Coral Programming Language Proposal

Rebecca Cawkwell, Sanford Miller, Jacob Austin, Matthew Bowers
rgc2137@barnard.edu, {ja3067, skm2159, mlb2251}@columbia.edu

September 19, 2018

1 Overview of Coral

The Coral programming language is an imperative and functional scripting language inspired by Python, but with support for optional static typing. It roughly aims to be to Python as TypeScript is to JavaScript. The syntax is quite similar to Python, with the addition of a few operators and the option of tagging a variable or function declaration with specific types. These types will not only be checked by the compiler, preventing errors due to invalid function usage in large codebases, but will also be used to optimize the performance of the language where possible. The goal is to create a language which is as convenient as Python, is as safe as an explicitly typed language like Java, and is more performant than existing scripting languages.

Our goals are:

- Python-style syntax with all its usual convenience, including runtime type inference where not explicitly specified.
- Type safety where desired, with type specifiers on variables and functions enforcing correct usage in large code bases.
- Potential optimizations due to types known at compile time. If the argument and return types of a function are given explicitly, it can be compiled into a function potentially as performant as equivalent C code.
- Seamless interplay between typed and untyped code. You dont pay a penalty if you dont type your code, and typed functions can be called with untyped arguments.

2 Language Details

2.1 Data Types and Operations

Coral's primitive data types are integers, floats, characters, and booleans. Strings are a built-in a class wrapping immutable arrays of characters. The language is in general not explicitly typed, and a wide range of automatic conversions will be performed. Control-flow consists of if-else statements, for loops, and while loops. Coral implements operators =, ==, !=, +, -, *, /, %, ++, −, +=, -=, <, >, =<, and >= for integers and floats, =, ==, !=, <, >, =<, and >= for chars, and =, ==, !=, !, &&, and || for booleans.

Other types may be added as needed. We will also have array types, but their exact syntax is yet to be determined. They will likely use an int[], float[], and char[] syntax. The language also supports custom-defined types, which can also function as keywords in typed functions.

Data Type:	Description:	Operations (unary and binary):	Examples:
int	An integral type, 4 bytes	=, ==, !=, +, -, *, /, %, ++, –, >>, <<, +=, -=, <, >, =<, >=	3 == 3 # True 3 * 4 # 12 3 > 4 # False 3++ # 4 x = 3
char	A character type, 1 byte	=, ==, !=, +, ++, –, <, >, =<, >=	'a' == 'a' # True 'b' == 'a' # False 'B'++ # 'C'
float	A floating point number, 8 bytes	=, ==, !=, +, -, *, /, %, ++, –, +=, -=, <, >, =<, >=	3.0 == 3.0 # True 3.0 * 4 # 12.0, converted 3.0 >4.0 # False 3.0++ # 4.0 x = 3.0

boolean	A boolean literal, with value either true or false	=, ==, !=, !, &&, ||	x = true
			! x # Returns false
			3 == 3 # Returns true

String (implemented as a class in standard library)	An immutable array of chars	=, ==, !=, <, >, =<, >= (lexicographical), + (concatenate)	x = "hi"
			y = "boye"
			x + y # Returns "hiboye"

2.2 Keywords

The following are reserved keywords in Coral:

`while, for, in, if, else, def, class, construct, return, true, false, int, float, char, bool, string, const, explicit, len, range`

2.3 Control Flow

The following keywords are reserved for control flow, and work like Python: if...else, while, for...in.

2.3.1 For...in Loops

For loops iterate over lists, as in Python, assigning each element to a given variable. For example, given a list mylist, this code iterates through it and prints the value of each element plus one:

```
for i in mylist:
    print(i + 1)
```

Generic for loops can be achieved with:

```
for i in range(1):
    print(i)
```

2.4 Functions

Coral supports function declarations in a standard Python style with

```
def foo(a, b):
    return a + b
```

Any objects can be passed to this function, and it will try its best at runtime to call a version of the given function which can support the passed types. Finding the right function version for the given objects will likely be implemented with a hash look-up under the hood (using a C backend for hash functions). The exact runtime environment has yet to be determined.

