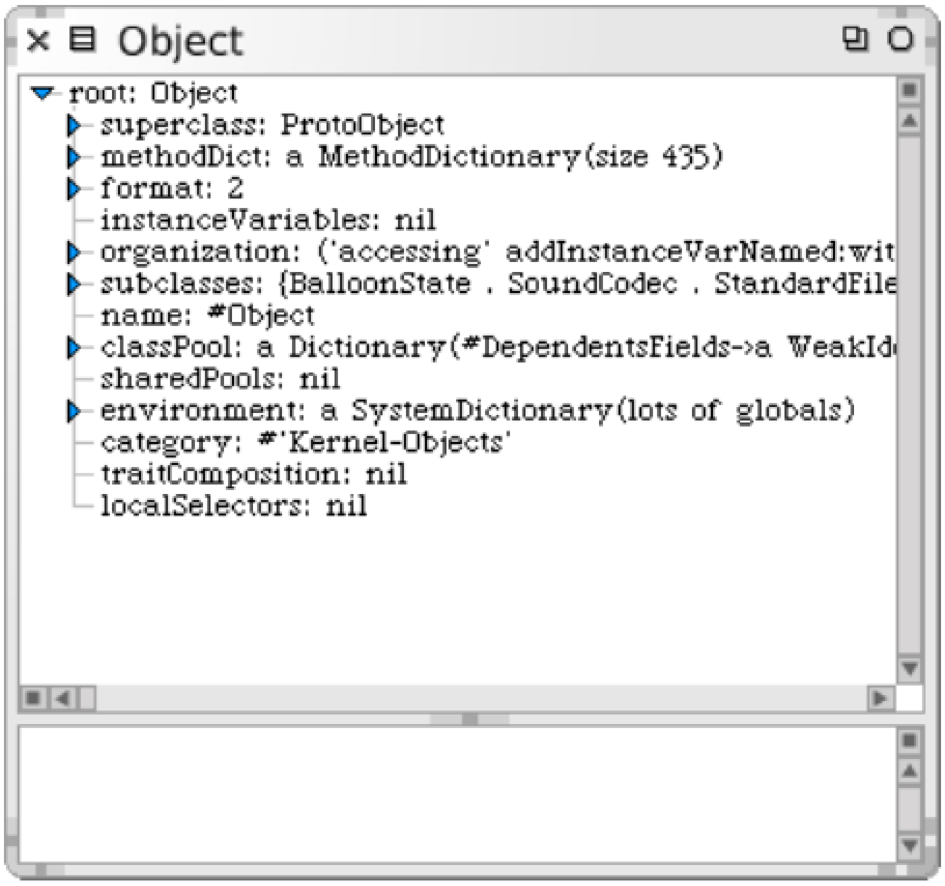
From our research, we have come across three different types of REPLs. The first of these types is the simplest to understand. However, it can only be created if the language one wants the REPL for, can be run line-by-line, and the language has loops and some sort of *eval()* function. This *eval()* function takes a string as input and will run that string as a line of code in the current program execution context. As an example here is a REPL written in Common Lisp for Common Lisp: "(*Loop* (Print (Eval (Read))))" (*REPL*, n.d.). We identified a second type of REPL for any language that can still be evaluated line-by-line but may be missing loops or an *eval()* function. The code of this type of REPL would look similar to a basic compiler. The minimal steps that this REPL would need to go through would be 1) to lex/tokenize an input string, 2) parse the stream of tokens into a parse tree, 3) evaluate the statement based on the parse tree structure, 4) update/maintain the context of the running REPL with the results from the previous step, and 5) loop back to the beginning ("*Making your own programming language",* n.d.). Lastly, the third type of REPL is sort of a "fake" REPL where each line input into the REPL is not being run alone. The REPL holds onto an entire program and adds to it for every valid line of code input. After every instance of input the REPL will run the compiler to compile and run the program it is holding onto in its entirety. This can be seen in Gore, the REPL for the Go language (Thomas Van Binsbergen et. al., n.d.). For the REPL we have built, because of the limitations of the DEX language and compiler, the REPL we created would be categorized in the third type of REPL defined here.

2-3. Live Programming

Dead programs or dead code are typically what programmers work with. Dead code is code that is written in some text file and then compiled into an executable that can no longer be modified. Once the program is running, no additional changes can be made. A live program on the other hand, is a program that is dynamic and changes over time (Bracha, n.d.). When working on live programs, there is a lot more focus on the coding environment. While the program is running, one can typically view the states of the variables one is using and also make changes which will immediately update the program as well.

To get a better idea of what live code and live programs looked like, we learned about Smalltalk. Smalltalk is an object-oriented dynamically typed reflective programming language (Squeak.org, n.d.). A reflective programming language is a language which allows for examining and modifying its own structure and behavior. Smalltalk is a lot closer to live code than dead code, and most Smalltalk systems run on a virtual machine where the entire program state is stored.

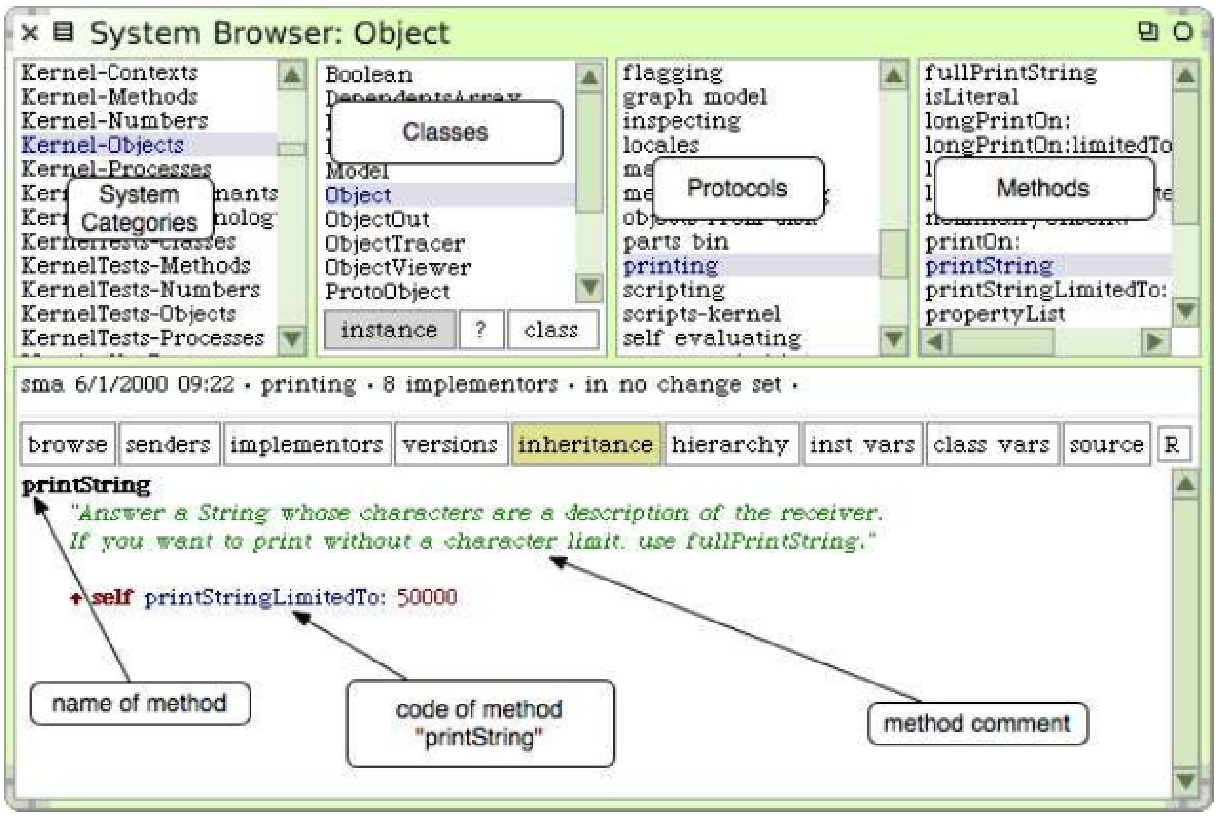
Squeak is one implementation of Smalltalk that we examined because it is a popular open source Smalltalk implementation with fast execution environments for all major platforms (Squeak.org, n.d.). Squeak runs on a virtual machine and saves all of the data from the virtual machine to an image file. Within Squeak, there are different menus that one can open that allows for the viewing of various program information. There is a workspace which acts similarly to a REPL where one can type lines of code and then run them and see the results. One can also choose to “explore” any object or expression and Squeak drills down into that object and shows various information about that object (Squeak.org, n.d.). Figure 5 shows the information in Squeak's object explorer.



```
root: Object
  superclass: ProtoObject
  methodDict: a MethodDictionary(size 435)
  format: 2
  instanceVariables: nil
  organization: ('accessing' addInstanceVarNamed:wit
  subclasses: {BalloonState . SoundCodec . StandardFile
  name: #Object
  classPool: a Dictionary(#DependentsFields->a WeakId
  sharedPools: nil
  environment: a SystemDictionary(lots of globals)
  category: #'Kernel-Objects'
  traitComposition: nil
  localSelectors: nil
```

Figure 5
Exploring an object
Note. From *Squeak by example*, by A. Black et. at, 2007.

Squeak has a system browser which is one of the main tools for programming. Within the system browser, one can view different objects and see how they can interact with other objects. This is also where one can add and modify classes and methods. Figure 6 shows the System Browser along with descriptions of what is shown.



sma 6/1/2000 09:22 · printing · 8 implementors · in no change set ·

```
printString
    "Answer a String whose characters are a description of the receiver.
    If you want to print without a character limit, use fullPrintString."

    ↑self printStringLimitedTo: 50000
```

Figure 6:

System Browser

Note. From *Squeak by example*, by A. Black et. at, 2007.

Squeak shows a lot of information that could typically only be seen while running a "dead program" when using a debugger. This information is useful as one can check the state of any object at any time, and even add in new classes or methods while a program is running. When we created the REPL for DEX, we aimed to make the environment allow the DEX program to behave closer to a live program than a dead one.

2-4. React

React is an open source frontend web development Javascript framework developed and released by Facebook in 2013 (JSConf, 2015). React is component based which means that applications made with React consist of UI elements bundled together as components that manage their own state. As an option when developing in React, users are able to write code in jsx syntax which is an extension of base Javascript. The syntax mixes HTML syntax into Javascript so that users are able to more easily create UI elements in their React components.

Developers currently have two options when it comes to creating React components. The first way is creating classes which represent a React component and overriding methods within those classes that determine the components' behavior during the components' lifecycle. The UI is created by returning UI elements from the class's render() method (Facebook inc., n.d.[c]). The second way is using React functional components. Functional components are all just Javascript functions that return UI elements like the render() method described. The difference is in how developers interact with the component's lifecycle which is done with React Hooks. These Hooks are functions designed to be used within a functional comonent's code to "hook" into the component's lifecycle. (Facebook inc., n.d.[a]) The framework on its own only supports the creation of single page applications and external libraries are required to handle routing for multi-page applications. The frontend of our full stack web-based REPL was implemented as a single page application using the React framework.

2-5. Typescript

Typescript is an extension of the Javascript language that was released by Microsoft in 2012 that is able to run both client and server-side and transcompiles to Javascript. The language enables static typing and is meant to be used in large projects where one would want the extra safety while coding that static typing delivers. Its syntax allows developers to declare types onto Javascript functions and variables. It also allows for the creation of custom types (Microsoft, n.d.). The React frontend of our web-based REPL was configured to be written in Typescript.

