Viper 🐍

An amalgamation of all our favorite language quirks.

A hosted copy of this manual may be found here.

Authors:

- Mustafa Eyceoz (me2680)
- Tommy Gomez (tjk2132)
- Trey Gilliland (jlg2266)
- Matthew Ottomano (mro2120)
- Raghav Mecheri (rm3614)

0 Contents 📌

1 Overview 🚁

Viper is a statically-typed imperative programming language that incorporates powerful functionality into a clean syntax. By requiring users to declare the types of functions and variables, Viper benefits from the safety mechanisms and increased efficiency of type checking. It also includes useful features like pattern matching, arrow functions, and an intuitive standard library. See the following sections for a complete introduction to the language.

↩️ Back to Contents 📌

2 Lexical Conventions 📝

2.1 Comments

Viper allows for multi-line comments that begin with an opening forward slash followed by a star (/*) and end with a closing backward slash followed by a star (*\). All content within the bounds of these symbols is ignored.

```
/* Single-line comments anyone? */

/* How about
multi-line?
*/
```

2.2 Identifiers

All user-defined identifiers (variable and function names) must begin with an ASCII letter and can contain any mix of ASCII letters and numbers.

Example valid identifiers:

```
lambda_bamba
pythonCython
RatGhav
V1P3RisTh3b3sT
```

Example invalid identifiers:

```
68vip
--!x
V*x
```

2.3 Reserved Keywords

Any Viper reserved keywords can not be used as user-defined identifiers. A list of reserved Viper keywords include:

```
/* control flow */
if else for while return skip abort panic in has

/* function and types */
func char int float bool nah string dict group

/* operators and literals */
and or is not true false
```

2.4 Scoping

Viper uses a pair of opening and closing curly brackets ({}) to represents a scope of statements within control-flow and function definitions. All statements within the scope must be followed by a semi-colon, but are not required to be on a new line or indentation as previous statements. The core of the control-flow statement following the keyword must be surrounded by parenthesis as well.

Example bracket scope usage:

```
func void foo() {
    print("bar");
}

count = 0
```

```
while (count < 10) {
    if (count % 2 == 0) {
        count += 1;
    }

    count += 1;
}
```

For more information on statements and scoping, see Section 5.

2.5 Literals

Literals are the values that primitive types within Viper take on within the source code.

Literals include:

- boolean
- char
- int
- float
- nah

2.5.1 Char Literals

Char literals represent a single ASCII character and expressed as a letter within single quotes. They also can represent escape sequences and special tokens such as '\t' and '\n'. These individual literals can be combined to make up a String when surrounded by double quotes. These character literals are always assigned to variables of the type *char*.

Examples of char literals:

```
'a'
'+'
'\n'
```

2.5.2 Int Literals

Int literals represent a whole decimal number as an integer and always takes on the *int* type.

Examples of int literals:

```
0
42
-70843
```

2.5.3 Float Literals

A float literal represents a decimal floating point number and always takes on the *float* data type. A float literal consists of a sequence of numbers representing the whole-number part, followed by an ASCII decimal point, followed by a sequence of numbers representing the decimal portion.

Examples of float literals:

```
123.45
-0.007
```

```
3.485
```

`2.5.4` Boolean Literals

Boolean literals are used to indicate the truth value of an expression and are represented by the *bool* data type. The two boolean literals used by Viper are the keywords *true* and *false*.

`2.5.5` String Literals

String literals are a sequence of chars surrounded by double quotes. These literals can be assigned to variables of the type *string*.

`2.5.6` Nah Literals

The nah literal represents a reference to a null value and always takes on the nah type. This literal is represented by *nah* made from ASCII characters.

Examples of string literals:

```
"Stringy123"
"ratghav merch boi"
"H3sKell >>>"
```

`2.5.7` List Literals

All list literals consist of an opening square bracket, a sequence of objects/values all of the same type sepereated by commas, and a closing square bracket. List literals must be assigned to variables of the type *list* wrapping the same type the array literal contains. List literals can contain array list within themselves, leading to multi-dimensional lists.

Examples of valid list literals:

```
[1, 2, 3]
["a", "b", "c"]
[[1], [10]]
```

Examples of invalid list literals:

```
[1, 'a', "3"]
[nah, "beach"]
[1, [5, 9]]
```

↩️ Back to Contents 📌

`3` Data Types 💾

Viper supports the same primitive and higher-order data types as many modern languages. Primitive types are supported natively, while higher-order types are implemented in Viper's Standard Library 📚.

`3.1` Primitive Data Types

The six primitive types supported by Viper are `char`, `int`, `float`, `bool`, `string` and `nah`. The table below summarizes their properties and declarations, with more details in the following sections.

