

Lesson 15

Proof of Cat

See [post](#)



Upcoming changes to Solidity

What would Solidity 1.0 and 2.0 look like ? [Video](#)

1.0

- Modifiers jump rather than in lining
- Try catch for custom errors
- Immutable reference types
- Generics
- Algebraic data types (composite types such as tuples)
- Operators for user defined types
- Standard Library

Motivation

- allow more pre compilation
- make the language extensible
- stop wasting memory

2.0

- Compiler changes - rewrite the compiler in Rust
 - Allow components in Rust
 - Have Rust bindings
- Language changes
 - Separate implementation storage
 - Improve clarity about state access / modification
 - Maybe remove inheritance
 - More control over storage layout

See [docs](#)

More talks from Devcon VI

Technical Details of the Solidity Compiler [Video](#)

Underhanded Solidity [Video](#)

Unlimited Size contracts [Video](#)

What's next in EVM ? [Video](#)

Symbolic computation [Video](#)

Verkle trees

<https://vitalik.ca/general/2021/06/18/verkle.html>

See [article](#)
and [Ethereum Cat Herders Videos](#)

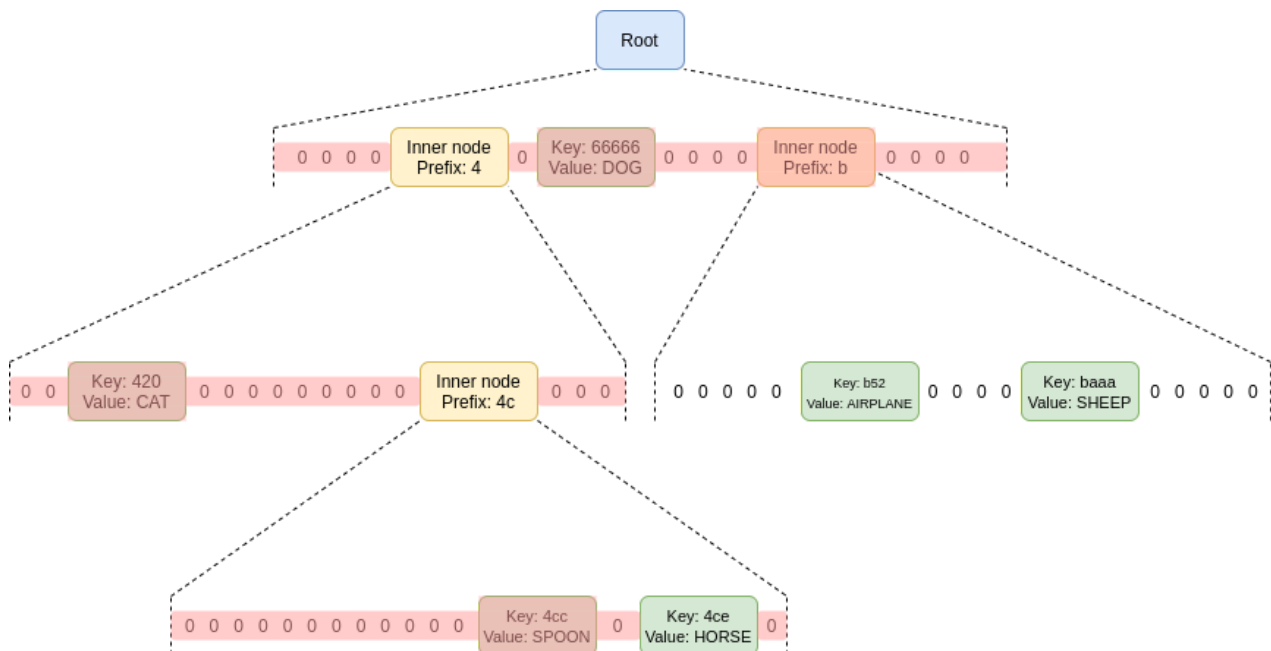
Like merkle trees, you can put a large amount of data into a Verkle tree, and make a short proof ("witness") of any single piece, or set of pieces, of that data that can be verified by someone who only has the root of the tree.

