Lesson 5

Plan for this week

Monday : Cairo
Tuesday : Starknet, Game development on Starknet
Wednesday : Cairo
Thursday : Cairo

Starknet / Cairo

Starknet

"StarkNet is a permissionless decentralized ZK-Rollup. It operates as an L2 network over Ethereum, enabling any dApp to achieve unlimited scale for its computation – without compromising Ethereum's composability and security."

https://www.starknet-ecosystem.com/

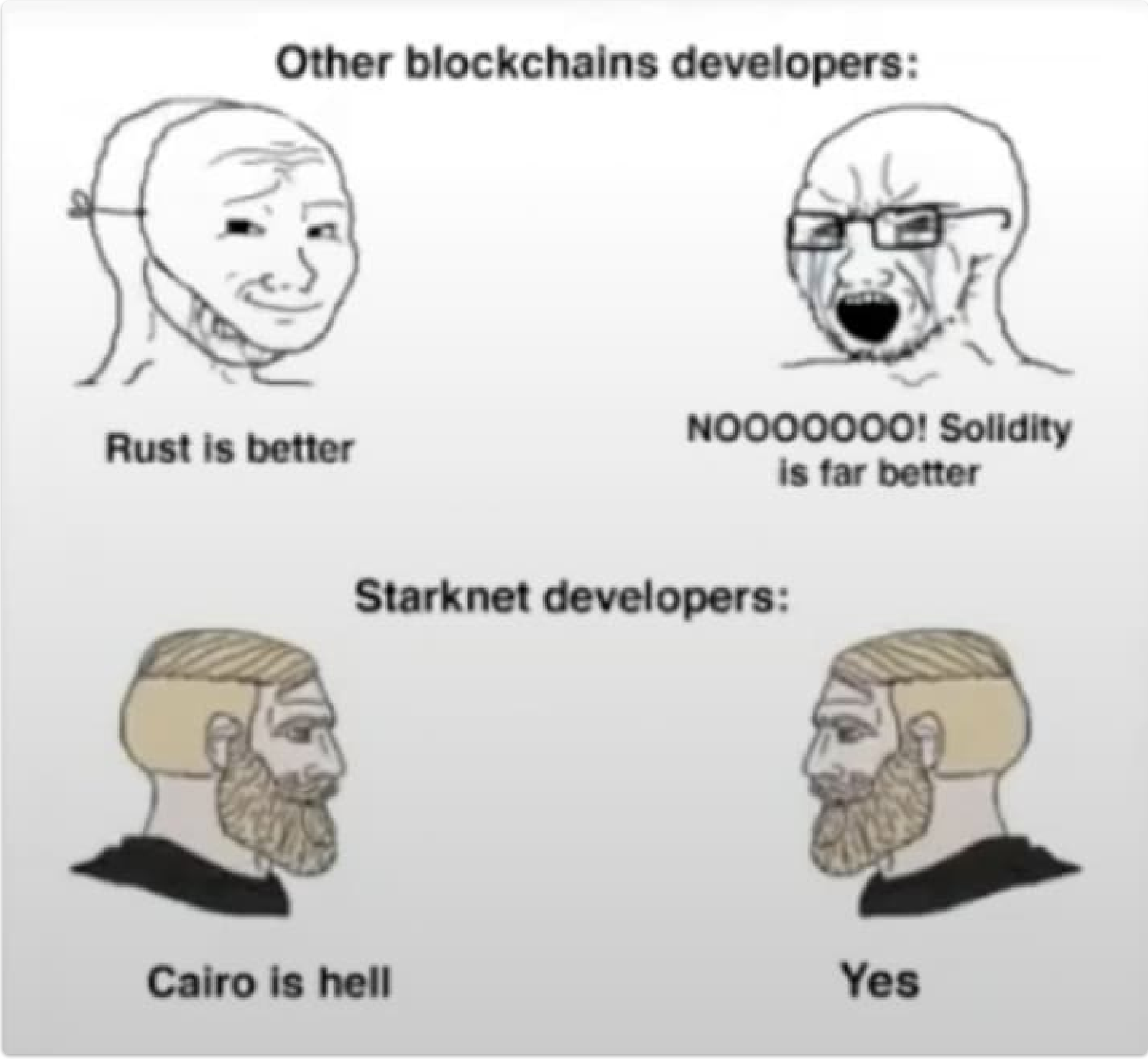
Starknet Components

1. **Prover**: A separate process (either an online service or internal to the node) that receives the output of Cairo programs and generates STARK proofs to be verified. The Prover submits the STARK proof to the verifier that registers the fact on L1.
2. **StarkNet OS**: Updates the L2 state of the system based on transactions that are received as inputs. Effectively facilitates the execution of the (Cairo-based) StarkNet contracts. The OS is Cairo-based and is essentially the program whose output is proven and verified using the STARK-proof system. Specific system operations and functionality available for StarkNet contracts are available as calls made to the OS.
3. **StarkNet State:** The state is composed of contracts' code and contracts' storage.
4. **StarkNet L1 Core Contract**: This L1 contract defines the state of the system by storing the commitment to the L2 state. The contract also stores the StarkNet OS program hash – effectively defining the version of StarkNet the network is running. The committed state on the L1 core contract acts as provides as the consensus mechanism of StarkNet, i.e., the system is secured by the L1 Ethereum consensus. In

addition to maintaining the state, the StarkNet L1 Core Contract is the main hub of operations for StarkNet on L1. Specifically:

- It stores the list of allowed verifiers (contracts) that can verify state update transactions
- It facilitates L1 ↔ L2 interaction

Cairo



Cairo is the programming language used for StarkNet It aims to validate computation and includes the roles of prover and verifier.
It is a Turing complete language.

"In Cairo programs, you write what results are **acceptable**, not **how to** come up with results."

