Viz: A Beginner's Language
Language Reference Manual

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

Table of Contents

1. 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!

2. Lexical analysis

We support the following token classes: keywords, literals, operators, delimiters, and identifiers. When scanning and generating tokens, our compiler will ignore all whitespace. However, it is good practice to use whitespace well when writing code for readability.

```
let whitespace = [' ' '\t' '\r' '\n']
```

2.1 Keywords

The following keywords are reserved and may not be used as identifiers.

```
func
if
else
elif
for
while
infinite_loop
return
true
false
... // this gets scanned as RANGE token for our for loop construct
step
as
struct
and
or
not
none
in
string
bool
float
list
int
```

2.2 Comments

Comments are treated as whitespace, thus they will be ignored. They cannot nest, and cannot appear within literals. Block comments begin with /* and end with */ and inline comments begin with // and will continue until a '\n' is reached.

```
{
     exception Viz_scan_error of string
}

rule token = parse
(* -------- whitespaces -------- *)
| "/*" {multi_comment lexbuf}
| "//" {single_comment lexbuf}
```

```
and single_comment = parse
| '\n' {token lexbuf} (* this is how we will end a single line comment *)
| _    {single_comment lexbuf} (* want to ignore the rest of the noise *)

and multi_comment = parse
| "*/" {token lexbuf} (* end of multi line comment, head back to token *)
| "/*" {raise (Viz_scan_error ("cannot nest multi-line comments"))} (* no nested comments *)
| _    {multi_comment lexbuf} (* want to ignore the rest of the noise *)
```

2.3 Literals

A literal is the source code representation of a value of a type, such as a number or string. Their validity will depend on the scanned data type it corresponds to.

Integer literals are a sequence of digits, which correspond to a whole number. There can be no decimal points present. Our integer literals follow OCaml standard and are capped at 32 bits. We only store positive numbers, because we will parse the MINUS token and use that to make the number negative in the Context Free Grammar.

```
let digit  = ['0'-'9']
let non_zero_digits = ['1'-'9']

(* we store everything as a positive magnitude number *)
| ('0'+) | non_zero_digits digit*  as lxm {LIT_INT(int_of_string lxm)}
```

Float Literals include an integer part, decimal point, and a fraction part. The integer part could be missing (example: .575) or could be present (example: 0.575), but the fractional part must always be there (non-example: 50., which is legal in OCaml). Our Float literals follow OCaml standard and are capped at 64 bits.

```
(* we store everything as a positive magnitude number *)
| ("0." "0"+) | digit* "." digit+ as lxm { LIT_FLOAT(float_of_string lxm) }
```

Boolean literals are one of the two valid sequences below, either true or false.

```
| "true" { LIT_BOOL(true) }
| "false" { LIT_BOOL(false) }
```

String Literals are sequences of characters that are surrounded by double quotes. We support the whole ascii set, and will scan any characters that live between and open and closing quote.

```
| '"' ([^ '"']* as lxm) '"' { LIT_STR(lxm) }
```

2.4 Operators

These operators below are lexical tokens that are reserved by the language. Their varied uses cases will be defined in the expressions section.

```
(* -------- arithmetic operators -------- *)
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '%' { MOD }

(* -------- assignment operators -------- *)
| '=' { ASSIGN }
| "+=" {PLUSEQ}
| "-=" {MINUSEQ}
| "*=" {TIMESEQ}
| "/=" {DIVEQ}
| "%=" {MODEQ}

(* -------- relational operators -------- *)
| "==" {EQ}
| "!=" {NEQ}
| ">=" {GTEQ}
| "<=" {LTEQ}
| ">" {GT}
| "<" {LT}
| "and" {AND}
| "or" {OR}
| "not" {NOT}
| "?" {QUESTION}
```

2.5 Delimiters

The following tokens serve as delimiters, and are reserved by the language for use.

```
(* -------- delimiters -------- *)
| "("  { LPAREN   }
| ")"  { RPAREN   }
| "["  { LBRACKET }
| "]"  { RBRACKET }
| "{"  { LBRACE   }
| "}"  { RBRACE   }
| ":"  { COLON    }
| ";"  { SEMI     }
| ","  { COMMA    }
| "."  { DOT      }
| "|"  { BAR      }
| "->" { ARROW    }
```

2.6 Identifiers

Our identifiers is how you are able to name and reference functions, structs, and variables. Identifier names are not allowed to be any of our reserved keywords. This rule is enforced by having the keywords get scanned at a higher precedence than our identifiers. We have three types of identifiers: struct and member definition (example: Shape), struct field member and function definition/reference (example: shape or func main()), variable declaration/reference (example: @my_int).

```
let uppercase = ['A'-'Z']
let lowercase = ['a'-'z']
let letter    = ['a'-'z' 'A'-'Z']

(* --------- IDs ------------ *)
(* capitalized UNCAP_ID, example: Shape, Person, Edge *)
| uppercase lowercase* as lxm { CAP_ID(lxm) }

(* uncapitalized UNCAP_ID, example: shape, person *)
| lowercase (digit | letter | '_')* as lxm { UNCAP_ID(lxm) }

(* variable access and decl need @ *)
| "@" letter (digit | letter | '_')* as lxm { ID_VAR(String.sub lxm 1 ((String.length lxm) -
1)) }
```

3. Type System

3.1 Primitive types

int

The integer type will store a signed whole number with 4 bytes.

```
@my_int = 5;
@my_neg_int = -5;
```

float

The float type stores signed fractional numbers with 4 bytes.

```
@my_float = 5.0;
@my_neg_float = -5.0;
```

bool

The bool type will represent a true or false value, and take up 1 byte of space.

```
@my_bool = true;
@my_bool2 = false;
```

none

The none type represents an empty return value, and can only be used by functions for return types.

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

string

An list of ASCII characters, wrapped with double or single quotation marks.

```
@my_string = "hello world!";
```

3.2 Complex Data Types

3.2.1 Builtin Abstract Data Types (ADTs)

For Viz builtins, we only support the list ADT, which is a collection of elements that may only contain the same type within the data structure. The supported data types include ints, floats, and strings.The list ADT is built in and generic, and we will use the following syntax to instantiate a list. A list can be instantiated like below:

```
@my_numbers: list|int| = [1, 3, 5, 7, 11, 13];
```

Lists can also be reassigned:

```
@my_numbers: list|int| = [1, 3, 5, 7, 11, 13];
@my_numbers = [1, 3, 5, 7];
```

We can then access list elements like by using the bracket operator, along with an index number. Our lists are 0-based, meaning that the first element is 0 and the last element is length of list - 1.

```
@my_numbers: list|int| = [1, 3, 5, 7, 11, 13];
print(@my_numbers[0]); // outputs 1
print(@my_numbers[5]); // outputs 13
@idx: int = 3;
print(@my_numbers[@idx]); // outputs 7
```

Unfortunately, we didnt have time to make this data structure more robust, as it proved to be more complicated that anticipated in the LLVM portion. However, in the list library we will provide some additional functionality limited to the few library functions that we implemented leveraging C.

3.2.2 User Defined Structs

Structs serve as our support for the Object Oriented Programming paradigm. Structs allow a user to define their own complex data type, which can be composed of a combination of underlying data types. These structs are essentially a wrapper, which allow the user to combine related variables within one container. Structs must be defined, before they can be used. Struct names must be capitalized, because they will become “types” that can be instantiated later on in the programs. See below for an example of how we define and instantiate a struct:

```
// define a struct before use
struct Person {
    @name: string;
    @age: int;
    @major: string;
    @school: string;
    @graduated: bool;
}

// instantiate a Person object
@my_person: Person;
```

Much like in C, if you try to access one of the field members before initializing it you will be accessing garbage values. The same applies to Viz structs, each of the variables are accessible, but will remain null until they are initialized. There are two ways of initializing Viz structs: directly maniputlaitng the members, or by having an “init” function that will execute that in a DRY manner (***preferred way***). See below for examples:

```
// manipulate a struct directly
@matt: Person;
@matt.name = "matt";
@matt.age = 29;
@matt.major = "Computer Science";
@matt.school = "Columbia University";
@matt.graduated = true;

// initializer function, similar to C
func init_person(@name: string, @age: int, @major: string, @school: string, @graduated:
bool): Person {

    @my_person: Person;
    @my_person.name = @name;
    @my_person.age = @age;
    @my_person.major = @major;
    @my_person.school = @school;
    @my_person.graduated = @graduated;

    return @my_person;

}

// declare and init a Person, using the “init” function.
@yanhao: Person = init_person("yanhao", 25, "Software Systems", "Columbia SEAS", true);
```

Lastly, structs can be nested, except for those of the same type. See below for an example and non-example:

```
// this is legal, provided Person is defined prior to struct Friends
struct Friends {
    @friend1: Person;
    @friend2: Person;
}
```

```
@my_group: Friends;
@my_group.friend1   = @yanhao;
@my_group.friend2   = @matt;

// this is illegal, because the struct is Recursively defined.
struct Person {
    @name: string;
    @age: int;
    @major: string;
    @school: string;
    @graduated: bool;
    @mom: Person;
}
```

3.3 Data Type Conversion Through Casting

Modern programming languages afford the ability to cast data types to and from one another. There are convenient cases where changing data types makes logical sense. Our compiler will detect that the user would like to convert data types. If the conversion is allowed, we will then execute the cast using underlying C library functions to assist us, which we linked in at compile time.

