Decentralized Finance and the Future of Money

Exercise 1: Smart Contracts

Luca Brilhaus

This report was written as a graded semester performance for the course "Decentralized Finance and the Future of Money", lectured by Prof. Dr. Hans Gersbach, Prof. Dr. Roger Wattenhofer and Dr. Bastian J. Bergmann

Spring semester 2023

ETH Zurich

Task 1

(4 points) Write an ERC20 token contract for your own token and include one additional functionality of your choice (e.g., a function which lets the token holder with the most tokens change the token name). Deploy the token and add its information to the shared drive (include the token code and address in your report).

For the sake of transparency, all transactions performed were executed using the EOA:

0x6b604a2A8134EBf49a6e90e946404145904BF858

In memory of my dog Anton, I created the ERC20 token Anton. Anton was born on July 21, 2004; hence, there is a total supply of 21'072'004 coins, and each coin has three decimals. Since Anton was a very good dog (like all other dogs), there should be a way of telling him (you can never tell them often enough). That is why I added the function sayGoodBoy(), telling Anton that he is a good boy. Anton can be found under the contract account address

0x5b67efeA0222aE80bf8263dA7114008112D79c34

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
import "./interfaces/IERC20.sol";
import "./libraries/SafeMath.sol";

contract AntonCoin is IERC20 {
  using SafeMath for uint256;
  uint256 public constant _totalSupply = 21072004*10**3;
  string public constant name = 'Anton';
  uint8 public constant decimals = 3;
  string public constant symbol = 'ANTON';
  mapping (address => uint256) private _balances;
  mapping (address => mapping (address => uint256)) private _allowed;
```

```
constructor () {
  _balances[msg.sender] = _totalSupply;
  emit Transfer(address(0), msg.sender , _totalSupply);
}
function totalSupply() external override pure returns (uint256) {
  return _totalSupply;
}
function balanceOf(address owner) public override view returns (uint256)
    {
  return _balances[owner];
}
/**
* Function to check the amount of tokens that an owner allowed to a
    spender.
*/
function allowance(address owner,address spender)public override view
    returns (uint256){
  return _allowed[owner][spender];
}
function transfer(address to, uint256 value) public override returns
    (bool) {
  require(value <= _balances[msg.sender]);
  require(to != address(0));
  _balances[msg.sender] = _balances[msg.sender].sub(value);
  _balances[to] = _balances[to].add(value);
  emit Transfer(msg.sender, to, value);
  return true;
}
function approve(address spender, uint256 value) public override returns
    (bool) {
  require(spender != address(0));
  _allowed[msg.sender][spender] = value;
  emit Approval(msg.sender, spender, value);
  return true;
```

```
}
function _transfer(address from, address to, uint value) private {
    _balances[from] = _balances[from].sub(value);
    _balances[to] = _balances[to].add(value);
    emit Transfer(from, to, value);
  }
function transferFrom( address from, address to, uint256 value) public
    override returns (bool){
  require(value <= _balances[from]);
  require(value <= _allowed[from][msg.sender]);
  require(to != address(0));
  _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
  _transfer(from, to, value);
  return true;
}
function increaseAllowance( address spender, uint256 addedValue ) public
    returns (bool){
  require(spender != address(0));

  _allowed[msg.sender][spender] = (
    _allowed[msg.sender][spender].add(addedValue));
  emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
  return true;
}
function decreaseAllowance(address spender,uint256 subtractedValue)
    public returns (bool){
  require(spender != address(0));
  _allowed[msg.sender][spender] = (
    _allowed[msg.sender][spender].sub(subtractedValue));
  emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
  return true;
}
function sayGoodBoy() public pure returns (string memory){
     return "You're a good boy!";
```