Primitive Type	Size	Description	Declaration/Usage
`char`	2 bytes	Represents single ASCII characters	`char a = 'a';` `char c = 'b' + 1;` `char newline = '\n';`
`int`	8 bytes	Stores signed integer values	`int pos = 12;` `int neg = -980;` `int sum = 4 + 5;`
`float`	8 bytes	Stores signed floating-point numbers	`float pos = 3.2;` `float neg = -29.7;` `float dec = 0.003;` `float whole_num = 2.0;`
`bool`	1 byte	Stores either `true` or `false`	`bool t = true;` `bool f = false;` `bool falsy = t and f;`
`string`	varies	Stores a sequence of `char`s representing a word	`string s = "yeehaw"` `string name = "mro";`
`nah`	1 byte	Viper's `null` value	`int nil = nah;` `char empt = nah;` `return nah;`

3.1.1 `char`

`char` is the type that represents single ASCII characters. In Viper, a `char` is represented as an ASCII character enclosed in single quotes. Special characters, like the newline and tab characters, are defined with an escape backslash (`'\n'` and `'\t'` , respectively). Each `char` behaves like an `int` in that it takes on the decimal value of its assigned ASCII character. Therefore, numerical operations that are valid for integers are also valid for `char` s.

3.1.2 `int`

`int` s represent signed integer values. The minimum value of an `int` is -2^{31}, and the maximum value is 2^{31} - 1. Negative integer values must be defined with a preceding minus (-) symbol, but positive integer values cannot be defined with a preceding plus (+) symbol.

3.1.3 `float`

`float` s represent signed floating-point numbers. To define a `float` at least one digit must precede a decimal point (.), and at least one digit must follow. For example, `.8` and `8.` are invalid, and result in syntax errors. These values are correctly defined as `0.8` and `1.0` , with padding zeroes to ensure that there is a least one digit on each side of the decimal point.

3.1.4 `bool`

`bool` s hold one of the two Boolean values: `true` or `false` . Expressions using the logical `and` , logical `or` , and equality operators are evaluated to `bool` s. For example, the expression `(1 < 2) and ('c' ==`

`'c')` evaluates to a `bool` with value `true`. Additionally, specific values of each primitive type evaluate to certain `bool` values. See the table below for details (note that `nah` always evaluates to `false`).

Primitive Type	`true` values	`false` values
`char`	All `char`s but `'\0'` and `''`	`'\0'` and `''`
`int`	[-2^{31}, -1], [1, 2^{31} - 1]	0
`float`	All `float`s but 0.0	0.0
`bool`	`true`	`false`
`string`	All non-empty `string`s	`""`
`nah`	n/a	`nah`

`3.1.5` `string`

The `string` type of Viper is implented as a `list` of `char`s. `string`s are defined with the standard double quote notation.

```
string name = "Ghav";
```

Internally, a `string` is stored as a sequence of defined chars, followed by the null terminal character `'\0'`. The `string` "rat" is internally `['r', 'a', 't', '\0']`.

`3.1.6` `nah`

`nah` is Viper's `null` value. It can be used to initialize any other data type, and is a valid return value for any function, regardless of the expected return type. Functions with no return value are declared with type `nah`.

`3.2` Higher-Order Data Types

Viper also supports various higher-order data types, including `list`, `string`, `group`, and `dict`. More details can be found in the Standard Library section.

Type	Description	Declaration/Usage

`list`	Ordered lists of any type	`int[0] array; /* Empty list */`
		`float[] scores = [9.7, 8.2];`

`group`	Lightweight structure to hold type-specified collections of data	`(int x, int y) coord = (3, -4);`
		`(string, int) name_id = ("Bon", 4432);`

`dict`	Key-value pairs with random access	`[int: int] pos; /* Empty */`
		`[string: (string, int)] items = [ "milk": ["dairy", 5], "apple": ["fruit", 3] ];`

`3.2.1` `list`

Like many languages, Viper supports random access `list`s of any data type. A `list` is defined by specifying a non-`list` data type, followed by at least one set of square brackets (`[]`). Multi-dimesional lists can be created with additional sets of square brackets. `list`s have fixed types, and can be created in the following ways:

```
/* 0. Empty lists: */
int[] dust;
float[][] edges;

/* 1. Size given implicitly by the length of the list literal: */
string[] cheese = ["chewy", "bendy", "wiggly"];
bool[][] outcomes = [[false, false], [false, true]];

/* 2. Size given implicityly by copy construction */
string[] glizzy = cheese;
```

`list`s can be accessed and modified directly by specifying indices in square brackets. Indices are integers in the range [0, length - 1). Attempting to access or modify an index outside this range throw errors.

```
int[] nums = [4, 0, 8];
nums[2] = nums[1];    /* Sets the last element to 0 */
nums[1] = 2;          /* Sets the middle element to 2 */

int error = nums[3]; /* Throws an error */
```

3.2.2 `group`

A `group` is an immutable type-specified collection of data. Any number of types can be specified, but their order is fixed. `group`s are declared as lists and are auto-inferred:

```
(string, int) order = ["Chicken Katsu", 17];
(float[2], string, bool) = [[0.1, 2.1], "boo", false];
```

Elements of `group`s can be accessed by passing an index into a set of parentheses. Like `list` indices, `group` indices are zero-indexed and must be in the range [0, length - 1).

```
(int, int) paws = [3, -2];
int x = paws[0]; /* Sets x to 3 */
```

