Smart Contract e Tokens

Thiago Alexsander da Costa Pereira[1]
[1]Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

This article aims to present and explain how an ERC-20 standard smart contract works on the *Ethereum blockchain*. It includes a step-by-step guide and explanations about the contract's functionality and how the Solidity language is used to create the contract. The entire project can be created and tested in Remix IDE.

Resumo

Esse artigo tem como objetivo apresentar e explicar como funciona um contrato inteligente no padrão ERC-20 na *blockchain Ethereum*. Contém um passo a passo e explicações do funcionamento do contrato e de como funciona a linguagem Solidity para criação do contrato. O projeto todos pode ser criado e testado na Remix IDE

1 Introdução

A tecnologia de *blockchain* é baseada em hashing, sendo assim a fundação para troca de criptomoedas e execução de contratos inteligentes. Essa tecnologia pode ser definida como um registro digital decentralizado distribuído, no qual o histórico e as informações de todas as transações feitas são mantidas em todos os computadores envolvidos na transação ??.

Ethereum é um protocolo de *blockchain* que pode ser usado por qualquer pessoa para ou serviço para criação e transação de espólios digitais.

1.1 Smart contract

Uma das aplicações da que podem ser realizadas com *blockchain* é a criação de Smart Contracts (Contratos inteligentes). Um Smart contract é um programa que é auto-executado na *blockchain*. E automaticamente impõe e executa contratos caso uma condição especificada previamente seja atingida. Além da automatização e a da decentralização, os contratos inteligentes também fornecem transparência, uma vez que são públicos na *blockchain*, imutabilidade, não sendo possível altera-lo depois de colocado na *blockchain* Antonopoulos & Wood (2018).

Os contratos inteligentes podem ser usados para diversas atividades, como serviços financeiros, criptomoedas decentralizdas, tokens e até jogos como jogos baseados em NFTs.

Corresponding author(s): Thiago Alexsander da Costa Pereira, Universidade Tecnológica Federal do Paraná

1.2 Tokens

Tokens são espólios digitais que existem na *blockchain*, podendo representar criptomoedas, certificados, acesso a serviços, contratos legais entre outros. São criados e geridos por contratos inteligentes, no qual deve especificar todos os detalhes de como funciona o token.

Existem dois tipos de tokens, *Fungible* (ERC-20) e *Non-Fungible* (ERC-721), de forma simplória, Fungible tokens são tokens que podem ser trocados por outros tokens do mesmo tipo e não perder valor na troca, um exemplo desse tipo de token são criptomoedas como Bitcoin e *Ethereum*. Por outro lado *Fungible* tokens são tokens unicos na *blockchain*, sendo assim nenhum outro token é igual, podendo então ter valores monetários diferentes, um exemplo de *fungible* tokens são NFTs.

1.3 Ferramentas para o desenvolvimento

Ethereum: Plataforma *open-source* de *blockchain* que permite criar e implantar contratos inteligentes, que serão executado na *blockchain* https://ethereum.org/en/what-is-ethereum/.

Solidity: Linguagem de programação desenvolvida para a criação de contratos inteligentes que são executados na *Ethereum blockchain* https://soliditylang.org/. Código 1 apresenta um código de exemplo da linguagem.

```
1  // SPDX-License-Identifier: GPL-3.0
2
3  pragma solidity >=0.8.2 <0.9.0;
4
5  contract Storage {
6
7      uint256 number;
8
9      function store(uint256 num) public {
10         number = num;
11     }
12
13     function retrieve() public view returns (uint256){
14         return number;
15     }
16 }
```

Código 1: Código Solidity

Remix IDE: Remix IDE é uma IDE para desenvolvimento de contratos inteligentes, podendo ser usado totalmente online pelo browser ou pelo desktop https://remix-project.org/?lang=en. Um exemplo de como é a IDE pode ser visto na Figura 1

