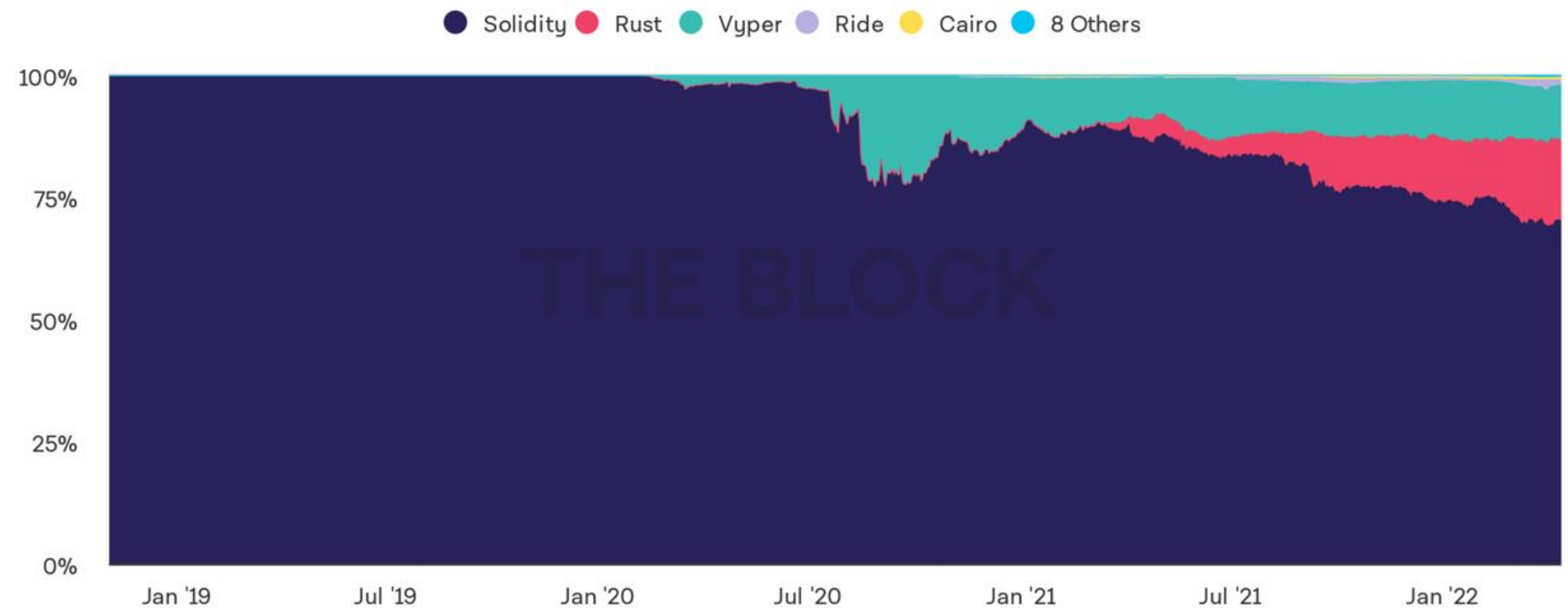
Rust Chapter 1 - 3

'22. 4. 19. Sigrid Jin

Share of Total Value Locked by Smart Contract Language



SOURCE: DEFILLAMA
UPDATED: APR 14, 2022

It stems from the Web History... Thread

As embedded devices like smartphones have evolved and sprung up over the past decade, programming languages supporting various platforms have become important.

Consequently, programming languages such as #JavaScript run on virtual machines have received attention. Even better, a web browser running JavaScript also supports an interface made with HTML/CSS.

Since the Web platform was very convenient to deploy such applications, people wanted to run more complex programs like games or AI models on top of the web browser, which gave birth to Web Graphics Library; WebGL.

cont'd!

However, the most significant overhead in web browsers was **JavaScript** since it has various but unnecessary features like type inference or Garbage Collection to provide a *convenient* user experience. These features make JavaScript very slow.