2-6. Spring and Spring Boot

Spring is a framework that can be used for creating Java web applications that greatly reduces the boilerplate code developers need to write. It is advertised as a fast and secure framework that increases developer productivity. Spring Boot is an extension of the Spring framework that further eliminates boiler plate code in Spring applications by auto-configuring apps with the most commonly used settings and dependencies (VMware. n.d.[a]; VMware. n.d.[b]; Baeldung, 2021). In our project Spring Boot was used for the backend when we incorporated our CLI REPL into a full stack web app.

3. Methodology

In this chapter we first describe in detail our rationale in choosing a starting point for our project. Next, we list the features we planned on implementing into our REPL. Lastly, we describe our plan for creating and integrating the CLI REPL into a web app and the accompanying UI mockups.

3-1. REPL Base

There are two approaches that we could have taken for developing a DEX REPL. The first option was working off the DEX CLI repo which allows users to run a DEX source file, making a REPL on the JVM. Our second option would be to build a web application off of the Composer-API, a web API that is used for F5's policy composer application which allows users to select common security policies and auto-generate DEX code for those policies.

3-1-1. Working off the DEX CLI

The DEX CLI is built directly off the DEX compiler in Java and allows users to give the program a DEX file and a JSON file input. The "data" bindings in the DEX file are first mapped to any fields specified in the JSON input that have matching names. Then the DEX program is compiled and run. The output of the CLI is a string that is a report of the DEX program.

If we chose to make the REPL off the DEX CLI, the prerequisites to get started were simple since we would have two working examples of the code we can run locally, one in Java and the other, a port of the same code in Clojure. With the DEX CLI we were certain we could run the DEX code and get the program report which is critical functionality for building a REPL. Also, our contact in F5 was the most familiar with the DEX CLI code, so we had quick access to help if needed.

However, creating a UI for a REPL off the DEX CLI may have been more work since development for offline apps using Java usually takes more work than just writing some HTML and CSS. One variant of the DEX CLI is written in Clojure and no one on the team had any experience with Clojure. That meant everyone may have needed to learn a new language for the implementation of the project.

There were two main choices for the language to implement the REPL if we worked off the DEX CLI. These two languages are Java and Clojure because currently there are two identical ports of the DEX CLI in those languages. We would implement the REPL in Java if we choose to work off of the DEX CLI because, of the relevant languages that compile to the JVM, we were the most comfortable with Java. Furthermore, the DEX compiler was also implemented in Java. Thus, the complexity of the codebase would be lower if we used the same language for the REPL. The only drawback of using Java rather than Clojure was that we would have missed out on several features of Clojure that can speed up development such as the interactivity of the Clojure REPL.

The simplest implementation of a DEX REPL would consist of a command line interface (CLI). This would be able to handle most of the main features fairly easily. However some more complicated user interactions would be better suited to a graphical UI. This would be best achieved by creating a web-based UI which could still be accomplished using the DEX CLI as a baseline, and a server providing a frontend web app.

3-1-2. Working off Composer-API

The Composer-API is a tool for users to write DEX policies without actually having to write any DEX code. This tool has a GUI frontend which accepts inputs into different menus and then based on those inputs, generates a DEX file. On the backend, the API has a lot of different endpoints, some of which allow for compiling complete or partial DEX code to verify its validity. If we worked off of the Composer-API, we would mainly be working with this backend API and making calls to it.

If we were to make a REPL with the Composer-API we would have been able to create a web app with a UI as a starting point rather than having to first create a CLI in a JVM language. The team has lots of web-based experience (Javascript, React, etc.), therefore we would have been able to get started with the project fairly quickly. The Composer-API may have contained many endpoints that would make implementing the logic for a REPL easier. One example is the ability to use an endpoint that compiles a DEX file rather than using the compiler to compile DEX code manually.

One unfortunate consequence of building the REPL off the Composer-API would have been that we could not have asked our contact in F5 for help with the code because he does not know much about how the Composer-API code works. As we did not have a contact to teach us about the Composer-API code, we did not know much about the API, which means we would have taken more time in the beginning of the project to research the functionality of each API endpoint. The API server is also written in Kotlin and no one on the team had much experience with Kotlin so researching the API would have been more difficult. Furthermore, we did not know if it was possible to “run” the DEX code with the API and get a report like the DEX CLI does. This is a critical functionality for us to implement a REPL. The API uses the DEX compiler under the hood and was not developed with supporting a REPL in mind which could have meant unnecessary clutter or unintended results that could cause problems for what we wanted to do in a REPL. If the API is calling functions in the DEX compiler it may be easier to cut out the middleman and use the DEX compiler directly ourselves like in the DEX CLI.

We would implement the REPL in React and Typescript if we chose to work off of the Composer-API and start off with a frontend web app. Typescript/React allows for comprehensive web design and components provide abstractions for large scale projects. This would make constructing more complex UIs simpler. Because of the team’s familiarity with the language and framework, these would be the fastest tools for creating our application. However, with most of the API code written in Kotlin, incorporating many different languages into the project right at the start could add more complexity than necessary.

3-1-3. Final Implementation Choice

We chose to implement the REPL in Java off the code for the DEX CLI. This is so we could have access to knowledgeable help through our contact at F5 if needed. Also, we could create a web-based UI for the REPL without using the Composer-API by setting up a server that uses the code for the CLI REPL. Lastly, the Composer-API was not built with supporting a

REPL in mind so there were most likely many unwanted side-effects in the Composer-API endpoints when used in a REPL.

3-2. REPL Features

This is a list of REPL features that we planned on supporting independent of any implementation details.

Redeclared/Redefined Bindings Update all Dependent References

The DEX REPL supports redefining of declarations. This means that when bindings, or any declaration is redefined, any and all past references to that previous declaration need to be redeclared as well. This does not keep the standard definition of permanent declarations as is common in most functional languages. As such this behavior is not typical of REPLs for functional languages, however it is common for imperative languages.

Allowing Testing of Snippets of Code and Immediately Returning Results

The DEX REPL functions as a proper REPL allowing for the testing of snippets of code and the immediate return of results. This is the standard behavior for all REPLs.

Undo Command (+ Command History)

In addition to the previously mentioned required features, this project supports some other useful tools including command history. This allows the user to cycle back through previously submitted commands which reduces the need to type similar lines repeatedly. Along with command history, the ability to undo a previous command, including all of its possible redeclarations, increase usability, allowing users to undo large changes that might have otherwise affected tens of previous declarations.

Allow for Inputting Multiple Lines of Code

In order to allow for the testing of small snippets of code, the DEX REPL also allows for inputs larger than just one line. Essentially, small programs could be run line-by-line and the REPL should run all lines in sequence only outputting results for the final line of code.

Allow Users to View the "State" of the Program

As users interact with the DEX REPL there is an interactive way to interact and view the state of the program. All declarations are visible allowing users to "drill down" into the details of specific variables. This inspection of values allows users to understand their program state in a more holistic way, allowing for debugging.

An Option to Save/Load the REPL Session

As users input new commands and update declarations the state of the specific DEX REPL session becomes more and more unique. To save this unique state, the REPL supports the option to save the REPL session allowing for users to return to a previous REPL session, or allow other users to use sessions created by colleagues. This would also require that the REPL has the ability to allow users to load previously saved sessions along with saving them.

Help / REPL Commands

With the addition of several features that are specific to the DEX REPL program itself and not just within the DEX language, this program will need specific commands for executing REPL options like saving sessions. This is where a “Help” command would need to be implemented along with other commands for previously mentioned features where the leading character is not one found in the natural DEX language. This would differentiate commands for the REPL from DEX code.

3-3. CLI REPL

We completed the CLI REPL using the DEX CLI described previously, writing code to directly interface with the DEX compiler. Our plan for the flow of the CLI REPL’s code was as follows. The CLI REPL structure consists of a loop that runs until the user quits the CLI REPL. Within this loop, the CLI REPL first receives an input from the user. After this, the input is checked to see if it is a CLI REPL command. If the line is a valid DEX statement, it will add it to the list of statements that form the state of the current DEX program and check for any binding redeclarations using the DEX compiler’s typechecker. If a binding is detected as redeclared the original binding will be replaced with the new one. Finally, the CLI REPL will parse all of the statements that it has received, compile, run, and print out a report.

3-4. Full Stack Web-Based REPL

After completing the CLI REPL we designed a frontend web UI to go along with the REPL. We implemented the web-based REPL in React and Typescript for the frontend of the web app. Typescript/React allows for comprehensive web design and components provide abstractions for large scale projects. This makes constructing more complex UIs simpler. Because of the team's familiarity with the language and framework, these were the fastest tools for creating the UI for our application. Also, it was requested by our sponsor that we use Typescript for our frontend language. However, configuring the backend server to serve a React app was more complex than serving static HTML, CSS, and JS files.

We implemented the backend of the web-based REPL in Java and Spring Boot. Java was used because the code for the CLI REPL is already written in Java and as a result, there was little configuration needed to interface the CLI code with the server code. No one on the team had experience with backend web development but Spring Boot is the most popular Java backend web framework so it has the largest community and amount of documentation in case we needed help.

3-4-1. Full Stack Design and Architecture

Client Sessions

The web-based REPL state is represented mainly by lists of strings. This is easily represented in a JSON object in the client. The entire web-based REPL state can then be sent to the backend for processing. This way any complexities with session handling and memory on the backend are eliminated.

Backend Server

The Spring Boot server serves the frontend of the app and uses the code from the existing CLI REPL to process and run DEX code received from the clients. To support this the code in the CLI REPL needed to be refactored to be stateless to be able to process web-based REPL states from many different clients at once. The function takes a web-based REPL state as input and outputs a new web-based REPL state along with string output from running the DEX code. This needed to be done if the backend was implemented to not hold any session data of the users and all web-based REPL state will be sent to the backend from the client. Using this implementation greatly reduced the complexity of the backend.

Client Server Communications

Due to the nature of a web-based REPL there is lots of communication between the client and the server. Therefore, it is most efficient to use a websocket connection rather than HTTP to handle all communications aside from serving the initial UI to the client.

3-4-2. Initial UI Considerations and Mockups

Baseline: Standard Command Line Interface

Assuming the bare minimum, the UI for the DEX REPL should at least be structured like a standard command line interface. Allowing the user to submit and execute one line of code at a time, displaying the result afterwards. This would be able to handle several of the main REPL features mentioned in previous sections for the CLI REPL and for more complicated text displays it would be possible to include more complex text displays akin to something resembling emacs.

Another option would be to structure the UI similarly to how a text editor would work. For this option, we would have something that looked like the REPLs in replit. On the left side, we would have a text editor interface where someone could edit a DEX program in its entirety. On the right, there would be an output area where we would show the DEX program report.

Sidebar: State Viewer/Saver/Loader

The side of the standard command line interface should include various amenities such as a state viewer, allowing for the user to save and load sessions. As well as a variable inspection tab which would allow for the user to “drill down” into declarations as was previously mentioned in the design for the features of the CLI REPL.

Multiple Concurrent States (dual windows)

Allowing the user to run multiple DEX REPL states concurrently would greatly increase efficiency in some aspects. The web UI of the REPL could allow for the user to create duplicate CLI windows which would create new sessions or allow for the user to load two separate sessions if needed.