Allowed casting conversions:

```
/* int casts */
float -> int
string -> int // "99", value becomes 99
bool -> int // true = 1, false = 0
int -> int // trivial

/* float casts */
int -> float
string -> float // “99.99”, value becomes 99.99
float -> float // trivial

/* string casts */
bool -> string // true becomes "true", false becomes "false"
float -> string
int -> string
string -> string // trivial
```

We are able to cast literals using the following syntax `|as <builtin-type>|`. See below for an example:

```
@my_int: int; // declare a variable
@my_int = |as int| 5.0; // cast the literal from float to int, value = 5
```

We are able to cast expressions. See below for an example:

```
@my_float: float; // declare a variable
@my_float = |as float| (50 * 10); // cast the int expression to float, value = 500.0
```

We are able to perform compound casts. See below for an example:

```
@my_str: string; // declare a variable
@my_str = |as string| (|as int| 5.89); // cast the compound expr to string, value = “5”
```

4. Execution model

4.1 Program Execution

Viz source code must be defined within a file containing a “.viz” extension. A Viz file may contain structs, functions, iteration, logic, and any statemenets that a user would like to include in a program. Our program is defined as a list of struct declarations and function declarations. Thus, we can define structs first and have them used at any point after, and a similar concept applies to functions. Here is our top level context free grammar definition of a Viz program:

```
program:
 /* nothing */ { ([], []) }
 | sdecls fdecls EOF { ($1, $2) }
```

A Viz program must always include a main function. If main is not defined then an error is thrown because there is no entry point for the code execution. Main stylistically can return nothing, a none type, or an int type. See below examples:

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

```
func main(): none {
    print("Hello World!");
    return;
}
```

```
func main(): int {
    print("Hello World!");
    return 0;
}
```

4.2 Blocks

As a group we spent a great deal of time implementing a real symbol table, which would allow for function and variable declaration anywhere. Each code block (scope) requires a set of statements that are to be executed within a pair of matching brace ("{" , "}") delimiters. Code blocks are self-contained and meant to provide organization to code. Code blocks can be nested.

```
func main(): int {
    {
        print("Hello World!");
    }
    return 0;
}
```

4.3 Functions

Functions are described in greater detail below, but in order to call one they must have been defined at a prior point. See below for an example:

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

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

5. Simple Statements and Expressions

At a high level, this is what our context free grammar looks like for statements. We have the ability to handle: variable declarations, expressions, blocks, control flow, and logic.

```
stmt:
 | expr SEMI { Expr $1 }
 | if_stmt { $1 }
 | block { $1 }
 | loop  { $1 }
 | return_statement SEMI { $1 }
 | id_block              { $1 }
 | vdecl var_init_opt SEMI  { VarDecl($1, $2) }
 | vdecl_list vdecl_list_init_opt SEMI {
     let var_list        = $1 in
     let vdecl_ty        = fst $2 in
     let vdecl_exp       = snd $2 in
     let create_var_decl = (fun var_name -> ((vdecl_ty, var_name), vdecl_exp)) in
     let list_of_decls = List.fold_left (fun lst var_name -> (create_var_decl var_name) ::
lst ) [] var_list
     in VarDeclList(list_of_decls)
   }
```

We will dive deeper into each of the statement types in the next few sections.

5.1 Variable Declarations

We will declare variables using the @symbol followed by the colon (":") then the variable type. Each of these variables will be tied to the specific scope that they were declared in. By carrying around this @symbol the program readability will improve, especially as we pass variables to functions.

```
@email: string;
@are_we_done_yet: bool;
@flt: float;
```

5.2 Variable Initialization

Variables must always be initialized by the programmer before use. If they are not initialized then a parse error will be thrown.

There are three ways to initialize variables in our language. Declare first, then initialize:

```
@email: string;
@are_we_done_yet: bool;
@flt: float;

@email = "md3420@columbia.edu";
@are_we_done_yet = false;
@flt = 0.1;
```

Declare and initialize at the same time:

```
@email: string = "md3420@columbia.edu";
@are_we_done_yet: bool = false;
@flt: float = 0.1;
```

Utilize a builtin construct called initializer lists. There are cases when you would want to initialize multiple variables to the same initial value. To support this functionality, we will start with a list of variable names (@symbol_1, …, @symbol_n) followed by an arrow (“->”) and then a pair including the variable type and value. This code will ensure that all variables before the arrow are of the same type, and have the same value. See example below:

```
// all variables will be ints, and have values of 0
@day_counter, @hour_counter, @min_counter -> (int, 0);
```

5.3 List Expressions

List expression are written by having 0 or more elements, separated by comma, and enclosed within square brackets. Within the list declaration, we will enforce the data types that are passed between the bar “|” operators. We don’t believe that these data type declarations are an impediment to the user because it is easier to read the code if you know what types are housed within the collections. Lists can only contain the same primitive types. We support the following list expressions:

```
// declaring  lists
@my_numbers_1: list|int| = [1, 3, 5, 7, 11, 13];
@my_numbers_2: list|float| = [1.0, 3.0, 5.0];
@my_strings: list|string| = [“hello”, “world”, “!”];
@my_numbers: list|bool| = [false, true, false, true];

// reassign list variables
@my_strings = [“just”, “reassigned”, “new”, “list”];
```

For more advanced list manipulations, please check out section 7.3 which details our list library methods.

5.4 Multiline Expressions

We realize that there are cases that breaking a larger statement up over multiple lines may improve readability. In order to do this, users can simply start a new line and keep writing statements if they are within a defined construct. Otherwise a “\” token will be needed to let the compiler know to treat everything after as whitespace.

```
@cs_courses: list|string| = [ "Programming Languages & Translators",
                              "Operating Systems",
                              "Advanced Software Engineering",
                              "Analysis of Algorithms" ];

@long_form_math: int = 100000 \
                       + 1;
```

5.5 Boolean Expressions

Logic is central to modern programming control flow. Boolean expressions will be needed when writing loops, and branching statements. Viz is strongly typed, so we require that the types involved in an operator match semantically. Boolean literals can be instantiated like in the below example:

```
// declare and instantiate
@reality: bool = false;
@fantasy: bool = true;

// declare then instantiate
@ok: bool;
@ok = true;
```

More often programmers will find themselves working with expressions, in order to do the branching and loop statements. The result of these logical expressions will be a truth value. Find below the logical expressions that are legal for ints and floats:

```
// boolean expression for int and float expressions
(expr) == (expr)
(expr) != (expr)
(expr) >= (expr)
(expr) <= (expr)
(expr) >  (expr)
```

```
(expr) <  (expr)
```

See below the logical operators that we support for boolean expressions:

```
// boolean expression for boolean expressions
(expr) == (expr)
(expr) != (expr)
(expr) and (expr)
(expr) or (expr)

// unary operator which negates truth value of expr
not (expr)
```

All of the string logical operators are supported using a linked string library written in C. Strings will not be compared by memory addresses, rather the contents of the strings (i.e. their characters). See below the logical operators that we support for string expressions:

```
// boolean expression for string equality checks
(expr) == (expr)
(expr) != (expr)

// these 4 operators follow lexicographical ordering
(expr) >= (expr)
(expr) <= (expr)
(expr) >  (expr)
(expr) <  (expr)

// example string comparisons
@comparison: string = "Boolean Comparison";
@comparison == "Boolean Comparison"; // true

"Boolean Comparison" < "Comparison";  // true
```

5.6 Arithmetic Expressions

Also central to programming is the ability for computers to perform arithmetic operations. The resulting type of the arithmetic will depend on the two operand types that were part of the expression. Recall that Viz is strongly typed, so the operand types that are a part of the expression need to make sense semantically. See below the arithmetic operators that are supported for ints:

```
// arithmetic expression for int results in int type
(expr) + (expr)
```

```
(expr) - (expr)
(expr) * (expr)
(expr) / (expr)
(expr) %  (expr)

// negate operator will make a positive number negative
// recall we only parse positive numbers
@neg_number: float = -10;
```

