UNIV - AN ETHEREUM-BASED GAME

CS2361 - Blockchain and Cryptocurrency
Project Report

Bhumika Mittal
Gautam Yajaman

Contents

1 Project Description

This project involves the implementation of our own ERC20 token - AshokaToken (ASH) as well as a frontend. This frontend will allow a user to connect their MetaMask wallet and play a quiz game. If they get all the questions correct, they are awarded with some ASH, which is minted and sent *directly* to their wallet. This process is seamless, and only requires the user to have the MetaMask extension installed in their browser.

2 Background

There are various types of ERC20 tokens on Ethereum, such as WETH, stETH, stablecoins, governance tokens, gaming tokens, etc. If a user wants to convert one token to another, we call this as an exchange. To execute an exchange, we need to answer two main questions:

1. What is the exchange rate?
2. How will seller and buyer communicate?

2.1 CeX: Centralised Exchange

A basic first approach would be to have some middlemen, who will facilitate this transaction. If Gautam wants to exchange his tokens, he will share his demand for Token B in exchange of Token A. The middlemen will quote the exchange rate and take Token A from him. In return, they give Gautam Token B as per the decided exchange rate. This process has several issues:

1. **Lack of transparency:** Who is deciding the exchange rate and how? The supply and demand at the exchange is not transparent.
2. **Security:** There is no guarantee that the middlemen will provide Token B after taking Token A from Gautam.
3. **Censorship:** If the middlemen refuse to allow Gautam to use their exchange, he won't be able to exchange his tokens.

2.2 DeX: Decentralized Exchange

To solve the above stated problems, let's look at the concept of a Decentralised Exchange (DeX). Here, the transactions take place between the user and a *liquidity pool*, without any middlemen. This method is accessible, transparent, and censorship free.

How to develop a DeX?: We can use a very elegant idea known as an Automated Market Maker. Liquidity providers deposit assets into an on-chain pool. Then users trade with the on-chain pool and the exchange rate is determined automatically (using well thought out formulas!). This solution is gas-efficient [1]

[1] Reference: Uniswap Whitepaper

3 AMM: Automated Market Maker

Automated Market Makers (AMMs) are a form of Decentralized Exchange (DeX) that employ algorithmic "money robots" to facilitate individual traders in the buying and selling of cryptocurrency assets. Unlike traditional order book trading where users trade directly with other individuals, in AMMs, users conduct trades directly through the AMM system.

Market makers are entities responsible for ensuring liquidity for an asset that can be traded on an exchange, which might otherwise be illiquid. They achieve this by purchasing and selling assets from their own accounts with the aim of generating a profit, often from the spread - the difference between the highest buy offer and the lowest sell offer. Their trading activities create liquidity, thereby reducing the price impact of larger trades.

3.1 Uniswap

Uniswap [2], an AMM, is a network of ETH-ERC20 exchange contracts, with each ERC20 token having a unique exchange contract. If a token doesn't have an exchange, anyone can create one using the Uniswap factory contract, which also serves as a public registry for all token and exchange addresses in the system.

Each exchange maintains reserves of both ETH and its corresponding ERC20 token. Anyone can become a liquidity provider for an exchange by depositing an equivalent value of both ETH and the relevant ERC20 token, which is different from buying or selling. Liquidity is pooled across all providers, and an internal "pool token" is used to track each provider's relative contribution. Pool tokens are created when liquidity is added to the system and can be destroyed at any time to withdraw a proportional share of the reserves.

Exchange contracts function as automated market makers for an ETH-ERC20 pair. Traders can swap between the two by adding to the liquidity reserve of one and withdrawing from the other. Since ETH is a common pair for all ERC20 exchanges, it can be used as an intermediary for direct ERC20-ERC20 trades in a single transaction. Users can specify a different recipient address for the purchased tokens.

3.2 Constant Product Formula: $x * y = k$ model

Consider an exchange market maker [3] with two tokens: PhineasCoin and FerbCoin (in reality, this would be ETH and an ERC20 token). This exchange will have reserves of both these tokens, say, x_0 of PC and y_0 of FC. This is referred to as the *liquidity pool*.

If Gautam wants to exchange some of his PC, say Δx, for some amount of FC, how is this amount (say Δy) determined? We use the constant product formula $x \times y = k$! At any given time, the amount of FC multiplied with the amount of PC should be equal to the constant k. This k is set based on the initial amount of FC and PC, when the liquidity pool was first established.