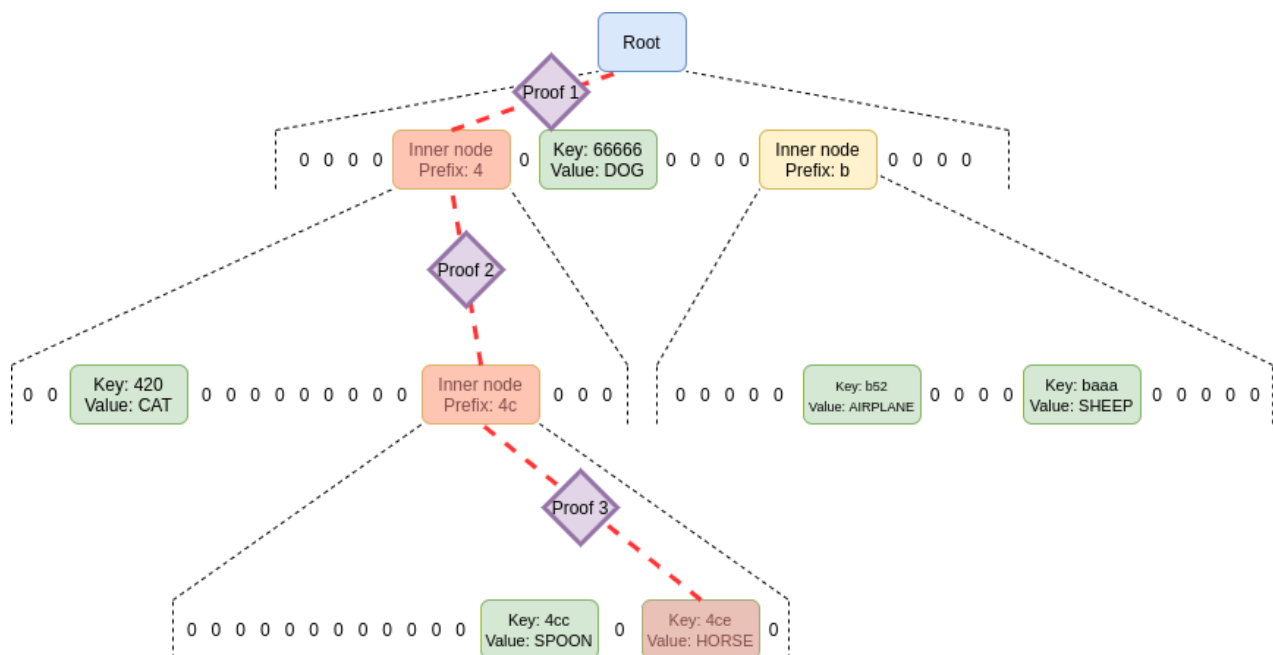
What Verkle trees provide, however, is that they are much more efficient in proof size. If a tree contains a billion pieces of data, making a proof in a traditional binary Merkle tree would require about 1 kilobyte, but in a Verkle tree the proof would be less than 150 bytes.

Verkle trees replace hash commitments with vector commitments or better still a polynomial commitment.

Polynomial commitments give us more flexibility that lets us improve efficiency, and the simplest and most efficient vector commitments available are polynomial commitments.

The number of nodes needed in a merkle proof is much greater than in a verkle proof





Vector commitments vs. Hash

- Vector commitments: existence of an “opening”, a small payload that allow for the verification of a portion of the source data without revealing it all.
- Hash : verifying a portion of the data = revealing the whole data.



Proof sizes

Merkle

Leaf data +
15 sibling
32 bytes each
for each level (~7)

= ~3.5MB for 1K leaves

Verkle

Leaf data +
commitment + value + index
32 + 32 + 1 bytes
for ~4 levels
+ small constant-size data

= ~ 150K for 1K leaves

Stateless Ethereum

The Ethereum world state contains all Ethereum accounts, their balances, deployed smart contracts, and associated storage, it grows without bound.

The idea of stateless Ethereum was proposed in 2017, it was realised that unbounded state is problematic, especially in providing a barrier for entry to people wanting to run nodes. Increasing the hardware requirements for a node leads to centralisation.