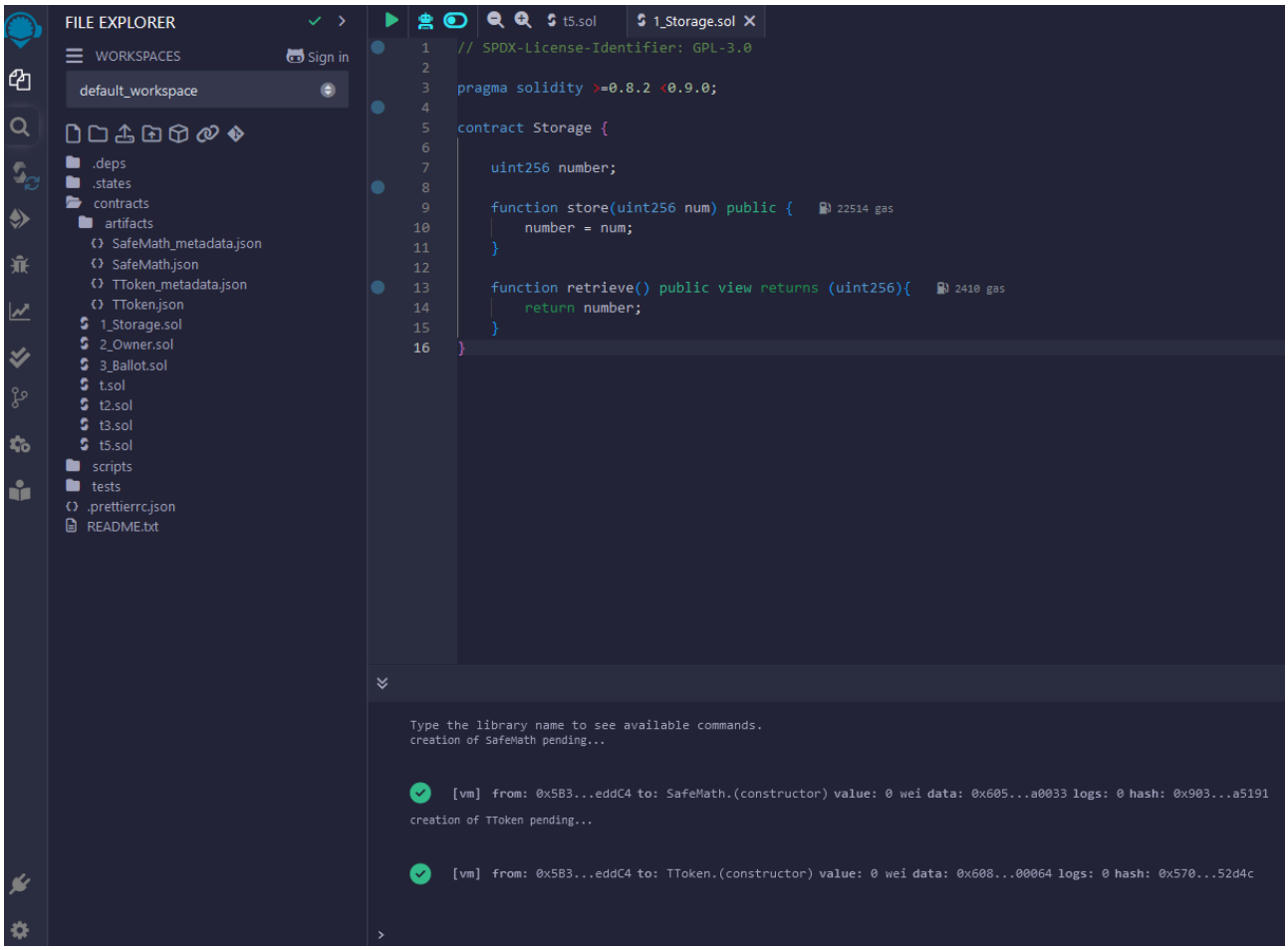

Figura 1: Remix IDE

1.4 Criação de Fungible token

O desenvolvimento de um contrato inteligente seguindo as especificações do ERC-20 ethereum (2024). Um contrato inteligente do tipo ERC-20 precisa implementar as seguintes especificações:

- Token Name: Nome do Token, exemplo ”TToken”
- Symbol: Simbolo que representa o token em três letras, exemplo: ”TTK”
- Decimal Points: Número de casas decimais para representar o token. Um número comun é 18.
- Total Supply: Número de tokens que será criado.
- BalanceOf: Deve retornar o saldo de tokens.
- Transfer: Transfere tokens do endereço que chama para outro endereço.
- Approval: Permite que um endereço gaste tokens em seu nome.
- Allowance: Número de tokens que o endereço pode usar em nome de outro endereço.
- TransferFrom: Transfere tokens entre endereços usando o allowance

Na IDE Remix crie um arquivo chamado token.sol na pasta contracts em seguida crie um contrato como mostra o código 2.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.20;

contract TToken {
    string public constant name = "TToken";
    string public constant symbol = "TTK";
    uint8 public constant decimals = 18;
}
```

Código 2: Código Contrato inteligente

Nessa etapa é criado o contrato com o nome de TToken, dentro deste contrato são instanciados o nome, simbolo e o número de casas decimais suportados pelo Token.

Em seguida, adicione as variáveis de saldo (*balances*), máximo de transação de token disponível (*allowed*) e número total de Tokens disponível (*totalSupply*) na *blockchain*, como mostra o Código 3.

No *solidity*, *Hashmaps* são chamadas de *mappings* e funcionam do mesmo jeito, essas hashs são usadas para associar um endereço com algum valor dentro do contrato.

```
    mapping(address => uint256) balances;

    mapping(address => mapping (address => uint256)) allowed;

    uint256 totalSupply_;
```

Código 3: Código Contrato inteligente

Eventos: Um evento na linguagem *Solidity* é uma maneira de criar um Registro (Log) que ficará na *blockchain* e acessível fora do contrato inteligente, definida com a *keyword event* antes da função alchemy (2024).

Esses logs não são acessíveis pelo contrato, pois dessa forma o contrato custa menos gás e esses dados podem ficar guardados na *blockchain* para outras buscas mais simples e que utilizam menos gás. Para chamar uma função de evento basta chamar o nome da função com a *keyword emit*.

O código 4 cria dois eventos; *Approval* para quando o endereço dono do token, aprova que outro endereço envie Tokens em seu nome. *Transfer* que é emitido quando um token é enviado de uma conta para outra.

```
    event Approval(address indexed tokenOwner, address indexed spender, uint
        tokens);
    event Transfer(address indexed from, address indexed to, uint tokens);
```

Código 4: Código Contrato inteligente

O próximo passo é criar o construtor, ele receberá a quantidade de tokens disponíveis, e coloca-los no endereço responsável pelo *deploy* do contrato. Como mostra o Código 5 o construtor recebe um número total de tokens iniciais, e instancia o *totalSupply* com este valor, e na *hash* do saldo coloca todo esse valor para o endereço criador do contrato.

```
    constructor(uint256 total) {
     totalSupply_ = total;
     balances[msg.sender] = totalSupply_;
    }
```

Código 5: Código Contrato inteligente

Funções do ERC-20:

No código 6 estão detalhadas as funções de *totalSupply*, *balance allowance* são simples, *totalSupply* retorna o valor inicial de tokens do contrato, *balanceOf* se utiliza da hash balances para buscar a

quantidade de tokens que o endereço possui, allowance retorna o número de tokens que aquele endereço ainda pode gastar e a função *approve* permite que um endereço gaste um quantidade de tokens especificada de outro endereço, quando isso acontece ela emite um registro de *Approval* com as informações.

```
1     function totalSupply() public view returns (uint256) {
2       return totalSupply_;
3     }
4
5     function balanceOf(address tokenOwner) public view returns (uint) {
6         return balances[tokenOwner];
7     }
8
9     function allowance(address owner, address delegate) public view returns (
          uint) {
10        return allowed[owner][delegate];
11    }
12
13    function approve(address delegate, uint numTokens) public returns (bool) {
14        allowed[msg.sender][delegate] = numTokens;
15        emit Approval(msg.sender, delegate, numTokens);
16        return true;
17    }
```

Código 6: Funções

A função *transfer* como mostra o Código 7 transfere uma quantidade de tokens de um endereço para outro, se utilizando das hashes e dá função *.sub e .add*, que subtrai e adiciona respectivamente, essas funções são especiais e serão explicadas mais adiante. Essa função faz o uso da o *emit Transfer* para registrar uma transação de tokens entre endereços e a quantidade de tokens que foi transferido.

