

Viz: A Beginner's Language Language Reference Manual

Matthew Duran (md3420) | Tester
Jakob Deutsch (jgd2150) | System Architect
Yanhao Li (yl4734) | Tester
Nicholas Wu (nkw2115) | Language guru
Jinsen Wu (jw4157) | Project Manager

1. Introduction

Viz is a general-purpose programming language that allows the visualization of data structures and the highlight of each operation step. It is imperative, statically scoped, statically and weakly typed like C++ 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, object-oriented and abstract data types like array, stack, and tree. For visualization, Viz will generate a styled HTML file for developers to examine each step of operation applied to data structures. Furthermore, Viz will be a helpful tool for beginners through friendly syntax which will allow users to not get bogged down by computer science fundamentals, but rather focus on writing code that executes a given task.

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, ['\t', '\n', '\r', ' ']. But please use whitespace well when writing code for readability.

2.1 Keywords

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

| | | | |
|------|----|------|------|
| func | if | else | elif |
|------|----|------|------|

| | | | |
|-------|----------|---------------|--------|
| for | while | infinite_loop | return |
| break | continue | try | catch |
| raise | link | true | false |
| use | in | step | as |

2.2 Comments

Comments are treated as whitespace. They cannot nest, and cannot appear within literals.

Inline Comments: //

Multiline Comments: begin with /* and end with */

2.3 Literals

A literal is the source code representation of a value of a type, such as a number or string.

| | |
|-------------|-------------------------|
| 15 | Integer literal. |
| 1.42 | Floating-point literal. |
| "Hello" | String literal. |
| true, false | Boolean. |

2.4 Operators

| | |
|------------|-----------------------|
| Arithmetic | +, -, *, /, % |
| Assignment | =, +=, -=, *=, /=, %= |
| Relational | ==, !=, >=, <=, >, < |
| Logical | and, or, not, ? |

2.5 Delimiters

The following tokens serve as delimiters.

| | | | | | |
|---|---|---|---|---|---|
| (|) | [|] | { | } |
| , | : | . | ; | | |

2.6 Identifiers

All identifiers should always start with '@'. I.e. "@variable_name"

let alpha = ['a'-'z' 'A'-'Z']

let digit = ['0'-'9']

let id = @ alpha (alpha | digit | '_')*

3.Type System

3.1 Primitive types

| | |
|---------|--|
| string | An array of ASCII characters, wrapped with double or single quotation marks. |
| int | 4 bytes, signed integer. |
| float | 4 bytes, floating-point number. |
| boolean | 1 byte, true/false value. |
| none | a type with no value. |

3.2 Complex Data Types

Each of these abstract data types (ADT) may only contain the same type within the structure. These five ADTs will be built in and generic, and we will use the following syntax to instantiate an object.

- ADT|data type|
- E.g. queue|string|, array|int|, treeNode|int|

Furthermore, since this is a beginner's language they will not need to worry about garbage collection and can do something like the example below without fear of memory management.

- Array|Array|int|, a hash table, for example.

| | |
|------------|--|
| array | A collection of elements of the same primitive data type. |
| queue | A first-in-first-out collection of same-type elements. |
| stack | A last-in-first-out collection of same-type elements. |
| linkedNode | The smallest unit of a doubly linked list. It has a value field and two reference fields "prev" and "next" that link to the previous node and next node. |
| treeNode | A tree data structure where each node has one parent and at most two children: left child and right child. The smallest unit of a binary tree. It has a value field and a parent and left child and right child. |

3.3 Viz() method for visualizing our ADTs

Viz() will be a built-in library function that will use the data structure object to display its contents. This will be done by writing the fields to an html file that will be generated, and which you could open, for either debugging or for a teaching tool for new students.

```
/*
Each ADT defined as a part of the language will have a viz() method which will
display the inner workings of the data structure in an html file which you can open up
and render in a browser.
```

```
Note: Any user defined objects will of course not have this implemented, unless they
did that themselves.
```

```
*/
```

```
@queue.viz(); // each of these ADTs will have their own html format as part of
@array.viz(); // viz standard library
@stack.viz();
@listHead.viz();
```

// example pictured below

@root: treeNode[int] = 1;

@left: treeNode[int] = 2;

@right: treeNode[int] = 5;

