Viz: A Beginner's Language
Final Report

Matthew Duran, Yanhao Li, Jinsen Wu, Jakob Deutsch, Nicholas Wu
{md3420, yl4734, jw4157, jgd2150, nkw2115}@columbia.edu

Table of Contents

1. Logistics

We have three members the that are graduating: Matt Duran, Yanhao Li, Jakob Deutsch. The remaining two members Jinsen Wu, and Nicholas Wu are not graduating this semester. Also, please see below for a link to our demo video:

```
https://youtu.be/BXfhnpdOyrg
```

2. Introduction

Viz is a general-purpose programming language that allows for the visualization of program execution. At first, we focused on visualizing data structures, but we have since pivoted to showing program execution in a step-by-step manner. We have implemented and deployed an interface, which will allow you to run our viz compiler and see the program execution of our source code at every possible step in the compilation process. Further, we mastered the dune build system which allows us to output these specific data structures to the console: Token List, Abstract Syntax Tree, Syntactically-Checked Abstract Syntax Tree, IR Code, and of course the Program Output.

Viz source code is imperative, statically scoped, statically, and strongly typed like Java but with simpler features and more intuitive syntax. Our programming language supports the most basic primitive data types, operations, and control flows. On top of the basics, we also include features such as garbage collection, struct (i.e. object) definition, and the list abstract data type. We have combined the best aspects of C, Python, and Java in order to make programming as simple as possible for beginners.

With our AWS deployed interface, users will have the ability to visualize the data structures that have been mentioned above. Our web application will contain a Docker container image bundled with our compiler, LLVM, and OCaml which will afford users the ability to easily execute our code and begin developing. Viz's beginner-friendly syntax will allow users to become comfortable with coding, and the interface will be a helpful tool for implicitly learning about how languages work under the hood from scanning through code generation. We put the skills we learned this semester into practice and produced a useful teaching tool for students that wish to learn about the compilation process and language design.

As a group, we put in a huge effort for this project, with over 400 commits. We hope you enjoy playing around with Viz!

3. Environment Setup

3.1 VizOnline Text Editor

You can access our online text editor compiler through the link below:

```
http://ec2-23-22-206-12.compute-1.amazonaws.com/
```

This is the recommended way to access our project since there is no environment setup required. However, if one prefers to run our project locally then please check out 2.2 Environment Setup.

3.2 Docker Viz Container

To run the project locally, you need to have OCaml (4.13.1), dune (>=3.0), and LLVM (11.0.0) or Docker. Docker is highly recommended since our container image has LLVM installed. Please visit https://github.com/4115-viz/viz to clone the compiler source code and check out more detailed instructions. See below for useful commands to work with our source code. Note that all commands should be done at the **root of the directory**.

To build the compiler:

```
dune clean && dune build
```

To build Viz docker image (contains LLVM bundled inside) and launch container. Note, building the container may take a while:

```
docker build -t viz .
./launch-container.sh
```

We provide a full set of sample programs in the test/programs directory. Feel free to use those to model your own code after. To run a test program in the **running** Viz container:

```
./viz test/programs/<test_file_name>.viz
```

To run all test programs:

```
./script-test-programs
```

To check the visualization of program execution steps:

```
dune exec -- vc test/programs/<test_file_name>.viz -ts // scan and print tokens
dune exec -- vc test/programs/<test_file_name>.viz -a  // generate abstract tree (AST)
dune exec -- vc test/programs/<test_file_name>.viz -s  // generate semantically checked AST
dune exec -- vc test/programs/<test_file_name>.viz -l  // generate and print llvm
```

4. Language Tutorial

4.1 Writing Programs

Writing programs in Viz is much like writing them in C. However, we have taken familiarity with other languages into consideration and modeled Viz toward them to reduce the learning curve. See below for some simple syntax to get you started:

4.1.1 Main Entry Point

func main() is always required in Viz since it is the main entry of Viz

```
func main(): none { statement… }
```

4.1.2 Varibale Declaration and Initialization

We support primitive types such as string, int, float and bool, and the list complex data list. See below for learning how to work with variables:

```
@x: bool; // declare
@x = true; // then initialize
@y: string = "good"; // declare and initialize
@y = "bad"; // reassign
@a, @b, @c -> (int, 0); // initializer list where all vars are ints with value 0
@l: list |int| = [1,2,3,4]; // declare and intialize a list
@l[0] = 1; // reassign a list element
```