In solidity we might write a statement to extract an amount from a balance, in Cairo we would write a statement to check that for the parties involved the sum of the balances hasn't changed.

General Points

- Cairo is a turing complete language for creating STARK-provable programs for general computation.

- It can be approached at a low level, it supports a read-only nondeterministic memory, which means that the value for each memory cell is chosen by the prover, but it cannot change over time.
- A great deal of syntactic sugar has been added to make it more user friendly
- There is a distinction between Cairo programs (stateless) and Cairo contracts (given storage in the context of Starknet)

Language Features

- Datatypes

  > - Felt – a field element
  > - Struct
  > - Tuples
  > - Pointers

- Functions

- Constants

- Literals

- Arrays

- Builtins

Cairo Low level

Although we shall approach Cairo from a higher level, it is useful to understand the concepts, and also useful for debugging to know the low level features.

Memory Model

From Cairo docs
Cairo supports a read-only nondeterministic memory, which means that the value for each memory cell is chosen by the prover, but it cannot change over time (during a Cairo program execution).

The memory cell is specified by square brackets, so
[x] gives us the value at address x.
If we think of the memory as 'write once' then a statement
[1] == 13 can mean either

1. Set the value at address 1 to be 13 , if it hasn't already been set , or
2. test whether the value at address 1 is 13.

There are 3 'registers' used
'ap' - the allocation pointer, to show where unused memory starts
'fp' frame pointer, this points to the function we are in, and variables in the function are then offsets from that.
'pc' - program counter, this gives us the current instruction.

Using this syntax, a simple statement could be

```
[ap] = [ap - 1] * [fp]; ap++
```

(the semicolon is part of the ap++ and is not used to end a statement as Solidity does)

We could write our code like this, but thankfully some syntactic sugar has been added to make our lives easier.