@cur: treeNode[int] = @root; // created pointer instead of new copy

@cur.left = @left; @cur.right = @right;

@left.left = 3; @left.right = 4; // indirectly assign left and right children with the same type will automatically create treeNode with the same type

@right.right = 9;

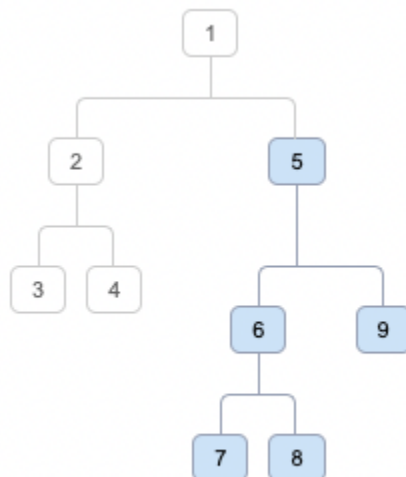
@cur = @right;

@cur.left = 6;

@cur = @cur.left; // cur is now pointed to the left child of previous cur

@cur.left = 7; @cur.right = 8;

@root.viz().highlight(@right); // below styled html will be generated



4. Execution model & style guide

4.1 Execution

To maintain simplicity Viz uses a module system that is broken up into files containing the (.viz) extension. A .viz file may contain classes, functions, and definitions that a user wants to include in a program. This is to maintain flexibility for the user to maintain code without restrictions. Structurally, Viz functions and class definitions are written in code blocks that may be called anywhere in the file following their declaration. Code blocks are self-contained and meant to provide organization to code. A (.viz) program is executed at a top-level with a (main) function, if none is present then code is executed sequentially in the order it is written as a script.

Each code block requires a pair of matching { } delimiters. Within code blocks each statement requires a semicolon and the compiler will throw a “Exception: Missing ;” error otherwise. The two following functions as separate .viz files would both compile and contain the same output to the terminal.

```
func main(@argc: int , @argv: array|string|): None {  
    print("Hello World!");  
    return;  
}
```

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

4.2 Exceptions

Exceptions are thrown upon compilation of a program. To help beginners understand errors, the code block where the error is found with the specific error highlighted is printed to the terminal. Exceptions can be handled using a (try...catch) clause inside a given code block. Viz also makes use of the (raise) statement to raise custom exceptions. If an exception is not handled the compiler will terminate the process and no program will be executed. The following example will throw an error to the terminal when compiling.

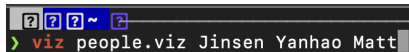
```
func my_test(): None {
```

```
print("A test")
return;
}
```

Exception: Missing a ;
On Line: print("A test")

4.3 Command Line Arguments

Command line arguments can only be passed to the program through the main function. By convention, they are named `@argc` and `@argv` to represent a count and array of contents respectively. To offer flexibility these names are customizable to the user. The following terminal command and code blocks will both output "JinsenYanhaoMatt3" to the terminal for a `people.viz` program.



```
> viz people.viz Jinsen Yanhao Matt
```

```
func main(@argc: int , @argv: array|string|): None {
    print(@argv);
    print(@argc);
    return;
}
```

```
func main(@count: int , @names: array|string|): None {
    print(@names);
    print(@count);
    return;
}
```

4.4 Classes

Classes are supported to serve as a support to Object Oriented Programming. We define classes in a similar manner to code blocks using the { } delimiters, the class keyword and passing in values upon declaration to the () delimiters. The following example creates a person class, and prints “Jake53 ” to the terminal.

```
class person(@name: string , @age: int){  
    name = @name  
    age = @age  
}  
  
@person_1: person = person("Jake",53);  
  
print(@person_1.name);  
print(@person_1.age);
```

5.The Import system

The import system within the Viz language is designed to be as simple and intuitive as possible. In order to import functionality from another file, the user simply declares the imports at the head of the .viz file and binds the imported file path to a name by which it will be referenced from then on. The syntax is as follows:

```
link "../CustomClassFile.viz" as Custom;  
link "../classes/Class1.viz" as Class1;  
link "../classes/utilities/Class1.viz" as Utilities1;
```

It is conventional for users to use capital letters at the start of an import name.

The import system works as follows. The user provides the filepath and filename to the desired file to link. The system then searches through the filesystem for the named file and compiles it and associates the results with the name given by the user. Users must be careful not to include circular references to files in their imported files as this will cause compilation errors such as an infinite loop.

