Lesson 4 - Development Tools / Cairo

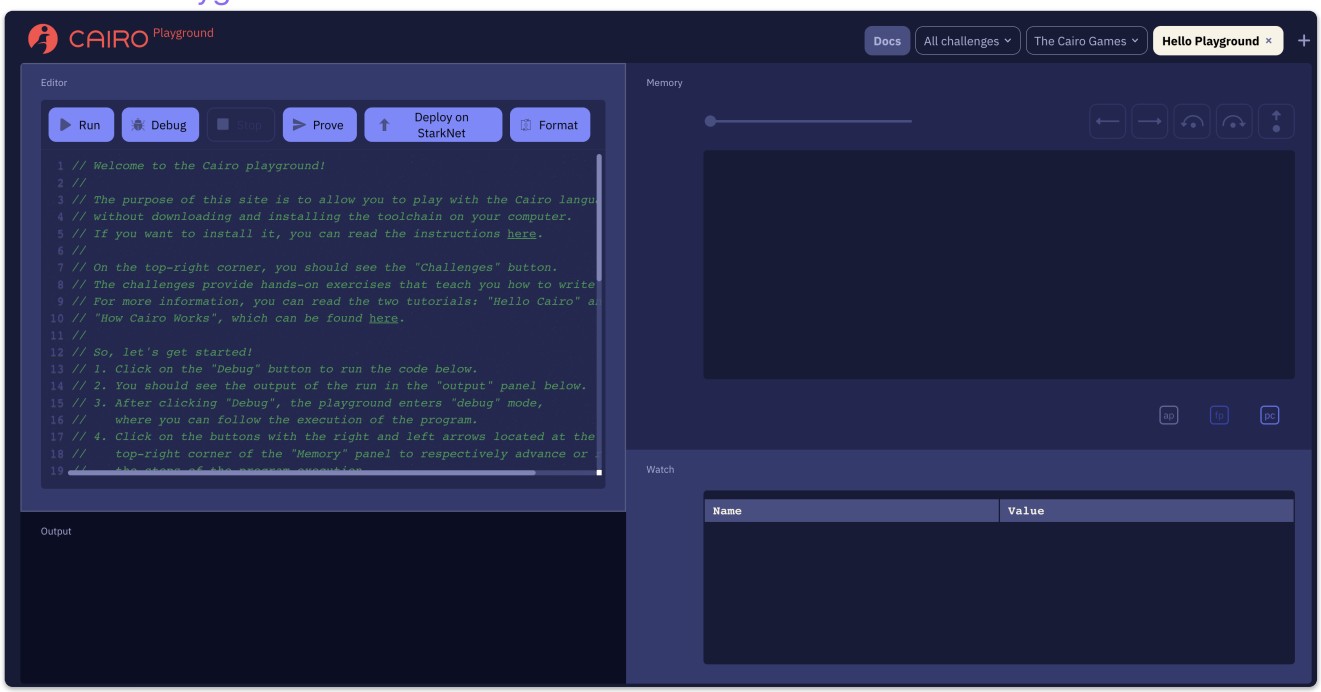
News

There was a twitter space discussion of Kakarot today.
Kakarot is a ZK-EVM written in Cairo , see repo

