Rivet

Release 0.1.0b

StunxFS

Jan 18, 2023

CONTENTS

A general-purpose programming language, focused on simplicity, safety and stability.

Rivet's goal is to be a very powerful programming language and at the same time easy to use, with a syntax inspired mainly by Zig, Rust and C# (which are the coolest languages I've ever seen), and by other languages such as Python, Lua, TypeScript, D, Go, etc.

Check out the *Overview* section for further information, including how to build the project.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 Overview

Rivet is a general purpose programming language designed for the development of stable and safe software.

Rivet uses the C programming language as its main backend.

Note: Before continuing, I assume you know how to use a console, otherwise you can read this tutorial: The Linux command line for beginners.

1.2 Run the compiler

1.2.1 Dependencies

- The compiler requires Python 3.
- **The Rivet compiler currently generates C code, so a C compiler, which supports C11,** is required to generate executables. Over time the compiler will add support for generating binaries directly without the need for a C compiler.

The compiler has been tested on **linux** and **windows**.

Just execute `python3 rivetc some_file.ri`.

You can see all available compiler options by using the `-h`/`--help` flag.

```
python3 rivetc -h
```

1.3 Hello World!

Let's start with the typical `Hello World!`:

We create a file called `hello_world.ri` with the following content:

```
import "std/console";

fn main() {
    console.println("Hello World!");
}
```

Then we compile that file:

```
$ python3 rivetc hello_world.ri
```

We'll get an executable called `hello_world` as output, so we run it:

```
$ ./hello_world
```

We should see this output:

```
Hello World!
```

Excellent! You have compiled your first program in Rivet!

1.4 Editor/IDE support

- LiteXL (Syntax-highlighting only).

1.5 Code Structure

1.5.1 Comments

```
// This is a single line comment.
/*
This is a multiline comment.
*/
```

You can use comments to make reminders, notes, or similar things in your code.

1.5.2 Entry point

In Rivet, the entry point of a program is a function named `main`.

```
func main() {
    // code goes here
}
```

1.5.3 Top-level declarations

On the top level only declarations are allowed.

```
import "module" { import_list, ... };

const Foo: int32 = 0;

let Foo: int32 = 0;

type Foo = int32;
```

(continues on next page)

(continued from previous page)

```
trait Foo { /* ... */ }

struct Foo { /* ... */ }

enum Foo { /* ... */ }

extend Foo { /* ... */ }

func foo() { /* ... */ }

test "Foo" { /* ... */ }
```

1.6 Functions

Functions contain a series of arguments, a return type, and a body with multiple statements.

The way to declare functions in Rivet is as follows:

```
func <name>(<args>) [return_type] {
    ...
}
```

For example:

```
func add(a: i32, b: i32) i32 {
    return a + b;
}
```

add returns the result of adding the arguments a and b.

Functions can have 0 arguments.

```
// `f1` returns a simple numeric value of type `i32`.
func f1() i32 {
    return 0;
}

// `f2` takes an argument of type `i32` and prints it to the console.
func f2(a: i32) {
    println("a: {}", a);
}

// `f3` takes no arguments and returns void.
func f3() { }
```

A function body is made up of 1 or more statements and can be empty.

```
func x() {
    /* empty body */
}
```

(continues on next page)

(continued from previous page)

```
func y() {
    let my_var = 1; // statement
}
```

1.6.1 Arguments

The arguments are declared as follows: `<name>: <type> [= default_value]`, for example: `arg1: i32`, `arg2: bool = false`.

The arguments are immutable.

They can also have default values, this bypasses the need to pass the argument each time the function is called: `arg1: i32 = 5`.

So, if we have a function called `f5` with a default value argument, we can call it in 3 ways:

```
func f5(arg1: i32 = 5) {
    println("arg1: {}", arg1);
}

f5(); // use the default value `5`
f5(100); // will print 100 instead of 5 to the console

// this uses a feature called `named argument`, which allows an optional
// argument to be given a value by its name in any order
f5(arg1: 500); // will print 500 instead of 5 to the console
```

1.7 Statements

Each statement must end with a semicolon.

1.7.1 Variables

Variables are like boxes that contain values.

Variables are declared as follows: `[mut] <name>[: <type>] = <value>;`. Example:

```
x: i32 := 1;
```

We have created a variable called `x`, which contains the value 1 and is of type `i32`.

The type of the variable can be omitted.

```
x := 1; // via inference, the compiler knows that `x` is an `i32`.
```

By default, all variables are immutable, that is, their values do not change. To change the value of a variable you have to declare it with `mut`.

```
mut x := 1;
x = 2; // this is valid
```

(continues on next page)

(continued from previous page)

```
y := 1;
y = 2; // error: `y` is immutable
```

Multiple values can be assigned on a single line via tuple-destructuring, example:

```
(a, b, c) := (1, 2, 3);
(c: i32, d: i32, e: i32) := (4, 5, 6);
(f, g, h) := tuple_fn();

// this is a short form for:

a := 1;
b := 2;
c := 3;

c: i32 := 4;
d: i32 := 5;
e: i32 := 6;

tmp_tuple_fn := tuple_fn();
f := tmp_tuple_fn.0;
g := tmp_tuple_fn.1;
h := tmp_tuple_fn.2;
```