Imported files are then used in the regular flow of a Viz program by calling the class methods using the dot convention. For example, if a user wanted to use the floor() function that was written in the Utilities1 class they would do so as follows:


```
link ./utilities/Class1.viz as Utilities1;

func main(@argc: int , @argv: array|string|): None {
    @floating_point_number: float = 3.67;
    @floor_result: int = Utilities1.floor(@floating_point_number);
    print(@floor_result);
    return;
}
```

6. Simple Statements and Expressions

6.1 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.

```
@car_speed: int = 50;
@email: string = "md3420@columbia.edu";
@are_we_done_yet: bool = false;
@flt: float = 0.1;
```

6.2 variable initialization

When variables are declared they do not need to be formally initialized by the programmer. If they are not initialized then they will be automatically by our compiler.

```
@counter: int; // this will be initialized to 0
@not_done: bool; // this will be initialized to true
@bank_balance: float; // this will be initialized to 0.0
@file_sentence: string; // this will be initialized to ""
```

There are cases when you would want to initialize multiple variables to the same initial value. To support this functionality, we will start with @symbol1, symbol2, ... then followed by the colon(:) and the variable type. The end value at the end of the expression will simply be copied amongst all of the variables. Our users do not need to worry about pointers or memory management.

```
@day_counter, @hour_counter, @min_counter: int = 0;
```

6.3 string literal expressions and formatting.

In this section, we will talk about completely analogous ways to manipulate string expressions for various use cases. We will also support an easier way to format strings, using the “{}” brackets within the string as a placeholder and then immediately following the end of the string use the reserved keyword ‘use’, followed by the variables to replace.

```
@normal_str: string = "Hello I am a normal string";  
  
// This is how we will format our strings, {} brackets are reserved tokens that  
// cannot be used within a string without the “\” character  
@output_str: string = "{} Pandemic began in \{} \{}" use @pandemic, @year;  
// @output_str becomes "Covid Pandemic began in {2022}"
```

6.4 array expressions

Array expression will be written by having 0 or more elements, separated by comma(,), enclosed within the square brackets. Within the array 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.

```
@empty_array: array[int] = [];  
@another_empty_array: array[int]; // this will initialize an empty array with integer  
type  
@int_array: array[int] = [0, 1, 2, 3];  
@dates: array[string] = ["01/01/2022", "02/22/2022"];
```

For readability, and ease of use we will allow the programmer to instantiate a long array of identical elements writing expression * integer.

```
@ten_zeros: array[int] = [0] * 5; // is equivalent to [0, 0, 0, 0, 0]
@ten_hellos: array[string] = ["hello"] * 2; // is equivalent to ["hello", "hello"]
```

6.5 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. We will use a semicolon(;) to determine the end of the expression.

```
@cs_courses: array[string] = [ "Programming Languages & Translators",
                               "Operating Systems",
                               "Advanced Software Engineering",
                               "Analysis of Algorithms" ];
@long_form_math: int = 100000
                  + 1;
```

6.6 boolean expressions

We want to create this expression in the simplest and commonest form. It is straightforward for boolean type. For other types, we will be using the four comparison operators in boolean expressions.

```
@reality: bool = false;
@fantasy: bool = true;
@ok: bool;

@reality; // false
@fantasy; // true
@ok; // false;

@comparison: string = "Boolean Comparison";
@comparison == "Boolean Comparison"; // true
```

```
0 == 1; // false
0 > 0; // false
0 >= 0; // true
0 < 0; // false
0 != 0; // false
```

6.7 Queue and Stack expressions

An easy interpretation of the queue is that it is a first-in-first-out array, where you can only get the first element in the array each time, and you can only push the element to the back of the queue. As for stack, as the name suggested, it is a first-in-last-out array, where you can only get the top element in the array, and the newly pushed array will become the new top element.

```
@q: queue|int|;
@q.push(1); // 1
@q.push(2); // 1 2
@q.size(); // output 2
@q.first(); // output 1
@q.pop(); // 2
@q.first(); // output 2

@s: stack|int|;
@s.push(1); // 1
@s.push(2); // 1 2
@s.first(); // output 2
@s.pop(); // 1
@s.first(); // output 1
@s.push(3); // 1 3
@s.first(); // output 3
```