EXAMPLE CODE

```
%builtins output
# Import the serialize_word() function.
from starkware.cairo.common.serialize import serialize_word

func main{output_ptr : felt*}():
    tempvar x = 10
    tempvar y = x + x
    tempvar z = y * y + x
    serialize_word(x)
    return ()
end
```

Datatypes

There is only one datatype - the field element (felt), although we can combine these into Structs.

Annoyingly the felt is a 252 bit integer, which gives problems when we want to fit a uint256 value into it.
We can put felts together into tuples for example

```
(a, b)
```

or if we want to name the tuple

```
(x : a, y : b)
```

We can create pointers

```
T*
```

where T is a type

and
Structs

```
struct MyStruct:
    member first_item : felt
    member second_item : felt
end
```

To create a struct once it has been declared, we use the syntax

```
let my_best_struct = MyStruct(
    first_item=12, second_item=13
)
```

Expressions

- An integer literal (e.g., `5`). Considered as of type `felt`.
- An identifier (a constant or a reference).
  E.g., `my_identifier`, `struct_name.member_name`, `reference_name.member_name`.
- An address register: `ap` or `fp`. Both have type `felt`.
- `x + y`, `x - y`, `x * y`, `x / y`, `-x` where `x` and `y` are expressions.
- `(x)` where `x` is an expression – same as `x` (allows to control operator precedence in the expression).
- `[x]` where `x` is an expression – represents the value of the member at the address `x`. If `x` is of type `T*` then `[x]` is of type `T`.
- `&x` where `x` is an expression – represents the address of the expression `x`. For example, `&[x]` is `x`.
- `cast(x, T)` where `x` is an expression and `T` is a type – same as `x`, except that the type is changed to `T`. For example, `cast(10, MyStruct*)` is `10`, thought as a pointer to a `MyStruct` instance.

Variables / References

Variables may be aliased or evaluated:

- Aliased
  - Value **Reference**: `let a = 5`.
  - Expression **Reference**: `let a = x`.
    - An alias may be evaluated with `assert a = b` (checks x equals b).
- Evaluated
  - **Temporary variable**: `tempvar a = 5 * b`.
  - **Local**: `local a = 5 * b`.
  - **Constant**: `const a = 5`.

So we can use the `let` keyword to create a reference

```
let a = 3
let b = 7
```

and we can change the reference

```
let a = 3
let a = 7
```

but we need to be aware of 'revoked' references, more of which later.

Constants

Similar to other languages we can define a contant that evaluates to a felt at compile time
For example

```
const PI = 3
```

Local References

We use the `local` keyword to define a variable that is local to a frame, these have the advantage of not being revoked.

```
local c = 13
```

A function that has local variables must include the statement

```
alloc_locals
```

Function outputs

The following format is used to reference the output from a function

`let (x) = my_function()`.

If you want to use a local reference, use

`let (local x) = my_function()`.

Assert

The assert statement is of the form

`assert <expr0> = <expr1>`

Its use is somewhat confusing that it can assert that a reference is equal to a particular value, or can set a memory location value if it hasn't already been set up (for example when assigning a value to an array)

Arrays

Dynamic arrays are created using `alloc()` this gives the address of the first element, and we can do arithmetic to get the rest of the elements.

```
alloc_locals

# Allocate a new array.
let (local array) = alloc()
# Fill the new array with field elements.
assert [array] = 1
assert [array + 1] = 2
assert [array + 2] = 3
assert [array + 3] = 4
```

The Cairo Common Library

This can be imported into our code and provides useful components

- alloc.
- bitwise.
- cairo_builtins.
- default_dict.
- dict.
- dict_access.
- find_element.
- set.

You can import these using for example

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

Functions

In simple form a function can be declared as

```
func function_name():
    # Your code here.
    return ()
end
```

We can also have `implicit arguments` (see below)

```
func func_name{implicit_arg1 : felt, implicit_arg2 : felt*}(
    arg1 : felt, arg2 : MyStruct*
) -> (ret1 : felt, fet2 : felt):
    # Function body.
end
```

Return statement

The return statement is of the following form, we can name the return values, or use their position

```
return (ret1=val1, ret2=val2)

return (2, b=3)  # positional, named.
```

but positional must come first.

