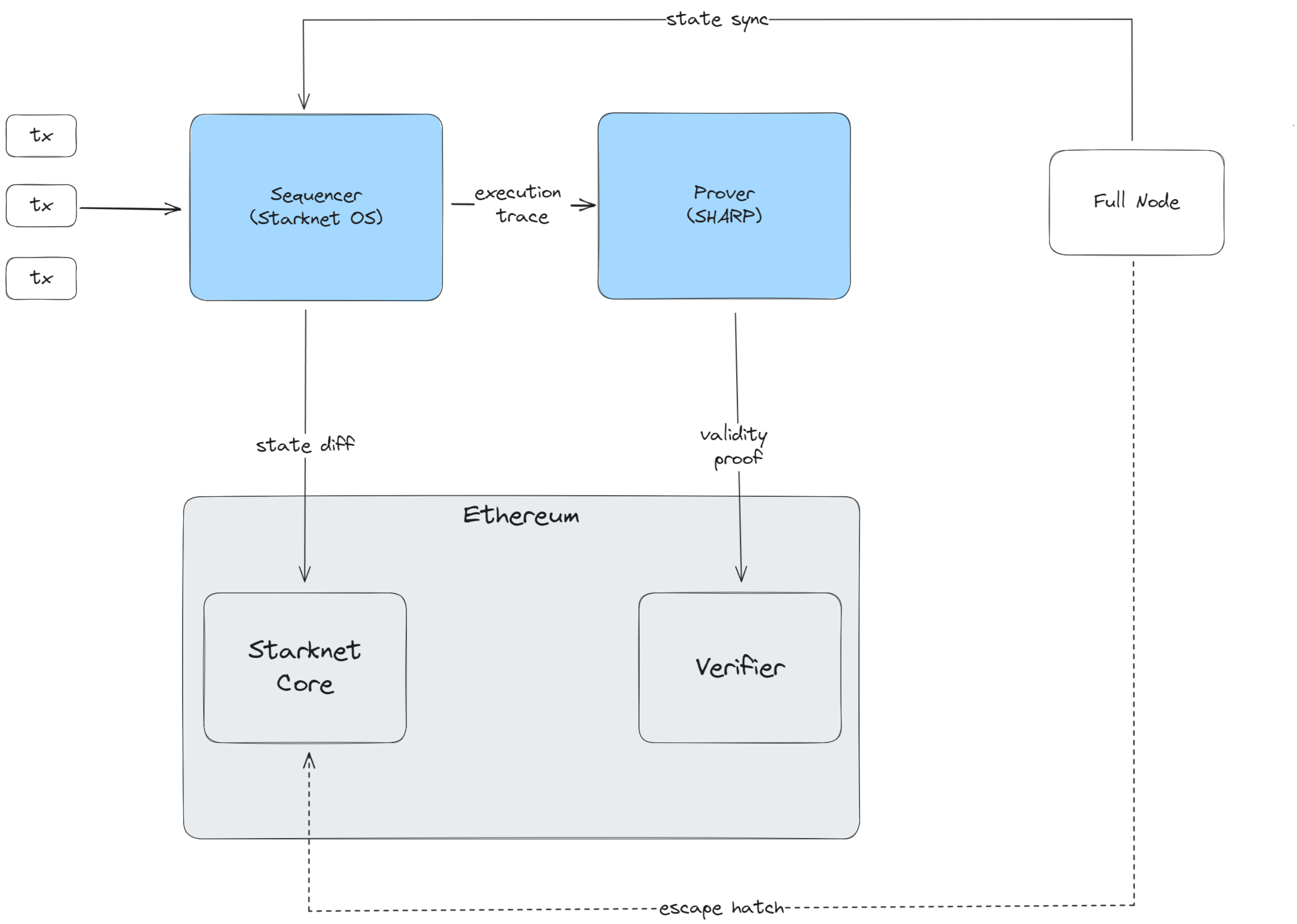
Lesson 4

Cairo / Starknet Introduction

Starknet Architecture Overview



Starknet Components

1. **Prover**: A separate process (either an online service or internal to the node) that receives the execution trace from the Sequencer 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