[2] Reference: Uniswap Documentation
[3] Reference: Building a Decentralized Exchange in Ethereum

The value of Δy can be calculated as follows:

$$(x + \Delta x) * (y - \Delta y) = k$$
$$-\Delta y = \frac{k}{(x + \Delta x)} - y$$
$$\Delta y = y - \frac{k}{(x + \Delta x)}$$

The process to calculate Δx in the event we are selling Δy is symmetric. This way of determining the value of Δy ensures that $x \times y = (x + \Delta x) \times (y - \Delta y)$. Consider $\alpha = \frac{\Delta x}{x}$ and $\beta = \frac{\Delta y}{y}$. We then have:

$$x \times y = (x + \Delta x) \times (y - \Delta y)$$
$$x \times y = (1 + \alpha)x \times (1 - \beta)y$$
$$1 = (1 + \alpha)(1 - \beta)$$

Hence we can compute both $x' = x + \Delta x$ and $y' = y - \Delta y$ based solely on the ratio of the number of tokens being sold and the number of tokens in the liquidity pool:

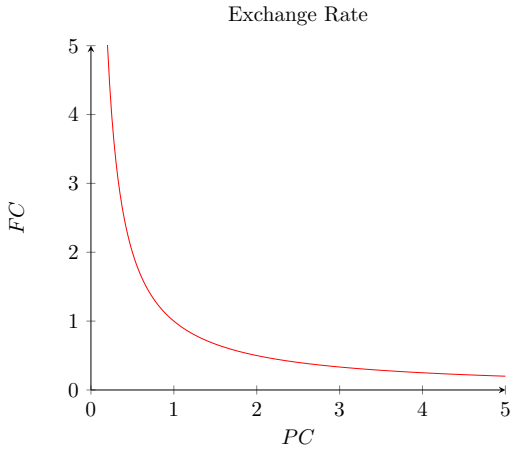
$$x' = x + \Delta x = x(1 + \alpha) = \frac{x}{(1 - \beta)}$$
$$y' = y - \Delta y = y(1 - \beta) = \frac{y}{1 + \alpha}$$

In the given example, Gautam is selling Δx of PC. If there are x PC already in the liquidiy pool, calculate $\alpha = \frac{\Delta x}{x}$ and then using the above formulae, we can easily find x' and y'. From y', we can compute Δy as $y - y'$.

A property of this constant product formula to note is:

$$\lim_{x \to 0} Price(PC) \to \infty$$

Hence the price of a token can increase without bound, so long as the demand for it is sufficiently high. The larger an exchange is in relation to the total reserves, the more price slippage will occur. Essentially, exchange contracts use the open financial market to determine the relative value of a pair and use that as a market making strategy:



Mathematically, let the pool have x units of PC and y units of FC. The marginal price [4] is given by

$$p = \frac{-dy}{dx} > 0$$

Using p, we can estimate the value of the tokens in the pool:

- Value of PC in the pool in terms of FC $= p.x$
- Value of FC in the pool in terms of PC $= x$

We want to have a balance such that $px = y$. This means

$$\boxed{-\frac{dy}{dx} = \frac{y}{x}}$$

For this equation to have an unique solution, the pool must maintain $x \cdot y = k$ where $k \in \mathbb{R}$.

3.3 Liquidity Provider Fee

We can incentivize liquidity providers (we will touch upon liquidity providers later) by sharing trading fees with them. When a fee is introduced in the constant product model, the formula for updating the token reserves and calculating the trade amounts Δx and Δy is modified to account for the fee.

Let ρ represent the fee rate for each token trade, where $0 \leq \rho < 1$. For example, $\rho = 0.003$ for a 0.3% fee. Define $\gamma = 1 - \rho$, representing the proportion of the trade not taken as a fee.

The new reserves are calculated as follows:

$$x'_\rho = x + \Delta x = (1 + \alpha)x = \frac{x + x\beta\left(\frac{1}{\gamma} - 1\right)}{1 - \beta}$$
$$y'_\rho = y - \Delta y = \frac{y}{1 + \alpha\gamma} = (1 - \beta)y$$

Using y'_ρ it is simple to compute Δy. Note that the introduction of fees causes the constant product to increase slightly for each trade.

$$x'_\rho \times y'_\rho > x \times y, \quad \text{when } \rho > 0,$$
$$x'_\rho \times y'_\rho = x \times y, \quad \text{when } \rho = 0 \text{ (no fee)}$$

We can rationalise this by realising that for Δx of PC put into the pool, only $\Delta y - \rho\Delta y$ of FC is given to the user, so the amount of tokens in the liquidity pool goes up.

This serves as a payout to liquidity providers, which is collected when they *burn* their pool tokens to withdraw their share of the total reserves. Guaranteed arbitrage opportunities from price fluctuations should drive a steady stream of transactions through the system, increasing the amount of fee revenue generated.

Sidenote: due to rounding errors, the value of k will slightly deviate from its true value.

[4] The price of a small amount of FC in units of PC

3.4 Updating Liquidity

3.4.1 Minting Liquidity

Minting liquidity refers to the process by which an investor can contribute to the liquidity pool of a decentralized exchange by depositing both tokens (PC and FC in our example). This will make them a *liquidity provider* (LP). The state of the liquidity pool can formally be represented as a 3-tuple (p, f, l), where p and f represent the amounts of PC and FC respectively and l refers to the amount of liquidity.