UI Mockups

We came up with the following four UI mockups for the web-based REPL. This first mockup in Figure 7 mimics the functionality of a traditional CLI REPL. Users can input statements in the text box on top and the output for that will be shown in the Output text box on the right. The large box on the bottom left is to show the current state of the DEX program, JSON data input, or Java FFI input. The tabs above the box are to swap between each of them. The DEX program state input box is not editable while the ones for the JSON and Java are. On the bottom of the window there are two buttons that allow the importing and exporting of the current web-based REPL state.

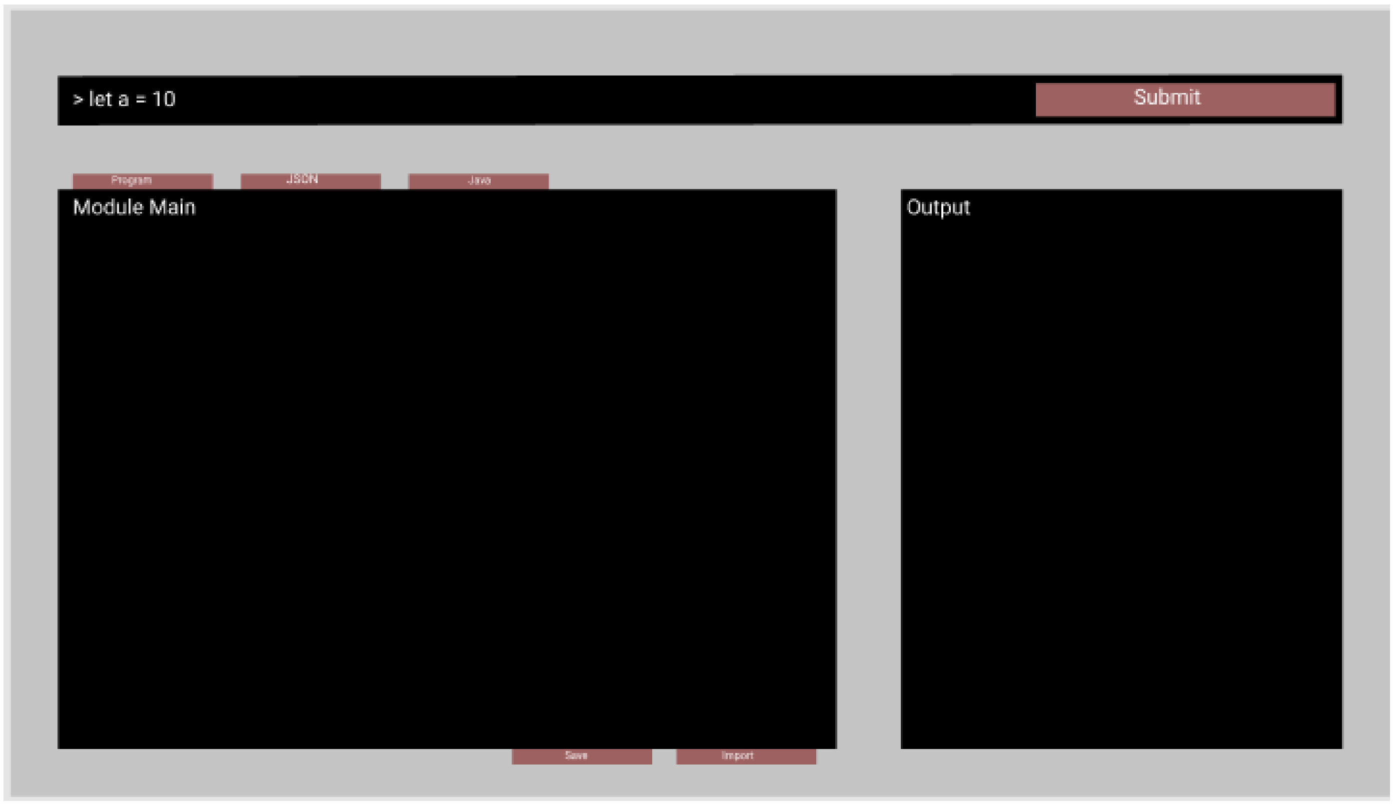
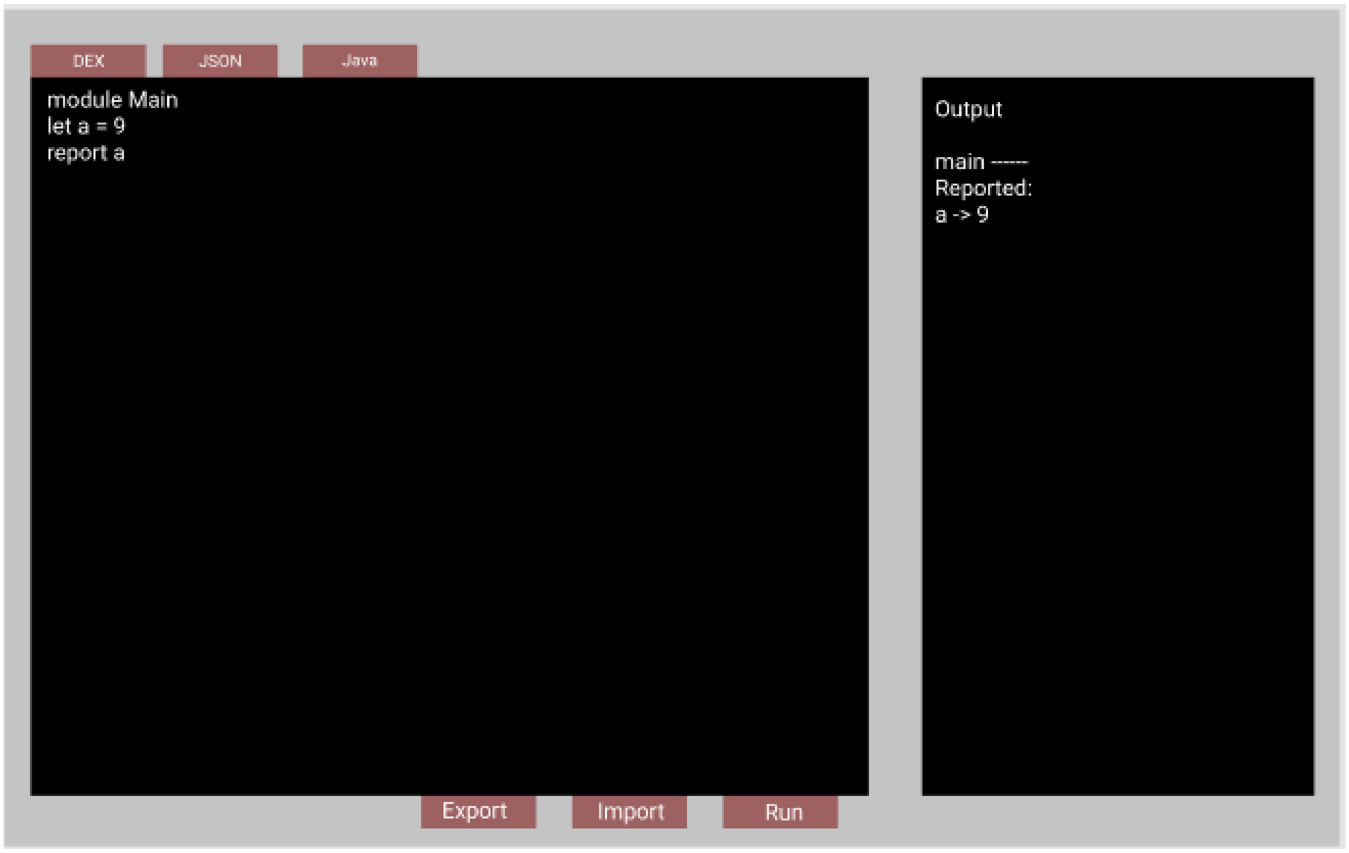

```
> let a = 10
```

Figure 7:

UI Mockup Option 1

This mockup shown in Figure 8 is a second option we considered and is similar to the first but does not include an input box at the top. Instead the DEX program state input box is now editable. This allows for the UI to act as more of a standard IDE where the entire program can be edited at the same time.



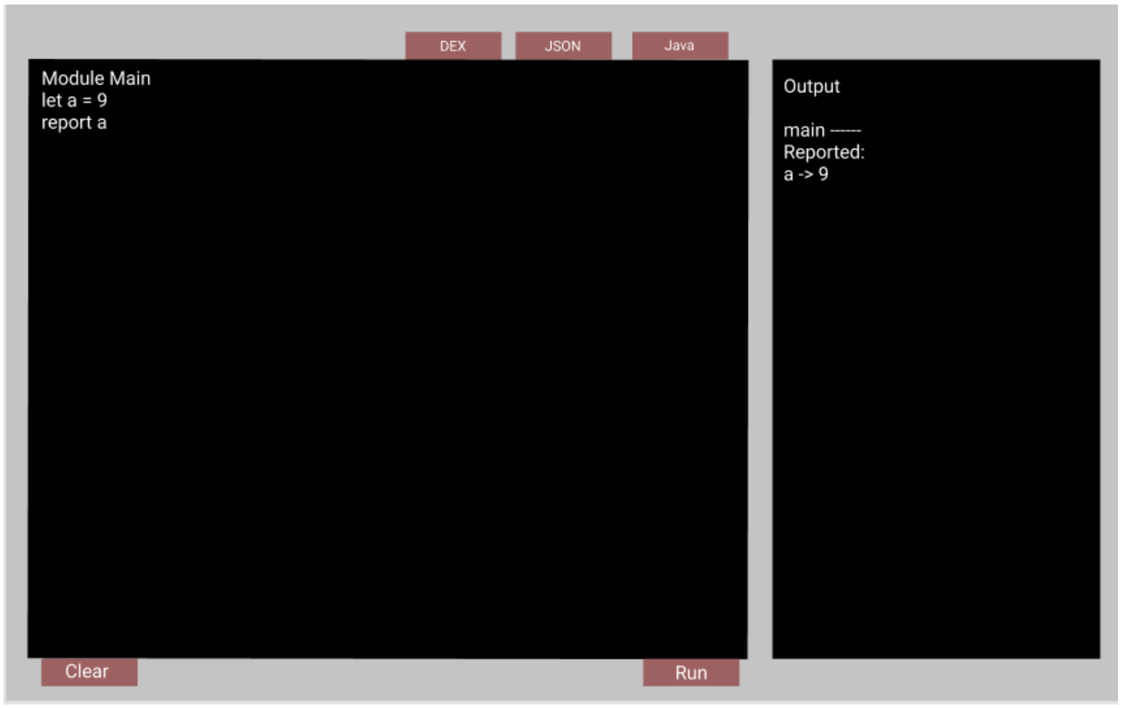
```
module Main
let a = 9
report a
```

```
main ------
Reported:
a -> 9
```

Figure 8:

UI Mockup Option 2

The mockups shown in Figure 9 and 10 are a third option we considered. Unlike the second mockup the tab that shows the DEX code will always be visible on the page. The text box where the DEX code can be edited is a scratch pad where the code in it is not run in its entirety but instead only the highlighted statements. Those highlighted statements act as the lines of input into a web-based REPL which accumulates its program state. The JSON and Java tabs will functionally behave the same way as they would in the previous UI options but would this time appear as a collapsable middle column in the UI. Because of the addition of the highlighting functionality in this option the user needs a way to view and edit the program state of the web-based REPL like in the CLI REPL. That is what is shown in Figure 10 in the middle column. There the program state at the time mirrors the content of the scratch pad and has trash can icons to the right of each line to allow for deleting from the program state. This would be the same feature as the “#del” CLI REPL command.