4.1.3 User Defined Structs

Structs serve as our support to the Object Oriented Programming paradigm. It allows user to define their own complex data type:

```
struct Person {
    @name: string;
    @age: int;
}

// instantiate a Person object
@my_person: Person;
@my_person.age = 3;
@my_person.name = “Matt”;
```

4.1.4 Function Declaration and Utilization

Declaration of function is similar to that of variable declaration. It starts with keyword func, followed by function name, return type, then block of statements:

```
func func_name() : type { statement… }
```

Users can call the functions that have already been defined prior. Also, we have built-in functions such as print():

```
func hello_world() : none {
  print("Hello World");
}

func main() : none {
   hello_world();
}
```

4.1.5 Control Flow and Iteration in Viz

If, Elif, Else Statements:

```
if (conditional expression) {
       Block...
} elif (conditional expression) {
       Block...
} ... {
       ...
} else {
       Block...
}
```

For Loop:

```
for k in start...<=end step n { // step defaults to 1 if “step n” not present
     Block...
}
```

While Loop:

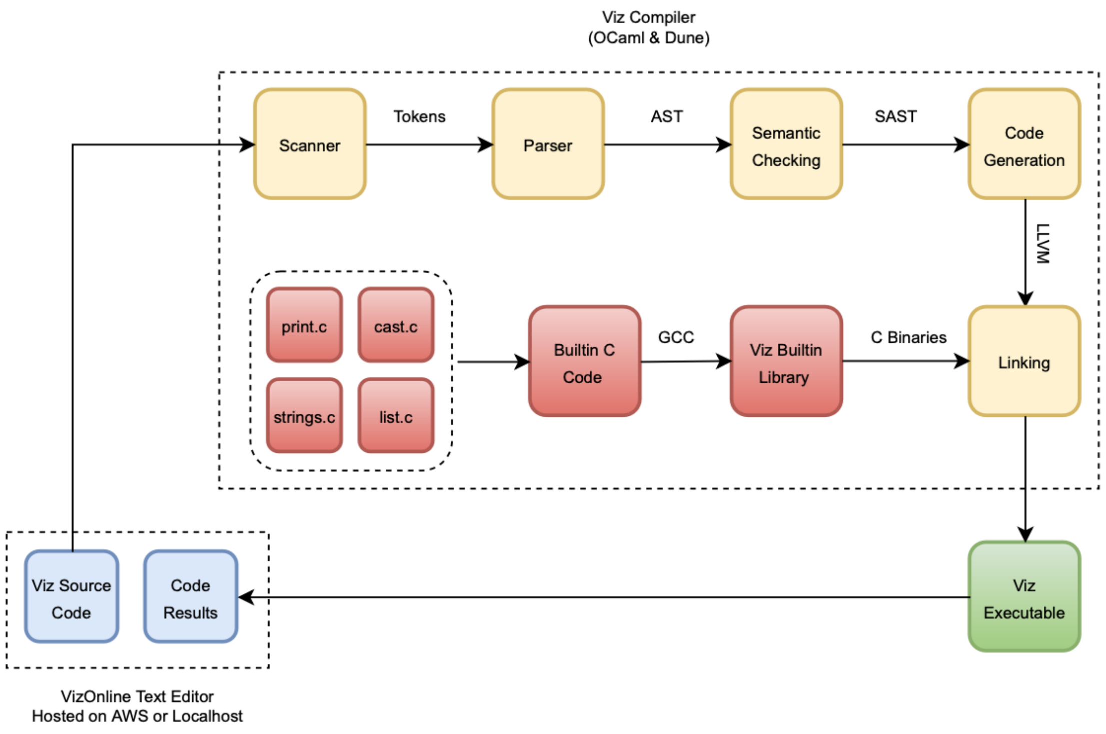
```
while boolean_condition {
       Block...
}
```

The tutorial above only covers the basic fundamentals of Viz’s many features. If you are interested in a comprehensive and detailed documentation of Viz, please see the Language Reference Manual we submitted along with this report. You can also check out the LRM at this link:

https://github.com/4115-viz/viz/blob/master/writeups/LRM/Viz_LRM.pdf

5. Architectural Design

5.1 Block Diagram of Major Components



5.2 Interfaces between Components

Viz Source Code:

Source code are programs stored in files with .viz postfix. This source code is the interface between the users and the compiler.

Scanner:

The scanner is the component defined in the scanner.mll file. It scans tokens that are allowed in Viz and perform lexical checks. When the source code is scanned, the scanner will handle the conversion of the source code into internal tokens and pass them to the parser.