See below the arithmetic operators that are supported for floats:

```
// arithmetic expression for float results in float type
(expr) + (expr)
(expr) - (expr)
(expr) * (expr)
(expr) / (expr)

// negate operator will make a positive number negative
// recall we only parse positive numbers
@neg_number: float = -10.0;
```

5.7 Variable Increment Assignment Operators

Now that we have defined what type of arithmetic operators are allowed for both integer and float types, we can introduce increment assignment. We want to provide programmers with shorthand for updating variables in the code. This syntactic sugar will likely be used within loop structures for code readability and simplicity. See below for the increment operators supported for ints:

```
@int_variable += (expr);
@int_variable -= (expr);
@int_variable *= (expr);
@int_variable /= (expr);
@int_variable %= (expr);
```

See below the increment operators supported for floats:

```
@float_variable += (expr);
@float_variable -= (expr);
@float_variable *= (expr);
@float_variable /= (expr);
```

6. Compound Statements and Expressions

6.1 Control flow

To evaluate the following expressions the conditional expression will need to be evaluated to a boolean in order to execute a specific branch or loop. Then depending on the expression value, you will enter one of the enclosing blocks of code.

6.1.1 If, Elif, Else Statements

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

The first if block will be entered if the conditional expression evaluates to true. If and only if it evaluates to false, then we can potentially enter the else if block of code. To enter any particular block of code all of the preceding conditionals have to evaluate as false. The else is a catch-all for all of the cases that have not been defined within the if blocks. The else case is also purely optional like in most programming languages.

6.1.2 If, Else Using Ternary ? Operator

There are many use cases where we would like to have a quick logical operator that can span one single line. This is the use of the “?” operator which will be defined as follows:

```
Boolean expression ? true-block... : false-block... ;
```

```
@stmt: string;
1 > 0 ? @stmt = "Hello, World" : @stmt = "Goodbye, World";
```

The @stmt will be available in whichever block that statement is scoped after it is done executing.

```
@stmt: string;
@my_bool: bool = true;
0 < 1 ? @stmt = "Hello, World" : @my_bool = false;
```

Furthermore, the true and false block could set or manipulate different types of expressions. In the example above we could manipulate a string variable in the true-block but set a boolean variable in the else-block.

6.1.3 For Loop

Looping is the next major paradigm that is used for modern programming tasks, and for beginners there are way too many “off by one” errors. Beginners often loop one extra or loop one less than the desired number of iterations due to loop syntax. Thus, we have come up with a little more readable syntax that will cut out these basic errors.

There is one type of for loop which is “ranged.” These ranged loops will have the following style:

```
// ending_num is allowed to be an integer or an integer variable
for counter in starting_num ... <ending condition> ending_num step step_number
```

The step and step_number will default to incrementing by 1 if those two components are omitted from the for loop declaration. Depending on what ending condition we place here, we can support a couple of different types of loops. Our position is that readability is key, and is helpful for quicker debugging instead of cryptic parsing of each line of code.

The below for loop allows us to increment from a starting number to the end, where the endpoint is not inclusive. Since step is omitted, we will increment by 1 in each iteration.

```
for i in start ... < end {
     Block...
}
```

In the below loop, we will iterate from start to end inclusively, the caveat being that we will now increment k by n after each loop iteration.

```
for k in start ... <=end step n {
     Block...
}
```

For this ranged for loop we currently support the following end conditions: <=, >=, >, <. The goal for all of these loop constructs is to allow for simpler code development and to allow for simpler debugging. Less time spent debugging off by one errors, adds more time for software development.

6.1.4 While Loop

While loops are best used in cases where we are unsure of the number of times we would need to iterate within a program. Thus, the loop begins by first evaluating the boolean in the loop condition. If the condition is true then the code block will be entered. When the boolean condition finally returns false, that block will be skipped and the code flow will continue from after the end of the block.

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

6.1.5 Infinite Loop

Infinite loops hold a special place in systems programming, and one great use case comes to mind being sockets programming and developing server code. We could produce an infinite loop by writing `while true { … }`, which will execute forever. However, since this situation arises in many different computer science contexts we feel that there is a necessity for a sleek and readable infinite_loop style.

Thus, the next time you write that web server, ditch the while true and use the following:

```
infinite_loop {
       // Code block executes forever...
}
```

6.2 Functions

Functions allow us to segment and reuse code logic in a DRY (Don't Repeat Yourself) manner. Modern languages like Python do not provide variable type inference, type enforcement, and return types. Thus, users can pass variables to functions with incompatible parameter types which can only be enforced at runtime. For beginners, it is best to understand what the function will take as input and return as output.

Functions are defined with the keyword func, and all of the loop body statements will be executed sequentially. The user must define a return type at the end of the function declaration, and can optionally add any arguments (with corresponding data types) to their function signature.

```
// All three of these functions are analogous
func my_first_func() : none {
        print("Hello World");
}

func my_first_func() : none {
        print("Hello World");
        return None;
}

func my_first_func() : none {
        print("Hello World");
        return;
}

// This would cause a parse error, due to a type mismatch
func my_first_func() : string {
        print("Hello World");
        return 1234;
}
```

More interesting functions would take typed input parameters from a caller, execute some statements with the data, and return a final typed expression according to the function stub. The input parameters will be copied and an object will be returned back to the caller. Copying the input argument by argument is an expensive operation, but we don't want our users to worry about memory management.

```
func how_many_days(@hours: int, @min: int): float {

       @seconds: int = @hours * 60 * 60;
       @seconds += @min * 60;
       @conversion: float = @seconds / (24 * 60 * 60);

       return @conversion; // return a copy of this variable
}
```

We also support struct return types, for an example using the init_function please check out section 3.2.2 on user defined structs.

7. Standard Library

No modern programming language would be complete without a standard library of builtin functions that can enrich the potential writable code. We didn't have the time to fulfill a robust viz standard library, but we were able to provide users with a small set of functions that can be used. They will be detailed in the following sections.

7.1 String Library

Strings are an important programming construct, and without predefined library functions their use cases would have been severely limited. Thus, using C we were able to write the following functions and comparison operators:

7.1.1 Operators

String Concatenation (+)

We provide the ability to concatenate two strings. Using C, this operator will use "strcat" in order to combine two char * lists into one and malloc the final string in memory. A pointer to the new string will be returned, and all memory management will be handled by Viz.

```
@greeting: string = "hello ";
@subject: string = "world!";
print(@greeting + @subject); // result will be "hello world!" printed to console
```

String Equality (== and !=)

These two operators can be particularly useful if we are searching for a string. These strings use the C "strcmp" function that checks if the two operands have the same underlying characters. The result of this expression will be a bool.

```
"hello " == "world"; // returns false
"hello " != "world"; // true
```

String Comparison (>=, <=, >, <)

These four operators can be particularly useful if we are sorting strings. These strings use the C “strcmp” function that checks the lexicographic ordering of the two operands. The result of this expression will be a bool.

```
"hello " < “world”; // returns true
"world" <= “world”; // true
"hello " > “world”; // returns false
"zeta" >= “world”; // true
```

7.1.2 Functions

strlen

This function will provide the user the ability to get the length of a string. This will be particularly useful if the programmer wanted to iterate through a string.

```
str_len("hello"); // length of string = 5
```

to_upper

This function provides the user the ability to convert a base string to one with all uppercase letters. All non-letters will remain unchanged. All memory management will be handled by viz.

```
to_upper("I’m Yelling!!!"); // string is now “I’M YELLING!!!”
```

to_lower

This function provides the user the ability to convert a base string to one with all lowercase letters. All non-letters will remain unchanged. All memory management will be handled by viz.

```
to_lower("I’M WhisPerIng..."); // string is now “i’m whispering...”
```

7.2 Printing Library

This is a basic function, which is very necessary for debugging code and of course printing output of your viz program to the console. We have spent the time developing a singular print function that will print any of our primitive types or any expressions that result in primitive types. Our two printing functions are:

```
// print functions
println(); // will print a “\n” character to the console.
print(expr); // where expr is an expression that evaluates to int, float, bool, or string

// under the hood these are how the functions are mapped
print(int expr) → print_int(int expr) → print(|as string| int expr)
print(float expr) → print_float(float expr) → print(|as string| float expr)
print(bool expr) → print_bool(bool expr) → print(|as string| bool expr)
print(string expr) → print(string expr) → print(|as string| string expr)
print(list expr) → print_list(list expr) → void print_list(int* arr) // using the C
library
println() → print(“\n”) → print(|as string| “\n”)
```

7.3 List Library

Lists in Viz are used to store multiple values in a single variable. The data type is defined as the collection of same primitive data type, such as int, float, bool and string.