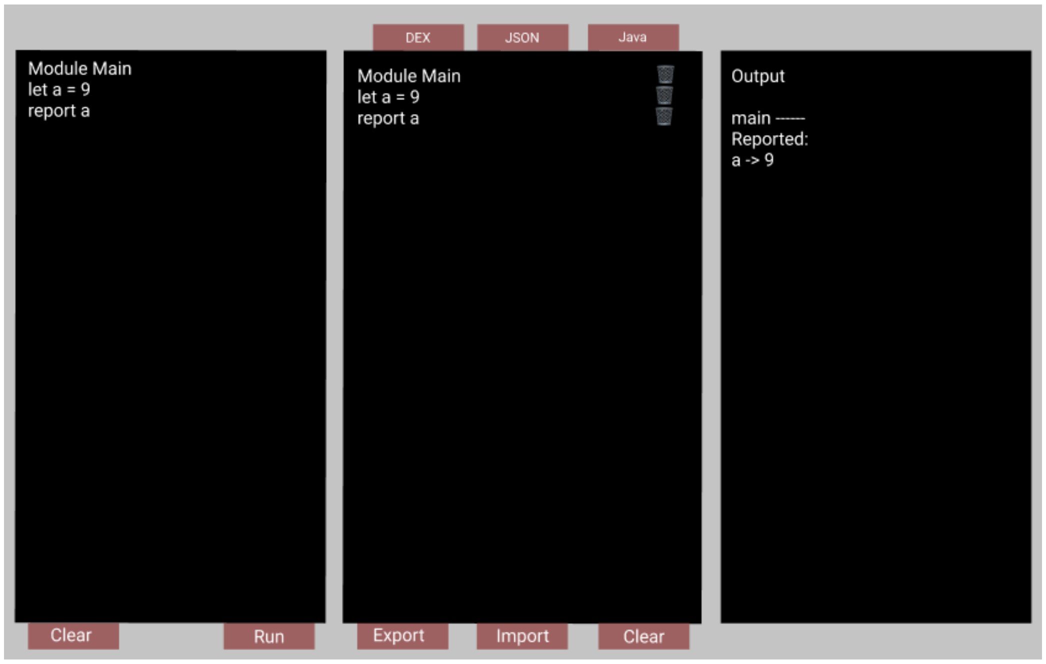
```
Module Main
let a = 9
report a
```

```
Output

main ------
Reported:
a -> 9
```

Figure 9:

UI Mockup Option 3 Screenshot 1



```
Module Main
let a = 9
report a
```

```
Module Main
let a = 9
report a
```

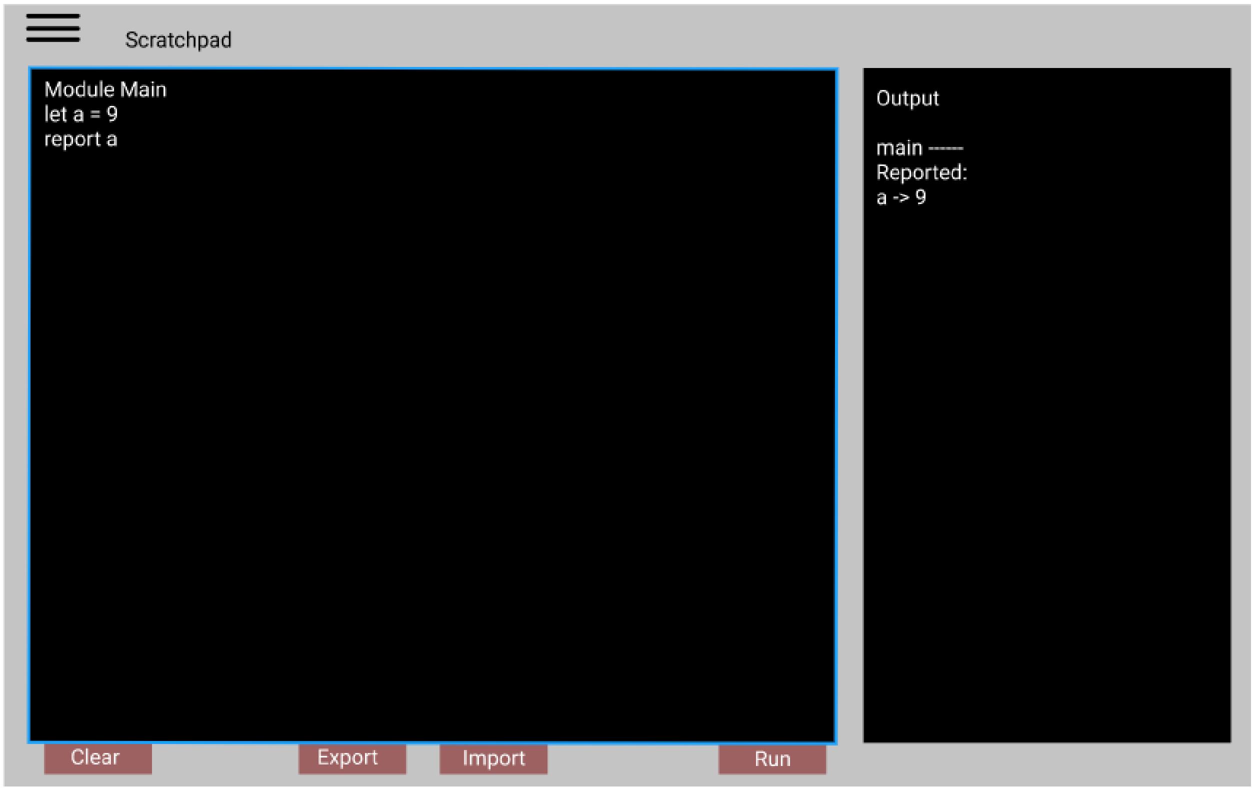
```
Output

main ------
Reported:
a -> 9
```

Figure 10:

UI Mockup Option 3 Screenshot 2

The last UI mockup option we created is shown in Figures 11, 12, and 13. All of the UI elements in this option have the same functionality as in the last option. However, the tabs used to swap between the different windows are now hidden in a hamburger menu on the top left. Also, the window to view the web-based REPL's program state is moved to be shown in the bottom right under the output window when it is toggled on. These mockups in Figures 11, 12, and 13 are what we ultimately decided to base our final UI implementation on for the web-based REPL.

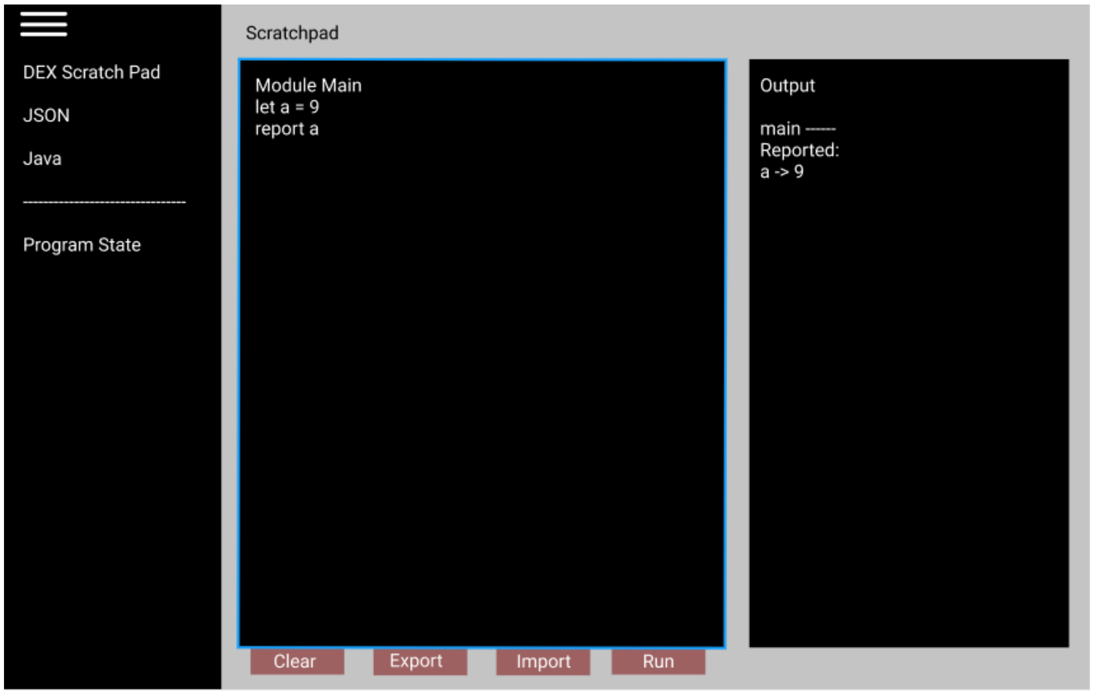


```
Module Main
let a = 9
report a
```

```
main ------
Reported:
a -> 9
```

Figure 11:

UI Mockup Option 4 Screenshot 1



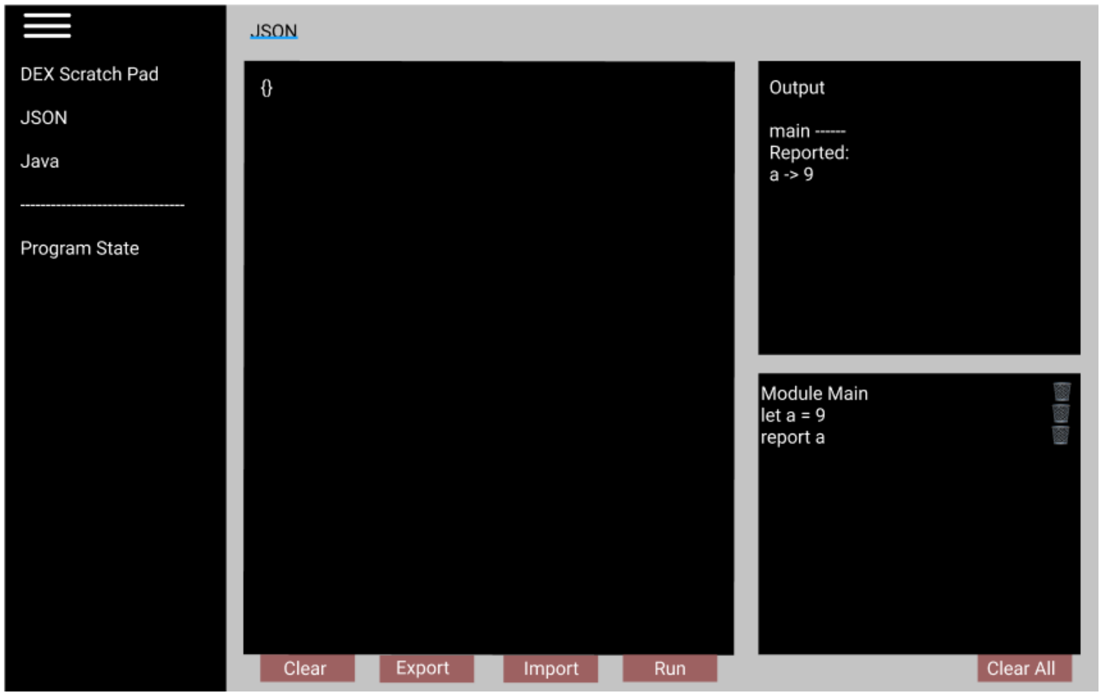
```
Module Main
let a = 9
report a
```

```
Output

main ------
Reported:
a -> 9
```