Parser:

This component is defined in the parser.mly file. The parser processes the stream of tokens passed in from the scanner. Here we use our context free grammar to make ensure that the source code is syntactically correct with precedence and associativity. We will output an AST (Abstract Syntax Tree) for the stream of tokens.

Semantic Checking:

Semantic Checking is done in the semant.ml file. It takes the AST generated by the parser and ensures the content is semantically correct by performing type-checking and raising various semantic errors where applicable. It also adds scope to our AST, which is our big addition to “MicroC”. We produce an advanced symbol table, that allows us to define nested scopes and access variables at varying levels of the table. At the end of semantic checking, we generate a SAST (Semantically Checked AST) and pass it to the code generator.

Code Generation:

Code generation is done in the irgen.ml file. This file converts descriptive types such as int, bool, and float into native LLVM primitives. We also use LLVM here to complete different operations like arithmetic operations and conditional flow. In the end, this file will build LLVM basic blocks from the input and generate an intermediate representation (IR) of the source code that can then be runnable on specific machines.

Linking:

> Linking is the interface between IR code and standard libraries that we built with C. It produces an output binary file and links the generated code and libraries containing built-in function definitions. It allows us to vastly expand the potential utilities of Viz. We wrote libraries to expand the versatility of our string type, typecasting, printing, and lists.

5.3 Group Member Contributions

All the members have contributed to each component based on the individual features, updates, and bug fixes.

6. Test Plan

6.1 Source code and Target Language Programs

We have plenty of interesting programs in the test/programs section of our repo that demonstrate the power of our language. Below we will provide two very simple code snippets for brevity. Note, that due to varying machines, some of the list tests may fail. We came to the conclusion that we wouldn't be able to support all machines, so some of the tests may fail. But, the functionality is pretty close to where it needs to be! See below for some code examples.

6.1.1 Hello-World Souce Code

```
func viz_greeting(): none {
    print("hello back from Viz Dev Team!!");
}

func main(): none {
    print("hello world: ");
    viz_greeting();
}
```

Target Language Program

```
; ModuleID = 'Viz'
source_filename = "Viz"
```

```
@str = private unnamed_addr constant [31 x i8] c"hello back from Viz Dev Team!!\00", align 1
@fmt = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.1 = private unnamed_addr constant [14 x i8] c"hello world: \00", align 1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1

declare i32 @printf(i8*, ...)

declare double @int_to_double(i32, ...)

declare double @str_to_double(i8*, ...)

declare i32 @double_to_int(double, ...)

declare i32 @bool_to_int(i1, ...)

declare i32 @str_to_int(i8*, ...)

declare i8* @bool_to_str(i1, ...)

declare i8* @int_to_str(i32, ...)

declare i8* @double_to_str(double, ...)

declare i8* @string_concat(i8*, i8*, ...)

declare i32 @str_len(i8*, ...)

declare i8* @to_upper(i8*, ...)

declare i8* @to_lower(i8*, ...)

declare i32 @string_equals(i8*, i8*, ...)

declare i32 @string_not_equals(i8*, i8*, ...)

declare i32 @string_lt(i8*, i8*, ...)

declare i32 @string_gt(i8*, i8*, ...)

declare i32 @string_lte(i8*, i8*, ...)

declare i32 @string_gte(i8*, i8*, ...)

define void @viz_greeting() {
entry:
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt, i32 0, i32 0), i8* getelementptr inbounds ([31 x i8], [31 x i8]* @str, i32 0, i32 0))
  ret void
}

define void @main() {
```

```
entry:
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt.2, i32 0, i32 0), i8* getelementptr inbounds ([14 x i8], [14 x i8]* @str.1, i32 0, i32
0))
  call void @viz_greeting()
  ret void
}
```

6.1.2 For-loop Soure Code

```
func main(): none {
    @x: int = 0;

    print("testing <= condition with default step 1");
    for @x in 1 ... <= 5 {
        print("========================");
        print("for loop <=, default step 1");
        print("iteration");
        print(@x);
        print("========================");
    }
    println();

}
```