```
1    function transfer(address receiver, uint numTokens) public returns (bool) {
2        require(numTokens <= balances[msg.sender]);
3        balances[msg.sender] = balances[msg.sender].sub(numTokens);
4        balances[receiver] = balances[receiver].add(numTokens);
5        emit Transfer(msg.sender, receiver, numTokens);
6        return true;
7    }
```

Código 7: Funções

Na função *transferFrom* no Código 8 que permite que um endereço permitido faça transferências de um endereço para outro, nela é possível ver que é preciso verificar com require, se o endereço dono dos tokens possui os tokens necessários para a transferência e se o endereço que está tentando realizar a transferência tem permissão do endereço dono para tal.

```
1  function transferFrom(address owner, address buyer, uint numTokens) public
       returns (bool) {
2        require(numTokens <= balances[owner]);
3        require(numTokens <= allowed[owner][msg.sender]);
4
5        balances[owner] = balances[owner].sub(numTokens);
6        allowed[owner][msg.sender] = allowed[owner][msg.sender].sub(numTokens);
7        balances[buyer] = balances[buyer].add(numTokens);
8        emit Transfer(owner, buyer, numTokens);
9        return true;
10   }
```

Código 8: Funções

A biblioteca *SafeMath* no Código 9 utilizada para subtrair e adicionar tokens nas operações, ela foi criada justamente para que erros de *underflow* ou *overflow* de inteiros não ocorram. As funções são bem simples apenas fazendo *asserts* para verificação das condições.

```
library SafeMath {
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
      assert(b <= a);
      return a - b;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
      uint256 c = a + b;
      assert(c >= a);
      return c;
    }
}
```

Código 9: Funções

O Código 10 apresenta o contrato inteiro.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.20;

contract TToken {

    string public constant name = "TToken";
    string public constant symbol = "TTK";
    uint8 public constant decimals = 18;


    mapping(address => uint256) balances;

    mapping(address => mapping (address => uint256)) allowed;

    uint256 totalSupply_;

    using SafeMath for uint256;

    event Approval(address indexed tokenOwner, address indexed spender, uint
       tokens);
    event Transfer(address indexed from, address indexed to, uint tokens);



   constructor(uint256 total) {
  totalSupply_ = total;
  balances[msg.sender] = totalSupply_;
    }

    function totalSupply() public view returns (uint256) {
  return totalSupply_;
    }

    function balanceOf(address tokenOwner) public view returns (uint) {
        return balances[tokenOwner];
    }

    function transfer(address receiver, uint numTokens) public returns (bool) {
        require(numTokens <= balances[msg.sender]);
        balances[msg.sender] = balances[msg.sender].sub(numTokens);
        balances[receiver] = balances[receiver].add(numTokens);
```

```
41          emit Transfer(msg.sender, receiver, numTokens);
42          return true;
43      }
44
45      function approve(address delegate, uint numTokens) public returns (bool) {
46          allowed[msg.sender][delegate] = numTokens;
47          emit Approval(msg.sender, delegate, numTokens);
48          return true;
49      }
50
51      function allowance(address owner, address delegate) public view returns (
            uint) {
52          return allowed[owner][delegate];
53      }
54
55      function transferFrom(address owner, address buyer, uint numTokens) public
            returns (bool) {
56          require(numTokens <= balances[owner]);
57          require(numTokens <= allowed[owner][msg.sender]);
58
59          balances[owner] = balances[owner].sub(numTokens);
60          allowed[owner][msg.sender] = allowed[owner][msg.sender].sub(numTokens);
61          balances[buyer] = balances[buyer].add(numTokens);
62          emit Transfer(owner, buyer, numTokens);
63          return true;
64      }
65  }
66
67  library SafeMath {
68      function sub(uint256 a, uint256 b) internal pure returns (uint256) {
69        assert(b <= a);
70        return a - b;
71      }
72
73      function add(uint256 a, uint256 b) internal pure returns (uint256) {
74        uint256 c = a + b;
75        assert(c >= a);
76        return c;
77      }
78  }
```

Código 10: Código completo

1.5 Deploy do contrato

No remix vá até a aba de compilação, selecione o commit ”0.8.26”, em configurações avançadas selecione a versão da EVM(*Ethereum* virtual machine) como defaul cancun, por ultimo selecione o contrato para compilar, no caso TToken (Mesmo nome da classe) por ultimo clique em compilar. A figura 2 apresenta esse menu de configuração para a compilação do contrato.