Elements of `groups` can also be deconstructed into variables of their base type. These variables are then accessible within the same scope as the deconstruction.

```
(int, int, int) t = [240, 130, 202];
(int r, int g, int b) = t;
print(r);
```

3.2.3 `dict`

A `dict` is a mapping of key-value pairs. The types of both keys and values must be specified at creation, and keys must be unique. `dict` literals are defined with square brackets (`[]`), in which a colon (`:`) separates key and values, and commas separate key-value pairs.

```
[int: string] map = [1: "one", 2: "two", 3: "three"]
```

`dict`s can be accessed and modified similarly to `list`s. Instead of using indices, however, `dict`s only accept key values. Attempting to use a key of an unexpected type, or using a key with no mapped value will result in an error.

```
/* Note: nested dicts */
[char: [string: int]] wordmap = [
        'a': ["add": 2, "and": 3],
        'b': ["blob": 1, "bap": 14],
        'd': ["doink": 1]];
[string: int] b_words = wordmap['b']; /* Retrieves ["blob": 1, "bap": 14] */
b_words["bing"] = 4; /* Adds key-value pair ["bing": 4] to b_words */
int no_no = b_words["balloon"]; /* Error: b_words has no key "balloon" */
int bad_idea = wordmap["a"]; /* Error: key type is char, not string */
```

Empty dicts can also be declared.

```
[string: int] rat = [];
```

↩️ Back to Contents 📌

4 Type System 🗃️

Viper utilizes a static typing system to benefit from the provided type safety and optimizations of a staticly typed compiled language.

4.1 Explicit Types

Viper requires explicit user-specified types for variable declarations, function parameters, control flow, and return types in function definitions. An explicit type is required when new variables, placeholders, and parameters are created and need a type to be referenced against.

Variable intialization and assignment:

```
string wow;
char x = 'y';
wow = "doge"; /* Note: not required with assignment when type of identifier has been
initialized */
```

Function definitions:

```
int func incrementer(int x) {
    x += 1;
}

int func sum (int c) => c + c;
```

Control-flow:

```
for (int i = 0; i <= 10; i++) {
    print(i);
}

for (int num: nums) {
    print(num);
}
```

4.2 Explicit Type Conversions

Viper utilizes casting functions available in the standard library to convert between types as needed. For example, casting up from an int to a float is a simple as wrapping an integer value expression in the *float* function.

Explicit type conversion functions include:

- str(x) - converts x to a string
- float(x) - converts x to a float
- int(x) - converts x to an int
- chr(x) - converts x to a char

Examples of using explicit type conversions:

```
int x = 1;
/* converts 1 into '1' */
char y = chr(x);

int x = 2;
int y = 5;
int z = x+y; /* 7 */
/* "257" */
string xyz = str(x) + str(y) + str(z);
```

Note: See Section 6 for more details on explicit type casting functions.

4.3 Implicit Type Conversions

As Viper is statically-typed, we can rely on user-specified types to infer the desired type of an output and convert values accordingly. This comes in handy for common programming tasks such as math operations and string concatenation.

Examples of implicit type conversion:

```
/* when 2 integers are divided, Viper's
type system is often able to infer which type the
user would like to return from the result
from hints such as the variable type. */
int x = 2/5; /* 0 */
float x = 2/5; /* .166666... */

/* when a series of concatenations
occurs starting with a string,
all following operands will be
converted to strings and then
concatenated. */
string num1 = "17";
```

```
int num2 = 38;
print(num1 + num2); /* "1738" */
```

When making an implicit type conversion, the Viper type system attempts to make an conversion based on the context of the values and types around it. If a conversion inference can not be made, Viper assumes the type that leads to the least loss of precision.

↩️ Back to Contents 📌

5 Statements 🗣️

Viper programs are composed of a list of statements. Statements are selector statements, iterator statements, jump statements, and function statements.

5.1 Selector Statements

Selector Statements are involved with Viper's control flow. These statements are the conditionals that Viper uses to control the flow of a program. These statements include the if statement and the if/elif/else statement.

5.1.1 If Statement

The if statement takes in a boolean expression within parentheses and runs the statements within its scope if the boolean expression returns true.

5.1.2 If / Else if / Else Statement

The if statement has optional statements that can come after it such as elif and else. Else if will be run if the previous if statement's boolean expression was false. An else if statement is like an if statement in that it takes in a boolean expression in parentheses and if the boolean expression returns a value of true, then the statements within its scope will be run. There can be infinitely many elif statements after an if statement. The else statement must come after the if and all else if statements, if any. The else statement will run the statements inside its scope if all the if statements and elif statements have a boolean expression that returns false.

```
if (a == b){
    print(a);
}
else if (a > b){
    print(b);
}
else{
    print("something is wrong");
}
```

If statements also can use a special keyword "has" to check if an element is in an array. The "has" keyword returns true if the element is in the array and false otherwise. The syntax is written by typing the name of the array, followed by "has" followed by the element.

```
if (arr has 42){
  print(true);
}
else{
  print(false);
}
```

5.2 Iterator Statements

Iterator Statements are involved with Viper's ability to loop through statements. These statements compose for loops and while loops.