To declare a list in Viz, users only need to provide the type information.

```
@my_list: list|int|;
```

Here, int is the data type for the list, and my_list is the list name. Users can also initialize the list while declaring them.

```
@my_list: list|int| = [1, 3, 5, 7, 11];
```

To access a single element in the list, users could use the subscript syntax.

```
@my_list[0]; //access the first element.
```

Different from arrays in C, Viz lists are dynamic, meaning that it does not require the size information being provided while declaring and will be automatically resized when necessary. Users could reassign a declared list variable with list of different length. For example:

```
@my_list: list|int| = [1, 3, 5];
```

```
@my_list = [7, 11]
```

Viz's list library also comes with three helper functions to manipulate the list, "push", "pop" and "list_len".

list_len

List_len accepts list|int| type and returns an int type containing the number of elements currently in the list.

```
@my_list: list|int| = [1, 3, 5];
print(list_len(@my_list)); //outputs 3
```

pop

Pop accepts an list|int| type and returns an int type that is the last element from the list. Calling pop on an empty list will result in a value of 0.

```
@my_list: list|int| = [1, 3, 5];
print(pop(@my_list)); // outputs 5
print(list_len(@my_list)); // outputs 2
```

push

Push accepts a list|int| type x and int type y and pushes element y onto list x at the last free position on the list. It returns no type.

```
@my_list: list|int| = [1, 3, 5];
push(@my_list, 7);
print(list_len(@my_list)); // outputs 4
print(pop(@my_list)); //outputs 7
print(list_len(@my_list)); // outputs 3
```

Current List Limitations

The current version of Viz only supports int list helper functions. String, bool, and float lists are not supported at this time. With helper functions on lists being one of the last features implemented, we found some unexpected edge cases across machines (mac intel v mac M1) and with respect to the docker container. This would have been addressed further if time constraints were not a factor. Regardless, this issue causes some minor failure in test cases across machines, but otherwise output is expected to be consistent.

8. VizOnline Text Editor

8.1 Overview

We have implemented an online text editor for our language in order to provide Viz programming language users a more interactive way of developing code. With this deployed web application, the user doesn't have to worry about software dependency issues by having the correct Dune, Ocaml, and LLVM versions installed locally on their machine. VizOnline affords users the ability to write, compile, and run viz source code in an online text editor. Furthermore, we provide a look into what happens under the hood during compilation through 6 different run options:

1) Running the code and displaying output
2) A look at the full build process
3) Displaying the parsed abstract syntax tree
4) Presenting the syntactically checked abstract syntax tree
5) Viewing the LLVM IR assembly code
6) Seeing how programs are scanned into tokens

8.2 How To Use VizOnline

Currently, VizOnline can be accessed by following the link provided below:

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

Unfortunately, due to server costs, we are unable to keep the hosted version online permanently. Therefore, we have included the source code for both the frontend and backend of the web interface in our frontend repository which can be found at the link below:

```
https://github.com/jakobgabrield/vizonline
```

In order to run the project locally, it requires that the machine has both node and npm installed in addition to opam, dune, and llvm which are required to compile and run the actual code.

If you do not already have all of these dependencies installed, if you are working on a mac on linux environment, you can do so by running the two following scripts sequentially:

```
// Installs node, npm, opam and dune (Follow default instructions by clicking ‘Enter/Return’
when prompted.)
./System_Setup_1.sh

// Installs llvm (Follow default instructions by clicking ‘Enter/Return’ when prompted.)
./System_Setup_2.sh
```

After you have node, npm, and llvm installed you can navigate into the **web-interface** directory of the Viz GitHub Repository and run the below commands in two separate terminal windows to start both the backend and frontend servers locally.

```
// Start Backend Server: This will start the frontend server on the default port (Usually
port 3000).
./launch-frontend.sh

// Start Frontend Server: This will install the required npm packages and then start the
backend server on PORT 5001 (Make sure that port 5001 is not in use or change the port
within the server.js file and on line 16 of the App.js file within the client/src
directory.)
./launch-backend.sh
```

9. Grammar

9.1 Scanner

```
{
   open Parser
   exception Viz_scan_error of string
}

(*
   actual scanner part
   everything below actually creates tokens
*)
let digit  = ['0'-'9']
let non_zero_digits = ['1'-'9']
let uppercase = ['A'-'Z']
let lowercase = ['a'-'z']
let letter    = ['a'-'z' 'A'-'Z']
let whitespace = [' ' '\t' '\r' '\n']

rule token = parse
(* -------- whitespaces -------- *)
| whitespace { token lexbuf}
| "/*" {multi_comment lexbuf}
| "//" {single_comment lexbuf}

(* -------- line continuation ---------*)
(*| "\\" { line_continuation lexbuf }*)
| "\\" { token lexbuf }

(* -------- keywords -------- *)
| "func" { FUNC }
| "if"   { IF }
| "else" { ELSE }
| "elif" { ELIF }
| "for" { FOR }
| "while" { WHILE }
| "infinite_loop" { INFINITE_LOOP }
| "return" {RETURN}
| "break" {BREAK} (* functionality unimplemented *)
| "continue" {CONTINUE} (* functionality unimplemented *)
| "try" {TRY} (* functionality unimplemented *)
| "catch" {CATCH} (* functionality unimplemented *)
| "raise" {RAISE} (* functionality unimplemented *)
| "link" {LINK} (* functionality unimplemented *)
| "use" {USE} (* functionality unimplemented *)
| "in" {IN}
| "step" {STEP}
| "as" {AS}
| "..." {RANGE} (* used in the for loop construct *)
| "struct" {STRUCT}

(* -------- types -------- *)
| "none" { T_NONE }
| "int" { T_INT }
| "string" { T_STR }
| "bool" { T_BOOL }
| "float" { T_FLOAT }
| "list" { T_LIST }

(* -------- arithmetic operators -------- *)
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '%' { MOD }

(* -------- assignment operators -------- *)
| '=' { ASSIGN }
| "+=" {PLUSEQ}
| "-=" {MINUSEQ}
| "*=" {TIMESEQ}
| "/=" {DIVEQ}
| "%=" {MODEQ}

(* -------- relational operators -------- *)
| "==" {EQ}
| "!=" {NEQ}
| ">=" {GTEQ}
| "<=" {LTEQ}
| ">" {GT}
| "<" {LT}
| "and" {AND}
| "or" {OR}
| "not" {NOT}
| "?" {QUESTION}

(* -------- literals -------- *)
| "true" { LIT_BOOL(true) }
| "false" { LIT_BOOL(false) }
| '"' ([^ '"']* as lxm) '"' { LIT_STR(lxm) }
| ('0'+) | non_zero_digits digit*  as lxm {LIT_INT(int_of_string lxm)} (* we store
everything as a positive magnitude number *)
| ("0." "0"+) | digit* "." digit+ as lxm { LIT_FLOAT(float_of_string lxm) } (* we store
everything as a positive magnitude number *)

(* -------- delimiters -------- *)
| "("  { LPAREN   }
| ")"  { RPAREN   }
| "["  { LBRACKET }
| "]"  { RBRACKET }
| "{"  { LBRACE   }
| "}"  { RBRACE   }
| ":"  { COLON    }
| ";"  { SEMI     }
| ","  { COMMA    }
| "."  { DOT      }
| "|"  { BAR      }
| "->" { ARROW    }

(* --------- IDs ------------ *)
| uppercase lowercase* as lxm { CAP_ID(lxm) } (* capitalized UNCAP_ID, example: Shape,
Person, Edge *)
| lowercase (digit | letter | '_')* as lxm { UNCAP_ID(lxm) } (* uncapitalized UNCAP_ID,
example: shape, person *)
| "@" letter (digit | letter | '_')* as lxm { ID_VAR(String.sub lxm 1 ((String.length lxm) -
1)) } (* variable access and decl need @ *)

(* -------- Other ----------- *)
| eof { EOF }
| _ as char {raise (Viz_scan_error ("unexpected character: " ^ Char.escaped char))}
```

```
and single_comment = parse
| '\n' {token lexbuf} (* this is how we will end a single line comment *)
| _    {single_comment lexbuf} (* want to ignore the rest of the noise *)
and multi_comment = parse
| "*/" {token lexbuf} (* end of multi line comment, head back to token *)
| "/*" {raise (Viz_scan_error ("cannot nest multi-line comments"))} (* no nested comments *)
| _    {multi_comment lexbuf} (* want to ignore the rest of the noise *)

(*
and line_continuation = parse
 | whitespace { line_continuation lexbuf }
 | _ { token lexbuf }
*)
```