Target Language Program

```
; ModuleID = 'Viz'
source_filename = "Viz"

@str = private unnamed_addr constant [41 x i8] c"testing <= condition with default step
1\00", align 1
@fmt = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.1 = private unnamed_addr constant [25 x i8] c"========================\00", align 1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.3 = private unnamed_addr constant [28 x i8] c"for loop <=, default step 1\00", align 1
@fmt.4 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.5 = private unnamed_addr constant [10 x i8] c"iteration\00", align 1
@fmt.6 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@fmt.7 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@str.8 = private unnamed_addr constant [25 x i8] c"========================\00", align 1
@fmt.9 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@str.10 = private unnamed_addr constant [1 x i8] zeroinitializer, align 1
@fmt.11 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1

declare i32 @printf(i8*, ...)

declare double @int_to_double(i32, ...)
declare double @str_to_double(i8*, ...)

declare i32 @double_to_int(double, ...)

declare i32 @bool_to_int(i1, ...)

declare i32 @str_to_int(i8*, ...)

declare i8* @bool_to_str(i1, ...)

declare i8* @int_to_str(i32, ...)

declare i8* @double_to_str(double, ...)

declare i8* @string_concat(i8*, i8*, ...)

declare i32 @str_len(i8*, ...)

declare i8* @to_upper(i8*, ...)

declare i8* @to_lower(i8*, ...)

declare i32 @string_equals(i8*, i8*, ...)

declare i32 @string_not_equals(i8*, i8*, ...)

declare i32 @string_lt(i8*, i8*, ...)

declare i32 @string_gt(i8*, i8*, ...)

declare i32 @string_lte(i8*, i8*, ...)

declare i32 @string_gte(i8*, i8*, ...)

define void @main() {
entry:
  %var_x = alloca i32, align 4
  store i32 0, i32* %var_x, align 4
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt, i32 0, i32 0), i8* getelementptr inbounds ([41 x i8], [41 x i8]* @str, i32 0, i32 0))
  store i32 1, i32* %var_x, align 4
  br label %while

while:                                            ; preds = %while_body, %entry
  %x = load i32, i32* %var_x, align 4
  %tmp = icmp sle i32 %x, 5
  br i1 %tmp, label %while_body, label %while_end

while_body:                                       ; preds = %while
  %printf1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt.2, i32 0, i32 0), i8* getelementptr inbounds ([25 x i8], [25 x i8]* @str.1, i32 0, i32
0))
```

```
  %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt.4, i32 0, i32 0), i8* getelementptr inbounds ([28 x i8], [28 x i8]* @str.3, i32 0, i32
0))
  %printf3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt.6, i32 0, i32 0), i8* getelementptr inbounds ([10 x i8], [10 x i8]* @str.5, i32 0, i32
0))
  %x4 = load i32, i32* %var_x, align 4
  %printf5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt.7, i32 0, i32 0), i32 %x4)
  %printf6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt.9, i32 0, i32 0), i8* getelementptr inbounds ([25 x i8], [25 x i8]* @str.8, i32 0, i32
0))
  %x7 = load i32, i32* %var_x, align 4
  %tmp8 = add i32 %x7, 1
  store i32 %tmp8, i32* %var_x, align 4
  br label %while

while_end:                                        ; preds = %while
  %printf9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]*
@fmt.11, i32 0, i32 0), i8* getelementptr inbounds ([1 x i8], [1 x i8]* @str.10, i32 0, i32
0))
  ret void
}
```

6.2 Test Automation

We used a script file run_viz_tests.sh to automate our test suite and create the easy-to-read output for the test results. Before we were able to perform code generation, and build executables we relied heavily on scanner, parser, and semantic print outputs. However, once we were able to go from end-to-end, we moved to running actual programs. We have 60 programs, most of which are comprehensive and provide actual use cases for users to learn the language. To run the test script file, follow the instructions below in root directory:

```
./launch-container.sh
./run_viz_test.sh
```

Output of the tests:

```
-------------------------------------
running master shell script from /Users/matt20d/Desktop/github_repos/cs4115/viz
-------------------------------------

Running Scanner Tests in: /Users/matt20d/Desktop/github_repos/cs4115/viz/test/scanner
... // actual tests being run
(11 / 11) Scanner Tests Passed
```

```
removing all the intermediate files in
/Users/matt20d/Desktop/github_repos/cs4115/viz/test/scanner

Running Parser Tests in: /Users/matt20d/Desktop/github_repos/cs4115/viz/test/parser
... // actual tests being run
(8 / 8) Parser Tests passed
removing all the intermediate files in
/Users/matt20d/Desktop/github_repos/cs4115/viz/test/parser

Running Semantic Tests in: /Users/matt20d/Desktop/github_repos/cs4115/viz/test/semantic
... // actual tests being run
(41 / 41) Semantic Tests Passed
removing all the intermediate files in
/Users/matt20d/Desktop/github_repos/cs4115/viz/test/semantic

Running Viz Test Programs in: /Users/matt20d/Desktop/github_repos/cs4115/viz/test/programs
... // actual tests being run
(65 / 65) Viz Executable Tests Passed
removing all the intermediate files in /Users/matt20d/Desktop/github_repos/cs4115/viz
```