5.2.1 For Statement

A for statement takes in an argument in the form of (assignment; condition; iterator), followed by a list of statements within its scope. The assignment creates a variable and initializes it to a given number. The condition is a boolean expression; if it returns true, the list of statements within the for statement's scope is run. The iterator changes the value of the variable in the assignment. Then the condition is checked with the new value and if it returns true, the statements are run again, otherwise the statements are not run again.

```
for (int i = 0; i < sizeof(arr); i++){
    print(arr[i]);
}
```

A for statement can take a second form as well. The second form of a for statement is an identifer, followed by the keyword in, followed by an object that is iterable. This statement will iterate over the values in the iterable object, using the identifier for each value, and run the statements in its scope. Once there is no elements left in the iterable object, the for statement will stop.

```
for (int element in arr) {
    print(element);
}
```

5.2.2 While Statement

A while statement takes in a boolean expression. If the boolean expression returns a value of true, the statements within its scope are run. After all statements are run, the boolean expression is evaluated again; if true then statements are run again, otherwise, the while statement is done. This process repeats until the boolean expression returns a value of false.

```
while (condition){
    print("chilling");
}
```

5.3 Jump Statements

Jump statements are statements located within the scope of an iterator statement which dictates how to proceed within the iterator statement.

5.3.1 Skip Statement

The skip statement appears in for statements and while statements. When the program encounters this statement, it will ignore any statements left in the iterator statement and go back to the beginning of the iterator statement.

```
for (int element in arr){
    if (element == 2) {
        print("I'm going to skip the remaining statements");
    }
    skip;
    print("This element isn't a 2");
}
```

5.3.2 Abort Statement

The abort statement appears in for statements and while statements. When the program encounters this statement, it will ignore any statements left in the iterator statement and leave the iterator statement, proceeding with other statements within the code, if any.

```
for (int element in arr){
    if (element == 2){
        print("found it");
    }
    abort;
}
```

5.4 Function Statement

Functions take input and may return output. Functions take the form of "returnType func functionName(parameter1, parameter2, ...)" The returnType is the type of the output that must be returned from the function. The func, is literally the word func. The functionName is the name of the function which must use the same convention as variables in Viper. The (parameter1, parameter2, ...), is the input of the function where each parameter is a variable. If a function is called, the statements in its scope will run, using any parameters given to the function and then returning the value of type, returnType, using the keyword return. Functions are called by writing the function name followed by a parantheses of parameters, if any.

```
nah func foo(){
    print("Hello World!");
}
foo();
```

5.4.1 Arrow Functions

Similar to arrow functions in Javascript, or Python lambda functions, users are able to define functions with arrow functions. Users are required to specify the type of the arrow function's return value and parameters. The syntax is as follows:

```
<ret_type> func <name> (<param_type> param1, ..., <param_type> paramN)
    => expression output;
```

Note that even with zero parameters or one parameter, the () are still necessary.

Example Function Calls:

```
int func y(int x, int y, int func z) {
    return z(x + y);
}
```

```
int func times (int a, int b) => a * b;
y(10, 20, times);
```

↩️ Back to Contents 📌

6 Expressions 🖥️

Expressions in Viper yield the recipe for evaluation. Expressions can be any data type in its simplest form and it can include operators in more complex forms. These include simple arithmetic expressions which yield a float or integer type, or boolean expressions which yield a true or false when evaluated.

6.1 Truth-Value Expressions

Truth-Value expressions in Viper are boolean expressions. They can include logical operators and when evaluated, must return a value of type bool.

6.2 Guard Expressions

Guard expressions are an alternative way of using conditional statements. When assigning a variable, Viper uses the symbol "??" to indicate the start of a guard expression. Each subsequent statement uses a "|", except the first one and last one, followed by a boolean expression, a ":", and then a value which fits the variable data type. If the boolean expression returns a value of true, then the expression to the right of the symbol ":" is used for the value of the variable. If the boolean expression is false, the program runs the next statement following the next symbol "|". The last statement in a guard expression contains a "??" followed by a value consistent with the data type for the variable. The first statement has neither a "|" nor a "??". This can be thought of as a combination of if, elif and else statements for assigning a variable.

```
int x = ??
4 == 4 : 42;
| 5 == 3 : 24;
?? 0;
print(x);
```

stdout:

```
42
```

↩️ Back to Contents 📌

7 Operators ➗

Operators are used on values to change them. This leads to interesting and complex expressions which can be useful. The different kinds of operators are Unary, Binary, Comparative, Logical and Variable.

7.1 Unary Operators

Unary operators act on only one value. These include the not operator, the increment operator and the decrement operator.

7.1.1 The NOT Operator

The NOT operator is given the symbol "!". When placed to the left of a bool, the value of the bool is flipped. If the value was true it is now false, vice versa.

```
bool example = true;
print(!example);
```

stdout:

```
false
```

`7.1.2` The Increment Operator

The increment operator is given the symbol "++". When placed to the right of an integer, the value of the integer is incremented by one.

```
int example = 0;
print(example++);
```

stdout:

```
1
```

`7.1.3` The Decrement Operator

The decrement operator is given the symbol "--". When placed to the right of an integer, the value of the integer is decremented by one.

```
int example = 0;
print(example--);
```

stdout:

```
-1
```

`7.2` Binary Operators

Binary operators act on two values. These include the addition operator, the subtraction operator, the multiplicative operator, the division operator, and the modulus operator.