5. **Starknet Full Nodes:** Can get the current state of the network from the sequencer. If the connection between the Sequencer and the Full Node fails for some reason, you can recreate the L2 current state by indexing date from the **Starknet L1 Core Contract** independently

Safe Intermediate Representation (Sierra)

A new intermediate level representation
Transactions should always be provable, even when a transaction fails
Asserts are converted to if statements, if it returns false we don't do any modifications to storage

Contracts will count gas
Still needs to be low level enough to be efficient
So the process would be
Cairo Smart Contract => Sierra => Cairo Assembly => Validity Proof

Sierra bytecode

- cannot fail
- counts gas
- compiles to Cairo with virtually no overhead


Cairo 1
Compile
Sierra
Compile
CASM
STARKNET

Starknet's current roadmap:

[Starknet Roadmap]

Starknet Open-Source Stack

Category	Project	Entity	Status	Open-Source
Full Node	Pathfinder	Equilibrium	In production	Yes
	Juno	Nethermind	In production	Yes
	Papyrus	StarkWare	Soon in production	Yes
	Deoxys	KasarLabs	In development	Yes
Execution Engine	Blockifier	StarkWare	In production	Yes
	starknet_in_rust	LambdaClass	Soon in production	Yes
Sequencer	SW Sequencer	StarkWare	In production	Yes
	Madara	Community	In development	Yes
	LC Sequencer	LambdaClass	In development	Yes
Prover	Stone	StarkWare	In production	Yes
	Platinum	LambdaClass	In development	Yes
	Sandstorm	Andrew Milson	In development	Yes

Some articles about version 2.0

Extropy [Starknet v0.12 Quantum Leap]

Starkware - [Quantum Leap]

Starknet Forum - [Syntax Proposal]

Starknet Resources

[Starknet Documentation]

[Starknet Book]

Cairo Resources

[Cairo Book]

[Cairo by example]

[Starknet by example]

Introduction to Rust

Core Features

- Memory safety without garbage collection
- Concurrency without data races
- Abstraction without overhead

Variables

Variable bindings are immutable by default, but this can be overridden using the `mut` modifier

```
let x = 1;
let mut y = 1;
```

Types

[Data Types -Rust book]

The Rust compiler can infer the types that you are using, given the information you already gave it.

[Scalar Types]

A *scalar* type represents a single value. Rust has four primary scalar types:

- integers
- floating-point numbers
- booleans
- characters

Integers

For example

```
u8, i32, u64
```

Floating point

Rust also has two primitive types for floating-point numbers, which are numbers with decimal points. Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in size, respectively.
The default type is `f64` because on modern CPUs it's roughly the same speed as `f32` but is capable of more precision. All floating-point types are signed.

boolean

The boolean type or bool is a primitive data type that can take on one of two values, called true and false . (size of 1 byte)

char

`char` in Rust is a unique integral value representing a Unicode Scalar value
Note that unlike C, C++ this cannot be treated as a numeric type.

Other scalar types

usize

`usize` is pointer-sized, thus its actual size depends on the architecture your are compiling your program for
As an example, on a 32 bit x86 computer, `usize = u32`, while on x86_64 computers, `usize = u64`.
`usize` gives you the guarantee to be always big enough to hold any pointer or any offset in a data structure, while `u32` can be too small on some architectures.
Rust states the size of a type is not stable in cross compilations except for primitive types.

Compound Types

Compound types can group multiple values into one type.

- tuples
- arrays
- struct

Tuples

Example

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Struct

```
struct User {
        name : String,
```

```
        age: u32,
        email: String,
}
```

Collections

- Vectors

```
let names = vec!["Bob", "Frank",
"Ferris"];
```

We will cover these in more detail later

Strings

Based on UTF-8 - Unicode Transformation Format

Two string types:

- `&str` a view of a sequence of UTF8 encoded dynamic bytes, stored in binary, stack or heap. Size is unknown and it points to the first byte of the string
- `String`: growable, mutable, owned, UTF-8 encoded string. Always allocated on the heap. Includes capacity i.e. memory allocated for this string.

A String literal is a string slice stored in the application binary (i.e. there at compile time).

