A First Look at Cairo 1.0: A Safer, Stronger & Simpler Provable Programming Language

Mathieu

Follow

Nethermind.eth

--

1

Listen

Share

By

Mathieu Saugier and Edgar Barrantes.

Special thanks to

[Jorik Schellekens

](https://uk.linkedin.com/in/jorik-schellekens-346567120)for reviewing.

Earlier this year, StarkWare announced Cairo 1.0 as an improvement to the Cairo programming language that will make it safer, better, and easier for writing contracts. With an improved syntax and a fully-fledged type system, Cairo 1.0 is paving the way toward an open-source Starknet stack.

This blog post will take you through the new features added to Cairo and discuss how they will improve the language. Keep in mind that Cairo 1.0 is still a work in progress, and new features are being added regularly, so the current state of the language is not what it will look like when it is fully released.

An improved syntax inspired by Rust

One of the key features of Cairo 1.0 is its improved syntax. The syntax of Cairo has been designed to be clean, simple, and easy to read, making it more intuitive and accessible for new users. This new syntax is inspired by Rust and borrows some of its key concepts, such as return statements, mutable variables, and pattern matching.

Variable declaration

In Cairo 1.0, all variables are declared using the let

keyword. It is now possible to change the value of a previously declared variable if it was declared as mutable, using the mut

keyword. Mutable variables hide from the developer the fact that the underlying memory is immutable, which makes programs easier to write.

Return statements

Like in Rust, functions have two ways of returning values. The first is by using the final expression of the function. In this case, the ;

is omitted at the end of the line. The second is by using the return

keyword. This can only be used to return a value earlier from within the function and can, for example, be used from inside if

statements.

Pattern matching

In Cairo 1.0, you can use the match

keyword to perform pattern matching, allowing you to easily match different patterns in your code. At the time of writing this post, pattern matching can be performed on enum variants and felts. Using match

statements allows developers to write more concise and expressive code, avoiding a series of if

and else

statements. It also helps to detect bugs during development by requiring the specification of a pattern for each possible value of the expression under test, which might be overlooked when using if statements.

Note: When using pattern matching with felts, while it should be possible to match other values in the future, only match zero (

match ... { 0 => ..., _ => ... })

is currently supported.

Enums

Enums are a way to define a new data type that can have a fixed set of values. By defining an enum, you can ensure that only valid values can be used, which can help prevent errors and improve the reliability of your code. Enums are also a way to group values, giving meaning to each variant and creating a logical structure for the data.

You can combine enums with pattern matching to avoid writing complex control flow constructs and have better control over the value used, making your code easier to understand and maintain.

Structs

Structs remain the same as in Cairo 0. It's a custom data type that provides a way to group values, defining a single entity that represents a collection of related values. However, unlike in the previous Cairo version, we can now define methods that can operate on the struct using a combination ofimpls

and traits

.

In order to destructure a struct into multiple variables, you can use the following syntax: let Point{x: a, y: b} = origin;

Ownership and references

Cairo 1 uses an ownership system similar to Rust. If you are unfamiliar with the ownership and borrowing system, we encourage you to read the ownership section in the Rust Book. When doing assignments (let x = y

) or passing functions arguments by value (foo(x)

), the ownership of the variable is transferred, which is known as a move

.

A variable that was moved can't be accessed later in the scope, and the compiler will throw an Variable was previously moved

error.

The example below shows how you can pass function arguments by value and by reference. When passing by value the argument to the pass_by_value

function, the ownership is transferred to the function and trying to access this variable in the let val_x = point_value.x

instruction will throw an error. When passed by reference to pass_by_ref

, the function is only borrowing

the value and returns the ownership after its execution so that it is possible to use it in the let ref_x = point_ref.x

instruction.

Derive macros

As you noticed in the Struct example above, we used a derive macro above our struct. These macros are used by the compiler to create new items from the code of a struct or an enum. Under the hood, the compiler generates this code:

If you remove this macro, the compiler will throw this error:

With the instruction let a = origin.x

, we're taking ownership of the origin

variable. Cairo 1 is similar to Rust on this point as a borrow checker makes sure no variable can be used twice and throws the Variable was previously moved

error. In the return instruction, we're trying to use origin

by returning origin.x

, hence the error. By implementing the Copy

trait for Point, values are implicitly copied when transferring ownership.