Figure 12:

UI Mockup Option 4 Screenshot 2



```
{}
```

```
Output

main ------
Reported:
a -> 9
```

```
Module Main
let a = 9
report a
```

Figure 13:

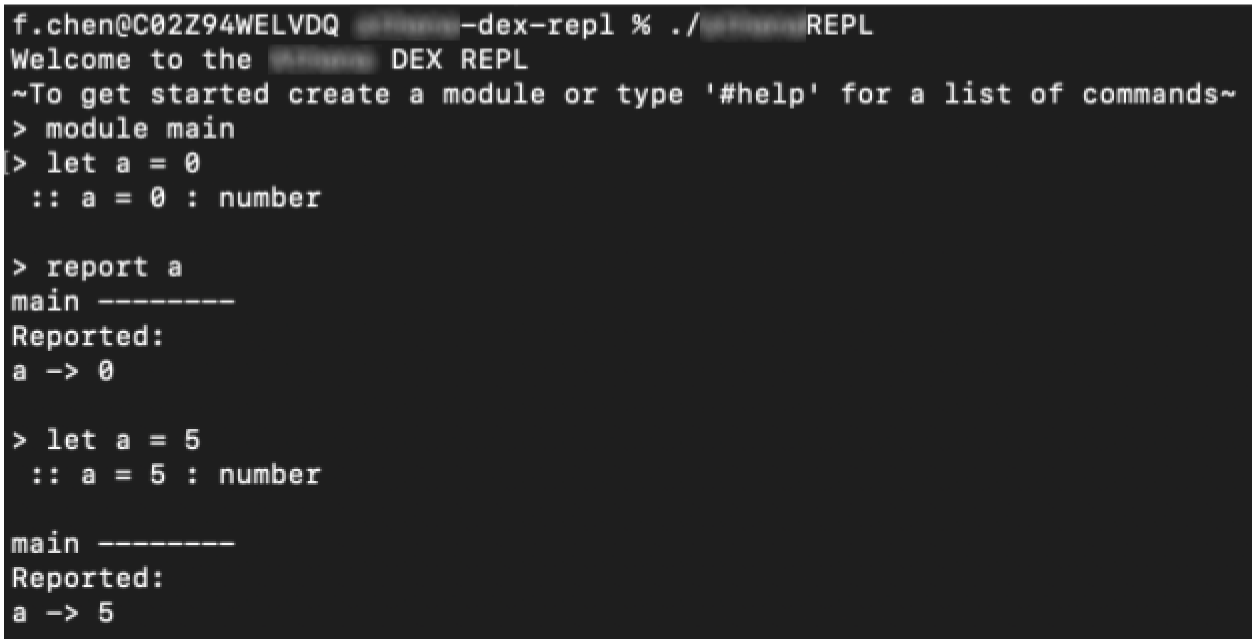
UI Mockup Option 4 Screenshot 3

4. Implementation

In this chapter we go over our implementation of the CLI REPL and the differences from the design described in the methodology. We do the same for the web-based REPL and also provide screenshots of the completed UI.

4-1. CLI REPL

Before diving into the implementation, to provide a complete overview of the finished CLI REPL the image shown in Figure 14 is an example of the CLI REPL being used. After using the executable, DEX code can be input, compiled, and run incrementally.



```
f.chen@C02Z94WELVDQ [redacted]-dex-repl % ./[redacted]REPL
Welcome to the [redacted] DEX REPL
~To get started create a module or type '#help' for a list of commands~
> module main
[> let a = 0
 :: a = 0 : number

> report a
main --------
Reported:
a -> 0

> let a = 5
 :: a = 5 : number

main --------
Reported:
a -> 5
```

Figure 14:

Usage of Completed CLI REPL

4-1-1. Planned Features

CLI REPL State

A class was created to hold all of the state information associated with the CLI REPL. This was done to separate the logic of processing the DEX code from the state information. The class holds the CLI REPL's state history, statements list, command history, JSON data input, FFI instance, and counter for CLI REPL created "let" bindings.

Main CLI REPL Functionality With Binding Redeclaration

In the design of the main CLI REPL functionality we thought that the DEX type checker would be needed to implement binding redeclaration. However, we did not explicitly use the type checker for any functionality associated with the base DEX REPL. This was done because the code to manually check for redefinitions is more robust. The only way to know if the type checker has found a redefinition error is to parse the error message it returns. Furthermore, we did not support redefining bindings when the DEX statements are in different categories (e.g., redefining a "let" binding with a "data" binding). This is because changes like this usually have catastrophic effects on the current CLI REPL state so it was decided to have the users use the "#del" CLI REPL command if they want to force the redefinition. The "#del" CLI REPL command removes a specified index from the CLI REPL state.

To maximize code reuse and organization the code to handle redefinitions were written using the Decorator and Factory patterns. We called the classes in question RedefHandler and RedefHandlerFactory. We handle redefinition by iterating through each DEX module in the CLI

REPL state and keep track of the names of the bindings in a list. We check the list of bindings anytime a new binding is defined. If a binding is redefined with the same category of binding (e.g., replacing a “let” binding with another “let” binding), we replace the original binding with the new binding. If there is a mismatch in the category of binding (e.g trying to replace a “let” binding with a “data” binding), we do not redefine the original binding and display an error in the CLI REPL.

CLI REPL Save and Load Session

The functionality to save and load CLI REPL sessions was implemented as the “#save” and “#load” CLI REPL commands and had no changes from its initial design.

JSON Data Input

In contrast to our initial design, to have “feature completeness” with the DEX language we decided to support JSON data input. We decided to use a JSON file as input for the DEX “data” bindings and not input from the command line so there was no need to manually handle the saving or editing of the JSON values. This feature was implemented as the “#loadJSON” CLI REPL command and loads a JSON file where the contents are set to a string that is parsed later for the DEX “data” bindings.

Undo

The functionality to undo commands in the CLI REPL was implemented as the “#undo” CLI REPL command. There were no changes to this feature’s implementation different from the initial design. As a companion to this feature it was decided to add the “#history” CLI REPL command which simply prints out the previous states of the CLI REPL to be viewed by the user. We thought this may be useful for some users if they were able to view the past states of the CLI REPL to see how many times they would need to use the "undo" command to reach their desired CLI REPL state.

Command History

The functionality to view the CLI REPL's command history was implemented as the "#cmd" CLI REPL command. There were no changes to this feature's implementation different from the initial design. This was done so that users would be able to view past commands that they input into the CLI REPL and reinput them again if desired.

Multi-line Input

It was decided to not implement the feature to have multi-line input in the CLI REPL. Multiple statements of DEX code can be input into the CLI REPL at once but they must all be on a single line. Because we used a Java Scanner as input into the CLI REPL and already had support for multiple DEX statements on a single line, it was decided that trying to allow "\n" characters in the CLI REPL input string was more trouble than it was worth implementing. We surmised that we would have most likely needed to manually parse the input string and figure out a way to distinguish between keys such as Enter vs. Shift + Enter all to have a slightly more aesthetic input string.

View State of Program / Inspect Variables

The functionality to view the program state was implemented as the “#show” CLI REPL command and did not have any changes from the initial design. However, the feature to view declarations and variables was not implemented. With the “#show” CLI REPL command and the DEX “report” statement there was no point in implementing a separate way to view declarations and variables. That is because with “#show” users are able to view the entire state of the CLI REPL including declared bindings in the program state removing the need to view declarations. With the DEX “report” statement users can print out bindings such as “let” and “data” bindings in their entirety removing the need to have a separate way for users to inspect variables.

4-1-1. Additional Features

There are a variety of CLI REPL features that we implemented but did not plan for in our initial design. These are described in the following sections.

Misc. CLI REPL Commands

There are two additional CLI REPL commands implemented that were not planned for in the initial design. They are the “#reset” and the “#quit” commands. “#reset” resets the state of the CLI REPL by emptying all the lists associated with the CLI REPL state. It was implemented because we thought it would be convenient if users could wipe the CLI REPL and not have to restart the entire CLI REPL program if they wanted to start from an empty state again. “#quit” simply exits the CLI REPL and closes the program. This was implemented so that the CLI REPL would have a way to “gracefully” close rather than forcing the users to use Ctrl + c to exit the program.

DEX FFI

We did not know how DEX FFI worked when designing the CLI REPL thus, no implementation plan was created. Support for DEX FFI feature was implemented so that the CLI REPL would be "feature complete" with the DEX language. The feature is contained within two CLI REPL commands "#loadFFI" and "#loadFFIClass". "#loadFFI" loads a Java source file, compiles it, instantiates an object of the contained class, and sets a variable in the CLI REPL code for later use when the compiler is compiling and running the DEX code. "#loadFFIClass" loads a compiled JVM class file, instantiates an object of the class, and sets a variable in the CLI REPL code for later use when the compiler is compiling and running the DEX code.

Printing of Data Types After Each CLI REPL Input

The print outs of the returned data types of each DEX statement after it is input into the CLI REPL and is fairly common among other REPLs of functional programming languages. It was implemented because the printing of types provides good feedback to the user while using the CLI REPL. The high level implementation is as follows. Given the most recent string of inputted DEX code, the CLI REPL compiles a miniature module which allows for only the most recent line to be parsed. This is done to get an AST from the parser that only contains information from the most recent input. For the types to be resolved correctly the type checker is run on the original CLI REPL state and the information on the types of each new statement is extracted using information from the AST fetched previously. Every DEX input outside of the declaration and importing of a module and the "report" statement should output some text to let the user know that the system has properly received their input.

Input of Arbitrary DEX Expressions

Users are able to input any valid DEX **expression** as an input into the CLI REPL (e.g. 2 + a) to see the computed value. This feature was added to increase the usability of the CLI REPL so that users would not need to always use the “report” statement to print out the computed value of DEX expressions. Our implementation of this feature is as follows. If the most recent line input into the CLI REPL causes a parse error, try to wrap the line in a temporary “report” statement which prints out the computed value of a valid identifier. If there is not another parse error, the most recent input was a valid DEX identifier. However, if there was another parse error, try to instead wrap the inputted line with a “let” binding. This works because the right side DEX “let” bindings only accept valid DEX expressions. The added “let” bindings are named with the scheme “_it#” where “#” is a number that counts up from 0. If the program is parsed again and there is no parse error then the last line input was a valid DEX expression. A temporary “report” statement is added after the “let” binding to compute and print out the value of the expression. Unlike the added “report” statements the added “let” bindings are not temporary so that users can access the computed values from them later.

4-2. Full Stack Web-Based REPL

In this section we describe the integration of the CLI REPL into a full stack web app. A diagram of the overall architecture is provided in Figure 15. Afterwards, we will discuss our completed UI designs and also additional UI features not planned for in the methodology.