`7.2.1` The Addition Operator

The addition operator is given the symbol, "+". It acts like addition in mathematics, i.e. it is written in between two values which result in the sum of the two values.

```
int example1 = 1;
int example2 = 2;
print(example1 + example2);
```

stdout:

```
3
```

`7.2.2` The Subtraction Operator

The subtraction operator is given the symbol, "-". It acts like subtraction in mathematics, i.e. it is written in between two values which result in the difference of the two values.

```
int example1 = 1;
int example2 = 2;
print(example1 - example2);
```

stdout:

```
-1
```

7.2.3 The Multiplicative Operator

The multiplicative operator is given the symbol, "*". It acts like multiplication in mathematics, i.e. it is written in between two values which result in the product of the two values.

```
int example1 = 1;
int example2 = 2;
print(example1 * example2);
```

stdout:

```
2
```

7.2.4 The Division Operator

The division operator is given the symbol, "/". It acts like division in mathematics, i.e. it is written in between two values which result in the quotient of the two values.

```
int example1 = 1;
int example2 = 2;
print(example1 / example2);
```

stdout:

```
0.5
```

7.2.5 The Modulus Operator

The modulus operator is given the symbol, "%". It acts like modulus in mathematics, i.e. it is written in between two values which result in the remainder of the two values when divided.

```
int example1 = 4;
int example2 = 2;
print(example1 % example2);
```

stdout:

```
0
```

7.3 Comparative Operators

Comparative Operators compare two values and returns a bool.

`7.3.1` The Greater Than Operator

The greater than operator is given the symbol, ">". When written in between two values, it returns false if the first value is less than or equal to the second value and returns true if the first value is greater than the second value.

```
int example1 = 2;
int example2 = 2;
print(example1 > example2);
```

stdout:

```
false
```

`7.3.2` The Greater Than Or Equal To Operator

The greater than or equal to operator is given the symbol, ">=". When written in between two values, it returns false if the first value is less than the second value and returns true if the first value is greater than or equal to the second value.

```
int example1 = 2;
int example2 = 2;
print(example1 >= example2);
```

stdout:

```
true
```

`7.3.3` The Less Than Operator

The less than operator is given the symbol, "<". When written in between two values, it returns true if the first value is less than the second value and returns false if the first value is greater than or equal to the second value.

```
int example1 = 2;
int example2 = 2;
print(example1 < example2);
```

stdout:

```
false
```

`7.3.4` The Less Than Or Equal To Operator

The less than or equal to operator is given the symbol, "<=". When written in between two values, it returns true if the first value is less than or equal to the second value and returns false if the first value is greater than the second value.

```
int example1 = 2;
int example2 = 2;
print(example1 <= example2);
```

stdout:

```
true
```

7.3.5 The Equals Operator

The equals operator is given the symbol, "==". When written in between two values, it returns true if the first value is equal to the second value and returns false if the first value is not equal to the second value.

```
int example1 = 2;
int example2 = 2;
print(example1 == example2);
```

stdout:

```
true
```

7.3.6 The Not Equals Operator

The not equals operator is given the symbol, "!=". When written in between two values, it returns true if the first value is not equal to the second value and returns false if the first value is equal to the second value.

```
int example1 = 2;
int example2 = 2;
print(example1 != example2);
```

stdout:

```
false
```

7.4 Logical Operators

The logical operators take in two bool values and returns a bool value. These operators include the AND operator and the OR operator. The HAS operator, which takes in an object containing elements and an element, returning a bool value.

7.4.1 The AND Operator

The AND operator is given the symbol, "&&". When written in between two bool values, it returns true if both values are true and false otherwise.

```
bool example1 = true;
bool example2 = false;
print((example1 && example2));
```

stdout:

```
false
```

7.4.2 The OR Operator

The OR operator is given the symbol, "||". When written in between two bool values, it returns false if both values are false and true otherwise.

```
bool example1 = true;
bool example2 = false;
print((example1 || example2));
```

stdout:

```
true
```

7.4.3 The HAS Operator

The HAS operator is given the symbol, "has". When written in between an object of elements and an element, it returns true if the element exists in the object of elements and false otherwise.

```
if (arr has 42){
  print(true);
}
else{
  print(false);
}
```

7.5 Variable Operators

Variable operators act on a variable and an integer. These include +=, -=, *=, and /=.