```
    }
}
```

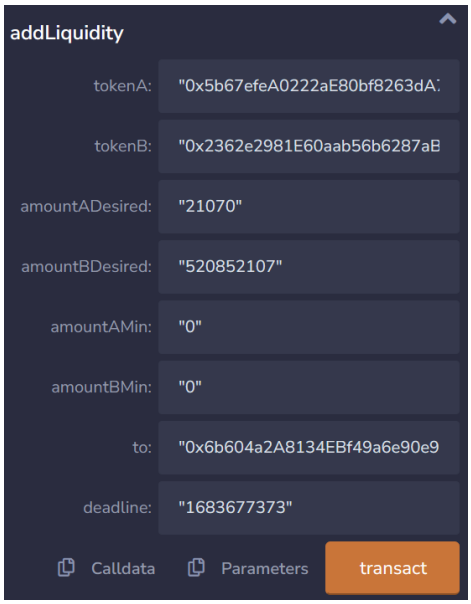
Solidity code for the ERC20 token ANTON

Task 2

(3 points) Add liquidity to a pair with your token (adding liquidity using the router will create the pair if it does not exist, you can also create the pair separately using the factory contract). In order to add liquidity or do any trades, you need to first give permission for the router to make trades on your behalf by calling the allow function of the token(s) smart contract.

In order to add liquidity to the ANTON-WETH pool, I allowed the factory and router to spend ANTON and WETH using their ERC20 approve function. For simplicity, I allowed both to spend the maximum amount of available ANTON and WETH in my Metamask wallet. The pool was created using the addLiquidity function of the router. I instantiated the pool with 21.070 ANTON and $5.20852107 * 10^{-10}$ WETH. The pool address is:

0x563b2008674cE273aAB1d4D0508B2365bF2e3B55

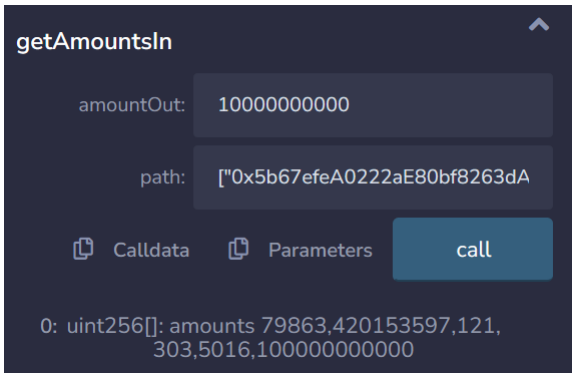


Task 3

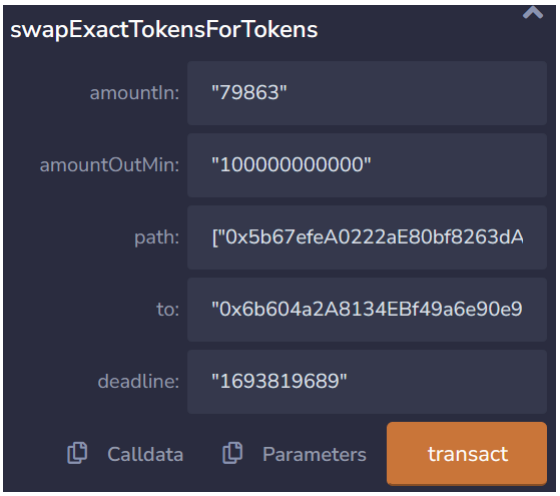
(3 points) Trade your token for 30 Defi tokens and add 20 Defi tokens to a pool with your token and the DEFI token. At the end of the project, your account should have at least 10 Defi tokens in its balance. Include in the report what token you traded to get the DEFI tokens, and how much.

In total, I needed three trades to trade 247.485 ANTON for 30 DEFI tokens. The first trade exploited an arbitrage opportunity between the DEFI-MOON, MOON-SUN and SUN-DEFI liquidity pools. This allowed me to trade 79.863 ANTON for 10 DEFI using the following path:

[”0x5b67efeA0222aE80bf8263dA7114008112D79c34”,
”0x2362e2981E60aab56b6287aB814da1716D2f4f6d”,
”0xDeD39E717E29f2f7d1C9Ac73B9BB067fC202B2B4”,
”0xD2fa1d3F21c1fD363ABbB1bd828106909A911f60”,
”0xCe126A100eDA0226405ee904eeF4F78f8afd7547”,
”0xDeD39E717E29f2f7d1C9Ac73B9BB067fC202B2B4”]



(a) Calculating first swap

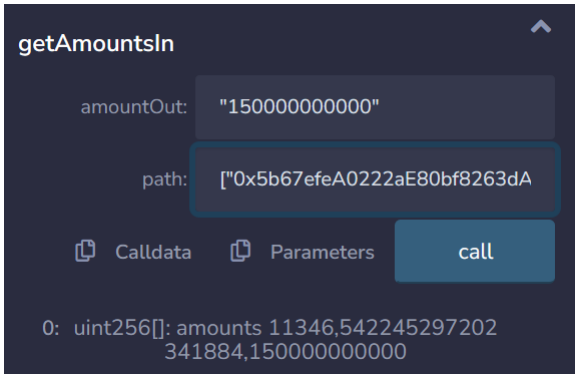


(b) Executing the first swap

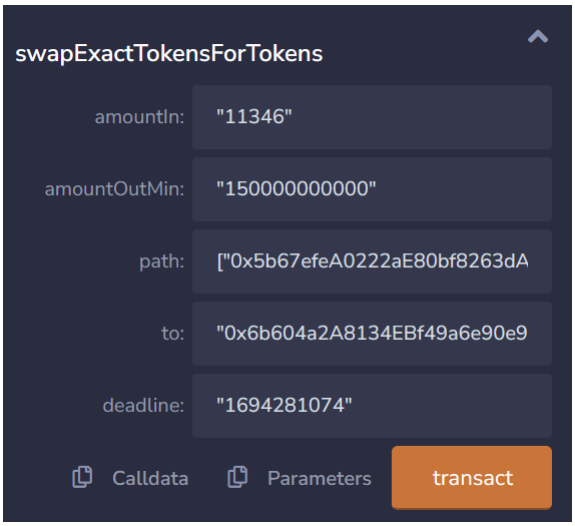
For the second and third trades, I used the WETH-DEFI pool. But first, I had to add some liquidity to the ANTON-WETH pool to ensure that I could swap a sufficiently large amount of WETH to obtain 15 DEFI. After adding liquidity using the router contract, I calculated the amount of ANTON I have to spend to obtain 15

DEFI using the getAmountsIn function. I traded 11.346 ANTON for 15 DEFI using the following path:

[”0x5b67efeA0222aE80bf8263dA7114008112D79c34”,
”0x2362e2981E60aab56b6287aB814da1716D2f4f6d”,