If Bhumika decides to deposit Δp PC, then she will also have to deposit $\Delta f = f' - f$ FC, where $\alpha = \frac{\Delta p}{p}$. This is to preserve the ratio of PC to FC and hence, the exchage rate does not change. The reserves of PC and FC are updated in the following manner:

$$p' = (1 + \alpha)p$$
$$f' = (1 + \alpha)f$$

Clearly, the constant k increases every time liquidity is minted, since $p \times f = k$ is strictly increasing. For this deposit, she will be awarded $\Delta l = l' - l$ liquidity such that $p : f : l$ remains the same. Hence $l' = (1 + \alpha)l$.

3.4.2 Burning Liquidity

Burning liquidity is dual to minting liquidity. Let the 3-tuple of the liquidity pool be (p, f, l), and let Bhumika own Δl liquidity. Then, Bhumika receives $\Delta p = p - p'$ PC and $\Delta f = f - f'$ FC respectively when liquidity Δl she owns is burnt. If $\alpha = \frac{\Delta l}{l}$, then:

$$p' = (1 - \alpha)p$$
$$f' = (1 - \alpha)f$$
$$l' = (1 - \alpha)l$$

This preserves the ratio $p : f : l$, and ensures that Bhumika receives the correct fraction of PC and FC she is owed. Any transaction fees she is entitled to are also paid out.

3.4.3 Impermanent Loss

Let Δl be the *share* of the liquidity that is owned by Bhumika. If $\Delta l = \frac{1}{20} \cdot l$, then when she burns her liquidity, she will receieve $\frac{1}{20}th$ of the reserves of PC and FC (assuming no additional liquidity has been minted). Note that this means that as per changes in the ratio of PC to FC, the number of tokens she receieves on burning liquidity will vary.

This exposes her to what is referred to as *impermanent loss*. There could be a scenario where the fiat value of her tokens was more before she deposited them, but most of the time this is counteracted by the trading fees she is entitled to as a liquidity holder.

3.5 Slippage

The exchange rate on a decentralized exchange is updated dynamically with every transfer. If Aarav tries to sell some of his PC, the exchange rate could have changed between the

time he placed his sell order and the time when it is actually fulfilled since many other users might be trying to swap currency at the same time.

This shift in the exchange rate between the exchange's quote price and actual price is called *slippage*. Aarav was fine with the quote price, but he might not be fine with the actual price. To handle this, we allow users to set a *maximum slippage percentage*.

3.5.1 Handling Slippage

If P_{quote} is the quote price and P_{actual} is the actual price, slippage S is calculated as follows:

$$S = \left(\frac{|P_{actual} - P_{quote}|}{P_{quote}} \right)$$

A transaction is executed only if $S \leq MS$, where MS is the maximum slippage percentage specified by Aarav.

4 DeX Implementation

The DeX Implementation we are using for this project is Uniswap v2. Here are some of the main functions offered by Uniswap:

4.1 Read-Only Functions

These functions do not modify the blockchain state. They are used to read or retrieve data from the blockchain without causing any state change. Since they don't alter the blockchain state, read-only function calls don't require any gas:

- `factory()`: Returns the factory address. The factory address is the address of the smart contract that creates liquidity pools.
- `WETH()`: Returns the canonical WETH address. Canonical WETH refers to Wrapped Ether, which is a token that represents Ether in a transferable token form.
- `getAmountsOut(amountIn, path)`: Returns output amounts for a given input amount and path.
- `getAmountsIn(amountOut, path)`: Returns input amounts for a given output amount and path.

4.2 State-Changing Functions

These are functions that modify the state of the blockchain. When these functions are called, they perform actions that change data or states on the blockchain, like transferring tokens or updating a variable. These changes are recorded on the blockchain after the transaction is confirmed:

- `addLiquidity(...)`: Adds liquidity to an ERC20 pool.
- `swapExactTokensForTokens(...)`: Swaps exact input tokens for output tokens.
- `swapTokensForExactTokens(...)`: Receives exact output tokens for input tokens.

- `swapExactETHForTokens(...)`: Swaps ETH for output tokens.
- `swapTokensForExactETH(...)`: Receives ETH for input tokens.
- `swapExactTokensForETH(...)`: Swaps tokens for ETH.
- `swapETHForExactTokens(...)`: Receives tokens for ETH.
- `swapExactTokensForTokensSupportingFeeOnTransferTokens(...)`: Swaps tokens with a fee on transfer.
- `swapExactETHForTokensSupportingFeeOnTransferTokens(...)`: Swaps ETH for tokens with a fee on transfer.
- `swapExactTokensForETHSupportingFeeOnTransferTokens(...)`: Swaps tokens for ETH with a fee on transfer.