Development Tools

Cairo Playground

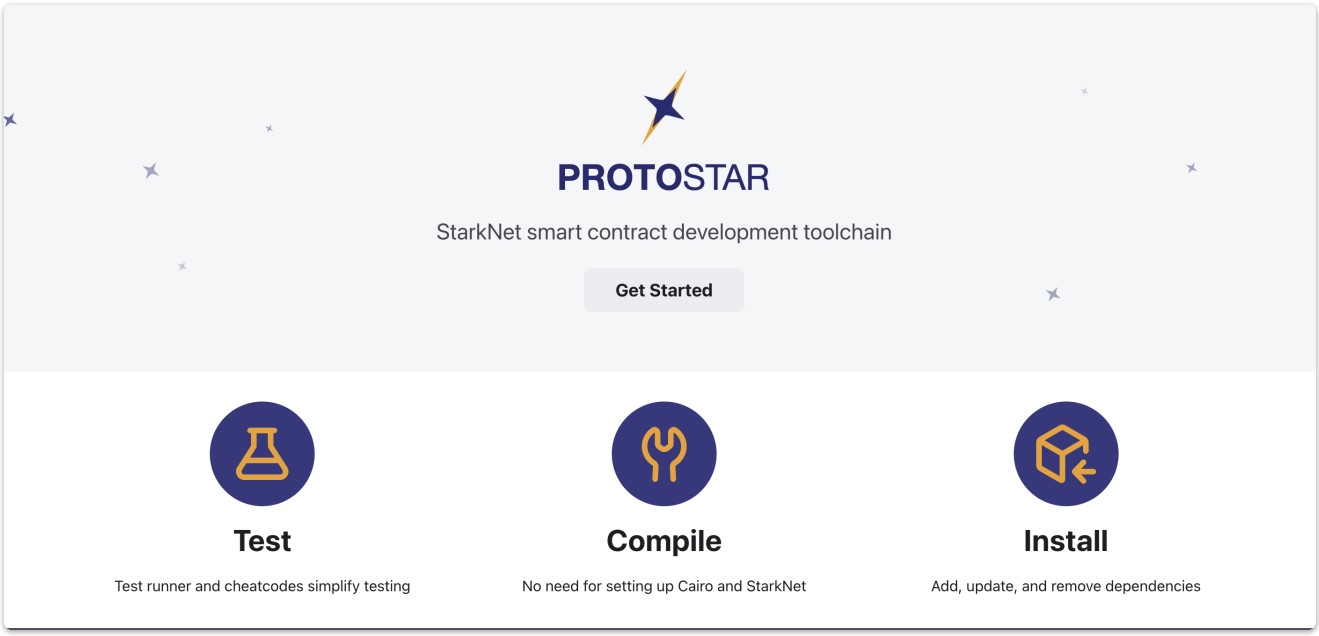
Web based IDE similar to Remix
See Cairo Playground



Features

Develop and run programs
Debug programs
Possible to send to the shared prover and deploy contracts

Protostar



See Protostar

Features

CLI toolchain
Unit test programs and contracts
Deploy contracts

We will be using protostar as our main tool for development

Creating a project

to create a new project use

```
protostar init
```

This will give a directory structure similar to this

```
drwxr-xr-x  7 laurencekirk  staff  224 16 Jul 05:39 ./
drwxr-xr-x  3 laurencekirk  staff   96 16 Jul 05:39 ../
drwxr-xr-x  9 laurencekirk  staff  288 16 Jul 05:39 .git/
drwxr-xr-x  2 laurencekirk  staff   64 16 Jul 05:39 lib/
-rw-r--r--  1 laurencekirk  staff  148 16 Jul 05:39 protostar.toml
drwxr-xr-x  3 laurencekirk  staff   96 15 Jul 10:03 src/
drwxr-xr-x  3 laurencekirk  staff   96 15 Jul 10:03 tests/
```

Configuration

This is specified in the .toml file

```
["protostar.config"]
protostar_version = "0.1.0"

["protostar.project"]
libs_path = "./lib"         # a path to the dependency directory

# This section is explained in the "Project compilation" guide.
["protostar.contracts"]
main = [
  "./src/main.cairo",
]
```

Compiling your programs / contracts

once you have specified the contracts in the protostar.toml file, run

```
protostar build
```

to compile them.

Deploying your programs / contracts

You need to specify the path to the compilation results.

```
$ protostar deploy ./build/main.json --network alpha-goerli
```

Testing your programs / contracts

Protostar will find the test file using its name, checking if it begins with `test_` prefix, and has `@external` functions, which names begin with `test_` .

A test looks like

```
@external
func test_sum{syscall_ptr : felt*, range_check_ptr}(){
    let (r) = sum_func(4,3);
    assert r = 7;
```

```
    return ();
}
```

You can run the tests, specifying the test directory

```
protostar test ./tests
```

Nile

See Nile

Features

CLI toolchain
Unit test programs and contracts
Deploy contracts

Starknet CLI

Starknet also offer a CLI
See installation instructions
It allows you to compile and run programs, and deploy contracts.