”0xDeD39E717E29f2f7d1C9Ac73B9BB067fC202B2B4”]

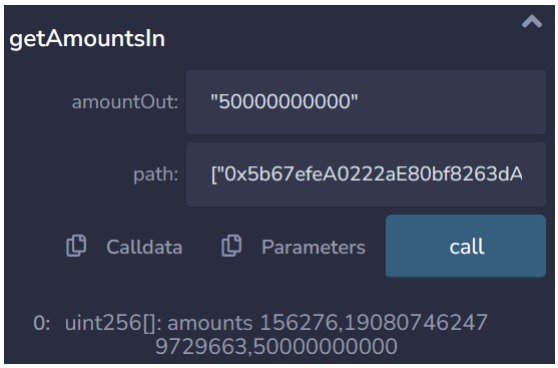


(a) Calculating second swap

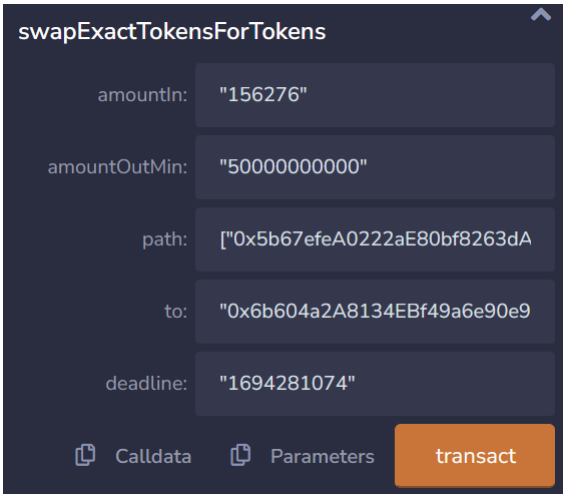


(b) Executing the second swap

For my last trade, I repeated everything from the second trade, using the same path, and obtained 5 DEFI for 156.276 ANTON.



(a) Calculating third swap



(b) Executing the third swap

After successfully trading my token for DEFI, I created the ANTON-DEFI pool which can be found at the following address:

0xe9e0abb7F63cfec9Ebc029ABE88eFF5f61D5D7F7

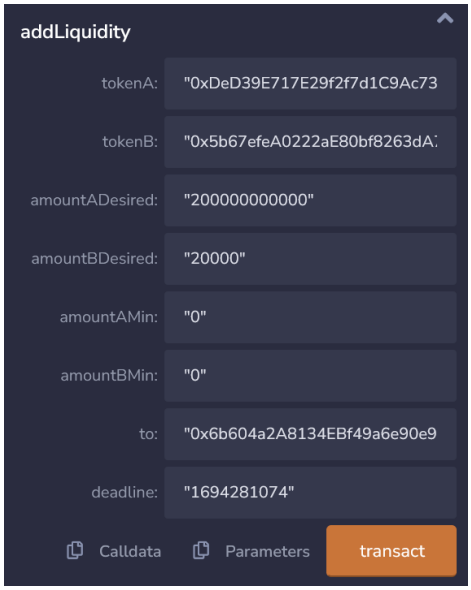


Figure 4: Generating ANTON-DEFI liquidity pool

The pool was instantiated such that 20 DEFI equals 20 ANTON.

Task 4

Write a smart contract with the following functions:

a) (5 points) A view function to get the pool balances of a pair of tokens in our DEX.

b) (5 points) A function which takes as input (address A, address B, address C, uint256 valA, uint256 valC) which performs a swap of valA of token A to token C using an intermediary token B (i.e., swaps A for B, then B for C) only if the resulting swap ends with at most valC amount of token C (this swap shoulfd be atomic: if the condition on valC is not met, all swaps should be voided). Your contract can either make use of the Router contract to perform the swap or itself perform the functionality of the swap. In the report, briefly describe how you implemented the function and what steps a user must take to call the function (e.g., who the user needs to approve what token).

The smart contract fulfilling both tasks can be found at the following address:

0x6E1D2f5651C73cE574A4804b6241d77f63699d75

Both functions will repeatedly use the factory address, the router address, and an invalid pair address. Therefore, they were initialized in advance.

a) The getPoolBalance function requests the user to input the addresses of the two tokens in the pool. The pool address is obtained by calling the factory's getPair function. In the next step, the smart contract verifies the existence of the requested pool balance. In the event of inexistence, the smart contract terminates and returns an error message "Invalid trading pair". If the pool exists on our DEX, the amount of tokens held by the pool is queried using the getBalanceOf function of the ERC20 tokens. Finally, the reserves are returned. Instead of calling the balanceOf function of each token, one could also interact with the pair directly using the IPair interface. This can be done as follows:

```
import './interfaces/IPair.sol';
contract SmartContract{
    function getPoolBalance(address token0, address token1) public
        view returns(uint256 reserve0, uint256 reserve1){
        address factory = 0x36C859726f1D72EBC32cA61B7Dc6A3092b416b66;
        address pairAddress =
            IUniswapV2Factory(factory).getPair(token0, token1);
        require(pairAddress != invalidPair, 'Invalid trading pair');
        (reserve0, reserve1, ) =
            IUniswapV2Pair(pairAddress).getReserves();
    }
}
```

Alternative code for the getPoolBalance function

b) The function performing the swap is called AtomicSwap. I wrote AtomicSwap in such a way, that the sender only has to allow SmartContract.sol to spend valA of token A from his balance. This requires that the factory and the router have permission to spend valA of token A held by the smart contract. In a similar manner to a), AtomicSwap checks whether the requested trading triple

exist, i.e. whether the pairs A-B and B-C exist. If one does not exist, the function terminates. Afterwards AtomicSwap verifies that the sender has sufficient balance of token A to perform the swap. In the next step, AtomicSwap validates whether the obtained amount of token C exceeds the limit given by valC. Now, AtomicSwap verifies that it has the allowance to spend valA of token A from the balance of the sender. Then, valA of token A is send to the address of the smart contract. Since no smart contract can spend more tokens than it owns, it allows the router and factory to spend valA after receiving valA from the sender. After all these preparations, the swap can finally happen. To avoid possible front-running attacks, I implemented a fall-back option, checking that the amount of token C received after the swap is at most valC. If an attack occurs (possible due to the large timestamp), the entire transaction is terminated and only the gas fee has to be paid, truly resulting in an atomic swap.