7. Summary

7.1 Group Member Contributions

```
Unit Tests: Jinsen, Yanhao, Matt, Nicholas
Test Infrastructure: Matt, Yanhao
Basic Literals: Jinsen, Yanhao, Matt, Jakob
Variable Declaration, Initialization, and Assignment: Jinsen, Yanhao, Matt, Jakob
List: Yanhao, Nicholas, Matt
Struct: Yanhao
Print function: Jinsen, Yanhao, Matt, Jakob
Arithmetic Operations: Jinsen, Matt, Yanhao
Typecasting: Jinsen, Matt
Conditional Operations: Nicholas, Matt, Yanhao
Function Declaration: Jinsen, Yanhao, Matt, Jakob
Return: Jinsen, Yanhao, Matt
Control Flows: Matt, Nicholas
Isolated Block Operation and Symbol Table: Jinsen
Standard Library: Matt, Yanhao, Nicholas
Docker Setup: Yanhao, Matt
Online Compiler: Matt, Jakob
LRM: Jinsen, Yanhao, Matt, Jakob, Nicholas
Final Report: Jinsen, Yanhao, Matt, Jakob, Nicholas
```

7.2 Takeaways

Jinsen: It has been a very challenging experience to build a language and compiler with Ocaml and LLVM. However, after learning how to create a programming language from scratch, I appreciate the languages I am using a lot more. This course has enriched my experience as a programmer. If not for this course, I would probably never seek out to learn how a language works. It takes a lot of time to build up enough knowledge to complete the final project so it is best to start early. At the end of the day, “it is hard to get it to compile, but once it compiles, it works”

Matt: I echo what Jinsen wrote wholeheartedly. I have been coding heavily for the past 4 years, and have wondered why some languages are slower than others and why certain languages are tailored to specific use cases. For instance, C is heavily used in systems programming, because it is compiled and fast to run high-performance systems. This course taught me what happens under the hood, and I am especially grateful for the rich functionality that comes with the major languages like Python, Java, and C/C++. Further, I will no longer complain about the nasty compiler error messages that I get in C++ because compiler design is complex, especially for a small novelty language like Viz. This project is the first one at Columbia that I realized we couldn’t “cram” for, it is best to start early and conquer byte-size chunks at a time.

Yanhao: The project gave me the chance to apply the knowledge I learned from the class in practice. The key takeaway is to start small, have a working MVP, and iteratively add new features to it. Time management and sticking to the timeline are also essential, we used the Github Project to track the progress, manage the issues and assign tickets. I’m very grateful to my Viz team members, they have been dedicated and diligent enough to work on the project throughout the whole semester. We conducted many code reviews and learned from each other along the way. I really appreciate the time and effort that everyone took.

Nicholas: To reiterate some of the sentiments of my teammates, going through the whole process of designing the language and building the compiler demystified a good deal of what was going on under the hood. When I look at programming languages now, I find myself thinking more about scope and how everything is assigned in memory. It was a challenging and difficult task overall, and halfway through we had a moment where we had to completely scrap our entire grammar and all the work we had accomplished because our grammar was too limited. It was difficult coming to terms with the fact that we had to start over again, all of us, to some degree, felt defeated by it. I owe a great deal of making it through this project to the hard work and effort of my teammates. And although it was a difficult task, it was single-handedly the most rewarding experience I had while working on a project at Columbia. One which taught me the value of setting up a well-defined architecture from the beginning, otherwise you will be beating yourself up about it later!

Jakob: Working on this project has definitely enhanced my appreciation for all the programming languages that I have come accustomed to using. I now understand, on a much higher level, the different components that go into building a programming language. This past summer, I became interested in learning how to build my own programming language. I watched a number of videos on YouTube and tried following along with them, but by doing so I didn’t really learn what was actually happening behind the scenes. Both this class and the final project have helped reveal all the information I had been missing and I am grateful for that. It was an amazing experience being able to implement a programming language from scratch. While it is a difficult task overall, it is a very rewarding experience.

7.3 Advice for Future Teams

Take this class and push through it. It may not be easy, but once you finish you will appreciate modern programming languages and compilers better. Don’t get stressed out, this course is tough for everyone. Start early, and do a little bit every day!