Viper

An amalgamation of all our favourite language quirks.

By:

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

Overview

Viper is a statically typed imperative programming language with similar syntax to Python and the safety mechanisms (an increased efficiency) of type checking. Viper forces the user to declare types of functions and variables, as one would do for C, yet in an easy to read and write syntax that mimics the simplicity of Python.

Our goals for Viper are:

- Python-styled syntax
- Types and Type Checking
- Choice of how to incorporate scope (whitespace or brackets) – get rid of Python's tab/spaces issues
- Arrow Functions/Lambdas

Note: The global scope for a Viper program is assumed to comprise the main function, unless one already exists.

What sort of programs would Viper be useful for?

Viper is perfect for almost all common scripting tasks generally associated with languages like Python or Javascript. It is still read top-down with the top-level acting as the default

“main” function, but there are huge efficiency gains from Viper being statically-typed and compiled rather than interpreted. Additionally, we combine favorite features of both languages, as well as flexible syntax options and many forms of efficient syntactic sugar to ensure that not only will the code run quickly, but writing it will also be feel fast and natural. We are aiming to optimize both the speed and experience from thought to output.

Basic Language Details

Data Types and Operations

The standard data types in Viper are integers, floats, booleans, and characters.
Strings are simply arrays of characters.
Viper also includes a null data type, which is defined with the keyword ‘“nah”’.
The primitive data structure which all other data structures will be built off is the array.

Data Type	Description	Operations	Examples
char	A 1 byte character	=, ==, !=, +, ++, –, <, >, =<, >=	a + b a >= b a <= b
int	A 8 byte number	=, ==, !=, +, -, *, /, %, ++, –, +=, -=, <, >, =<, >=	a = 1 a > b a == b
float	An 8 byte decimal number	=, ==, !=, +, -, *, /, %, ++, –, +=, -=, <, >, =<, >=	a = 1 a > b a == b
bool	A 1 byte boolean value	=, ==, !=, !, &&	a == b a != b !(a == b) (a && b)

Data Type	Description	Operations	Examples
nah	A 1 byte none type	=, ==, !=	a == b a != b

The standard library will consist of data structures such as stacks, queues, hash maps, etc.
Viper will use imperative-style control-flow mechanisms such as the for loop and while loop.
Viper will also use if/else/elif statements.
Viper will be able to perform addition, subtraction, multiplication, division, compare (greater than, less than, equals), modulus, powers, concatenation, and increment/decrement. We will also have arrays and tuples of primitives. Just like in OCaml, tuples can also contain elements of multiple datatypes.
An array of `char` types would constitute a string, and we plan to implement a `string` class in our standard library.

Keywords

Keyword	Usage
char	Declares a character
int	Declares an integer
float	Declares a floating-point number
bool	Declares a boolean
nah	Declares our equivalent of a nulltype
panic	Throws an exception
func	Defines a function
return	Specifies the return value of a function
abort	Our equivalent of a break statement

Keyword	Usage
skip	Skips the loop iteration - equivalent of continue
for/while	Defines a for or while loop, respectively
if/else/elif	Controls the flow of if, else, and elif statements
in	Specifies direct, index-free iteration
true	true boolean value
false	false boolean value

Control Flow

Control flow mechanisms resemble for/while loops in either Python, or C:

```
for int element in arr:
    print(element)
```

```
for (int i = 0; i<sizeof(arr); i++){  # More on indentation vs explicit scopin
    print(arr[i]);
}
```

```
while (condition):
    print("chilling")
```

Viper uses if/else/elif conditionals like Python:

```
if a == b:
    print(a)
elif a > b:
    print(b)
else:
    print("something is wrong")
```

The abort keyword is the equivalent of break in Python; it stops the loop:

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

The skip keyword functions much like continue in Python; it rejects all the remaining statements in the loop and returns the control back to the top of the loop:

```
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")
```

Functions

Functions in Viper resemble function calls in either Python, or Go. A basic function may be defined and invoked as follows:

```
nah func foo():
    print("Hello World!")
foo()
```

Viper also allows for explicit scoping, rather than using indentation. This allows us to move to a more well-defined scoping system, especially when we want to escape Python's well known tabs/spaces confict:

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

Viper also supports arrow functions, more on which may be found below. However, a sample arrow function may either be anonymous, or assigned to a function type variable:

```
int func apply(int x, int func f):
    return f(x)
```

```
int squared = apply(10, int (int x) => x * x)
```

An assigned arrow function may look as follows:

```
func f = int (int a, int b) => a + b
int result = f(10, 20)
```

Comments

As many other popular scripting languages use # to denote single-line comments, we feel it is natural to continue this tradition.
However, a pain point of Python is the lack of “real” multi-line comments so we will implement multi-line comments using `/*` and `*/` tokens.