```
pragma solidity >=0.8.18;
/**
 * @title SmartContract
 * @author Luca Brilhaus
 * @notice Get pool balances and implement atomic triple swap
 */
 import './interfaces/IERC20.sol';
 import './interfaces/IFactory.sol';
 import './interfaces/IRouter.sol';

 contract SmartContract{
    address invalidPair = 0x0000000000000000000000000000000000000000;
    address factory = 0x36C859726f1D72EBC32cA61B7Dc6A3092b416b66;
    address router = 0x3141e4602Fd9B3b08029816D0C09ab177Fdc9dFA;
   // A view function to get the pool balances of a pair of tokens in our
      DEX.
    function getPoolBalance(address token0, address token1) public view
       returns(uint256 reserve0, uint256 reserve1){
      // get Address of liquidity pool
      address pairAddress = IUniswapV2Factory(factory).getPair(token0,
```

```
        token1);
    require(pairAddress != invalidPair, 'Invalid trading pair');
    reserve0 = IERC20(token0).balanceOf(pairAddress);
    reserve1 = IERC20(token1).balanceOf(pairAddress);
}
function AtomicSwap(address A, address B, address C, uint256 valA,
    uint256 valC) public{
    // Path for swapping
    address[] memory path = new address[](3);
    path[0] = A;
    path[1] = B;
    path[2] = C;
    // Trading pairs must exist
    address pairAddress1 = IUniswapV2Factory(factory).getPair(A, B);
    address pairAddress2 = IUniswapV2Factory(factory).getPair(B, C);
    require(pairAddress1 != invalidPair, 'Pair does not exist');
    require(pairAddress2 != invalidPair, 'Pair does not exist');
    // Holder has sufficient balance
    require(IERC20(A).balanceOf(msg.sender)>=valA, 'Insufficient
        balance');
    // Swap results in more than valC
    require(IUniswapV2Router01(router).getAmountsOut(valA,
        path)[path.length-1]<=valC, 'Swap results in more than valC');
    // Check if contract is allowed to spend valA from msg.sender
    require(IERC20(A).allowance(msg.sender, address(this))>=valA,
        'Contract not allowed to spend valA');
    // Transfer funds to contract address
    IERC20(A).transferFrom(msg.sender, address(this), valA);
    // Approve factory and router to perform swaps
    IERC20(A).approve(factory, valA);
    IERC20(A).approve(router, valA);
    // Time stated for Swap to be executed
    uint timelock = block.timestamp + 100000;
    // Perform swap
```

```
        uint[] memory amounts =
            IUniswapV2Router01(router).swapExactTokensForTokens(valA, 0,
            path, msg.sender, timelock);
        // Swap results in at most valC
        require(amounts[path.length-1] <= valC, 'Swap results in more than
            valC');
    }
}
```

Solidity code for SmartContract.sol