String vs str

`String` - heap allocated, growable UTF-8
`&str` - reference to UTF-8 string slice (could be heap, stack ...)

String vs &str - StackOverflow
Rust overview - presentation
Let's Get Rusty - Strings

Arrays

Rust book definition of an array:

"An array is a collection of objects of the same type `T`*, stored in contiguous memory. Arrays are created using brackets* `[]`*, and their length, which is known at compile time, is part of their type signature* `[T; length]`*."*

Array features:

- An array declaration allocates sequential memory blocks.
- Arrays are static. This means that an array once initialized cannot be resized.
- Each memory block represents an array element.
- Array elements are identified by a unique integer called the subscript/ index of the element.
- Populating the array elements is known as array initialization.
- Array element values can be updated or modified but cannot be deleted.

Array declarations

```
//Syntax1: No type definition
let variable_name =
[value1,value2,value3];

let arr = [1,2,3,4,5];

//Syntax2: Data type and size specified
let variable_name:[dataType;size] =
[value1,value2,value3];

let arr:[i32;5] = [1,2,3,4,5];

//Syntax3: Default valued array
let variable_name:[dataType;size] =
[default_value_for_elements,size];

let arr:[i32;3] = [0;3];

// Mutable array
let mut arr_mut:[i32;5] = [1,2,3,4,5];

// Immutable array
let arr_immut:[i32;5] = [1,2,3,4,5];
```

Rust book definition of a slice:

Slices are similar to arrays, but their length is not known at compile time. Instead, a slice is a two-word object, the first word is a pointer to the data, and the second word is the length of the slice. The word size is the same as usize, determined by the processor architecture eg 64 bits on an x86-64.

[Arrays - TutorialsPoint]
[Arrays and Slices - RustBook]

Numeric Literals

The compiler can usually infer the type of an integer literal, but you can add a suffix to specify it, e.g. `42u8`

It usually defaults to `i32` if there is a choice of the type.

Hexadecimal, octal and binary literals are denoted by prefixes
`0x` , `0o`, and `0b` respectively

To make your code more readable you can use underscores with numeric literals
e.g.

```
1_234_567_890
```

ASCII code literals

Byte literals can be used to specify ASCII codes
e.g.
`b'C'`

Conversion between types

Rust is unlike many languages in that it rarely performs implicit conversion between numeric types, if you need to do that, it has to be done explicitly.

To perform casts between types you use the `as` keyword

For example

```
let a = 12;
let b = a as usize;
```

Enums

See docs

Use the keyword `enum`

```
enum Fruit {
    Apple,
    Orange,
    Grape,
}
```

You can then reference the enum with for example

```
Fruit::Orange
```

Functions

Functions are declared with the `fn` keyword, and follow familiar syntax for the parameters and function body.

```
fn my_func(a: u32) -> bool {
    if a == 0 {
        return false;
    }
    a == 7
}
```

As you can see the final line in the function acts as a return from the function
Typically the return keyword is used where we are leaving the function before the end.

Loops

Range:

- inclusive start, exclusive end
  `for n in 1..101 {}`
- inclusive end, inclusive end
  `for n in 1..=101 {}`

- - inclusive end, inclusive end, every 2nd value

```
for n in (1..=101).step_by(2){}
```

We have already seen for loops to loop over a range, other ways to loop include
`loop` - to loop until we hit a `break`
`while` which allows an ending condition to be specified
See [Rust book] for examples.

Control Flow

If expressions

See [Docs]

The `if` keyword is followed by a condition, which *must evaluate to bool* , note that Rust does not automatically convert numerics to bool.

```
if x < 4 {
        println!("lower");
} else {
        println!("higher");
}
```

Note that 'if' is an expression rather than a statement, and as such can return a value to a 'let' statement, such as

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else {
6 };

    println!("The value of number is: {}",
```

```
number);
}
```

Note that the possible values of `number` here need to be of the same type.

We also have `else if` and `else` as we do in other languages.

Printing