Plugins

Plugins are available for popular IDEs offering some degree of language support

VSCode plugin for Cairo :
Hardhat plugin

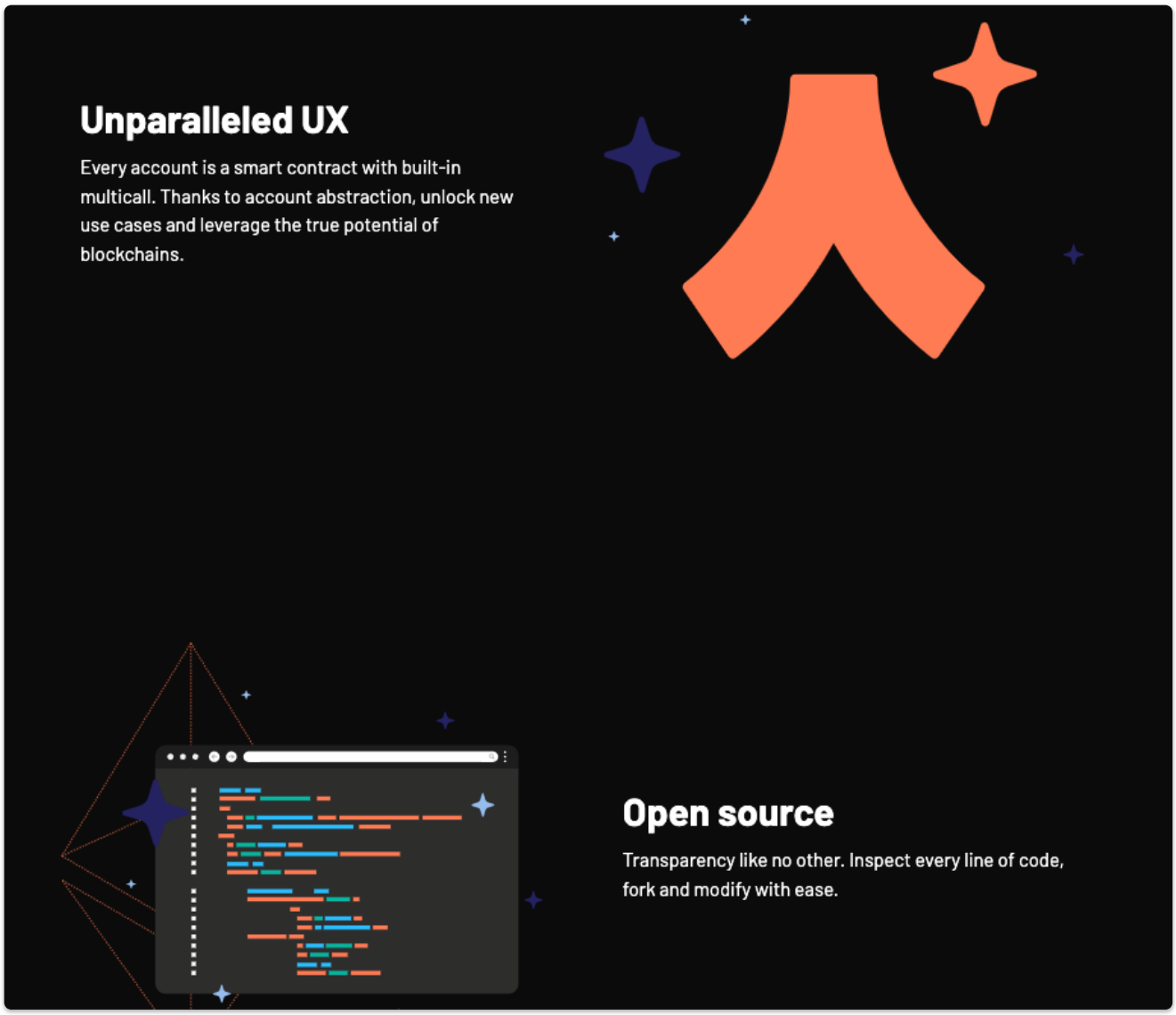
Foundry experimental

Wallets for Starknet

Braavos

Available as an app for a browser extension
See Docs
See Extension
Twitter : @myBraavos
Blog

ArgentX



See Extension

They have guides to Ethereum / DeFi

We shall see how they use account abstraction to improve the user experience in games.

Game without interruption

Session keys let you connect your wallet when you start playing and stay in the game until the end of the session.

No popups to approve every action!

Cairo continued

Builtins revisited

Builtins are predefined optimized low-level execution units which are added to the Cairo CPU board to perform predefined computations which are expensive to perform in vanilla Cairo
The available builtins are

1. output - to output values, these are seen by the verifier.
2. signature - to allow checking of ecdsa signatures.
3. bitwise - to carry out bitwise operations on felts
4. pedersen - to supply the pedersen hash function.
5. range check - to compare integers and check they fall in a certain range.