9.2 Parser

```
%{
open Ast
%}

/* arithmetic */
%token PLUS MINUS TIMES DIVIDE MOD

/* assignment */
%token ASSIGN PLUSEQ MINUSEQ TIMESEQ DIVEQ MODEQ

/* relational */
%token EQ NEQ GTEQ LTEQ GT LT AND OR NOT QUESTION

/* keywords */
%token FUNC IF ELSE ELIF FOR WHILE INFINITE_LOOP RETURN BREAK
%token CONTINUE TRY CATCH RAISE LINK USE IN STEP AS RANGE STRUCT

/* type */
%token T_NONE T_STR T_INT T_BOOL T_FLOAT T_LIST

/* delimiters */
%token SEMI LPAREN RPAREN LBRACE RBRACE COLON COMMA LBRACKET RBRACKET DOT BAR
%token EOF ARROW /*LINECONTINUATION*/

/* split id into two, nothing changes outside of parser file */

%token <string> UNCAP_ID /* function names and struct member */
%token <string> ID_VAR /* variable access or assign */
%token <string> CAP_ID /* struct names */
%token <string> LIT_STR
%token <int> LIT_INT
%token <float> LIT_FLOAT
```

```
%token <bool> LIT_BOOL

/* precedence following C standard*/
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN PLUSEQ MINUSEQ TIMESEQ DIVEQ MODEQ
%left COMMA
%left SEMI
%left OR
%left AND
%left EQ NEQ
%nonassoc LBRACKET RBRACKET
%left LT GT LTEQ GTEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT NEG
%right BAR /* bar is used in typecast, this precedence is like c cast right assoc */


%start program
%type <Ast.program> program

%%


program:
 /* nothing */ { ([], []) }
 | sdecls fdecls EOF { ($1, $2) }

fdecls:
  /* nothing */ { [] }
| fdecl fdecls { $1 :: $2 }

// structs declarations
sdecls:
  /* nothing */ { [] }
| sdecl sdecls  { $1 :: $2 }

/* @x: string; */
vdecl:
 | ID_VAR COLON typ {($3, $1)}

typ:
 | T_NONE { NoneType }
 | T_STR { StrType }
 | T_INT { IntType }
 | T_BOOL { BoolType }
 | T_FLOAT { FloatType }
 | T_LIST BAR typ BAR { ListType(Some($3), None) }
 | CAP_ID { StructType($1) }

/* function declaration */
```

```
fdecl:
 /* func with args */
 | FUNC UNCAP_ID LPAREN formals_opt RPAREN COLON typ LBRACE stmt_list RBRACE
 {
   {
     rtyp = $7;
     fname = $2;
     formals = $4;
     body = $9;
   }
 }

/* struct declaration */
sdecl:
 /* struct definition, more like a C struct currently */
 /* I guess we need locals if we want class functions */
 | STRUCT CAP_ID LBRACE members_list RBRACE
 {
   {
     name = $2;
     members = $4;
   }
 }

members_list:
 vdecl SEMI { [$1] }
 | vdecl SEMI members_list { $1::$3 }

/* formals_opt */
formals_opt:
 /*nothing*/ { [] }
| formals_list { $1 }

formals_list:
 vdecl { [$1] }
 | vdecl COMMA formals_list { $1::$3 }

stmt_list:
 /* nothing */ { [] }
 | stmt stmt_list  { $1::$2 }

stmt:
 | expr SEMI { Expr $1 }
 | if_stmt { $1 }
 | block { $1 }
 | loop  { $1 }
 | return_statement SEMI { $1 }
 | id_block              { $1 }
 | vdecl var_init_opt SEMI  { VarDecl($1, $2) }
 | vdecl_list vdecl_list_init_opt SEMI {
     let var_list        = $1 in
     let vdecl_ty        = fst $2 in
     let vdecl_exp       = snd $2 in
     let create_var_decl = (fun var_name -> ((vdecl_ty, var_name), vdecl_exp)) in
     let list_of_decls = List.fold_left (fun lst var_name -> (create_var_decl var_name) ::
lst ) [] var_list
     in VarDeclList(list_of_decls)
   }

vdecl_list_init_opt:
 | ARROW LPAREN typ RPAREN            { ($3 , None    )   }
 | ARROW LPAREN typ COMMA expr RPAREN { ($3 , Some($5) ) }

vdecl_list:
 | ID_VAR { [$1] }
 | ID_VAR COMMA vdecl_list { $1 :: $3 }

var_init_opt:
 | { None }
 | ASSIGN expr { Some($2) }

return_statement:
 /* return; i.e. for nonetype return */
 | RETURN                              { Return NoneLit }
 | RETURN expr                         { Return $2      }
/*logic_expr: within control flow, we dont want SEMI involved in our exprs
 | expr { $1 } */

loop:
 | WHILE LPAREN expr RPAREN stmt           { While ($3, $5)  }
 | INFINITE_LOOP stmt        { While (BoolLit(true), $2)  }
 /* for counter in starting_num ... <ending condition>ending_num step step_number */
 | FOR ID_VAR IN LIT_INT RANGE end_condition end_item increment stmt
           {
               let var_init   = Assign(Id($2), IntLit($4)) in (* ex: i = 0 *)
               (*let predicate  = Binop(PostfixExpr(Id($2)), $6, IntLit($7)) in *) (* ex: i
< 5 *)
               let predicate  = Binop(PostfixExpr(Id($2)), $6, $7) in (* ex: i < 5 or i <
@len where @len = 5 *)
               let update     = Assign(Id($2), Binop(PostfixExpr(Id($2)), Add, $8) ) in (*
ex1: i=i+1, ex2: i=i+(-1) *)
               let block_code = $9 in
               For(var_init, predicate, update, block_code)
           }

/* this allows us to support an int final value, or a value stored in a variable */
end_item:
 | LIT_INT { IntLit($1) }
 | ID_VAR  { PostfixExpr(Id($1)) }

end_condition:
 | LT   { Less   }
 | GT   { Great  }
 | GTEQ { Geq    }
 | LTEQ { Leq    }
 /*| EQ   { Eq     }*/ /* this makes no sense, we should get rid of */

increment:
 /* dont define increment, default to 1 */ { IntLit(1) }
 | STEP LIT_INT { IntLit($2)} /* supports a positive step */
 | STEP MINUS LIT_INT { IntLit($3 * -1) } /* supports a negative step */

block:
 | LBRACE stmt_list RBRACE                 { Block $2 }

id_block:
 | LBRACE stmt_list RBRACE  SEMI            { ID_Block $2}

if_stmt:
 | expr QUESTION expr COLON expr SEMI  { If($1, Expr($3), Expr($5)) } /* (1 > 2) ?
print("true") : print("false") */
 | IF LPAREN expr RPAREN block %prec NOELSE    { If($3, $5, Block[]) } /* covers if */
 | IF LPAREN expr RPAREN block else_stmt       { If($3, $5, $6) } /* covers if/else */
 else_stmt:
 | ELIF LPAREN expr RPAREN block %prec NOELSE { If($3, $5, Block[]) }
 | ELIF LPAREN expr RPAREN block else_stmt { If($3, $5, $6) }
 | ELSE stmt { $2 }

expr:
 | postfix_expr { PostfixExpr($1) }
 /* literal */
 | string_literal { StrLit($1) }
 | LIT_INT   { IntLit($1)   }
 | LIT_BOOL  { BoolLit($1)  }
 | LIT_FLOAT { FloatLit($1) }
 | LBRACKET exprs_opt RBRACKET { ListLit($2) }

 /* arithmetic */
 | expr PLUS   expr { Binop($1, Add,   $3)   }
 | expr MINUS  expr { Binop($1, Sub,   $3)   }
 | expr TIMES  expr { Binop($1, Mult,  $3)   }
 | expr DIVIDE expr { Binop($1, Div,   $3)   }
 | expr MOD    expr { Binop($1, Mod,   $3)   }

 /* logical binary ops */
 | expr  EQ      expr { Binop($1, Eq, $3)      }
 | expr  NEQ     expr { Binop($1, Neq,   $3)   }
 | expr  LT      expr { Binop($1, Less,  $3)   }
 | expr  GT      expr { Binop($1, Great, $3)   }
 | expr  LTEQ    expr { Binop($1, Leq,   $3)   }
 | expr  GTEQ    expr { Binop($1, Geq,   $3)   }

 /* logical ops */
 | expr  AND    expr       { Binop($1, And,   $3)   }
 | expr  OR     expr       { Binop($1, Or,    $3)   }
 | MINUS expr %prec NEG { Unop(Neg, $2) }
 | NOT expr { Unop(Not, $2) }

 /* assignment */
 | postfix_expr ASSIGN expr { Assign($1, $3) } /* struct member */
 | ID_VAR PLUSEQ expr { Assign(Id($1), Binop(PostfixExpr(Id($1)), Add, $3))}
 | ID_VAR MINUSEQ expr { Assign(Id($1), Binop(PostfixExpr(Id($1)), Sub, $3))}
 | ID_VAR TIMESEQ expr { Assign(Id($1), Binop(PostfixExpr(Id($1)), Mult, $3))}
 | ID_VAR DIVEQ expr { Assign(Id($1), Binop(PostfixExpr(Id($1)), Div, $3))}
 | ID_VAR MODEQ expr { Assign(Id($1), Binop(PostfixExpr(Id($1)), Mod, $3))}
  /* remove clarifying parens */
 | LPAREN expr RPAREN { $2 } /* (expr) -> expr. get rid of parens */

 /* function call */
 | UNCAP_ID LPAREN exprs_opt RPAREN { FuncCall($1, $3) }

 /* just need to ensure that this is right associative */
 | BAR AS typ BAR expr {TypeCast($3, $5)}

/* allowing for multi-line strings */
string_literal:
| LIT_STR   {  $1 }
| LIT_STR string_literal { $1 ^ $2 } /* concat strs */

// Match the following patterns
// ""
// "expr, expr, expr"
exprs_opt:
 /*nothing*/ { [] }
 | exprs { $1 }

exprs:
 expr  { [$1] }
 | expr COMMA exprs { $1::$3 }

postfix_expr:
 /* variable access */
 | ID_VAR { Id($1) }
 | postfix_expr DOT UNCAP_ID { MemberAccess($1, $3) }
   /* List subscript [] */
 | postfix_expr LBRACKET expr RBRACKET { Subscript($1, $3) }
```