7.5.1 The += Operator

The += operator is written in between a variable on the left hand side and an integer on the right hand side. The integer value on the right hand side is added to the variable value, which is updated as the new value for the variable.

```
int example1 = 1;
example1 += 1;
print(example1);
```

stdout:

```
2
```

7.5.2 The -= Operator

The -= operator is written in between a variable on the left hand side and an integer on the right hand side. The integer value on the right hand side is subtracted from the variable value, which is updated as the new value for the variable.

```
int example1 = 1;
example1 -= 1;
print(example1);
```

stdout:

```
0
```

7.5.3 The *= Operator

The *= operator is written in between a variable on the left hand side and an integer on the right hand side. The integer value on the right hand side is multiplied by the variable value, which is updated as the new value for the variable.

```
int example1 = 1;
example1 *= 1;
print(example1);
```

stdout:

```
1
```

7.5.4 The /= Operator

The /= operator is written in between a variable on the left hand side and an integer on the right hand side. The integer value on the right hand side divides the variable value, which is updated as the new value for the variable.

```
int example1 = 1;
example1 /= 1;
print(example1);
```

stdout:

```
1
```

7.5.5 The = Operator

The = operator is written between a variable name on the left hand side and a value on the right hand side. The value on the right hand side is assigned as the value for the variable on the left hand side. If the variable exists already, the value of the variable is overwritten, otherwise a new variable is created.

```
int example1 = 1;
print(example1);
```

stdout:

```
1
```

7.5.6 The Ternary Operator

The Ternary Operator is given the symbol "?". This operator provides a short hand for an if-else statement and saves the result in a variable. When using the ternary operator in assigning a variable, Viper expects a boolean expression followed by the ternary operator "?". After the ternary operator, a value that matches the type of the variable being assigned is expected, followed by a ":" and another value that matches the type of the variable being assigned. If the boolean expression returns a truth value of true, then the first value is assigned to the variable, otherwise the second value is assigned.

```
int x = 5 < 10 ? 42 : 0;
print(x);
```

stdout:

```
42
```

`7.6` Precedence of Operators

The precedence of operators is important for determining how to write programs in Viper. It is important to note that any expression within parentheses has the highest precedence.

`7.6.1` Precedence of Unary Operators

Unary operators receive the highest precedence, second to parentheses.

`7.6.2` Precedence of Binary Operators

The multiplicative operator, division operator, and modulus operator are left associative and have a higher precedence than the addition operator and the subtraction operator. The addition and subtraction operator are also left associative.

`7.6.3` Precedence of Comparative Operators

The >, >=, <, <= operators are given higher precedence than the != and == operators.

`7.6.4` Precedence of Logical Operators

The and operator is given higher precedence than the or operator.

`7.6.5` Precedence of Variable Operators

Variable operators are given a lower precedence than binary operators and are right associative.

↩️ Back to Contents 📌

`8` Scope 👀

Viper uses curly braces to define scope. For example, a for loop can be established in a number of different ways:

```
for string elem in list {
    print(elem);
}

/* Is the same as: */

for string elem in list
{
    print(elem);
}

/* Is the same as: */

for string elem in list
{ print(elem); }
```

This will function in the same manner as expected with function definitions, conditionals, etc.

↩️ Back to Contents 📌

9 Standard Library 📚

Viper's standard library includes methods and functionalities that are used in nearly every program. This is to balance the tediousness of requiring numerous lines of imports and keeping compilation quick and program bloat low.

9.1 Math Functions

Viper provides built-in methods for common arithmetic operations.

9.1.1 `sqrt()`

`sqrt()` returns the square root of the given `int`, `float`, or `char` as a `float`. If a `char` is given, the decimal value of its ASCII symbol is used.

```
float four = sqrt(16); /* Returns 4.0 */
float two = sqrt(four); /* Returns 2.0 */
float eight = sqrt('@'); /* Returns 8.0 */
```

9.1.2 `pow()`

`pow()` can be polymorphically used into two ways. If only one `int`, `float`, or `char` input is given, it returns the square (x^2) of that input. If two `int`, `float`, or `char` inputs are given, it returns the `float` result of raising the first input to the power of the second. If `char` s are given, the decimal values of their ASCII symbols are used.

```
float one_four_four = pow(12); /* Returns 144.0 */
float a_milly = pow(10.0, 6); /* Returns 1000000.0 */
```

9.1.3 `floor()`

`floor()` takes a `float` input and returns the `int` result of truncating the `float` 's decimal components.

```
int zero = floor(0.999); /* Returns 0 */
int whole_num = floor(72.0); /* Returns 72 */
```

9.1.4 `ceil()`

`ceil()` does the opposite of `floor()`. It takes a `float` input and returns the closest `int` greater than or equal to the given value.

```
int five = ceil(4.1); /* Returns 5 */
int four = ceil(4.0); /* Returns 4 */
```

9.1.5 `round()`

`round()` takes a `float` input and returns the closest `int` to the given value. Values of ending in .5 always round to the next greatest `int`.

```
int three = round(3.2); /* Returns 3 */
int also_three = round(3.3); /* Returns 3 */
int neg_three = round(-3.5); /* Returns -3 */
```

9.1.6 `min()`

`min()` takes either two `float` s, two `int` s, or two `char` s as input and returns the smallest value between the two. If `char` s are given, the decimal values of their ASCII symbols are used. The function is overloaded, so the return type is the same as the input type.

```
int negative1 = min(-1, 1); /* Returns -1 */
float gpa = min(5.7, 4.0); /* Returns 4.0 */
char a_char = min('b', 'a'); /* Returns 'a' */
```