They have their own area of memory set aside for their use, and hence need implicit arguments in functions.
To use the builtins you need to specify them at the beginning of your program, for example

```
%builtins output pedersen range_check ecdsa bitwise
```

Revoked references

The compiler substitutes a reference with the thing it refers to, but it may find a situation where it doesn't know how to do this.
For example is we have
`let a = [ap -1];`
and later in our code we use `a`, it will substitute that with `[ap -1]`
In order to do this, it needs to understand how `ap` will change, and it may not be able to do this unambiguously.

From the documentation
"If there is a label or a call instruction between the definition of a reference that depends on `ap` and its usage, the reference may be *revoked*, since the compiler may not be able to compute the change of `ap` (as one may jump to the label from another place in the program, or call a function that might change `ap` in an unknown way).""

The way to solve this is to use local variables which depend on `fp` rather than `ap` for example

`local a = 13`

in order to use local variables we must explicitly add

`alloc_locals;`

Loops / Recursion

Although loops are possible in Cairo, they are restricted in what they can do and so instead we use recursion.
Loops will (hopefully) be fully supported in Cairo v 1.0
For an example of recursion see the cairo playground Recursion challenge

Error Messages / Scope Attributes

See documentation

Scope atrributes are specified for a code block by surrounding it with the `with_attr` statement

```
with_attr attribute_name("Attribute value"){
    # Code block.
}
```

The attribute value must be a string, and can refer to local variables only. Referring to a variable is done by putthing the variable name inside curly brackets
(e.g., `"x must be positive. Got: {x}."`).

At present, only one attribute is supported by the Cairo runner: `error_message`. It allows the user to annotate a code block with an informative error message. If a runtime error originates from a code wrapped by this attribute, the VM will automatically add the corresponding error message to the error trace.

Strings

Strings are not natively supported as a datatype, since everything fundamentally is a felt.
We can create string literals
`[ap] = 'hello';`

which the compiler encodes into a felt

`[ap] = 0x68656c6c6f;`

There is a utility scripts to convert strings into a felt in our repo

Useful Libraries

Import the libraries using this format

```
from starkware.cairo.common.bitwise import bitwise_operations
```

1. Math.cairo
    - `assert_not_zero()`.
    - `assert_not_equal()`.
    - `assert_nn()`.
    - `assert_le()`.
    - `assert_lt()`.
    - `assert_nn_le()`.
    - `assert_in_range()`.
    - `assert_le_250_bit()`.
    - `split_felt()`.
    - `assert_le_felt()`.
    - `abs_value()`.
    - `sign()`.
    - `unsigned_div_rem()`.
    - `signed_div_rem()`.
2. Common Library

- alloc.
- bitwise.
- cairo_builtins.
    - This has structs
        - BitwiseBuiltin
        - HashBuiltin
        - SignatureBuiltin
- default_dict.
- dict.
- dict_access.
- find_element.
- set.

3. Bool comparison of felts

- equal
- either
- both
- neither
- not

4. Uint256
   This has a struct to hold the values and 2 operations on the values

- `Uint256`
- `uint256_add()`
- `uint256_mul()`

The value is split into 2 parts high and low with
Low = least significant u251, High. =. most significant u251

We need the implicit argument `range_check_ptr` for the functions.
Functions include

- uint256_check
- uint256_add
- uint256_mul
- uint256_sqrt
- uint256_lt
- uint256_le
- uint256_unsigned_div_rem
  See the repo for others

