한 번에 끝내는
블록체인 개발 A to Z

Chapter 3

Defi 기초 컨셉 구현

Chapter 3

Defi 기초 컨셉 구현

기초적인 AMM 구현

Goal

AMM(CSMM, 유동성 공급 및 스왑)을 간단하게 구현해본다.

유동성 공급/제거

- 유동성 공급은 나의 토큰을 Exchange Contract로 보내는 것이다.
- 유동성 제거는 Exchange Contract에 넣은 나의 토큰 지분을 가져오는 것이다.
- 유동성 공급에 대한 더 정확한 표현은 나의 토큰을 Exchange Contract가 가져가는 것이다.

```
contract Exchange {
    IERC20 token;

    constructor(address _token) {
        token = IERC20(_token);
    }

    function addLiquidity(uint256 _tokenAmount) public payable {
        token.transferFrom(msg.sender, address(this), _tokenAmount);
    }

    function removeLiquidity() {
        msg.sender.transfer(eth_amount);
        token.transfer(msg.sender, tokenAmount);
    }
}
```

유동성 공급

```
contract Exchange {

    function addLiquidity(uint256 _tokenAmount) public payable {
        token.transferFrom(msg.sender, address(this), _tokenAmount);
    }

}
```

- 유동성 공급에 대한 더 정확한 표현은 나의 토큰을 Exchange Contract가 가져가는 것이다.
- 내가 직접 Exchange Contract에게 토큰을 transfer 하는 것이 아니라 내가 Exchange Contract의 addLiquidity 함수를 호출하고 Exchange Contract가 ERC20 토큰 Contract의 transferFrom 함수를 통해 나의 토큰(msg.sender)을 Exchange Contract(address(this))로 전송한다.
- Exchange Contract가 내 토큰을 가져가기 위해서 addLiquidity 전에 approve가 호출이 되어 있어야 한다.

유동성 제거

```
function removeLiquidity() {
    msg.sender.transfer(ethAmount);
    token.transfer(msg.sender, tokenAmount);
}
```

- Exchange Contract에 들어있는 ETH를 나에게 전송한다.
- Exchange Contract가 가지고 있는 ERC20을 나에게 전송하는 것이기 때문에 transfer 함수로 전송한다.

CSMM

- CSMM(Constant Sum Market Maker)은 유동성 풀의 합이 일정한 알고리즘이다.
- CSMM이 독자적으로 사용되는 디파이 서비스는 없다.
    - 그런데 왜 CSMM(Constant Sum Market Maker)인가?

      - 간단하게 디파이의 수식이 어떻게 계산 되는지를 알아 볼 수 있다.

      - 간단하게 디파이의 스왑을 구현 해볼 수 있다.
- CSMM의 특징
    - 슬리피지가 없다. (항상 토큰을 1:1 비율로 교환해준다.)
    - 풀의 유동성이 0이 될 수 있다.

CSMM

1. $x + y = k$가 기본 공식이다. x, y 각각 유동성 풀에 들어있는 토큰의 개수이다.
2. $x + y = k$ 공식에서 스왑 후에도 k는 변하지 않는 다는 것이다.
3. 스왑 후 $x + y$는 $(x + \Delta x) + (y - \Delta y)$ 가 될 것이다. Δx는 Input 토큰의 개수, Δy는 Output 토큰 개수이다.
4. 따라서 $(x + \Delta x) + (y - \Delta y) = x + y$ 라는 공식이 만들어 진다.
5. 공식을 분해해보면 아래와 같이 나온다.

$$x + y + \Delta x - \Delta y = x + y$$

6. 최종적으로 $\Delta x = \Delta y$ 가 나온다.
7. 즉, 스왑 할 때 내가 Input으로 넣은 토큰과 Output으로 나오는 토큰의 개수가 같다.

스왑

```
// ETH -> ERC20
function ethToTokenSwap() public payable {
    uint256 inputAmount = msg.value;
    // calculate amount out (zero fee)
    uint256 outputAmount = inputAmount;
    //transfer token out
    IERC20(token).transfer(msg.sender, outputAmount);
}
```