```
println!("Hello, world!");

println!("{:?} tokens", 19);
```

Option

We may need to handle situations where a statement or function doesn't return us the value we are expecting, for this we can use Option.

Option is an enum defined in the standard library. The `Option<T>` enum has two variants:

- `None`, to indicate failure or lack of value, and
- `Some(value)`, a tuple struct that wraps a `value` with type `T`.

It is useful in avoiding inadvertently handling null values.

Another useful enum is `Result`

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Matching

A powerful and flexible way to handle different conditions is via the `match` keyword
This is more flexible than an `if` expression in that the condition does not have to be a boolean, and pattern matching is possible.

Match Syntax

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

Match Example

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
```

```
            Coin::Penny => 1,
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => 25,
        }
    }
```

The keyword `match` is followed by an expression, in this case `coin`

The value of this is matched against the 'arms' in the expression.

Each `arm` is made of a pattern and some code

If the value matches the pattern, then the code is executed, each arm is an expression, so the return value of the whole match expression, is the value of the code in the arm that matched.

Matching with `Option`

```
fn main() {
    fn plus_one(x: Option<i32>) ->
Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }

    let five = Some(5);
    let six = plus_one(five);
    let none = plus_one(None);
}
```

Installing Rust

The easiest way is via rustup

See [Docs]

Mac / Linux

```
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Windows

See details [here]

download and run [`rustup-init.exe`].

Other [methods]

Cargo

See the docs

Cargo is the rust package manager, it will

- download and manage your dependencies,
- compile and build your code
- make distributable packages and upload them to public registries.

Some common cargo commands are (see all commands with --list):

```
build, b       Compile the current package
check, c       Analyse the current package
and report errors, but don't build
                         object files
clean          Remove the target directory
doc, d         Build this package's and its
dependencies' documentation
new            Create a new cargo package
init           Create a new cargo package
in an existing directory
add            Add dependencies to a
manifest file
```