9.3 Semantic Checking

```
(* Semantic checking for the Viz compiler *)

open Ast
open Sast
open Ast_fmt
```

```
module StringMap = Map.Make(String)
module StringHash = Hashtbl.Make(struct
 type t = string
 let equal x y = x = y
 let hash = Hashtbl.hash
end)

(* Used to query a struct using it's name from a StringMap *)
type struct_symbol = {
 members: typ StringMap.t
}

(* Semantic checking of the AST. Returns an SAST if successful,
  throws an exception if something is wrong.

  Check each function *)

let check ((structs: struct_def list), (functions: func_def list)) =

 (* Verify a list of bindings has no duplicate names *)
 let check_binds (kind : string) (binds : bind list) =
   let rec dups = function
       [] -> ()
     | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
       raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
     | _ :: t -> dups t
   in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
 in

 (* built-in declarations *)
 let built_in_decls =
   let add_built_in_function map (ret_type, name, f) = StringMap.add name {
     rtyp = ret_type;
     fname = name;
     formals = f;
     body = [] } map
     in List.fold_left add_built_in_function StringMap.empty [
                                                               (NoneType, "print", [StrType,
"x"]);
                                                               (NoneType, "print_int",
[IntType, "x"]);
                                                               (NoneType, "print_float",
[FloatType, "x"]);
                                                               (NoneType, "print_bool",
[BoolType, "x"]);
                                                               (NoneType, "println", []);
                                                               (NoneType,"print_list",
[(ListType (Some(IntType), None)), "x"]);
                                                               (IntType, "str_len", [StrType,
"x"]);
                                                               (StrType, "to_upper",
[StrType, "x"]);
                                                              (StrType, "to_lower",
[StrType, "x"]);
                                                              (IntType, "list_len",
[StrType, "x"]);
                                                              (IntType, "list_len_int",
[(ListType (Some(IntType), None)), "x"]);
                                                              (IntType, "pop", [(ListType
(Some(IntType), None)), "x"]);
                                                              (StrType, "push", [(ListType
(Some(IntType), None)), "x"; IntType,"x"]);
                                                              ]
 in

 (* Add Struct to the symbol table *)
 let add_struct map (s:struct_def) =
   let name = s.name in
   let add_member map (member:bind) =
     match member with
     | (_, m_id) when StringMap.mem m_id map ->
       failwith ("Duplicate struct member: " ^ m_id)
     | (m_typ, m_id) -> StringMap.add m_id m_typ map
   in
   let (symbol:struct_symbol) = {
     members = List.fold_left add_member StringMap.empty s.members;
   }
 in
   match s with (* No duplicate structs or redefinitions of built-ins *)
    | _ when StringMap.mem name map -> failwith ("duplicate struct " ^ name)
    | _ ->  StringMap.add name symbol map
 in

 (* Add function name to symbol table *)
 let add_func map fd =
   let name = fd.fname in
   match fd with (* No duplicate functions or redefinitions of built-ins *)
    | _ when StringMap.mem name built_in_decls ->
      failwith ("function " ^ name ^ " may not be defined, it is reserved")
    | _ when StringMap.mem name map ->
      failwith ("duplicate function " ^ name)
    | _ -> StringMap.add name fd map
 in

 (* Collect all struct names into one symbol table *)
 let struct_decls = List.fold_left add_struct StringMap.empty structs
 in

 (* Collect all function names into one symbol table *)
 let function_decls = List.fold_left add_func built_in_decls functions
 in

 (* Return a struct from our symbol table *)
```

```
let find_struct name: struct_symbol =
  try StringMap.find name struct_decls
  with Not_found -> raise (Failure ("unrecognized struct: " ^ name))
in

(* Given a struct symbol, and the member name, return the type of that member*)
let find_struct_member (symbol: struct_symbol) (member_id:string): typ =
  try StringMap.find member_id symbol.members
  with Not_found -> raise (Failure ("unrecognized member: " ^ member_id))
in


(* Return a function from our symbol table *)
let find_func s =
  try StringMap.find s function_decls
  with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

let _ = find_func "main" in (* Ensure "main" is defined *)

let check_struct (s:struct_def) =
  (* Make sure no members are void or duplicate *)
  check_binds "struct members" s.members;
  (* body of check_struct *)
  {
    sname = s.name;
    smembers  = s.members;
  }
in

let check_func func =
  (* Make sure no formals are void or duplicates *)
  check_binds "function formals" func.formals;

  (* Raise an exception if the given rvalue type cannot be assigned to
     the given lvalue type *)
  let check_assign (lvaluet:typ) (rvaluet:typ): typ = match lvaluet with
  (* If the lhs and rhs are both list and have the same type, using the rhs list length *)
    | ListType(l_list_typ, _) -> (match rvaluet with
      | ListType(r_list_typ, _) when l_list_typ = r_list_typ -> rvaluet
      | _ -> failwith ("illegal assignment " ^ fmt_typ lvaluet ^ " = " ^ fmt_typ rvaluet)
    )
    | _ when lvaluet = rvaluet -> lvaluet
    | _ -> failwith ("illegal assignment " ^ fmt_typ lvaluet ^ " = " ^ fmt_typ rvaluet)
  in

  (* Build local symbol table of variables for this function *)
  let symbols : typ StringMap.t =
    List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
      StringMap.empty (func.formals)
  in
  (* Query a variable from our local symbol table *)
  let type_of_id (id: string) (symbols : typ StringMap.t) : typ =
    try StringMap.find id symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^ id))
  in

  let uninited_symbols = StringHash.create 10 in

  (* Return a semantically-checked expression, i.e., with a type *)
  let rec check_expr (symbols: typ StringMap.t) (e: expr) : (sexpr * typ StringMap.t) =
    match e with
      IntLit x -> ((IntType, SIntLit x), symbols)
    | FloatLit x -> ((FloatType, SFloatLit x), symbols)
    | StrLit x -> ((StrType, SStrLit x), symbols)
    | BoolLit x -> ((BoolType, SBoolLit x), symbols)
    | NoneLit   -> ((NoneType, SNoneLit), symbols)
    | ListLit (a : expr list) -> (match a with
      | [] -> ((ListType (None, Some(0)), SListLit []), symbols)
      | (x :: _) as xxs -> let ((t, _), symbols) = check_expr symbols x in
        let sa = List.map (fun i -> match (check_expr symbols i) with
          | ((i_t, _), _) when i_t != t -> raise (Failure ("Invalid list literal. Expect
type " ^ fmt_typ t ^ ", found" ^ fmt_typ i_t))
          | ((t', se'), _) -> (t', se')) xxs
        in
        let len = List.length xxs in
        ((ListType (Some(t), Some(len)), SListLit sa), symbols)
    )
    | Assign(pe, e) ->
      let (l_t, _) as l_spe = match pe with
        | Id id -> (type_of_id id symbols, SId(id))
        | _ -> check_postfix_expr symbols pe
      in
      let ((r_t, _) as r_se, symbols) = check_expr symbols e in
      let t = check_assign l_t r_t in
      (* If the left side is a variable, removed the symbol from uninited symbol table *)
      let new_symbols = (match pe with
      | Id id ->
        (if StringHash.mem uninited_symbols id then StringHash.remove uninited_symbols id;
        (* update the id's type in the symbol table, in the case that list has been
reassigned *)
        StringMap.add id t symbols)
      | _ -> symbols) in
      ((t, SAssign(l_spe, r_se)), new_symbols)

    | Binop(l, bo, r) as ex->
      let ((ltype, l'), symbols) = check_expr symbols l in
      let ((rtype, r'), symbols) = check_expr symbols r in
      (* we can only do binop on operands of same type *)
      let compatible_types = (ltype = rtype) in
      (* throw error, or return final_type for supported binops *)
      let final_type =
        if compatible_types = false then
```

```
          raise (Failure ("incompatible types for binary operator " ^
                fmt_typ ltype ^ " " ^ fmt_op bo ^ " " ^
                fmt_typ rtype ^ " in " ^ fmt_expr ex))

        else
          (fun my_op -> match my_op with
          | (Add | Sub | Mult | Mod | Div) when ltype = IntType && rtype = IntType ->
IntType
          | (Add | Sub | Mult | (*Mod |*) Div) when ltype = FloatType && rtype = FloatType
-> FloatType
          | (Eq | Neq) -> BoolType
          | Add when ltype = StrType && rtype = StrType  -> StrType
          | (Leq | Geq | Less | Great) when (ltype = IntType && rtype = IntType ||
                                             ltype = FloatType && rtype = FloatType ||
                                             ltype = StrType && rtype = StrType) ->
BoolType
          | (And | Or) when (ltype = BoolType && rtype = BoolType) -> BoolType
          | _ -> raise (Failure ("No operator (" ^ fmt_op bo ^ ") " ^ "to handle type (" ^
                        fmt_typ ltype ^ ", " ^ fmt_typ rtype))
          ) bo
      in ((final_type, SBinop((ltype, l'), bo, (rtype, r'))), symbols)
    | Unop(op, e) as ex ->
      let ((t, e'), symbols) = check_expr symbols e in
      let ty = match op with
        Neg when t = IntType || t = FloatType -> t
      | Not when t = BoolType -> BoolType
      | _ -> raise (Failure ("illegal unary operator " ^
                              string_of_uop op ^ fmt_typ t ^
                              " in " ^ fmt_expr ex))
      in ((ty, SUnop(op, (t, e'))), symbols)
    | FuncCall(fname, args) as call ->
      let fd = find_func fname in
      let param_length = List.length fd.formals in
      if List.length args != param_length then
        raise (Failure ("expecting " ^ string_of_int param_length ^
                        " arguments in " ^ fmt_expr call))
      else
      let check_call (ft, _) e =
        let ((et, e'), _) = check_expr symbols e in
        if fname = "print" then (et, e') (*convert to string*)
        else if fname = "list_len" then (et, e')
        else(check_assign ft et, e')
      in
      let args' = List.map2 check_call fd.formals args in