Therefore, four prominent vendors, which are Google(Chrome), Apple(Safari), Microsoft(Edge), and Mozilla(Firefox), came together to develop a new programming language for the Web. And that is WebAssembly(Wasm).

Wasm is a low-level bytecode format so that developers usually write the code with other high-level programming languages like Rust. Also, Wasm supports server applications like Node.js, a JavaScript runtime supporting server applications.

con'td!

Blockchain mainnets integrate Wasm, and blockchain provides the open financial system to the internet, Web3. As a result, a Rust programmer can be a full-stack developer who can implement business logic on a blockchain, a server application, and an interface.

Rust with Wasm will be full-stack (business logic, server, client) in the Web3 era. However, Rust is hard to learn, so it is important to grow the Rust community and develop some frameworks that make Rust easy to use as TypeScript for JavaScript.

Why Rust for WebAssembly?

- Because languages like C++ or Go have large runtimes that must be included in the WebAssembly binary, they're only really practical for greenfield projects (i.e., they're practical only as JavaScript replacements).
- This Go wiki article on Wasm says the smallest achievable binary size uncompressed is around 2MB; this mirrors what I've seen. For Rust, which ships with an extremely minimal runtime (basically just an allocator), the "hello, world" example compiles to 1.6KB on my machine without any post-compile size optimizations (which could bring it down further).
- "Mozilla is thrilled to join the Rust community in announcing the formation of the Rust Foundation."

Why Rust for Blockchain?

Rust offers zero-cost abstractions and assumes the best practice design and development guidelines as defaults. Programmers only need to be explicit when they have to digress from the best initial choice. As a result, Rust is very fast and memory-efficient, while also very reliable.

Rust also doesn’t have a garbage collector, which means there would be no indeterministic incident (caused by the language, not framework) during the runtime.

What is Cargo?

A build system in addition to a package manager

```
cargo --version
```

```
cargo new hello_cargo
```

cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
authors = ["SongKJ00 <gowithdsm@gmail.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

main.rs

```
fn main() {
  println!("Hello, world!")
}
```

```
$ cargo build // compile
$ ./target/debug/hello_cargo
Hello, world!
```

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
    Running `target/debug/hello_cargo`
Hello, world!
```

Auto Reload

```
fn main() {
    println!("Hello, world!");
    println!("Have a nice day!");
}
$ cargo run
   Compiling hello_cargo v0.1.0 (/Users/jypthemiracle/Desktop/rust-project/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.20s
     Running `target/debug/hello_cargo`
Hello, world!
Have a nice day!
```

```
$ cargo check
    Checking hello_cargo v0.1.0 (/Users/jypthemiracle/Desktop/rust-project/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.07s
```

execution binary to `target/release`

```
$ cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
$ cargo build --release
    Finished release [optimized] target(s) in 0.0 secs
```

숫자 맞히기 게임

요구사항

- 1과 100 사이의 임의의 정수, 난수값을 생성한다.
- 플레이어가 어떤 값을 예상했는지 묻고 플레이어는 예측값을 입력한다.
- 플레이어의 예측값과 생성된 난수값을 비교하여 비교 결과를 알려준다.
- 플레이어의 예측값이 생성된 난수값과 일치하면 축하 메시지를 출력 후 종료한다.

예측값 처리

```
// src/main.rs

use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Fail to read line");