```
run, r        Run a binary or example of
the local package
test, t       Run the tests
bench         Run the benchmarks
update        Update dependencies listed
in Cargo.lock
search        Search registry for crates
publish       Package and upload this
package to the registry
install       Install a Rust binary.
Default location is $HOME/.cargo/bin
uninstall     Uninstall a Rust binary
```

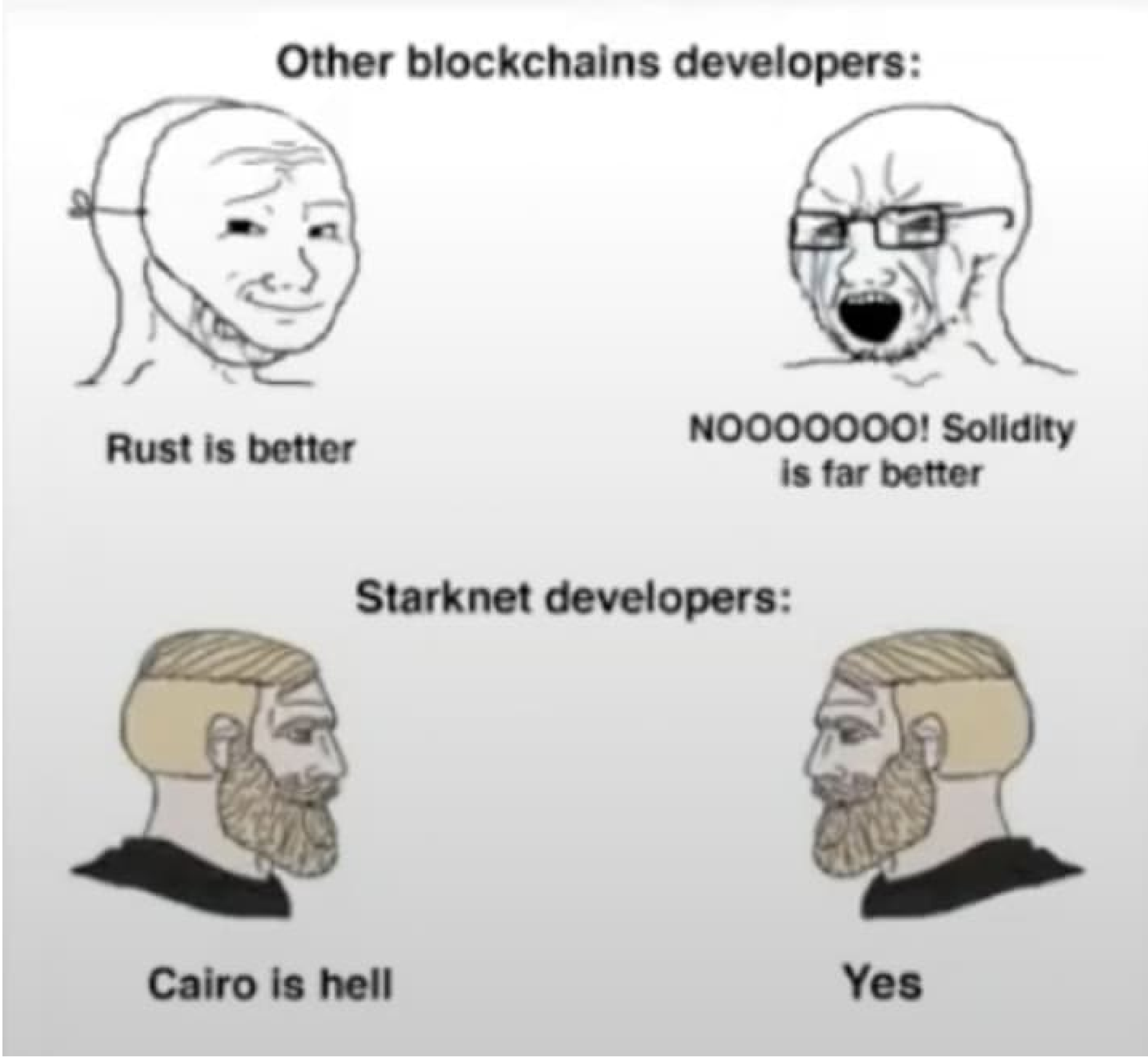
See `cargo help` for more information on a specific command.

Useful Resources

[Rustlings]
Rust by [example]
Rust Lang [Docs]
Rust [Playground]

Rust [Forum]
Rust [Discord]

Cairo



This Cairo 0 meme *should* soon be retired

The new Cairo meme is (by @sylvechv):

Introduction

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

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.
- There is a distinction between Cairo programs (stateless) and Cairo contracts (given storage in the context of Starknet)
- The Cairo white paper is more readable than some, but this describes Cairo version 0

Documentation

The Cairo book is a good resource for the Cairo language

Cairo Language Introduction

From the Introduction in the Cairo book

"Cairo is a programming language designed for a virtual CPU of the same name. The unique aspect of this processor is that it was not created for the physical constraints of our world but for cryptographic ones, making it capable of efficiently proving the execution of any program running on it.
This means that you can perform time consuming operations on a machine you don't trust, and check the result very quickly on a cheaper machine. While Cairo 0 used to be directly compiled to CASM, the Cairo CPU assembly, Cairo 1 is a more high level language.
It first compiles to Sierra, an intermediate representation of Cairo which will compile later down to a safe subset of CASM.
The point of Sierra is to ensure your CASM will always be provable, even when the computation fails."

Cairo is based on Rust syntax and semantics, but there are some differences

Data types

Cairo is a statically typed so the compiler needs to know the data type.

Scalar types

- Felts
  A field element - `felt252`
  Arithmetic on `felt252` is done `mod p` with `p` = $2^{251} + 17 * 2^{192} + 1$
  Division doesn't work as you might expect, since we are dealing with integers.
  In Cairo, the result of x/y is defined to always satisfy the equation $(x/y) * y == x$. If y divides x as integers, you will get the expected result in Cairo (for example `6 / 2` will indeed result in `3`).

- Integers Integer types are based on the `felt252` type

type	size
u8	8 bits
u16	16 bits
u32	32 bits
u64	64 bits
u128	128 bits
u256	256 bits
usize	32 bits

`u256` is implemented as a struct

```
struct u256 {
    low: u128,
    high: u128,
}
```

See github for the details of the Integer type.

Cairo operators

A full list is given in [Appendix B](#) of the Cairo Book

Booleans

```
Booleans are one `felt252` in size.
```

Strings

Strings are not natively supported yet, but there is a short string literal possible by converting to a `felt252`. The maximum length of a short string is `31` characters.