Functions themselves are first class objects, and can be passed by name to other functions, as in:

```
def map(f, my_list):
    for i in range(len(my_list)):
        my_list[i] = f(my_list[i])
```

2.5 Comments

Single-line comments use the pound symbol (#). Multi-line comments are denoted with a /# #/ notation. For example:

```
x = 3  # I really should have chosen a more descriptive name

/# This is a
 multi-line
 comment
/#
```

2.6 Memory

The Coral language will be ”pass by name,” as in Python, and all memory management will be handled internally by a simple garbage collector.

3 Optional Types

3.1 Typed Variables

All variables in Coral can be assigned a type using the syntax

```
int x = 5
```

This tells the compiler that the symbol x is an integer. Any future value assigned to it must also be of integer type. The command

```
int x = 3
x = 3.0
```

will fail. However, explicitly-typed variables can be combined seamlessly with untyped variables, as in

```
int x = 5
y = 3

z = x + y
```

There will be no issue here. The operation will simply run as expected; only x is explicitly typed, so addition with the untyped y returns whatever type is appropriate in context. Objects can be casted to each other when explicitly typed, as in

```
int x = 3
float y = (float) x
```

3.2 Typed Functions

Function arguments and return values can also be typed. For example, you can declare a function which takes two integers and returns an integer:

```
def int foo(int x, int y):
    return x + y
```

This function expects two integer inputs, and expects to return an integer. If it is given an argument which is not an integer, or if it returns something other than an integer, it will throw an error. A function can also be declared with only some typing, i.e.

```
def foo(int x, y):
    return x + y
```

which only requires that the first argument is an integer.

3.2.1 The Explicit Keyword

A function can be tagged with the explicit keyword, which tells the compiler that every type in the function is defined at compile time. For example, this GCD function

```
def explicit int gcd(int a, int b):
    while (a != b):
         if a > b:
            a -= b
        else:
            b -= a
    return a;
```

is declared explicit, and the return type and the types of all variables used in the body are indeed all explicitly defined. This allows the compiler to compile this function into highly efficient code. Instead of having to do runtime type-checking at every stage, a version of the function will be compiled with the appropriate types.

Depending on how intelligent we are able to make the parser, this keyword might be unnecessary, but we are including it to give a sense for the kinds of optimizations we intend to incorporate. We might also not make any optimizations, which would render this totally irrelevant.

3.3 Standard Library

A list with append, insert, pop, remove, and count methods is built into the standard library. Dicts (hash tables) may also be added, and even additional data types as well as time allows. The standard library will also include generic functional programming constructs like map.

3.4 Object-Oriented Programming

Coral supports basic classes, but without any sort of inheritance mechanism. Classes are simply objects with associated methods, local variables, and constructors. They are defined as follows:

```
class MyClass:
    def construct(a, b):
        this.a = a
        this.b = b

    def add():
        return this.a + this.b
```

Objects belonging to a class are instantiated like this:

```
MyClass instance = MyClass(3, 4)
instance.add() # returns 7
```

4 Examples

```
# Printing a list with static typing
String[] my_list = ["1", "1.0", "one"] # Explicitly typed list

while(i < len(my_list)):
    print(my_list[i])
    i++

# Printing a list without static typing using a for loop
my_list = ["1", "1.0", "one"]
for x in my_list:
    print(x)

# Declarations
```

```
bool x = true
int[] x = [1, 2, 3] # Array literal
x = 'a' # Single quotes automatically initialize chars
x = "Hello World!" # Double quotes automatically initialize strings

# Array errors
String[] wont_compile = ["1", 1] # All of the elements in a typed array must be of the same type
also_wrong = [1, 2] # Elements in an array must be the type of the array if it is specified.
```