    println!("You guessed: {}", guess);
}
```

Prelude

- 모든 Rust 프로그램이 자동으로 가져오는 것들의 목록.
- Rust는 모든 crate의 root에 `extern crate std;`을 추가하고, 모든 module에 `use std::prelude::v1::*;`을 추가한다.

io

- `io`는 Prelude에 포함되어 있지 않아 우리가 use를 통해 가져왔던 녀석이다.
- 그리고 `stdin`은 방금 본 `String::new`와 마찬가지로 io의 연관 함수다.
- `io::stdin`는 터미널로부터 표준 입력을 받는 핸들의 인스턴스를 반환한다. 그리고 표준 입력 핸들의 메서드 `read_line`을 통해 사용자가 입력한 값을 얻어올 수 있다.

가변성

- 얻어온 값은 `read_line`의 인자로 전달된 변수에 저장된다. `read_line`에 의해 변경되기 때문에 그것의 인자는 가변성을 가지고 있어야 한다.
- `&`는 값을 복사하지 않고 접근하기 위한 참조에 이용되며, 참조는 기본적으로 불변성을 띄기 때문에 가변성을 갖고 참조하겠다는 의미에서 `&mut guess`와 같이 인자를 전달한다.
- 변수를 대여하여 사용할 경우 그것은 기본적으로 불변성을 가진다. 대여한 변수를 수정하고 싶다면 `&`와 함께 `mut` 키워드도 붙여 주어야 한다.
- 물론 가변 참조를 하기 위해서는 원본 변수 자체도 가변성을 띄어야 한다.
- 불변성을 띄는 변수를 가변 참조 하려고 하면 오류가 나는 것을 볼 수 있을 것이다.

```rust
// src/main.rs

fn main() {
    let mut s = String::from("hello");

    change(&mut s);
    println!("string: {}", s);
}

fn change(s: &mut String) {
    s.push_str(", world");
}
```

가변 참조를 사용할 때 유의해야 할 점

- 한 순간에 한 자료에 대한 가변 참조는 최대 하나만 존재할 수 있다.
- `r1`과 `r2`가 둘 다 `s`를 가변 참조한다는 것이 에러의 내용이다.
- 학습 키워드: `데이터 레이스 data race`

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;
```

`println!` ?

- 매크로는 다른 코드를 작성하는 코드다. 이런 작업을 meta-programming이라고 한다.
- 매크로를 사용하면 개발자가 직접 그 코드를 작성하는 것보다 생산적이다.
- 매크로와 함수의 가장 큰 차이 중 하나는 매개변수다. 함수는 그 시그니처에 정해져 있는 매개변수의 자료형과 개수를 정확하게 따라야 한다.
- 반면 매크로는 그 수가 가변적이다. 가령 `println!("hello");` 처럼 하나의 인자를 전달할 수도 있고, `println!("hello {}", name);` 처럼 두 개의 인자를 전달할 수도 있다.
- 그리고 실행 시간에 호출되는 함수와 달리 매크로는 컴파일러가 코드의 의미를 해석하기 전에 그것이 나타내는 코드로 대치된다. 이는 함수가 하지 못하는 몇몇 기능들을 하게 해준다.
- `주어진 자료형에 어떤 트레이트를 구현한다거나?`

`let mut guess = String::new();`

- Rust에서 변수는 `let` 키워드를 통해 선언한다.
- Rust의 변수는 기본적으로 불변성을 가지고 있으며 `mut` 키워드로 가변성을 명시해주어야 그 값을 변경할 수 있다.
- `=` 는 우측값을 좌측값에 `bind` 하겠다는 의미인데, `guess` 변수에 `String::new` 함수의 반환값을 `bind` 한다는 것이다.

```
// src/main.rs

fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}

// error: cannot assign twice to immutable variable
```