```
let short_string = 'Encode ZKP';
```

Type conversion

You can convert between types with the `try_into` and `into` methods
For example

```
use traits::TryInto;
use traits::Into;
use option::OptionTrait;

fn main() {
    let my_felt252 = 10;
    // Since a felt252 might not fit in a
u8, we need to unwrap the Option<T> type
    let my_u8: u8 =
my_felt252.try_into().unwrap();
    let my_u16: u16 = my_u8.into();
    let my_u32: u32 = my_u16.into();
    let my_u64: u64 = my_u32.into();
    let my_u128: u128 = my_u64.into();
    // As a felt252 is smaller than a
u256, we can use the into() method
    let my_u256: u256 = my_felt252.into();
    let my_usize: usize =
my_felt252.try_into().unwrap();
    let my_other_felt252: felt252 =
my_u8.into();
```

```
    let my_third_felt252: felt252 =
my_u16.into();
}
```

Differences between Cairo and Rust

Loops

Cairo only has one kind of loop for now: `loop`.

```
use debug::PrintTrait;
fn main() {
    let mut counter = 0;

    let result = loop {
        if counter == 10 {
            break counter * 2;
        }
        counter += 1;
    };

    'The result is '.print();
    result.print();
}
```

Collections

```
use dict::Felt252DictTrait;

fn main() {
    let mut balances: Felt252Dict<u64> =
Default::default();

    balances.insert('Alex', 100);
    balances.insert('Maria', 200);

    let alex_balance =
balances.get('Alex');
    assert(alex_balance == 100, 'Balance
is not 100');

    let maria_balance =
balances.get('Maria');
    assert(maria_balance == 200, 'Balance
is not 200');
}
```

`Span` is a struct that represents a snapshot of an `Array`. It is designed to provide safe and controlled access to the elements of an array without modifying the original array. Span is particularly useful for ensuring data integrity and avoiding borrowing issues when passing arrays between functions or when performing read-only operations.

All methods provided by `Array` can also be used with `Span`, with the exception of the `append()` method.

Turning an Array into span

See [Docs]

To create a `Span` of an `Array`, call the `span()` method:

```
let span = array.span();
```

Scarb installation

Follow the installation steps from here

It is recommended to install it via `asdf` as this will help us switch seamlessly to a upgrade/downgrade `scarb` whenever we need.

```
asdf plugin add scarb
asdf install scarb latest
asdf global scarb latest
asdf reshim scarb
```

Once you have completed the installation, we can check if it was successful. Run the following command:

```
scarb --version
>>
scarb 2.3.1 (0c8def3aa 2023-10-31)
cairo: 2.3.1
(https://crates.io/crates/cairo-lang-
compiler/2.3.1)
sierra: 1.3.0
```

> You should receive the version of Scarb along with the version of Cairo.
>
> Note: A good place to check which Cairo version is currently supported on Starknet is the Version notes.

Starkli CLI

Starkli CLI is used to declare and deploy your smart contracts on Starknet. For more information about the installation process please follow the steps from here.

Once installed, run the following command to verify if the installation was successful:

```
starkli --version
>>
0.1.19 (3fd85cf)
```

If the installation has been successful, you should see the installed version of Starkli displayed.

In case you want to find out more about Starkli, you can check the Starkli Book.

A good introduction is given in this article

Creating an account

```
starkli signer keystore new demo-key.json
starkli account oz init demo-account.json
--keystore ./demo-key.json
```

```
starkli account deploy demo-account.json --keystore ./demo-key.json
```

Starknet Foundry Installation

See repo

See Book

Pre requisites

- Rust
- Scarb

`curl -L https://raw.githubusercontent.com/foundry-rs/starknet-foundry/master/scripts/install.sh | sh`

via asdf

`asdf install starknet-foundry latest`

Windows Installation

See details

Forge version:

```
snforge --version
>>
forge 0.9.0
```

Cast version:

```
sncast --version
>>
forge 0.9.0
```

Starknet Foundry consists of two modules:

- Forge - which is a testing framework
- Cast - all-in-one tool for interacting with Starknet smart contracts