9.1.7 `max()`

`max()` takes either two `float` s, two `int` s, or two `char` s as input and returns the largest value between the two. If `char` s are given, the decimal values of their ASCII symbols are used. The function is overloaded, so the return type is the same as the input type.

```
int positive1 = max(-1, 1); /* Returns 1 */
float big = max(8.78, 9.9); /* Returns 9.9 */
char e_char = max('e', 'a'); /* Returns 'e' */
```

9.1.8 `trunc()`

`trunc()` takes a `float` and `int` as input, and returns the `float` truncated to the number of decimal points specified by the `int` . The `int` must be greater than zero.

```
float whee = trunc(0.123456789, 3); /* Returns 0.123 */
float almost_whole_num = trunc(whee, 1); /* Returns 0.1 */
float bad_bad_bad = trunc(0.99, 0); /* Throws an error */
```

9.2 Primitive Type Casting Functions

Viper's standard library provides methods for casting between types for ease of use and readability. Type casting functions include:

9.2.1 `char()`

`char()` converts to `char` s. The input can be an `int` in range [0, 127], for which the output is the `char` corresponding to the ASCII value of the `int` . The input can also be a `string` , for which the output is the `char` value of the first character in the `string` . For empty strings, `char()` returns an empty `char` (`''`). Passing any other types or `nah` to `char()` results in an error.

```
char int_chr = char(36); /* Returns '$' */
char str_char = char(str(true)); /* Returns 't' */
char no_dont_do_it = char(33.4); /* Throws an error */
```

9.2.2 `int()`

`int()` casts certain types to integer values. Given a `char`, `int()` returns the decimal value of the `char`'s ASCII code. Given a `float`, `int()` returns the result of truncating all decimal components. Given a `bool`, `int()` returns 0 for values of `false`, and 1 for values of `true`. Passing any other types or `nah` to `int()` results in an error.

```
int chr_int = int('R'); /* Returns 82 */
int str_int = int(7.999); /* Returns 7 */
int zero = int(false); /* Returns 0 */
int one = int(true); /* Returns 1 */
```

9.2.3 `float()`

`float()` casts `int`s and `char`s to float values. Given an `int`, `float()` returns the `float` equivalent of that `int`, appending a decimal point and a single 0. Given a `char`, `float` does the same thing with the decimal value of the `char`'s ASCII code. Passing any other types or `nah` to `float()` results in an error.

```
float int_float = float(333); /* Returns 333.0 */
float char_float = float('&'); /* Returns 38.0 */
float noooooo = float("if you smoke"); /* Returns an error */
```

9.2.4 `bool()`

`bool()` converts any data type to either `true` or `false`, depending on the value. It's called implicitly when using a non-boolean type in a boolean expression For example, the following code implicitly calls `bool()` on `x`:

```
int x = 0;
if (x) {
    print("Yes");
}
```

See the below table for details on what values `bool()` maps to true and result in `true` and what values result in `false`:

Type	`true` values	`false` values
`char`	All `char`s but `'\0'` and `''`	`'\0'` and `''`
`int`	$[-2^{31}, -1], [1, 2^{31} - 1]$	0
`float`	All `float`s but 0.0	0.0
`bool`	`true`	`false`
`string`	All non-empty `string`s	`""`
`nah`	n/a	`nah`

9.2.5 `str()`

`str()` converts any type to a `string`, which is useful for printing. See the below examples for type-specific details.

```
string char_str = str('z'); /* Returns "z" */
string int_str = str(30); /* Returns "30" */
string float_str = str(34.5); /* Returns "34.5" */
string bool_str = str(false); /* Returns "false" */
```

9.3 Miscellaneous Functions

These functions are unclassified, and are useful in a variety of situation.

9.3.1 `print()`

`print()` prints the given `string` to a new line of standard output. It throws errors if it encounters a type other than `string`, and prints an empty new line if no inputs are given. To get around this, use the str() method with `string` concatenation (+).

```
print("Hola Mundo");
print();
print(str('a'));
print("I have " + str(388) + " bananas");
float nose = -0.3;
print(str(true) + str(nose));

/* print(4999); Uncommenting this line throws an error */
```

stdout:

```
Hola Mundo

a
I have 388 bananas
true-0.3
```

9.3.2 `len()`

The `len()` function is an all-purpose method that returns the `int` size of `string`s, `list`s, `group`s, and `dict`s. For `string`s, `len()` returns the number of characters in the `string` (excluding the null terminator at the end of its underlying `list` of `char`s). For `list`s, `len()` returns the number of elements in the `list`. For `group`s, `len()` returns the number of elements in the `group`, and for `dict`s, `len()` returns the number of key-value pairs.

```
int str_length = len("Nice sock!\n\t"); /* str_length = 12 */
int list_length = len([3, 1, 6, 76]); /* list_length = 4 */
(int, float) groupie = (8, 8.8);
int group_length = len(groupie); /* group_length = 2 */
int dict_length = len(["word": 3,
                       "knees": 5,
                       "port": 90]); /* dict_length = 3 */
```

9.4 Lists

List functionality is provided through the standard library. Lists are mutable, static sequences of a single data type with fixed sizes. Lists are accessed and modified with square brackets ([]). See Higher-Order Data Types: lists for more details on instantiation and modification. The following sections describe additional list-specific operations implemented in the list api. These operations are all called on instances of lists, and thus take the form `list.operation(parameters)`.