- OK

```
// src/main.rs

fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

`mut` 키워드를 추가한 것만으로 문제 없이 잘 실행되는 것을 확인할 수 있다.

그런데 불변성을 가진 변수는 다른 언어에서의 상수와 같은 개념이라고 생각해도 될까?

정답은 `NO`다. Rust의 상수는 따로 존재한다.

- `let` 대신 `const` 키워드를 통해 선언되며 `mut` 키워드를 사용할 수 없다.
- 그리고 컴파일 시점에 자료형이 명확하면 자료형을 생략할 수 있는 변수와 달리 상수의 경우 항상 자료형을 명시해주어야 한다.
- 상수는 전역 범위를 포함하여 어디에서나 선언 가능하며, 반드시 상수 리터럴을 bind해야 한다.
- 상수의 식별자는 대문자만을 사용하며, 여러 단어로 이루어질 경우 `_`로 구분한다.

난수 생성

- 1과 100 사이의 임의의 정수, 난수값을 생성한다.
- 크레이트 `crate`는 라이브러리 또는 바이너리 소스 파일의 집합
- 안에 들어 있는 소스 코드를 가져다 쓸 수 있는 라이브러리 크레이트와 `main` 함수가 있어 실행 가능한 바이너리 크레이트가 존재하는데, `rand` 크레이트는 라이브러리 크레이트에 해당한다.
- `rand` 크레이트와 같은 외부 크레이트를 사용하려면 `Cargo.toml`에 그것을 의존 패키지로 등록해주어야 한다. `Cargo.toml`의 `[dependencies]` 아래에 의존 패키지를 추가할 수 있다.
- `cargo build`가 처음 실행되었을 때 이 연관된 패키지들의 버전이 `Cargo.lock`에 기록되는데 우리는 `Cargo.lock`으로 인해 의존 패키지가 업데이트되었을 때 업데이트 버전이 우리 프로젝트와 호환되지 않더라도 기존 버전을 유지하며 사용할 수 있다.
- Cargo.toml에 지정된 조건에 맞는 가장 최신 버전을 사용하고 싶다면 `cargo update` 가능

```rust
use std::io;
use rand::Rng; // 난수 생성을 위해 use rand::Rng;가 추가되었다.
// Rust의 난수 생성기는 운영체제로부터 시드 값을 받는다.

fn main() {
    println!("Guess the number!");

    let secret_number =

        rand::thread_rng() // 난수 생성기를 반환
        .gen_range(1, 101); // 난수를 생성한다.

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Fail to read line");

    println!("You guessed: {}", guess);
}
```

비교하기

- 플레이어의 예측값과 생성된 난수값을 비교하여 비교 결과를 알려준다.
- 표준 입력은 문자열의 형태로 읽어온다. 즉, 난수 정수값과 직접 비교하려고 하면 자료형이 다르기 때문에 불가능하다.

src/main.rs

```rust
use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    // 컴파일 시점에 자료형이 명확한 경우에만 자료형을 생략할 수 있다.
    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Fail to read line");

    // Rust는 shadowing이라는 기능으로 기존의 값을 가리고 같은 이름의 변수를 선언할 수 있다.
    // shadowing을 사용하면 기존의 값은 그 변수의 이름으로 접근할 수 없다.
    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

    println!("You guessed: {}", guess);
}
```

Shadowing?

- 새로 선언한 변수로 기존의 변수의 값을 가리는 것
- 기존에 선언된 변수와 같은 이름의 변수를 let으로 선언하면 그 변수는 가려진다.
- 대입 연산은 `=` 의 우측 연산이 먼저 수행된 후 값을 `bind` 하므로 `shadowing` 을 할 때 기존 변수의 값을 활용할 수 있다.
- `shadowing` 없이 그냥 대입할 땐 기존의 것과 자료형이 같아야 하지만, `shadowing` 을 할 경우 서로 다른 자료형이더라도 상관 없다.

Shadowing?

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;

    println!("The value of x is {}", x);

    let spaces = "    ";
    let spaces = spaces.len();

    println!("The value of spaces is {}", spaces);
}

// The value of x is 12
// The value of spaces is 4
```

예측값과 난수값의 비교

```
use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1, 101);

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Fail to read line");

    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {

        // Ordering은 Result와 마찬가지로 열거형이다.

        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

반복하기

- 유의미한 게임이 되도록 하기 위해서 반복문을 사용하여 플레이어의 예측값이 생성된 난수값과 일치하면 축하 메시지를 출력 후 종료한다. 에 해당하는 부분을 작성한다.

```
// loop를 사용하여 무한 반복을 하게 수정

loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Fail to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        // match 구문의 Ordring::Equal에 해당하는 arm의 실행문을 코드블록으로 바꾸어
        // 반복에서 벗어나기 위한 break;를 추가하였다.
        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            },
        }
    }
```

IO 에러 핸들링

- 프로그램은 사용자가 숫자가 아닌 값을 입력하면 사용자가 다시 입력하도록 한다.
- 이제 `guess.trim().parse()` 가 성공했을 경우에만 그 결과가 guess에 `bind` 되고 그렇지 않을 경우 `continue` 를 만나 반복문의 시작 부분으로 돌아간다.