5 Project Components

The project consists of three major components:

1. *token.sol*: This is a smart contract written in Solidity. We will utilize the standard ERC20 implementation from the OpenZeppelin project, which includes functionality for creating our custom token named "AshokaToken" (couldn't think of anything better :)). The token incorporates a Role-Based Access Control system defined within the smart contract, with two roles: DefaultAdmin (sudo user) and Minter (a user authorized to mint the token).
2. *uniswap.sol*: This Solidity smart contract serves as a simplified version of Uniswap v2 Router[5]. A Router defines how trades occur across various pair contracts (pools), specifying details such as the number of hops, involved tokens, etc. However, it cannot alter the core logic; it only determines how trades are routed. We deploy our ASH coin using the Factory Protocol[6], responsible for creating new pair contracts. Since the router collaborates with all pairs deployed via the Factory contract, we utilize the addLiquidityETH and swapExactETHForTokens functions. The Uniswap UI interfaces primarily with this component.
3. *webApp*: This frontend is developed using HTML/CSS and JavaScript, deployed on Netlify. It is accessible to anyone with no installations required, other than the MetaMask extension in the browser.

6 Creating and Deploying our Own Token

We used OpenZeppelin's `ERC20.sol` smart contract implementation for *fungible tokens* [7]. Each AshokaToken is indistinguishable from another, and hence they are interchangeable. There are a few key components of `ERC20.sol`:

1. **Token Information**:

[5]Reference: Router02 Documentation
[6]Reference: Factory Protocol Documentation
[7]Reference: OpenZeppelin's ERC20 Documentation

- `name`: A human-readable name for the token (AshokaToken)
- `symbol:` An identifier for the token (ASH)
- `decimals:` Solidity only supports integer numbers. To work around this, `ERC20.sol` has this field, which specifies the number of decimal places of the token. ASH uses 18 decimal places, meaning that the smallest unit is `0.000000000000000001`.

2. **Functions:** `ERC20.sol` has many in-built functions that are used to support the token:
   - `totalSupply()`: This function returns the total number of tokens in existence. It's useful for understanding the entire scope of the token supply.
   - `balanceOf(account)`: Returns the amount of tokens that are held by a specified address. This function is key for tracking token distribution and ownership.
   - `transfer(account, amount)`: Enables a token holder to transfer a specified amount of their tokens to another address. This function is the primary method for token movement in the ERC20 ecosystem.
   - `allowance(owner, spender)`: Returns the remaining number of tokens that a spender is allowed to spend on behalf of the owner. This is important for delegated token transfers.
   - `approve(spender, amount)`: Allows a token holder (owner) to authorize another address (spender) to transfer up to a specified number of tokens on their behalf. This function is crucial for enabling third-party spending of tokens.
   - `transferFrom(from, to, amount)`: Facilitates the transfer of tokens from one address to another using the allowance mechanism. It allows a spender to transfer tokens on behalf of the owner to a third address.
3. **Key Events:** Key events in Ethereum are used to record important actions that occur within a smart contract. They serve as a way to emit information that external entities, like user interfaces, off-chain applications, or other smart contracts, can detect and use. For ERC20 tokens, there are two key events defined by the standard:
   - `event Transfer(from, address to, transfer_amt)`: The `Transfer` event is emitted whenever tokens are moved from one account to another. This includes both direct transfers (using `transfer`) and delegated transfers (using the `transferFrom` function). The event logs the sender's address, the receiver's address, and the amount of tokens transferred.
   - `event Approval(owner, spender, spend_amt)`: The `Approval` event is emitted when a token holder approves another account to spend a specified number of tokens on their behalf, using `approve`. This event logs the token owner's address, the spender's address, and the amount of tokens approved for spending.

   These events are crucial in the ERC20 standard because they provide a transparent and traceable way to track token movements and approvals on the blockchain. When these events are emitted, they get recorded on the blockchain and can be observed

and reacted to by external applications, such as wallets or dApps. This makes events an essential tool for integration and interaction with Ethereum smart contracts.

Since we want different levels of authorization, we use the role-based access control offered by `AccessControl` [8]. These are the key features of `AccessControl` from `AccessControl.sol`:

1. **Role-Based Access Control (RBAC):** RBAC offers flexibility with multiple roles like 'moderator', 'minter', or 'admin', each having specific permissions.
2. **Granting and Revoking Roles:** Roles are dynamically manageable, with an associated admin role for each, responsible for granting and revoking roles. The `DEFAULT_ADMIN_ROLE` acts as a universal admin for all roles.
3. **Querying Privileged Accounts:** Allows for tracking and querying which accounts hold specific roles, essential for verifying system properties and ensuring secure role assignments.

Using OpenZeppelin allows our token to have the same functionality as any ERC20 token on the *real blockchain*. We have successfully deployed this token on the Goerli Ethereum Testnet. The code for our token contract is given below (we used the assistance of OpenZeppelin's Solidty Wizard)[9]:

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/AccessControl.sol";

contract AshokaToken is ERC20, AccessControl {
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

    constructor(address defaultAdmin, address minter) ERC20("AshokaToken",
    ↪  "ASH") {
        _grantRole(DEFAULT_ADMIN_ROLE, defaultAdmin);
        _grantRole(MINTER_ROLE, minter);
    }

    function mint(address to, uint256 amount) public onlyRole(MINTER_ROLE)
    ↪  {
        _mint(to, amount);
    }
    function RewardUser(address to, uint256 amount) public {
        _grantRole(MINTER_ROLE, to);
        _mint(to, amount);
        _revokeRole(MINTER_ROLE, to);
    }
}
```

[8] Reference: OpenZeppelin's Access Control Documentation
[9] Reference: OpenZeppelin's Solidity Wizard

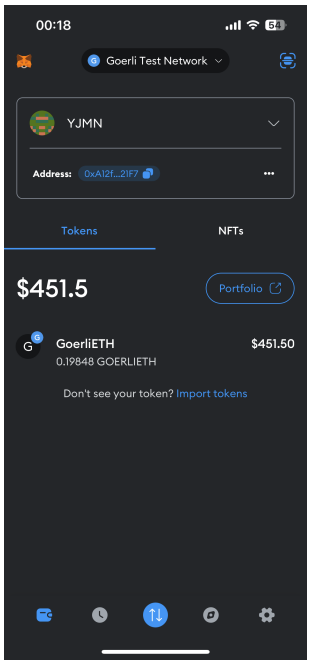
6.1 Minting Tokens

New tokens are minted using the `mint` function defined in the token contract. Calling this function with the recipient address and amount to mint will add those tokens directly to the recipient address. Note that `amount` is actually an unsigned integer. This raises the question - how does one mint, say, 0.5 ASH? This is where the `decimal` field of the ERC20 standard comes into play. Since ASH has 18 decimal places, one needs to use $0.5 * 10^{18}$ as the value for `amount`. The value displayed to the end-user will be divided by 10^{18}, showing them the true value of ASH they have actually received.

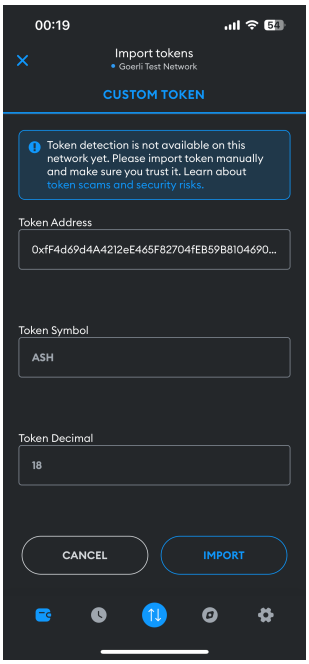
Minting tokens is analogous to a country printing more money. For example, the US Federal Reserve printed over $3 trillion in 2020 alone[10].

6.2 Using Our Token

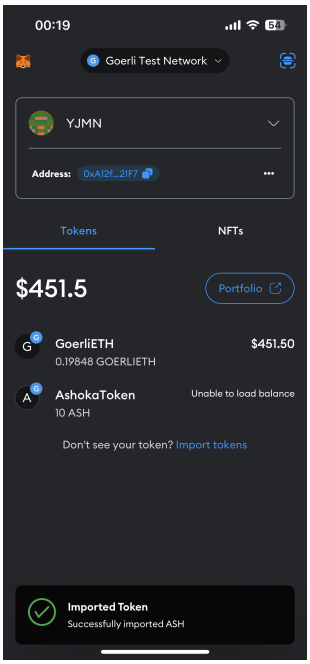
The contract address for AshokaToken is `0x2a20E78F558CB8F659ABc1bC242801D56B9f05c0`. If Prof. Jhawar wants to, he can easily add this token to his MetaMask [11] wallet in 2 simple steps:



(a) Make sure you are on Goerli Test Network and click on 'Import tokens'



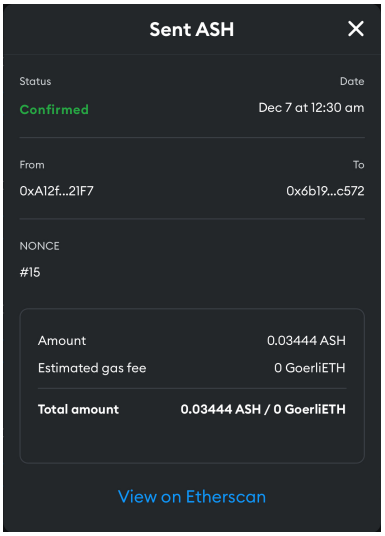
(b) Paste the contract address, and click import



(c) All done!

Now, transfer of ASH from his wallet to another wallet is possible. For example, here is a transaction from Gautam to Bhumika's wallet[12]:

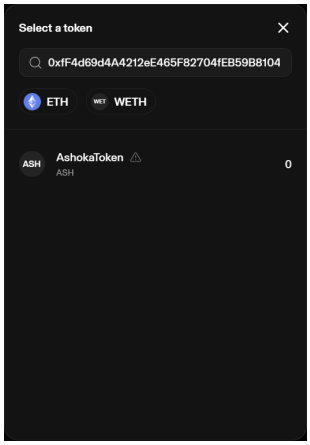
[10] Reference: The US printed more than $3 trillion in 2020 alone. Here's why it matters today
[11] Reference: MetaMask
[12] TxID: 0x795123b9ee1a0b73465cdd4ca1e7eaa7266c7d7d712d20ba7516ce7eb454211d



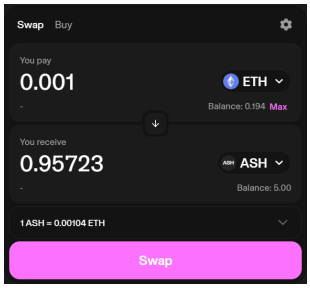
We have already established an ASH-ETH liquidity pool on Uniswap, and this enables anyone to perform exchanges on the pool, provided they have their Goerli Testnet MetaMask wallet linked. Users can also add liquidity to the pool, giving them a liquidity share.

6.2.1 Exchanging Tokens

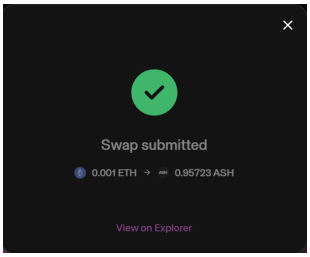
Swapping tokens is a seamless process through this webpage:



(a) Click on 'Select a token', paste the contract address, and click 'AshokaToken'



(b) Click 'Swap'. You will have to approve the transaction through a MetaMask pop-up

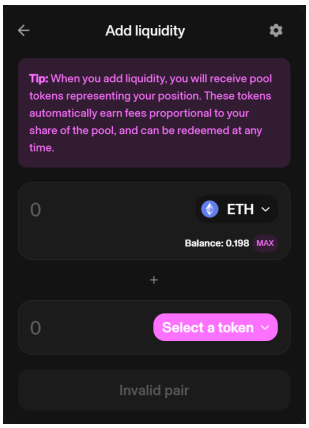


(c) All done!

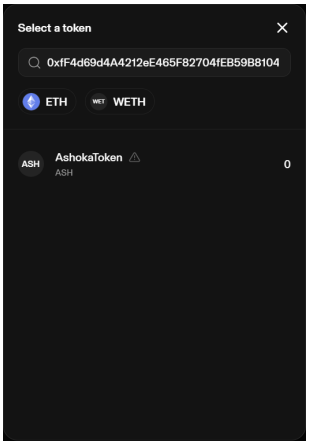
Transactions can take anywhere between seconds and minutes to be approved. Maximum slippage is set to auto by default, however, a custom value can be set by clicking the cog at the top right. If approval takes longer than the approval deadline (which can be set in the same place), it is discarded.

6.2.2 Adding Liquidity

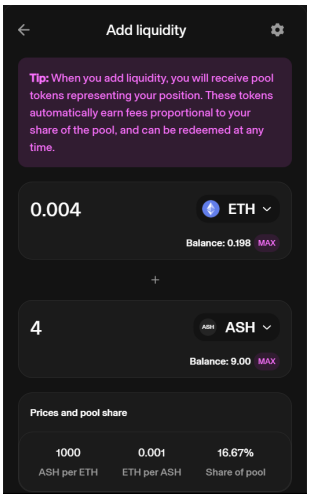
Liquidity can be added in a few simple steps using this webpage:



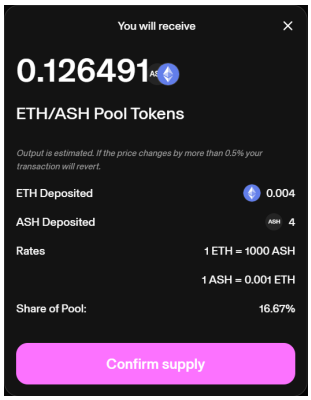
(d) Click on 'Select a token'



(e) Paste the contract address, and click 'AshokaToken'



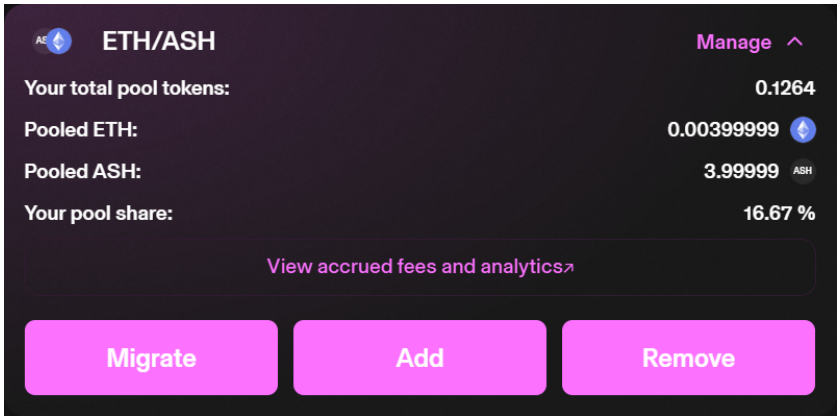
(f) Click approve, and a MetaMask pop-up will appear. Approve that as well.



(g) Click 'Confirm supply'

6.2.3 Burning Liquidity

Burning liquidity is as easy as adding it: Clicking on 'Remove' will move the user to a page where they can specify the percentage of their liquidity they wish to burn. Upon approval, the user will be granted their tokens, plus any fees accrued.



6.3 What Gives a Token Value?

A question that many people (including us) will have is - you made a token... so what? What actually gives a token value? To better understand this, we will be using Dogecoin[13] as a case study.