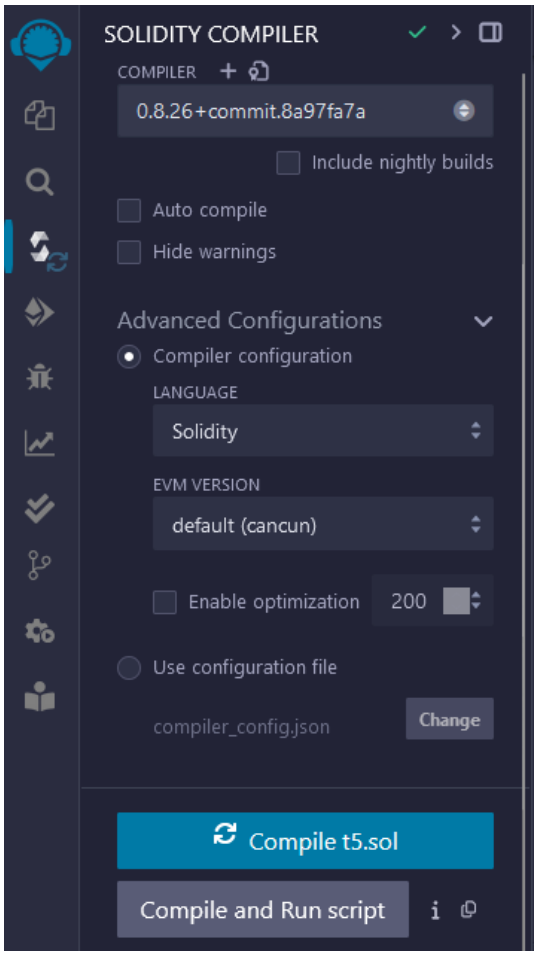

Figura 2: Remix menu de compilação

No menu de Deploy do remix selecione o enviroment Remix VM (cancun), selecione o contrato TToken, este é o contrato que será usado para o deploy. No botão de Deploy lembre-se de iniciar o deploy com um número de tokens disponíveis neste caso 200. A figura 3 mostra como deve estar configurado.

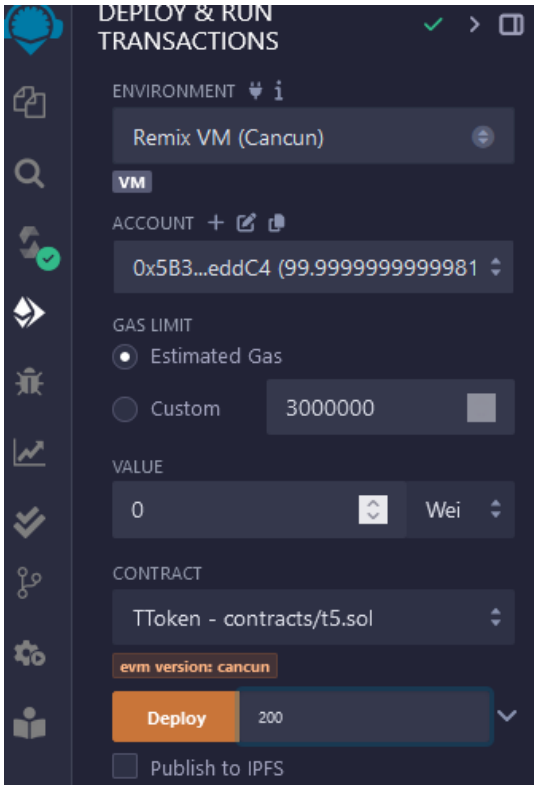

Figura 3: Remix menu de Deploy

Após clicar em deploy será possível ver o output de debug no terminal do remix, a figura 4 mostra o resultado dessa ação, ao clicar o log se expande para mostrar mais informações sobre o deploy.