6.8 Binary Tree and Linked List expressions

We want our users to have an easier time implementing Binary Tree and Linked List and to help them understand how they work better. Therefore, we offered built-in datatypes `treeNode` and `linkedNode`. Users can easily build a tree with `treeNode` and a

linkedList with linkedNode. Each node is fundamentally a pointer to memory location. A treeNode has left and right children which are also treeNodes. A linkedNode has previous and next which are also linkedNodes. By default, the attachments to both types of nodes are none. The type of node must be declared during initialization like other data structures. Each node can only connect to nodes with the same type or None.

```
@root: treeNode[int] = 1; // initialize a treeNode with type integer and value 1
@root.left; // None
@root.parent; // none; root has no parent
@root.left = 0; // initialize a new treeNode as the left child of root node with value 0
@root.left.parent; // 1, left child has a parent with value 1
@right: treeNode[int] = 3;
@root.right = right; // assign treeNode right as the right child of root node
@right.parent; // 1, right has a parent with value 1
@current: treeNode[int] = @root.left; // @current is pointer pointing to the left child of
root node
@current = -1; // update the left child of root node to -1
@root.left; // -1

@head: linkedNode[string] = "Hello"; // initialize a linkedNode with a type string and
value "Hello"
@head.next; // None
@head.next = "World";
@head = @head.next;
@head.previous; // "Hello"
@head; // "World"
```

7. Compound Statements and Expressions

7.1 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.

7.1.1 if/elif/else

```

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 a 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.

7.1.2 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; // initialized to empty 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; // initialized to empty 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*.

7.1.3 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 put these notions to bed.

There will be two types of loops, one that is “ranged” and one that is “collection-iterable.” The ranged loops will have the following style:

```
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

```
for n in start...=end {  
    Block...  
}
```

Here, the start and endpoints are inclusive (thanks to the “=” ending condition), and we will increment by 1 in each iteration. Now we could iterate 10 times by having the endpoints be [start = 1, end = 10]. This allows for more readable code, as opposed to the usual [start = 0, end = 9] which tends to trip up beginners.

The next loop allows us to increment from a starting number to the end, where the endpoint is not inclusive.

```
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...  
}
```

Next up we have the “collection-iterable” loop type which will allow us to iterate through each of the elements of a collection without having to think about starting and ending endpoints. All major languages support this construct and allow for simpler collection passing. The loop construct is as follows:

```
for item in collection { Block... }
```

See below for an example:

```
for number in [2, 7, 8] {  
    Block...  
}
```

The goal for all of these loop constructs is to allow for simpler code generation and to allow for easier debugging. Less time spent debugging off by one error, adds more time for software development.

7.1.4 while loop

While loops are best used in cases where we are unsure of the number of times we would need to iterate. 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 will continue.

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


7.1.5 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...  
}
```

7.1.6 break and continue keywords

Viz will also support `break` and `continue` keywords for more control with the loop flow. Usually, these two keywords would be embedded within control logic within a loop and executed if a condition is met.

When a `break` expression is encountered the innermost loop where it is contained would be exited. We only allow `break` statements to be executed within the body of a loop

```
// this is an error  
If @my_bool == true {  
    break;  
}  
  
// this is legal, and the expected behavior  
Infinite_loop {  
    If @event_bool == true {  
        break;  
    }  
}
```

When a continue expression is encountered, then control would immediately return to the loop condition in which it is enclosed. Continue statements provide value when you have certain conditional checks within a loop execution, and should be used wisely.

We have 3 loop structures defined and each of them handles the continue expression slightly differently. For the for loop, we go back to the expression within the loop control with an incremented counter. For the while loop, we go back to the boolean expression for another conditional check and then proceed according to the truth outcome. Lastly, for the infinite loop, we will return to the top of the infinite_loop, which essentially is always true, and we continue another iteration. Lastly, much like the break statement we cannot call continue from outside a loop body.

```
// this is an error
If @my_bool == true {
    continue;
}

// this is legal, and the expected behavior
Infinite_loop {
    If @event_bool == true {
        continue;
    }
}
```

7.2 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 define functions with incompatible parameter types that can only be enforced at runtime. For beginners, it is best to understand what the function will return, as well as compatible variable types.

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
}
```

8. Viz LRM references

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