Calling functions

Suppose we have a function named `foo` ,
we can all it in 4 ways

```
foo(x=1, y=2)  # (1)
let x = foo(x=1, y=2)  # (2)
let (ret1, ret2) = foo(x=1, y=2)  # (3)
return foo(x=1, y=2)  # (4)
```

Option (1) can be used when there is no return value or it should be ignored.
Option (2) binds `x` to the return value struct.
Option (3) unpacks the return value into `ret1` and `ret2`.
Option (4) is a tail recursion – after `foo` returns, the calling function returns the same return value.

You need to be consistent with named arguments
as shown where we call foo

```
func foo(y : felt) -> (z : felt):
        return (z=y + 1)
end
```

```
# x defined earlier
let (z) = foo(y=x)
```

Unpacking return values

```
let (a, b) = foo()
let (_, b) = foo()
let (local a, local b) = foo()
let (local a, _) = foo()
```

Implicit arguments

From documention

```
from starkware.cairo.common.cairo_builtins import HashBuiltin

func hash2{hash_ptr : HashBuiltin*}(x, y) -> (z : felt):
    # Create a copy of the reference and advance hash_ptr.
    let hash = hash_ptr
    let hash_ptr = hash_ptr + HashBuiltin.SIZE
    # Invoke the hash function.
    hash.x = x
    hash.y = y
    # Return the result of the hash.
    # The updated pointer is returned automatically.
    return (z=hash.result)
end
```

The curly braces declare `hash_ptr` as an *implicit argument*. This automatically adds an argument **and** a return value to the function. If you're using the high-level `return` statement, you don't have to explicitly return `hash_ptr`. The Cairo compiler just returns the current binding of the `hash_ptr` reference.

Hints

A hint is a piece of code that the prover runs to initialize some values. It is written in python code within a `%{ %}` block.
The hint can interact with the program's variables/memory.
The hint is not seen by the verifier.

For example

```
# Set the value of res using a python hint.
%{
        import math
        # Use the ids variable to access the value of a Cairo variable.
        ids.res = int(math.sqrt(ids.n))
%}
```

You need to use an assert statement to test the thing you are trying to prove.
For example, if you are showing the res is the square root of n

```
assert n = res * res
```

Note that hints are not available in Cairo smart contracts, so Cairo programs can have privacy, but Cairo smart contracts on Starknet are completely public.

Output

We can use the component serialise_word to output values from our Cairo code
This is a convenient way to 'log' values when we are testing our code.

```
from starkware.cairo.common.serialize import serialize_word

...


serialize_word(1234)
```

This output is seen by the validator.

Entrypoint

The entrypoint to a Cairo program is the main function

```
# Use the output builtin.
%builtins output

func main{output_ptr : felt*}():
        ...
    serialize_word(1234)
    return ()
end
```

We need the output_ptr argument for the builtins output

Error Messages / Scope Attributes

See documention

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

```
with_attr attribute_name("Attribute value"):
    # Code block.
end
```

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.

Useful Tools

Cairo Playground

Similar to Remix with Solidity, it allows you to write, run and debug Cairo programs
Link

Protostar

We will be using protostar for the practical exercises.

Setting up Protostar

See documentation and this useful medium article

LINUX / MAC

1.

```
curl -L https://raw.githubusercontent.com/software-mansion/protostar/master/install.sh | bash
```

2. Restart the terminal.
3. Run `protostar -v` to check Protostar and cairo-lang version.
   It adds to your PATH in .bashrc, you may need to move that line to .bash_profile If it doesn't find protostar.
   You may also need to update your version of git

WINDOWS

Windows is not supported.

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 ()
end
```

You can run the tests, specifying the test directory

```
protostar test ./tests
```