      let func_name = match fname with
        | "print" ->
          (match (fst (List.hd args')) with
          | IntType -> "print_int"
          | FloatType -> "print_float"
          | StrType -> "print"
          | BoolType -> "print_bool"
          | NoneType -> failwith "Does not support print None type"
```

```
            | ListType _ -> "print_list"
            | StructType _ -> failwith "Does not support print custom Struct type"
          )
        | "str_len" ->
          (
            match (fst (List.hd args')) with
            | StrType -> "str_len"
            | _ -> raise (Failure ("cannot get string length of non string"))
          )
        | "to_upper" | "to_lower" ->
          (
            match (fst (List.hd args')) with
            | StrType -> fname
            | _ -> raise (Failure ("invalid type for to_upper/to_lower"))
          )
        | "push" -> (
            match (fst (List.hd args')) with
            | ListType (Some(IntType), _) -> "push"
            | _ -> raise (Failure ("cannot push to parameter"))
          )
        | "pop" ->
          (
            match (fst (List.hd args')) with
            | ListType (Some(IntType), _) -> "pop"
            | _ -> raise (Failure ("cannot pop from parameter"))
          )
        | "list_len" ->
          (
            match (fst (List.hd args')) with
            | ListType (Some(IntType), _) -> "list_len_int"
            | ListType (Some(StrType), _) -> raise (Failure ("need to support str arr len"))
            | ListType (Some(FloatType), _) -> raise (Failure ("need to support float arr
len"))
            | ListType (Some(BoolType), _) -> raise (Failure ("need to support bool arr
len"))
            | _ -> raise (Failure ("cannot get array length of non list"))
          )
        | _ -> fname
        in
        ((fd.rtyp, SFuncCall(func_name, args')), symbols)
      | TypeCast(ty, expr) ->
        let ((ty_exp, var), symbols) = check_expr symbols expr in
         (
          let type_cast_err e1 e2 =
            raise (Failure("Cast type not supported from " ^
                          fmt_typ e1 ^ " to " ^
                          fmt_typ e2)) in

           match ty with
              | IntType -> (
                if ty_exp = FloatType || ty_exp = IntType || ty_exp = BoolType || ty_exp =
StrType
```

```
                    then ((ty , STypeCast(ty, (ty_exp, var))), symbols)
                else type_cast_err ty_exp ty
              )
              | FloatType -> (
                if ty_exp = IntType || ty_exp = FloatType || ty_exp = StrType
                    then ((ty ,STypeCast(ty, (ty_exp, var))), symbols)
                else type_cast_err ty_exp ty
              )
              | StrType ->
                if ty_exp = IntType || ty_exp = FloatType || ty_exp = BoolType || ty_exp =
StrType
                then ((ty ,STypeCast(ty, (ty_exp, var))), symbols)
                else type_cast_err ty_exp ty
              | _ -> type_cast_err ty_exp ty
         )
      | PostfixExpr pe -> (match pe with
        | Id id -> if StringHash.mem uninited_symbols id
          then failwith ("uninitialized local variable '" ^ id ^ "' used.")
        | _ -> ());
        let spe = check_postfix_expr symbols pe in
        ((fst spe, SPostfixExpr spe), symbols)

   and check_postfix_expr (symbols: typ StringMap.t) (pe: postfix_expr) : spostfix_expr =
     match pe with
      | Id id -> (type_of_id id symbols, SId id)
      | Subscript(list_pe, idx_expr) ->
        let (list_typ, _) as list_spe = check_postfix_expr symbols list_pe in
        let (idx_sexpr, _) = check_expr symbols idx_expr in
        let (idx:int) = match (idx_sexpr) with
          | IntType, SIntLit x -> x
          | IntType, SPostfixExpr (_ , SId _) -> 0
          | t, _ -> failwith (String.concat "" ["Subscript operator [] expect index to be
int, got: "; fmt_typ t])
        in
        let (list_ele_typ, (len:int)) = match (list_typ) with
          | ListType (Some(t), Some(l)) -> (t, l)
          | ListType (None, _) -> failwith "Runtime error: List type unknown."
          | ListType (_, None) -> failwith "Runtime error: List length unknown."
          | t -> failwith (String.concat "" ["Subscript operator [] expect list, got: ";
fmt_typ t])
        in
        if idx >= len then failwith "Index out of range."
        else (list_ele_typ, SSubscript(list_spe, idx_sexpr))
      | MemberAccess (pe, member_id) ->
        (* Check the type of the given postfix expression is a struct *)
        let spe = check_postfix_expr symbols pe in
        let struct_name:string = match spe with
          | (StructType name, _) -> name
          | _ -> failwith "The type of the given postfix expression must be Struct."
        in
        (* Check the given struct existed *)
        let s_symbol = find_struct struct_name in
        (* Check the memebr_id exists in the given struct *)
        let member_type = find_struct_member s_symbol member_id in
        (member_type, SMemberAccess(spe, member_id))
    in

    let check_bool_expr symbols e =
      let ((t, e'), _) = check_expr symbols e in
      match t with
      | BoolType -> (t, e')
      |  _ -> raise (Failure ("expected Boolean expression in " ^ fmt_expr e))
    in

    (* Here we need to ensure that you are following
      correct fdecl format. see below
      added this in to check two function types:
        1) func main(): none { ... }
        2) func main():
    *)
    (*let check_function_decl = true
          {
        styp = f.typ;
        sname = f.name;
        sparams = f.params;
        sbody = check_program f.body;
      } something like ensure that these fields can be filled
    in *)
    let rec check_stmt_list symbols = function
        [] -> []
      | Block sl :: sl'  -> check_stmt_list symbols (sl @ sl') (* Flatten blocks *)
      | s :: sl ->
          let (stmt, new_symbols) = check_stmt symbols s
          in
          stmt :: check_stmt_list (match s with
                                  | ID_Block _ -> symbols
                                  | _ -> new_symbols) sl

    and check_list_var_decl symbols ((list_ele_typ, id), list_e):(svar_decl * typ
StringMap.t) =
      let ((list_t, list_sx) as list_sexpr, symbols) = check_expr symbols list_e in
      (* match rhs expression type *)
      let (bind, sexpr) = (match list_t with
        (* rhs is empty list, we need to deduce the empty list's type *)
        | ListType (None, _) ->
          let deduced_expr_t = ListType(Some(list_ele_typ), Some(0)) in
          ((deduced_expr_t, id), Some((deduced_expr_t, list_sx)))
        (* rhs has the same type as lhs *)
        | ListType (Some(expr_ele_t), _) when expr_ele_t == list_ele_typ ->
          ((list_t, id), Some(list_sexpr))
        | _ -> failwith "Type mismatch, expected list.")
      in
      let new_symbols = StringMap.add id list_t symbols in
      ((bind, sexpr), new_symbols)
```

```
  and check_var_decl symbols (((t, id) as b, e):var_decl):(svar_decl * typ StringMap.t) =
    match e with
    | None -> (* rhs is empty *)
      let new_symbols = StringMap.add id t symbols in
      (* Struct don't need to be initialized *)
      (match t with
      | StructType _ -> ()
      | _ -> StringHash.add uninited_symbols id true);
      ((b, None), new_symbols)
    | Some(e) -> match t with (* match variable's type *)
      | ListType (Some(list_ele_typ), None) -> (* Special case: lhs is list type *)
        check_list_var_decl symbols ((list_ele_typ, id), e)
      | _ -> let ((expr_t, _) as e_sexpr, symbols) = check_expr symbols e in
        if t <> expr_t then
          failwith (String.concat "" [
            "Type mismatch for variable: '"; id; "'. ";
            "Expect '"; (fmt_typ t); "'";
            ", Got: '"; (fmt_typ expr_t); "'"])
        else let new_symbols = StringMap.add id t symbols in
        ((b, Some(e_sexpr)), new_symbols)

  and check_var_decl_list symbols (dl: var_decl list): (svar_decl list * typ StringMap.t) =
match dl with
      | [] -> ([], symbols)
      | x :: xs ->
        let (svar_decl, new_symbols) = check_var_decl symbols x in
        let (svar_decl_list, new_symbols) = check_var_decl_list new_symbols xs in
        (svar_decl :: svar_decl_list, new_symbols)

  (* Return a semantically-checked statement i.e. containing sexprs *)
  and check_stmt (symbols : typ StringMap.t) = function
    (* A block is correct if each statement is correct and nothing
       follows any Return statement.  Nested blocks are flattened. *)
      Block sl -> (SBlock (check_stmt_list symbols sl), symbols)
    | ID_Block sl -> (SID_Block (check_stmt_list symbols sl), symbols)
    | Expr e ->
      let sexpr, symbols = check_expr symbols e in
      (SExpr sexpr, symbols)
    | If(e, st1, st2) ->

      (*(* note: we need to check for st2 being a No_op *)
      (* this case is if without else *)
      if check_stmt st2 = SNo_op then
        SIf(check_bool_expr e, check_stmt st1, SBlock([]))
      (* this case is if/else *)
      else *)
        (SIf(check_bool_expr symbols e, fst (check_stmt symbols st1), fst (check_stmt
symbols st2)), symbols)
      | While(e, st) ->
        (SWhile(check_bool_expr symbols e, fst (check_stmt symbols st)), symbols)
      | For(e1, e2, e3, st) ->
```

```
          let se1, symbols = check_expr symbols e1 in
          let se3, symbols = check_expr symbols e3 in
          (SFor(se1, check_bool_expr symbols e2, se3, fst (check_stmt symbols st)), symbols)
      | Return e ->
        let (t, e'), symbols = check_expr symbols e in
        if t = func.rtyp then (SReturn (t, e'), symbols)
        else raise (
            Failure ("return gives " ^ fmt_typ t ^ " expected " ^
                     fmt_typ func.rtyp ^ " in " ^ fmt_expr e))
      | VarDecl (((t, id), _) as vd) -> (match t with
        | NoneType -> raise (Failure ("Variable type cannot be none: '" ^ id ^ "'"))
        | _ -> let (svd, sym) = check_var_decl symbols vd in (SVarDecl(svd), sym))

      | VarDeclList var_decls_list ->
        let (s_var_decl_list, new_symbols) = check_var_decl_list symbols var_decls_list
        in
        (SVarDeclList s_var_decl_list, new_symbols)

    in (* body of check_func *)
    { srtyp = func.rtyp;
      sfname = func.fname;
      sformals = func.formals;
      sbody = check_stmt_list symbols func.body
    }
  in
  (List.map check_struct structs, List.map check_func functions)
```