4-2-1. Full Stack Design and Architecture

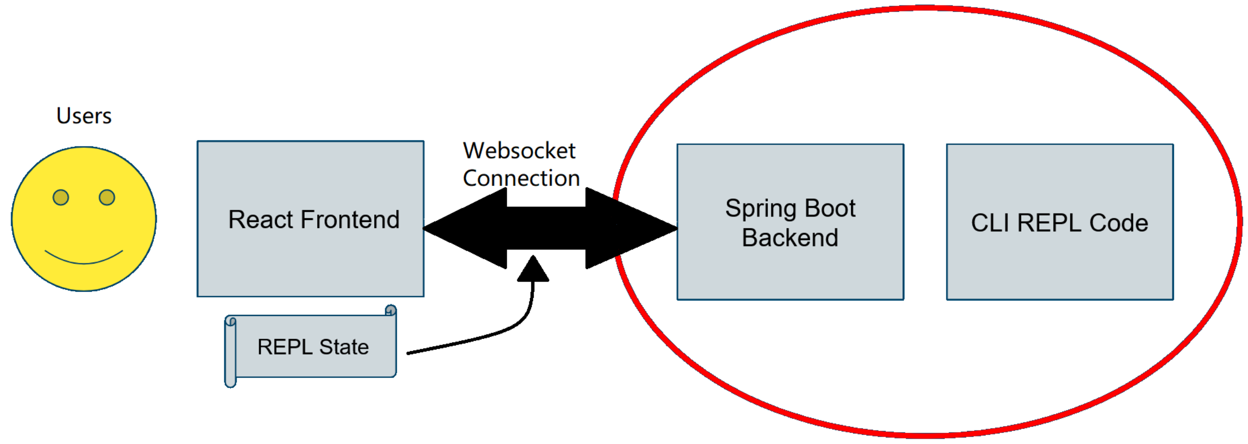


Figure 15:

Web-Based REPL Basic Code Architecture

Backend Server

The built React app is placed into the Spring Boot server's static folder and served up as static files. The web-based REPL is a single page application so no additional configuration was required in the backend.

Client Sessions

The web-based REPL state for the clients are contained within a React Context that provides all of its children components access to both the values of the current web-based REPL state and functions to modify that state.

Client Server Communications

There was no change in the implementation for this from the initial design. The websocket connection for the web-based REPL is managed by the same React Context that manages the web-based REPL state. The Context first tries connecting to the web socket endpoint directly after the web app loads and if the connection ever closes it will continue to retry the connection. The current web-based REPL state is sent through the web socket connection to the backend to be processed.

Integration of CLI Code Into the Backend Server

The only change to the initial design of refactoring the CLI REPL code to be stateless is the need to implement different REPL modes when using the function. This was done because practically all of the commands implemented for the CLI REPL are environment specific and do not work for the web-based REPL. Three REPL modes were implemented, represented by an enumerator, CLI, PURE_DEX, and EXPRESSION. CLI mode is for the CLI REPL and allows both the input of CLI REPL commands and arbitrary DEX expressions in addition to valid DEX statements. PURE_DEX mode only allows valid DEX statements to be input. This mode is currently unused. Lastly, EXPRESSION mode allows for the input of arbitrary DEX expressions but not CLI REPL commands. This is the mode used for the web-based REPL.

4-2-2. UI

The following in Figures 16, 17, and 18 show the completed UI for the web-based REPL. The layout used is similar to the last option in the UI mockups. Figure 16 shows the UI when the web-based REPL is initially loaded. The user is presented with a DEX Scratch Pad in a text window on the left and an Output window on the right. Under those is a menu that opens up when the gear icon is moused over and a button to run the DEX code in the web-based REPL. In the top left corner is a hamburger menu tray that opens when clicked.

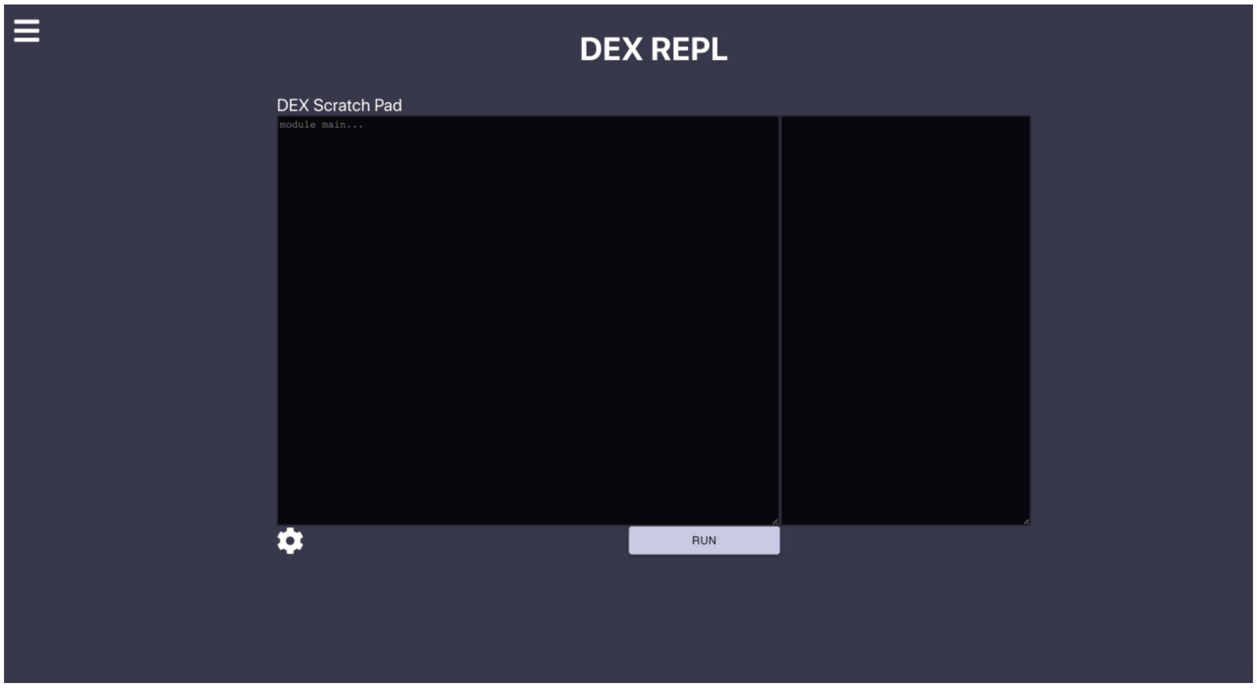
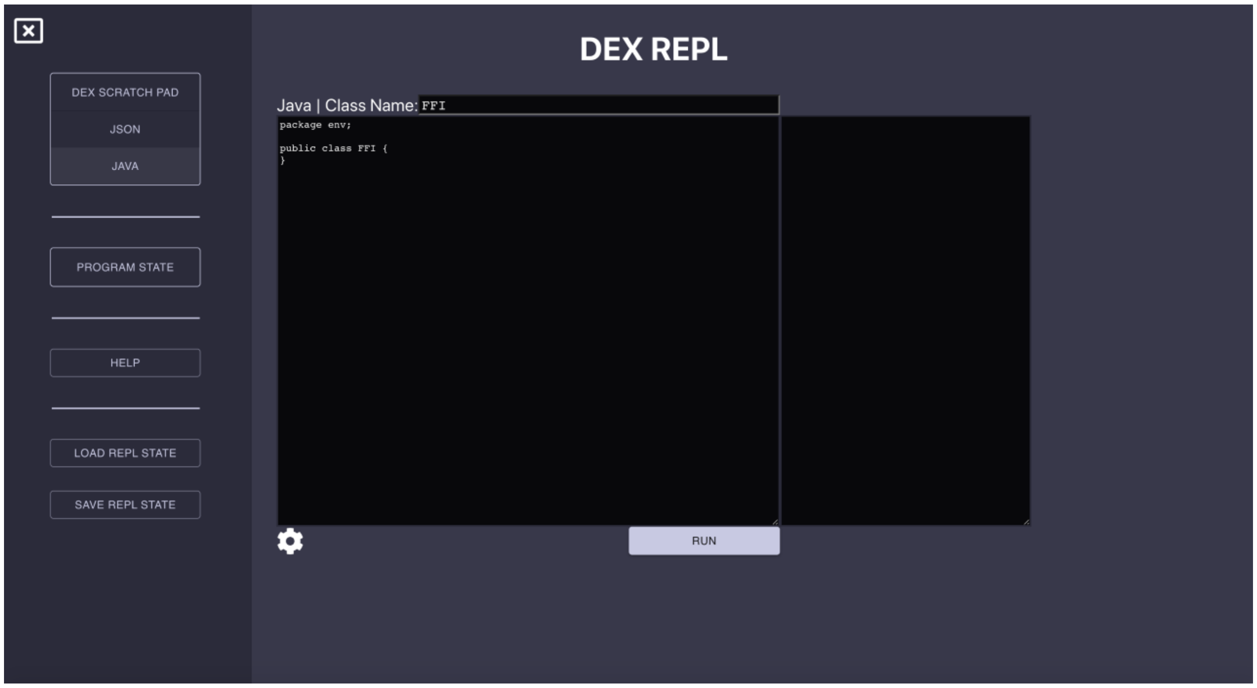


Figure 16:

Completed UI Screenshot 1

The image shown in Figure 17 has the hamburger menu open on the left. Here, the user has clicked on the “Java” button so the DEX Scratch Pad has been replaced with a Java input window.



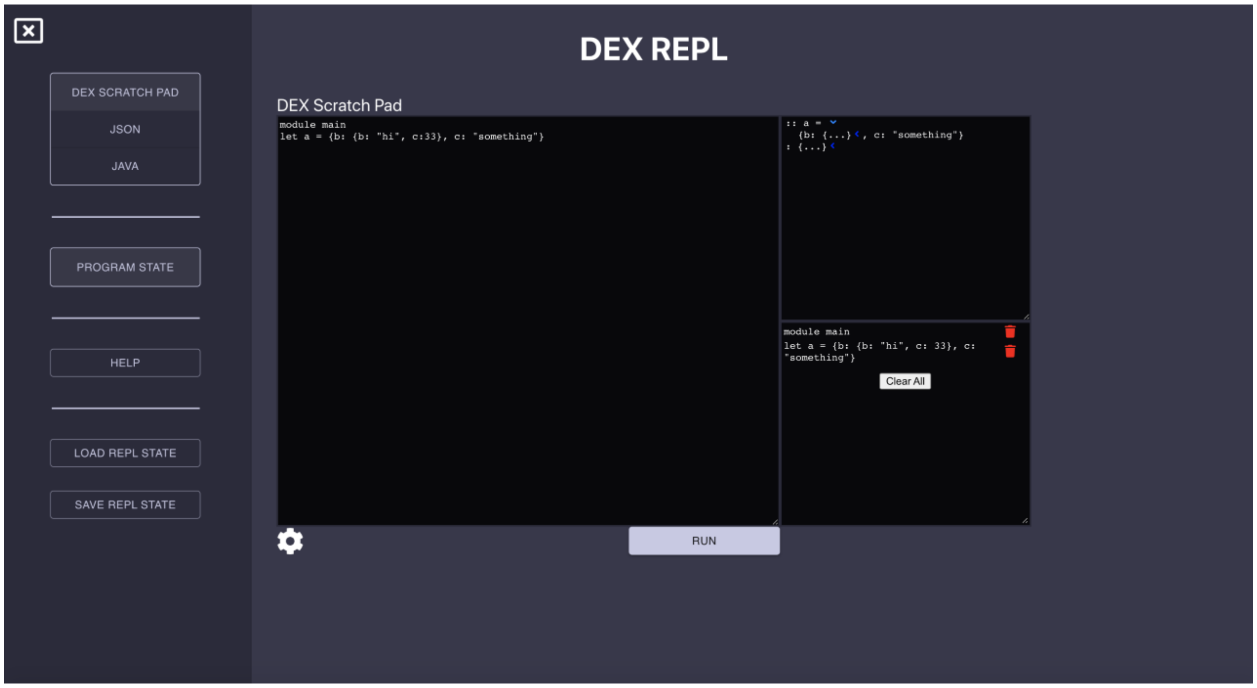
```
package env;

public class FFI {
}
```