6.3.1 Introduction

Dogecoin (DOGE), a cryptocurrency that started as a joke, provides an intriguing case study into how cryptocurency tokens can acquire real-world value. This digital currency, which features the face of the Shiba Inu dog from the "Doge" meme [14], was created by software engineers Billy Markus and Jackson Palmer. Despite its satirical inception, Dogecoin has witnessed significant market growth, highlighting the unpredictable nature of cryptocurrency valuations.

6.3.2 Origins

Dogecoin was launched on December 6, 2013, as a playful alternative to Bitcoin, aiming to reach a broader demographic and to distance itself from the controversial aspects of other cryptocurrencies. Its website quickly gained popularity, and within two weeks, Dogecoin's market value reached $8 million. This rapid growth was partly driven by its online community on platforms like Reddit.

6.3.3 Public Attention

- **Early Volatility:** Shortly after its launch, Dogecoin's value surged nearly 300% in just 72 hours.
- **Social Media Influence:** In July 2020, a TikTok trend aimed at driving Dogecoin's price to $1 significantly boosted its value, illustrating the impact of social media on cryptocurrency prices.
- **Celebrity Endorsements:** Public figures like Elon Musk, Snoop Dogg, and Mark

[13] Reference: Dogecoin's Wikipedia page
[14] Reference: Doge - Know Your Meme

Cuban have influenced Dogecoin's value through social media endorsements and practical implementations, like Dallas Mavericks accepting Dogecoin for purchases [15].

- **NASCAR ties:** In 2014, the Dogecoin community sponsored NASCAR driver Josh Wise, raising 67.8 million DOGE (about $55,000) to fund his race car, which sported a Dogecoin/Reddit paint scheme in the Aaron's 499. This unique collaboration reached a peak when Wise, backed by a vigorous online fan vote driven by the Dogecoin Reddit community, secured a spot in the Sprint All-Star Race, outrunning established names like Danica Patrick. This highlighted Dogecoin's real-world influence and marked a significant crossover between cryptocurrency and mainstream sports.