9.4.1 `append()`

`append()` adds an input element to the end of a specified list. Because lists have fixed sizes, the original list remains unmodified, and `append()` returns a new list with the input element attached. The type of the input to `append()` must match the type of the list.

```
int[] channels = [31, 44, 21];
int[] new_channels = channels.append(54);
/* new_channels contains: [31, 44, 21, 54] */
```

9.4.2 `prepend()`

`prepend()` works in the same way as `append()`, but it adds the input element to the front of the list.

```
string[] cities = ["NEWY", "BOST", "MIAM"];
string[] more_cities = cities.prepend("ATLA");
/* more_cities contains: ["ATLA", "NEWY", "BOST", "MIAM"] */
```

9.4.3 `remove()`

`remove()` takes in an `int` and removes the element at the index specified by the `int`. If the list has no such index, an error is thrown. Because lists have fixed sizes, the original list remains unmodified, and `remove()` returns a new list with the specified input element removed.

```
char[] notes = ['a', 'c', 'd', 'c'];
char[] less_notes = notes.remove(2);
/* less_notes contains: ['a', 'c', 'c'] */
```

9.4.4 `join()`

`join()` takes another list as input and appends it to the list that `join()` is called on. Because lists have fixed sizes, the original lists are unmodified, and `join()` returns a new list with the elements of the second list appended to the elements of the first. The type of both lists must match, or an error is thrown.

```
float[] class1 = [0.8, 0.8, 0.9];
float[] class2 = [0.7, 0.75, 1.0];
float[] all = class1.join(class2);
/* all contains: [0.8, 0.8, 0.9, 0.7, 0.75, 1.0] */
```

9.4.5 `sub()`

`sub()` takes two `int` s as input, and returns the sublists that fall between those two indices of the list. The element at the start index is included in the sublist, but the element at the end index is not. If the first index is out of the range of the list's indices, an error is thrown. If the second index is less than 0 or greater than the size of the list, an error is throw. Note that the second index is permitted to be one greater than the index of the list's last element, because the element at the second input is not included in the sublist. `sub()` returns a brand new list and leaves the original unmodified.

```
int[] nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
int[] less_than_5 = nums.sub(0, 5);
int[] middle_3 = nums.sub(4, 7);
/* less_than_5 contains: [0, 1, 2, 3, 4] */
/* middle_3 contains: [4, 5, 6]
```

`9.5` Groups

Check back soon for the `group`s API.

`9.6` Dicts

Check back soon for the `dict`s API.

↩️ Back to Contents 📌

`10` Sample Code 🧩

Example programs written in Viper below.

`10.1` Fizzbuzz examples:

```
/* standard fizzbuzz
for-loop solution */
for (int i = 1; i <= 100; i++) {
   if (i % 15 == 0) {
       print("fizzbuzz");
   } else if (i % 3 == 0) {
       print("fuzz");
   } else if (i % 5 == 0) {
       print("buzz");
   } else {
       print(i);
   }
}

/* fizzbuzz for-loop with nested ternary operator
valid, but overly complex solution */
for (int i = 0; i <= 100; i++) {
    (i % 15 == 0)
        ? print("fizzbuzz")
        : (i % 3 == 0)
            ? print("fizz")
            : (i % 5 == 0)
                ? print("buzz")
                : print(i);
}
```

```
/* fizzbuzz for-loop with pattern-matching
valid, short, and easily comprehensible solution */
for (int i = 0; i <= 100; i++) {
    string output = ??
        i % 15 == 0 : "fizzbuzz"
        | i % 3 == 0 : "fizz"
        | i % 5 == 0 : "buzz"
        ?? i;

    print(output);
}
```

10.2 Int list sum examples:

```
/* Printing an average of a list
of ints, (almost) C-style */
int[] nums = [1, 2, 3, 4];
int sum = 0;

for(int i = 0; i < len(nums); i++) {
    sum = sum + nums[i];
}

float avg = sum/len(nums);
print(avg);

/* Printing an average of a list
of ints using Viper conventions */
int[] nums = [1, 2, 3, 4];
int sum = 0;

for (num in nums) {
    sum += num;
}

float avg = sum/len(nums);
print(avg);

/* Printing an average of a list
of ints using Viper standard library */
int[] nums = [1,2,3,4];
float avg = sum(nums)/len(nums);
print(avg);
```

10.3 GCD Function

```
int func recursiveGCD(int a, int b) {

    int func conditional (int x, int y) =>
```

```
        x == 0 ? y : y == 0 ? x : nah;

    int func swappedGCD (int x, int y) =>
        x > y ? recursiveGCD(x-y, y) : recursiveGCD(x, y-x);

    int check = conditional(a, b);

    return check == nah ? swappedGCD(a, b) : check;
}
```

↩️ Back to Contents 📌