```
println!("Please input your guess.");

let mut guess = String::new();

io::stdin()
    .read_line(&mut guess)
    .expect("Fail to read line");

let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

println!("You guessed: {}", guess);
```

Data Types

- Rust는 컴파일 시점에 모든 변수의 자료형이 결정되어야 하는 정적 타입 언어다.
- 따라서 모든 변수의 자료형이 확정되어야 정상적으로 컴파일된다.
- 자료형이 명시되지 않아도 컴파일 시점에 자료형이 명확한 변수에 대해서는 Rust 컴파일러가 자동으로 자료형을 지정해주지만 그렇지 않은 경우에는 반드시 자료형을 직접 명시해주어야 한다.

Scalar 자료형

정수 자료형 Integer

- i8, i16, i32, u8, u16, u32
- 만약 우리가 정수를 사용할 때 자료형을 명시해주지 않는다면 Rust 컴파일러는 기본적으로 i32로 선언한다.

부동소수점 자료형 Floating Point

- 현재 대부분의 CPU에서는 f64 연산을 f32만큼 빠르게 수행할 수 있는데, 정확도는 `f64`가 더 높으므로 Rust는 부동 소수점 기본 자료형으로 `f64`를 사용한다.

문자 자료형 Character

Compound 자료형

튜플 자료형 Tuple

튜플은 서로 다른 자료형의 여러 값을 하나의 자료형으로 그룹화할 때 사용한다.

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
let (x, y, z) = tup;
println!("The value of x is: {}", x);
println!("The value of y is: {}", y);
println!("The value of z is: {}", z);

// tup.0
```

배열 자료형 Array

- 배열은 튜플과 달리 동일한 자료형의 여러 값을 하나의 자료형으로 그룹화할 때 사용한다.
- 어떤 언어의 배열은 그 길이가 유동적이지만 Rust의 배열은 고정되어 있다.
- 튜플은 소괄호(`()`)를 통해 생성하는 반면 배열은 대괄호(`[]`)를 통해 생성한다.
- 배열을 선언할 때 자료형을 명시해줄 경우 `let arr: [i32; 5];` 와 같이 그 크기와 함께 명시한다.
- 배열의 초기값을 설정할 때 모두 동일한 값을 사용하고 싶다면, `let arr = [0; 3];` 와 같이 작성하면 `let arr = [0, 0, 0];` 와 동일한 배열이 생성된다.
- 원소 접근 시 배열 `arr` 에 대해 `arr[0]` 와 같이 `[]` 에 인덱스를 지정하여 접근한다.

함수의 본문

```
fn main() {
    let x = 5;

    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of x is {}", x);
    println!("The value of y is {}", y);
}
```

- 구문은 세미콜론으로 끝나며 어떤 동작을 실행하는 것 그 자체로, 결과값이 존재하지 않는다.
- 반면 표현식은 세미콜론을 찍지 않으며 어떤 결과값을 가진다.
- 반환값에 해당하는 부분은 표현식이므로 세미콜론을 붙이면 안된다는 점을 유의하자.
- 구문과 마찬가지로 반환값이 없는 함수의 반환값을 bind하려고 하면 빈 튜플 `()` 이 `bind` 될 것이다. 반환값의 자료형을 명시할 땐 함수 소괄호와 중괄호 사이에 `->` 과 함께 적는다.

```
fn main() {
    let x = plus_one(5);
    println!("The value of x: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}

// The value of x: 6
```

if without else in Rust

```
// error[E0317]: `if` may be missing an `else` clause
fn return_number(is_even: bool) -> u32 {
    if is_even {
        42
    // } else {
    //     43
    // }
}

fn main() {
    println!("{}", return_number(true));
    println!("{}", return_number(false));
}
```

```
fn return_number(is_even: bool) -> u32 {
    if is_even {
        42
    }; // semicolon ignores the result!
    43
}
```