Example:

```
# who decided foo and bar would be fun words to use for code examples?
```

```
/* old code here:
func foo() {
    print("bar");
}
*/
```

Memory

The Viper language will be call by value like Python is, and all memory management will be handled internally by a simple garbage collector.

Unique Features

Statically Typed Variables

As we wanted to develop a compiled scripting language that shared our favorite aspects of other languages, we decided that using static typing would make full use of the compiler and help to alleviate the runtime errors that make dynamically typed languages so tedious to use on systems where safety and security is key. Defining the type would be C-style in that variables must be declared or initiated with a type. Function return types and parameters must also be specified.

Examples:

```
int big_number;
float small_number = 1.0;

func int add1(int a) {
    return a+1;
}
```

Scope Definition Options

Scope in Python is traditionally defined with whitespace.
Viper retains this option, while also giving users the alternative (via curly braces) to take a more traditional approach and avoid whitespace concerns.
With this method, everything within the scope will be equivalent to four added spaces of indentation.
Note that if this method is used, whitespace will be ignored for everything within the scope and every statement within a scope defined by `{}` must end with a semicolon.
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);
}

# Is the same as:

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

Examples of snippets that wouldn't work are:

```
for string elem in list
{
    for char letter in elem:
        print(letter)
}
```

Once you use traditional scoping, whitespace is ignored. The other way around would work fine though:

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

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

Arrow Functions

Similar to arrow functions in Javascript, or Python lambda functions, users will be able to define functions on the fly 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> (<param_type> param1, ..., <param_type> paramN) => expression outpu
```

```
<ret_type> (<param_type> param1, ..., <param_type> paramN) => {
    return complex expression output
}
```

```
<ret_type> (<param_type> param1, ... , <param_type> paramN) => :
    return complex expression output
```

Additionally, these arrow functions can be assigned to function variables:

```
func x = <ret_type> (<param_type> param1, ...,<param_type> paramN) => {
    return expression output
}
```

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

```
func myFunc = <ret_type> () => expression output

<ret_type> (<param_type> param) => expression output
```

Example Function Calls:

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

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

Anonymous Function Call Example

```
nah (int a, int b) => {
    print(a);
    print(b);
} (10, 20);
```

Additional Features

The following features make writing Viper simple and easy.

Ternary operator

Viper supports a JavaScript-like ternary operator for variable assignment. Unlike JavaScript, however, these operators can be chained together with the `|` symbol (similar to what we see in OCaml's pattern matching):

```
int x = <boolean_exp> ? <output_if_true> : <output_if_false>
```

An example with chaining:

```
int y = rand_int()
int x =
      (y < 0) ? -1    # Set x to -1 if y < 0
    | (y == 0) ? 0    # Set x to 0 if y is 0
    | (y < 5) ? 1     # Set x to 1 if y is in the range [1, 5]
    : 2
# If none of the above are true, set x to 2.
# This catch-all case must be last in the chain.
```

Iterator indexing

Viper makes an iterator's index available to the user, even when iterating directly over elements using the `in` keyword. We plan to either accomplish this by enforcing that the `in` operator returns a tuple, which can then be unpacked within the loop body. The second tuple value can also be implicitly ignored.

```
int[] array = [3, 2, 1]
for int num, int idx in array:
    print(idx)
```

stdout:

```
0
1
2
```

We expect to include other examples of syntactic sugar in the future.

A Cool Example in Viper

Let's look at function overloading, which is now made possible in Viper (due to explicit typing)

```
func add = int (int x, int y) => x + y;
func add = float (float x, float y) => x + y;
func add = char (char x, char y) => x + y;

int intResult = add(10, 20);
float floatResult = add(10.1, 10.2);
char charResult = add('a', 'b');
```

Another cool example could be something like the GCD function:

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

    func conditional = int (int x, int y) =>
        x == 0 ? y : y == 0 ? x : nah;

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

    int check = conditional(a, b);
    if (check == nah) {
        return swappedGCD(a, b);
    }
    return check;
}
```