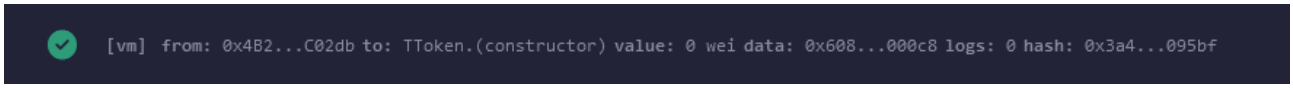

Figura 4: Remix deploy log

Ainda no menu de deploy é possível realizar operações sobre o contrato, como, transferir tokens, checar quantidade de tokens disponível e as outras operações que foram implementadas no contrato.

Na parte superior do menu de deploy, existe uma lista de ”accounts”, endereços que podem ser utilizados para testar essas operações, o primeiro endereço ficou como o endereço que criou o contrato e então possui todos os tokens iniciais, para testar, copie este primeiro endereço da conta e cole na função balanceOf e a execute. A figura 5 apresenta o resultado da execução dessa função, apresentando o que este endereço possui 200 tokens.

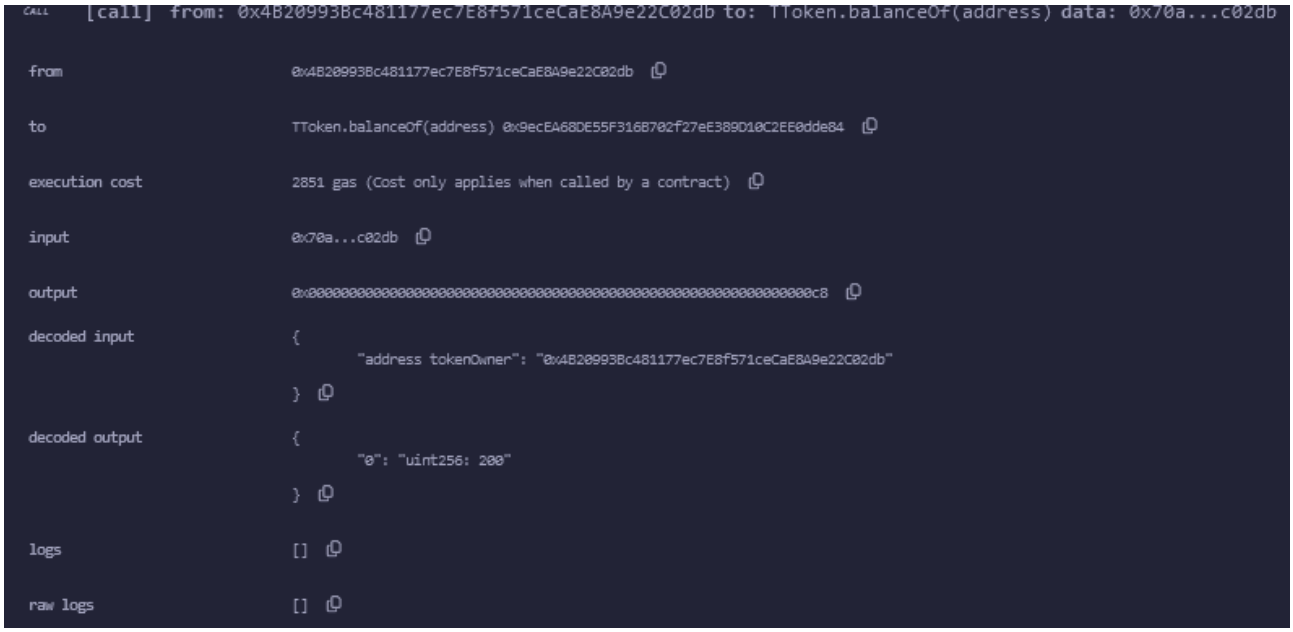

Figura 5: Remix balanceOf log

Agora, copie o endereço de outra conta na disponível na lista para fazer a transferência de tokens dessa conta para outra. A figura 6 demonstra a execução dessa operação, transferindo 50 tokens, Após a transferência ser concluída repita o passo anterior para as duas contas e verá que uma possui 150 tokens e a outra possui 50.