The second error is Variable not dropped

. In Cairo 1, every variable that is declared must be either returned at the end of the function or dropped. Custom types need to implement the Drop

trait so that they can be dropped after they're used.

Traits

The trait

keyword is used to define functionalities a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. You can think of a trait as an interface for a specific type so that every type that implements the trait must also implement all of its functions.

Implementations

The impl

keyword is primarily used to define implementations on traits. Trait implementations are used to implement a specific behavior for a trait, meaning that all the functions from the trait must be implemented.

Starknet contracts

StarkNet contracts are modules annotated with a #[contract]

decorator. The storage values of a contract are defined under the Storage

struct and can be either types from the core library or mappings.

- The constructor is a function annotated with a #[constructor]

decorator

- Events are empty functions annotated with a #[event]

decorator

- View and external functions are annotated with #[view]

or #[external]

- Other functions are not a part of the contract’s ABI

Generics

Cairo 1.0 supports generics, which are a flexible way to write code that can work with multiple types rather than being tied to a specific type, avoiding code redundancy. You can define a generic type by adding a placeholder type within angle brackets <>

after the name of the type or function. Generics are not implemented yet, so running the code below will fail, but this is how it should work.

The example below shows how you can use functions with generics to avoid code redundancy. In this example, we define a swap

function that takes two parameters of type T

and returns the values swapped.

Type literals

Numeric literals are fixed values for numbers, such as 1

, 2

, or 3

.

These literals are felts

by default, but using a suffix _

we can specify a specific type for our literal, such as uint128

. You can use type literals as function arguments, improving code clarity and readability compared to the previous version of Cairo.

A developer-friendly core library

Arrays

Working with arrays becomes easier with Cairo 1.0. The core library exports an array

type along with associated functions that allow you to easily get the length of the array you're working with, append an element or get the element at a specific index. It is particularly interesting to use the ArrayTrait::get()

function because it returns an Option

type, meaning that if you're trying to access an index out of bounds, it will return None

instead of exiting your program, meaning that you can implement error management functionalities. In addition, you can use generic types with arrays, making arrays easy to use compared to the old way of managing pointer values manually.

Dictionaries

Cairo 1 also supports dictionaries in the core library. A dictionary is a data structure that stores key-value pairs. The key is used to look up a specific value, much like looking up a word in a print dictionary. It is currently possible to create dictionaries that map felts to u128

, felt

, u8

, u64

and Nullable::

.

Because short strings are actually felts, it is possible to create string keys in a dictionary.

Debugging

The latest release alpha2 introduced support for basic debugging functions. You can now print messages during the execution of a Cairo program.

Panic

The panic

function is used to raise a panic, which indicates that the program encountered an error. It takes an Array

from the core library filled with felts as a parameter. You can use felt string as error messages, as long as your string can fit into a felt. Panics make it easier for developers to detect bugs and debug them by assigning specific error codes to each situation and are also useful for users who interact with smart contracts if a transaction fails.

Integer types

Multiple integer types are available in the core library: u8, u16, u32 (usize), u64, u128

, and u256

. Even though they rely on the basic type felt

under the hood, it is safer to use them if the context is appropriate. They support arithmetic operators, meaning that you can, for example, sum them with let sum = 1_u128 + 2_u128

. These operators automatically check for overflows and panic if one is detected. This greatly improves the clarity of the code compared to using functions for basic arithmetic operations.

Assertions

assert

is a function that verifies a boolean condition and panics if the condition is not satisfied. The first parameter is the boolean check to perform, and the second one is the error code to panic with if the assertion fails. The example below shows how you can combine type literal and arithmetic operations with assertions.

Conclusion

While Cairo 1.0 is still early and may change a lot, this first look into the codebase has taught us a lot regarding the future of the language. The steepness of the Cairo learning curve will decrease, making the transition from other programming languages easier. With the addition of a package manager and a proper development framework for testing and deployment, writing contracts on StarkNet will become as easy as on any other blockchain.

How do Cairo fundamentals stack up against EVM and Solidity? Learn more [here

](/nethermind-eth/cairo-fundamentals-stacked-up-against-evm-and-solidity-1d8d4e12b2c3).