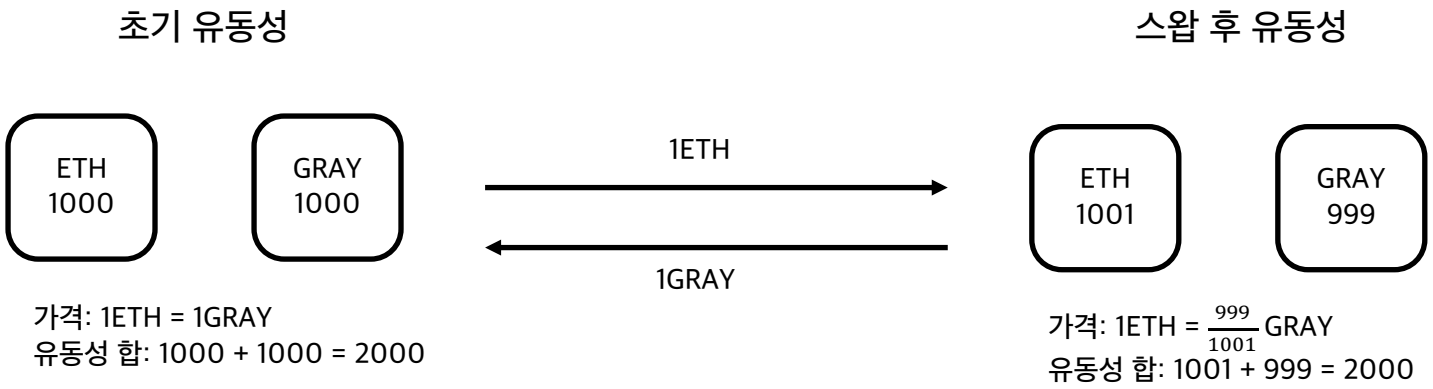
- ETH -> ERC20 스왑의 경우 Exchange Contract로 Input토큰은 ETH가 되고, Output토큰은 ERC20이 된다.
- Output 토큰의 개수는 보낸 ETH의 개수와 같다.
    - CSMM에 의해 $\Delta x = \Delta y$이기 때문이다.
- 계산된 outputAmount를 나에게(msg.sender) 보내준다.

구현 및 테스트

---

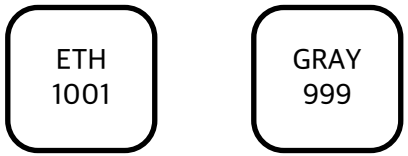
git clone https://github.com/GrayWorld-io/lec_fc_defi -b csmm

Detail Swap


초기 유동성
ETH
1000
GRAY
1000
1ETH
1GRAY
스왑 후 유동성
ETH
1001
GRAY
999
가격: 1ETH = 1GRAY
유동성 합: 1000 + 1000 = 2000
가격: 1ETH = $\frac{999}{1001}$ GRAY
유동성 합: 1001 + 999 = 2000

Detail Swap

스왑 후 유동성



$1000 + 1$ $1000 - 1$

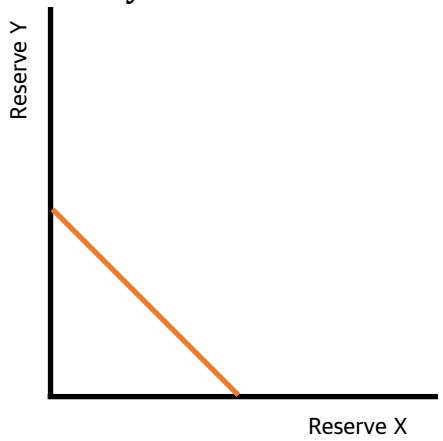
가격: 1ETH = $\frac{999}{1001}$ GRAY

```
function getPrice(uint256 inputReserve, uint256 outputReserve)
    uint256 numerator = inputReserve;
    uint256 denominator = outputReserve;
    return numerator / denominator;
}
```

- InputAmount와 OutputAmount의 값이 1:1로 일정하다.
- 그렇기 때문에 슬리피지가 존재하지 않는다.
- 토큰의 가격 비율은 1:1이 아닐 수 있다.
- 외부 시장의 토큰의 가격에 따라 차액거래자들이 활동한다.
- OutputReserve의 개수가 선형적으로 감소할 수 있고 0이 될 수 있다.
- 이는 풀의 고갈을 야기 할 수 있다는 문제점이 있다.

CSMM

$$x + y = k$$
$$y = -x + k$$



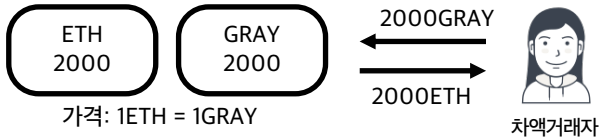
초기 유동성

스왑 후 유동성

극단적으로 초기 유동성을 모두 스왑 해버리면 풀에 토큰이 0개가 된다.

현재 유동성

외부 가격
1ETH = 100,000GRAY



외부의 가격에서 1:1 비율이 깨진 경우에 차액거래자들이 CSMM 유동성 풀에서 스왑하여 유동성 풀이 고갈 될 수 있다.

CSMM

- 본격적으로 CPMM 기반의 AMM을 구현한다.