Figure 17:

Completed UI Screenshot 2

The screenshot shown in Figure 18 is an image of what the UI may look like when being used. In Figure 14 the DEX Scratch Pad is open with some code written in it. The code has been run so the corresponding output is in the top left window. In the hamburger menu the “Program State” button has been clicked so the window on the bottom right that shows the program state is now open.



```
module main
let a = {b: {b: "hi", c:33}, c: "something"}
```

```
:: a =
  {b: {...}, c: "something"}
: {...}
```

```
module main
let a = {b: {b: "hi", c: 33}, c:
"something"}
```

Figure 18:

Completed UI Screenshot 3

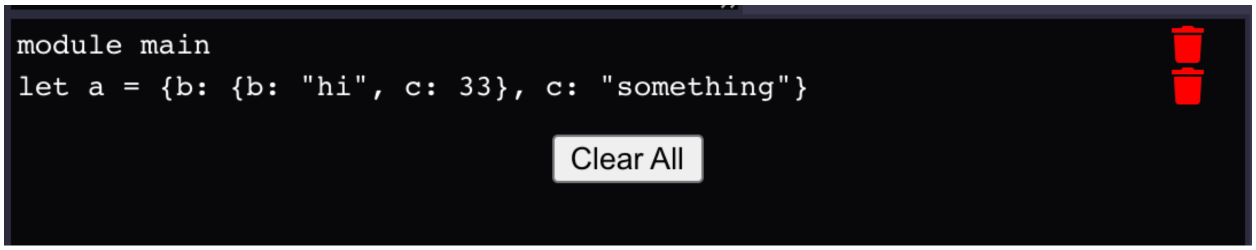
Many of the following UI features were implemented to increase usability and put in place of the CLI REPL commands.

Running Highlighted Code

While writing DEX in the web UI, users are able to highlight one or many statements of DEX and run them as if they were being input into a CLI REPL. This is how our web-based REPL still works as a REPL and not just a DEX editor on the web.

Program State Window

The program state window is a collapsible window that allows users to see the current list of statements stored in the web-based REPL state. From this window users can also either delete individual statements from the web-based REPL state or clear the web-based REPL state entirely. This feature was implemented in place of the “#del” CLI REPL command. Figure 19 shows this window with some example code inside.



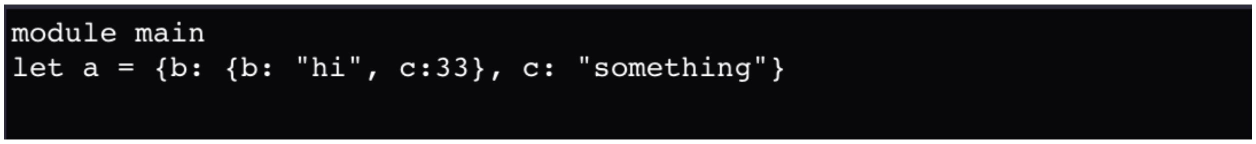
```
module main
let a = {b: {b: "hi", c: 33}, c: "something"}
```

Figure 19:

Web-Based REPL Program State Window

DEX Struct Folding

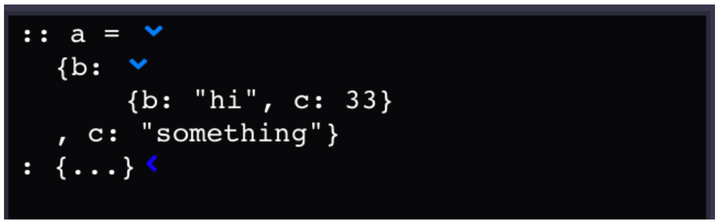
To present DEX structs in a more organized way any “{}”s in the output are automatically folded. Users are able to click on an icon to the right of the folded struct to unfold and refold any values. This was implemented by writing a parser to capture the values between pairs of matching brackets and generating jsx based on the result. An example of this is shown in Figures 20 and 21.



```
module main
let a = {b: {b: "hi", c:33}, c: "something"}
```

Figure 20:

DEX Program that Contains Structs



```
:: a = ⌄
  {b: ⌄
      {b: "hi", c: 33}
  , c: "something"}
: {...} <
```

Figure 21:

Output of DEX Code from Figure 20

File Importing and Exporting

This feature was implemented in place of the “#save”, “#load”, “#loadJSON”, and “#loadFFI” CLI REPL commands. The program can save and load DEX, JSON, and Java files. Saving is accomplished by detecting the current window the user is editing (DEX, JSON, or Java) and then prompting the user to enter the name they would like to save the file as. Once the user enters the name of the file, the program writes the contents of the current window to an output file which is placed into the default download location of the user’s browser.

To load files, the program uses the HTML input tag with the type of “file”. This allows the user to choose any file from their computer to load into the program. Once the file is loaded, the current web-based REPL window is updated with the file and displays the contents of the file.

Web-Based REPL State Saving

State saving and loading is similar to how files are imported and exported. Instead of just saving or loading a single DEX, JSON, or Java file, it takes all the data from those three windows and saves it as a JSON file that represents the state of the web-based REPL as a whole. When loading, the same process is repeated but in reverse. The user selects a JSON file and the program extracts the information from the file and updates the web-based REPL state, overwriting anything that was previously there.

5. Evaluation

At the beginning of this project there was a relatively small but straightforward list of requirements for what should be constructed. These instructions were kept short and simple on purpose to allow for the team to expand upon these initial guidelines if they had enough time. A successful project would not just meet the specified project requirements, but it would go beyond those specifications to provide additional useful capabilities to the DEX development teams.

5-1. Project Requirements

Requirements

This project had a straightforward and clear goal. As outlined previously, while a REPL is not required for development in the DEX language, the implementation of such a tool may increase productivity. However, there are different considerations that must be taken into place when designing a REPL specific to any given language. For this particular project, while the end goal could be provided by just one application, there were important design decisions that needed to be made.

The baseline requirement for this project was to create a command line interface capable of acting as a CLI REPL which would allow the user to submit and run DEX code line-by-line. The CLI REPL would need to remember the state of the DEX code as the user essentially writes the program line-by-line. This is standard for how many other REPLs of functional languages work aside from how our DEX REPL handles redeclarations. While functional languages have to keep declarations permanent to preserve the functional nature of their language, a requirement of this project is to allow and persist declaration changes. Should any binding need to be rebound the DEX REPL needs to handle this change along with any other bindings that might have relied on the original binding.

Additional Features

While the specified requirements for this project were fairly minimal, this left a large amount of room for innovation and additional features that could be implemented. Though not specifically required, they should provide a much better user experience.

To improve the user experience there are many aspects that can be improved. Some improvements focused on actions that can be done more efficiently. The first improvement in this area is with the ability for the user to input multiple lines of code at once. Instead of waiting for a report after each line, concatenating these statements significantly reduces development time. Also, the ability to save and load the state of the DEX REPL can eliminate the need to copy many lines of DEX code just to return to a previous state of the DEX REPL.

Other improvements to user experience include reducing user confusion. By implementing commands that show the user more information about the current state of the program, they can better understand the internal system, allowing the user to more effectively use the software. By, for example, showing the program state the user can review the condition of the current DEX program and continue development right from where the user might have left off.

Finally, another improvement to the user experience is in user error forgiveness. As users make mistakes, it is important to allow the user to correct those mistakes. By adding reset, delete, and undo commands regardless of what the user may mistype, the program state can be corrected.

Optional User Interface

While a basic CLI implementation does meet all of the requirements for a functioning DEX REPL, it is not as nice to use as an application with a Graphical User Interface (GUI). It was suggested that, given an adequate amount of extra time, this project be extended to include the creation of a web-based application with a UI acting as a shell for the underlying CLI REPL. A web-based REPL could utilize buttons and multiple windows to allow for users to access all of the same information without the need for as much typing, while simultaneously providing the user with a clearer representation of the CLI REPL.

5-2. Project Testing

JUnit Testing for the CLI

The command line interface based REPL is the backbone for the entire project and as such it is important that all functionality be tested in such a way as to ensure that each function is robust and bug-free. With regard to handling state and ensuring that redefinitions were correctly handled several suites of JUnit tests were administered. These tests covered basic DEX inputs as well as some invalid inputs in an attempt to uncover any potential bugs in the underlying structure. Testing the CLI was especially important when production switched to a web UI focused effort since the web-based REPL had to just use the underlying CLI framework.

The tests were not limited to DEX inputs. The majority of the tests for the CLI were testing CLI REPL commands and additional features that were not included in the base DEX language. These included testing FFI and JSON loading and the saving and loading of DEX files. Other tests included ensuring that the CLI REPL would handle the many different types of redeclarations correctly. Although these extra tests took more time to complete, a well rounded testing base was essential to confidently move onto the next stage of the project.

Manual Integration Testing for the Web UI

Testing changed once production shifted to the web-based REPL. Rather than having to ensure that the fundamental structure of the REPL would continue to work, as was necessary during development of the CLI REPL, this web-based REPL used the CLI as a starting point and primarily affected the user's interactions. Under the assumption that the CLI REPL works as intended, production of the web UI relied solely on manual integration testing. This was mostly done in an effort to try and save time, allowing for as many new features to be added to the UI as possible. While this testing strategy might be ill-advised in larger development scenarios, on a team of only three people, new changes were easy to keep track of and test.

The first and most important set of manual tests is to ensure that the DEX scratch pad is able to submit and report DEX code just as accurately as the CLI REPL can. The other two supplementary aspects of DEX code submission, the Java and JSON text areas, also need to be tested in a similar fashion. All functionality converted from the original CLI REPL commands need to be checked next, which includes the several different saving and loading features of the web UI. Because browsers handle file loading and saving it is also important to test these features on multiple browsers, specifically those most commonly used. In addition to saving and loading, the program state was made into its own window which is important to test to make sure that its state is always consistent.

Once the functionality of the web-based application was decided upon, testing each feature after any UI change became a generally simple task. Running simple DEX statements that included JSON and Java FFI elements and comparing their outputs against expected results ensured the original functionality from the original CLI REPL. Similarly, testing each save and load button for different file types and across several browsers is important. Then, once all functional elements had been examined, including other simpler buttons, ensuring the alignment and various visual aspects of the UI was the final test.

5-3. Requirement Evaluation

Requirements Completed

By using the DEX CLI as a starting point, the creation of a CLI REPL happened organically. Through proper implementation and rigorous testing, the CLI based REPL was able to function as a REPL in every typical way as well as allowing for redefinitions. By relying on a modified version of the DEX CLI to interact with the DEX compiler, the CLI REPL was able to run in a loop waiting for code input. When an input was received, the CLI REPL used the DEX compiler to check for valid DEX syntax, then compiled the state to check for redeclarations. By manually parsing the new inputs the CLI REPL updated the state of the DEX code a final time before sending that information to the compiler to receive a new output. Every required feature mentioned previously can be found implemented in even the basic CLI REPL. The main structure of a CLI REPL was clear cut, and the addition of redeclarations for every binding type in DEX was an extra step in the evaluation. While the other features mentioned were optional enhancements this core framework lays the groundwork for everything else built on top of it.