10. Compiler Codebase

Please feel free to go to our public repo in order to pull our code and begin writing your own .viz source code!

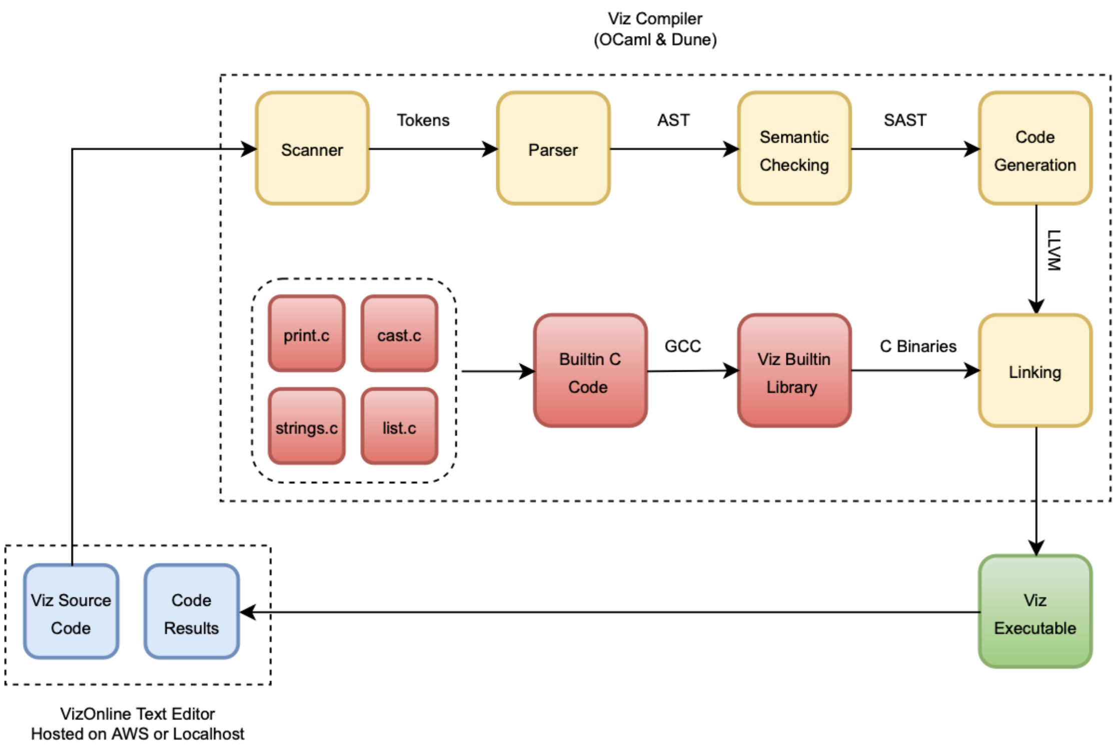
```
https://github.com/4115-viz/viz
```

11. Viz Demo

Please feel free to check out our Viz compiler demo at the link below:

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

12. Architecture Design



13. LRM References

1) https://www.bell-labs.com/usr/dmr/www/cman.pdf
2) https://docs.python.org/3/reference/
3) http://www.cs.columbia.edu/~sedwards/classes/2016/4115-fall/lrms/rusty.pdf
4) https://go.dev/ref/spec
5) https://doc.rust-lang.org/stable/reference/