6.3.4 Milestones

- **Historic Price Movements:** Dogecoin's value reached new heights in 2021, peaking at over $0.50 and achieving a market capitalization close to $50 billion. These events reflect both the speculative nature of cryptocurrencies and their susceptibility to external influences[16].
- **Dogecoin Foundation:** The re-establishment of the Dogecoin Foundation in August 2021, with advisors like Ethereum co-founder Vitalik Buterin, added legitimacy to the coin.

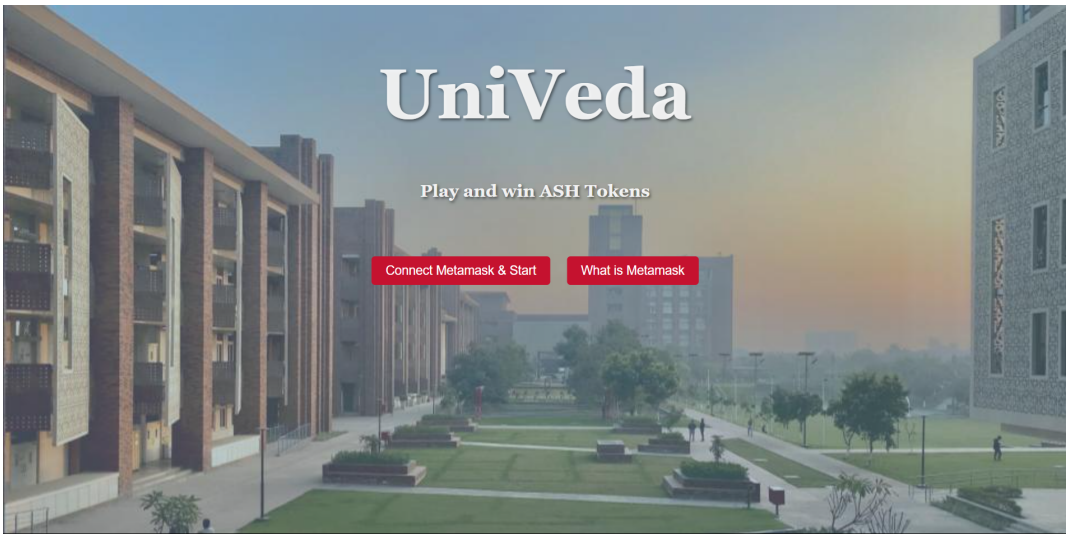
6.3.5 Conclusion

Dogecoin's journey from a humorous meme to a cryptocurrency with significant market capitalization exemplifies how digital tokens can acquire real-world value. Factors like community support, social media trends, celebrity influence, and practical use cases play substantial roles in determining the value of cryptocurrencies. Dogecoin's case is a testament to the dynamic and often unpredictable nature of the crypto market, where perceived value can sometimes override traditional financial metrics. Today, DOGE still has a value of $0.1.

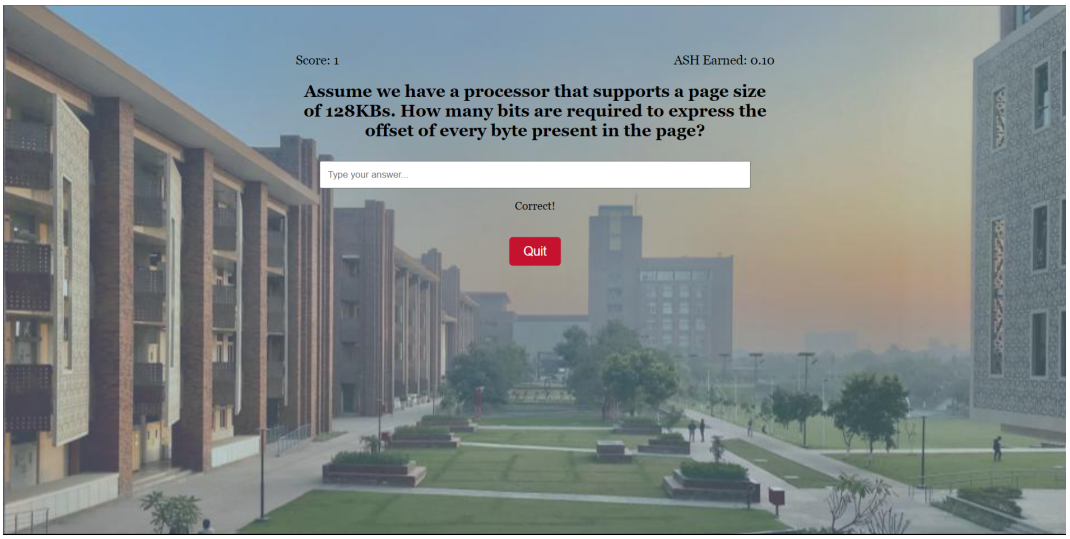
[15]Reference: Mark Cuban boasts Mavericks are largest Dogecoin merchant in the world
[16]Reerence: Dogecoin surges 30% to a record above 50 cents as speculative crypto trading continues

7 UniVeda: Our Own dApp

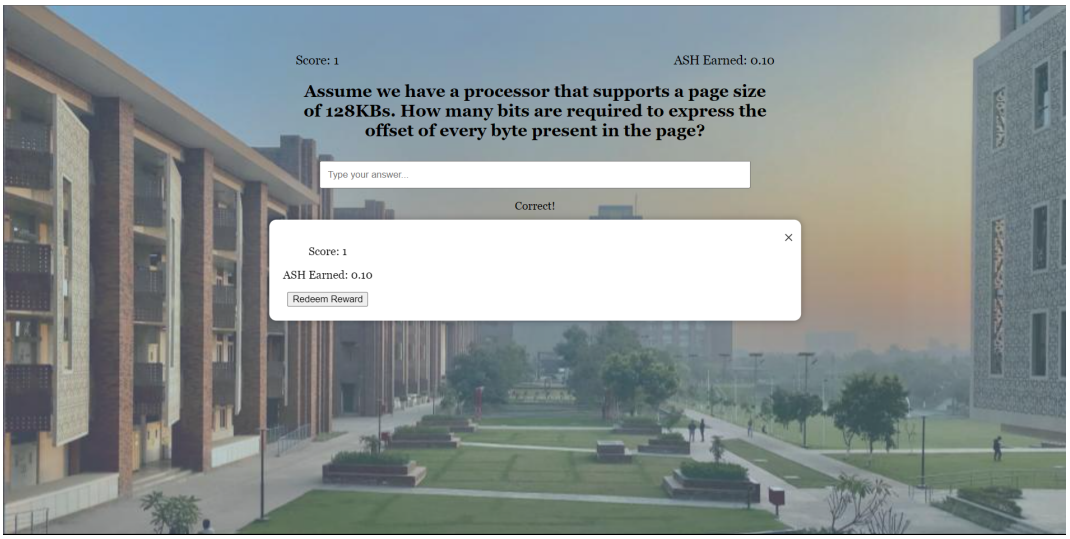
We have written and deployed our own dApp called Univeda, accessible through this link. This is a platform with computer science based questions, but can easily be edited to support any kind of puzzle hunt.



The home screen prompts the user to connect to MetaMask. After connecting to MetaMask, the user can answer up to seven questions. For each correct answer, the user gets 0.1 ASH, which they can redeem before exiting the platform.



Upon either completing all 7 questions, or clicking quit, the user will be presented with the following screen:



Click redeem reward, and a MetaMask wallet popup will ask for the user's approval. Upon clicking approve, the appropriate amount of ASH is minted directly to the user's wallet.