Additionally, before actual development began on the CLI REPL it became clear that the DEX -CLI would not be able to handle all aspects of DEX. This included the Java FFI (Foreign Function Interface) and module system which were not properly integrated into the DEX CLI and would require extra work to get running in this CLI REPL. While originally only designed to handle one module at a time, the CLI REPL was configured to accept multiple modules including partial modules while still allowing for redefinitions and imports across modules. The final addition for the CLI REPL to be "feature complete" with the DEX language was the Java FFI which required two CLI REPL commands that would allow the user to load either a Java file or JVM class file.

Importantly, although the required elements outlined in the beginning of this chapter were set in place well before this project began, as development happens, new requirements tend to arise. Additional required features include type declaration and lazy variable binding and reporting.

While originally the CLI REPL would not report any output unless the program specifically stated to report some binding. This proved to be less than satisfactory because it did not give the user enough information on session progression. To remedy this, the CLI REPL regurgitates any bindings with the binded name and type of the value. This allows for the user to be sure that their bindings were read by the CLI REPL and work as intended. Also, since binding is a large part of DEX, this means that after almost every command the user receives some sort of feedback, whether a variable or the type and name of the binding created.

The other requirement that came after development started has to do with improving user experience. Since certain actions are more common than others, having to repeatedly type out common phrases can become tedious. So, a new requirement to add lazy let binding along with reporting was implemented. Instead of needing to type out every let binding, users can type an expression and the CLI REPL infers the user wants to create a new binding and temporarily report the value. This is handled by the initial DEX syntax parser. If any string of DEX is incomplete, the CLI REPL assumes that either a report should occur or a "let" binding should occur. If neither are possible then the CLI REPL reports an error but otherwise the inferred action is able to take place.

Optional Features Completed

Once the initial requirements of this CLI REPL were completed, the project still had time to expand by implementing various ideas brainstormed in the design document. Of these additional features mentioned previously in this chapter: allowing multiple line input from users, saving and loading CLI REPL state, state reporting, as well as undo, delete, and reset commands were all successfully implemented.

While this CLI REPL tended to consider DEX a “black box” and mostly left parsing and compiling to the DEX compiler, allowing multiple line inputs was simple to implement because of the way DEX is structured. Each individual statement was clear to the compiler even without the end of line character, so the CLI REPL was easily able to handle multiple DEX statements in the same single line input.

After the CLI REPL was able to properly report on DEX code the next step was to allow interaction with the CLI REPL itself. CLI REPL commands, which utilized a precursor symbol unused in the DEX language, allowed for the user to not only compile and run DEX code but also interact more with the state. The CLI REPL already had to maintain a state and so commands allowing the user to view the current status of each state was trivial but ultimately useful for user comprehension. With the state of the CLI REPL came the ability for the user to modify it by undoing or deleting specific lines that were no longer of use. Through a combination of commands including: “#undo”:, “#delete <val>”, and “#show” users were able to view their DEX state and delete specific lines, as well as undo their previous statements. Having the undo feature did mean that in addition to the current state of the CLI REPL, a certain number of previous states must also be stored in memory. Other significant additions to the CLI REPL commands included the ability to load and save the state which was accomplished through the use of simple DEX files. These commands provided the CLI REPL with more complexity, however the only way for the project to reasonably expand beyond this point would be to develop a proper web-based REPL.

After completing the CLI REPL, the construction of a backend server utilizing the CLI source began. This Spring Boot backend became the baseline for the web-based REPL which was ultimately built on a React frontend. The web-based REPL succeeds in masking the CLI REPL, providing a much nicer interface for the user while also allowing for all the same CLI REPL commands, either handled with the text editors present or with buttons and menus. Program output and program state are both easily accessible through the UI, as well as integrated DEX, JSON, and Java FFI text areas allow the user to type code in a way that more closely emulates other coding environments. With different windows and states to save, the import and saving functionality of the web-based REPL is much improved from the CLI REPL. Not only can the entire web-based REPL state be saved and loaded, but so can each individual text area as well as the DEX program state. Finally, the web-based REPL includes a short help manual which explains each button and function to the user.

5-4. Project Evaluation

Given the original required features, if a CLI REPL was all that were to be completed, the user would simply have a command line interface. This would only allow them to evaluate DEX expressions line-by-line while the state of the CLI REPL is preserved but is not accessible to the user. The application of DEX would be “feature complete”, however, usability would be at the bare minimum. There would be no room for error on the part of the user unless they were able to redeclare their mistake, assuming they remembered how this mistake happened in the first place. With the original CLI REPL, using the application was like working with a black box. The addition of commands to view and edit the CLI REPL state add to the ability of the user to comprehend and check up on their instance of the program.

The next large step in usability comes with the implementation of the web-based UI. By moving away from the CLI REPL, not only can every aspect of the program state become clearer, but the user experience is improved from input to ease of use by utilizing separate windows and specific buttons for otherwise complex commands. While the UI in the web-based REPL is much more complex than the simple CLI REPL, once the user learns how the web-based REPL works, aspects have been designed to make the process go faster.

6. Conclusion

The DEX configuration language is a tool that F5 uses to compose security policies. At the beginning of the project, DEX development happened in a fairly closed off, niche system which offered little experimentation and few live development tools. While there existed three development tools in the IntelliJ plugin, the DEX Tour, and the DEX CLI, none of those tools had the depth and live development capabilities that F5 desired.

To address this need,we designed and implemented a Read Evaluate Print Loop for DEX. This REPL needed to support variable redeclarations and include features of DEX not supported in the other development tools. To accomplish this goal, we first identified the DEX CLI as a good base for our DEX REPL. The resulting CLI based REPL we built allowed users to input consecutive lines of DEX code, constructing a DEX file as they evaluate individual lines of code. Our CLI REPL added DEX features not supported in the existing development tools. The most significant of these was the loading of Java Foreign Function Interface data which up to this point did not have an easy way to be tested. We also added many quality of life CLI REPL commands to help users, such as REPL commands to view the current DEX program state, undo previously input commands, and save the current state of the REPL.

After implementing the CLI REPL, we ported over the DEX REPL into a web-based UI which was a lot more user friendly than the CLI implementation. The core CLI REPL functionality was maintained and we added additional features such as saving the entire web-based REPL state, including JSON data and Java FFI data. This Web UI met and exceeded the initial expectations for the project.

7. Future Work

This project has many avenues to explore in the future which expand on the capabilities of the DEX REPL. Some of these future works could be accomplished within a relatively short time frame, while others would expand the project past its initial scope.

As with almost all user interfaces, there are always more exact and general improvements to be made to the look of the program and the user experience. As the project matures and begins to see active use, adjustments to the UI could be made to make the user experience more streamlined. The first is the implementation of a multi-windowed setup that would allow for the user to create and interact with multiple concurrent web-based REPL states. Currently one could still open the program in two separate browser tabs at once to experience the same effect but given enough time this would be a useful feature to have consolidated into the one application instance. Another UI improvement would be adding the option for more user customization. This would include allowing the user to change things such as font type, sizes, and colors. This could improve the user experience, as most developers have their own preferences for how a development environment looks. Such UI improvements would be relatively straightforward to implement, as most UI modifications would not actually involve changing the mechanics of how the program actually works.

A database could be added to the web-based REPL to give users the option to save and load files from the cloud. This would be useful because then users would be able to access their data over multiple machines without having to send save files between them.

The web-based REPL could be linked up to the F5 authentication system. This would allow persistent user profiles where the user could save their data. This user data could include things like personal UI configurations, and individual save and load files. Adding an F5 login could be done by using something like OAuth to link up to F5's outlook login.

Adding debugging capabilities to the DEX REPL could also be useful. Currently if there are any errors, the DEX REPL just prints out the error that the compiler has returned. This can be frustrating for a user since these error messages are usually vague, sometimes just saying that there is a syntax error somewhere in the program. Adding debugging capabilities would speed up the user's workflow as they would be spending less time debugging their code and more time actually coding. Some basic features such as breakpoints and stepping through code could be easily implemented, although other debugging features could be more complicated and would require a much deeper understanding of the DEX language than was required to create this DEX REPL.

Another long term project could be expanding the DEX REPL to allow for the construction of DEX code through UI elements. This idea has already been partially implemented by the Policy Composer through the Composer-API and is how most DEX programming is actually done. This would allow programmers less experienced in DEX to also use this REPL. Implementing this idea would involve a deeper investigation of the Policy Composer to see its inner workings and build upon its ideas.

References

About F5 | careers - investor - contact - leadership. (n.d.). Retrieved October 12, 2021, from https://www.f5.com/company.

Baeldung. (2021, March 24). *A comparison between Spring and Spring Boot*. Baeldung. Retrieved December 1, 2021, from https://www.baeldung.com/spring-vs-spring-boot.

Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Casssou, D., & Denker, M. (2007). *Squeak by example*. Square Bracket Associates

Bracha, G. (2021, October 10-18). *A language for configuring security policies*[Conference Presentation]. *SPLASH 2021, Chicago, IL, United States. https://2021.splashcon.org/details/conflang-2021-papers/1/A-Language-for-Configuring-Security-Policies*

Bracha, G. (n.d.). *How is a programmer like a pathologist?*. Retrieved October 11, 2021, from https://blog.bracha.org/exemplarDemo/exemplar2021.html?snapshot=BankAccountExemplarDemo.vfuel

Enterprise security solutions overview: Shape Security. (2020). Retrieved October 12, 2021, from https://www.shapesecurity.com/solutions.

Facebook inc. (n.d.). *Introducing hooks*. React. Retrieved December 1, 2021, from https://reactjs.org/docs/hooks-intro.html.

Facebook inc. (n.d.). *React – A JavaScript library for building user interfaces.* Retrieved October 21, 2021, from https://reactjs.org/.

Facebook inc. (n.d.). *React.component*. React. Retrieved December 1, 2021, from https://reactjs.org/docs/react-component.html.

Gruening, R., DuLong, S., & Lancaster, H. (2020, January 24). *F5 completes acquisition of Shape Security.* F5 press release. Retrieved October 12, 2021, from https://www.f5.com/company/news/press-releases/f5-completes-acquisition-of-shape-security.

JSConf. (2015, August 5). *Tom Occhino and Jordan Walke: JS Apps at...* YouTube. Retrieved December 7, 2021, from https://www.youtube.com/watch?v=GW0rj4sNH2w.

Microsoft. (n.d.). *JavaScript with syntax for types.* TypeScript. Retrieved December 1, 2021, from https://www.typescriptlang.org/.

REPL. CLiki. (n.d.). Retrieved October 11, 2021, from https://www.cliki.net/REPL.

Squeak.org. (n.d.). *Welcome to Squeak/Smalltalk*. Retrieved October 12, 2021, from https://squeak.org/

Thomas Van Binsbergen, L., Merino, M. V., Jeanjean, P., Van Der Storm, T., Combemale, B., & Barais, O. (n.d.). *A principled approach to REPL interpreters*. 1–17. https://doi.org/10.1145/3426428.3426917

VMware. (n.d.). *Spring Boot*. Spring. Retrieved December 1, 2021, from https://spring.io/projects/spring-boot.

VMware. (n.d.). *Why Spring?* Spring. Retrieved December 1, 2021, from https://spring.io/why-spring.