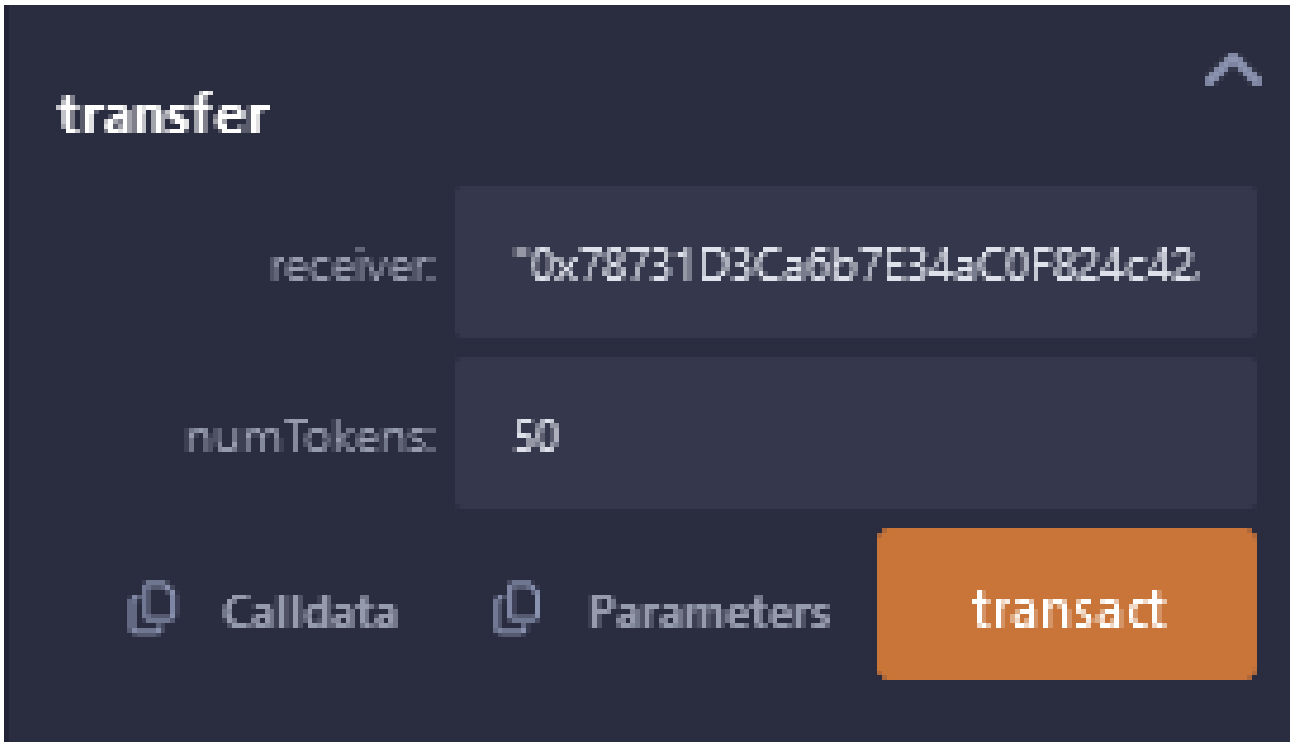

Figura 6: Remix função transfer tokens

1.6 Considerações finais

E é apenas uma pequena parte do que os contratos inteligentes podem oferecer, como dito anteriormente eles permitem fazer muitas outras coisas, como NFTs e sistemas de votações. Esse contrato também pode ser recriado utilizando o Ganache CLI e o MetaMask, bastando apenas modificar na IDE do Remix qual Environment será utilizado para o deploy do contrato.

Referências

alchemy. 2024. Alchemy solidity basics. https://docs.alchemy.com [Accessed: (03/12/2024)].

Antonopoulos, A.M. & G. Wood. 2018. *Mastering ethereum: Building smart contracts and dapps*. O'Reilly Media, Incorporated. https://books.google.com.br/books?id=SedSMQAACAAJ.

ethereum. 2024. Understand the erc-20 token smart contract. https://ethereum.org/en/developers/tutorials/understand-the-erc-20-token-smart-contract/ [Accessed: (03/12/2024)].