Example of using Uint256 in Cairo

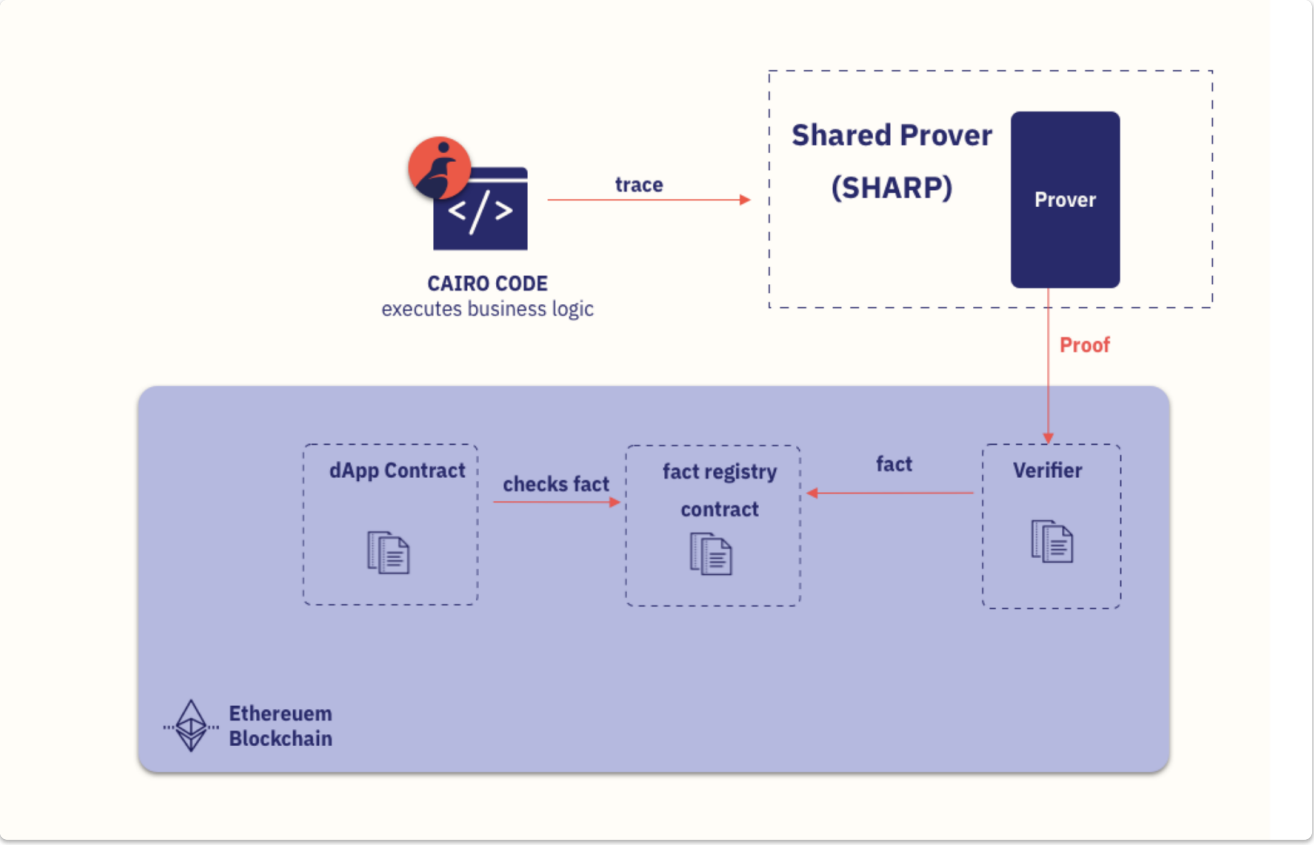
```
%builtins output range_check

from starkware.cairo.common.uint256 import (uint256_add, Uint256,
    uint256_mul)
from starkware.cairo.common.serialize import serialize_word

func main{output_ptr : felt*, range_check_ptr}():
    alloc_locals
    local num1 : Uint256 = Uint256(low=0,high=10)
    local num2 : Uint256= Uint256(low=0,high=3)
    let (local mul_low : Uint256, local mul_high : Uint256)
    = uint256_mul(num1, num2)
    serialize_word(mul_high.low)

    return ()
end
```

5. Felt Packing

   The idea of this library is to be able to store multiple smaller felts into one bigger felt.

   As an example it is possible to store 62 felt of size 8 bits (0-255) into one unique felt.

Shared prover for cairo programs



There are more details of the process here
You can try this process with the prove button in cairo playground

you can monitor the progress

SHARP status tracking

Job key: 27ad68f1-3788-4c4e-888b-7d1da1c85793
Program hash: 0x049a748653632ec760b53cb9830fb30e989b6a12fc8e15345fcb3ebfa79cf376
Fact: 0xf6fe2af6e4ec2f4247e9d536e0b79c2b64538d9da58c7fc9f8417e8ecfdf58c9
Current status: Job validated. Waiting for train to be created and proved...
Created -> **Processed** -> **Train proved** -> **Registered**

Once your fact is registered, you can query it using the isValid() method here.

This page reloads the data every few seconds, you don't have to refresh it manually.

Next Week

- Cairo contracts
- Rollup theory
- Account abstraction
- Game development