The aim of Stateless Ethereum is to mitigate unbounded state growth.

Two paths were initially proposed : **weak statelessness** and **state expiry**:

- State expiry:
remove state that has not been recently accessed from the state (think: accessed in the last year), and require witnesses to revive expired state. This would reduce the state that everyone needs to store to a flat ~20-50 GB.
- Weak statelessness:
only require block proposers to store state, and allow all other nodes to verify blocks statelessly. Implementing this in practice requires a switch to [Verkle trees](#) to reduce witness sizes.

However according to this [roadmap](#) it may make sense to do both together.

State expiry without Verkle trees requires very large witness sizes for proving old state, and switching to Verkle trees without state expiry requires an in-place transition procedure (eg. [EIP 2584](#)) that is almost as complicated as just implementing state expiry.

See the full proposal [here](#)

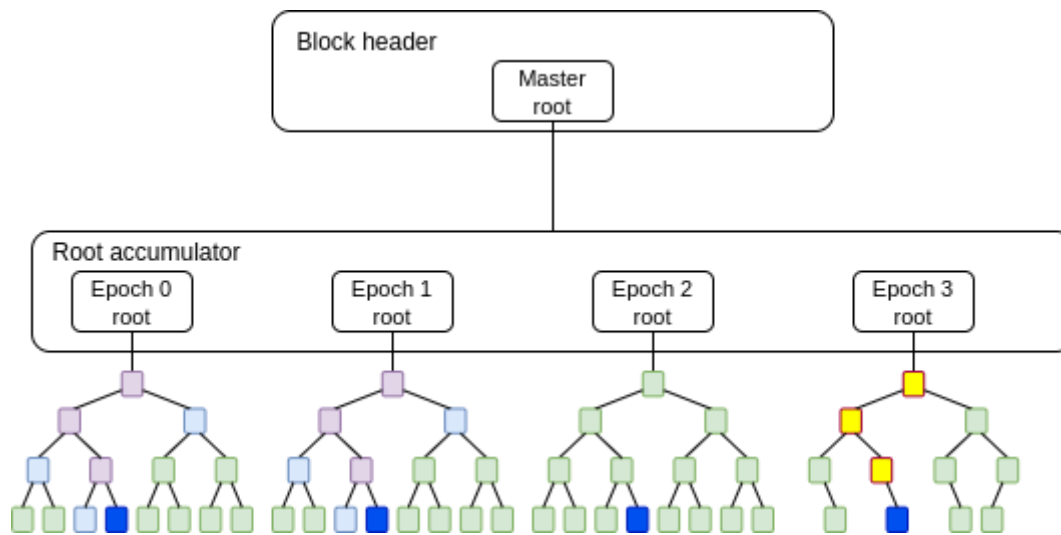
The core idea is that there would be a state tree per epoch (think: 1 epoch ~= 8 months), and when a new epoch begins, an empty state tree is initialized for that epoch and any state updates go into that tree.

Full nodes in the network would only be required to store the most recent two trees, so on average they would only be storing state that was read or written in the last ~1.5 epochs ~= 1 year.

Block producers will in addition to the block provide a 'witness' that the data is required to execute the transactions in the block.

There are two key principles:

- Only the most recent tree (ie. the tree corresponding to the current epoch) can be modified. All older trees are no longer modifiable; objects in older trees can only be modified by creating copies of them in newer trees, and these copies supersede the older copies.
- Full nodes (including block proposers) are expected to only hold the most recent two trees, so only objects in the most recent two trees can be read without a witness. Reading older objects requires providing witnesses.



Suppose the dark-blue object was last modified in epoch 0, and you want to read/write it in a transaction in epoch 3.

To prove that epoch 0 really was the last time the object was touched, we need to prove the dark-blue values in epochs 0, 1 and 2.

Full nodes still have the full epoch 2 state, so no witness is required.

For epochs 0 and 1, we do need witnesses: the light blue nodes, plus the purple nodes that can be regenerated during witness verification.

After this operation, a copy of the object is saved in the epoch 3 state.

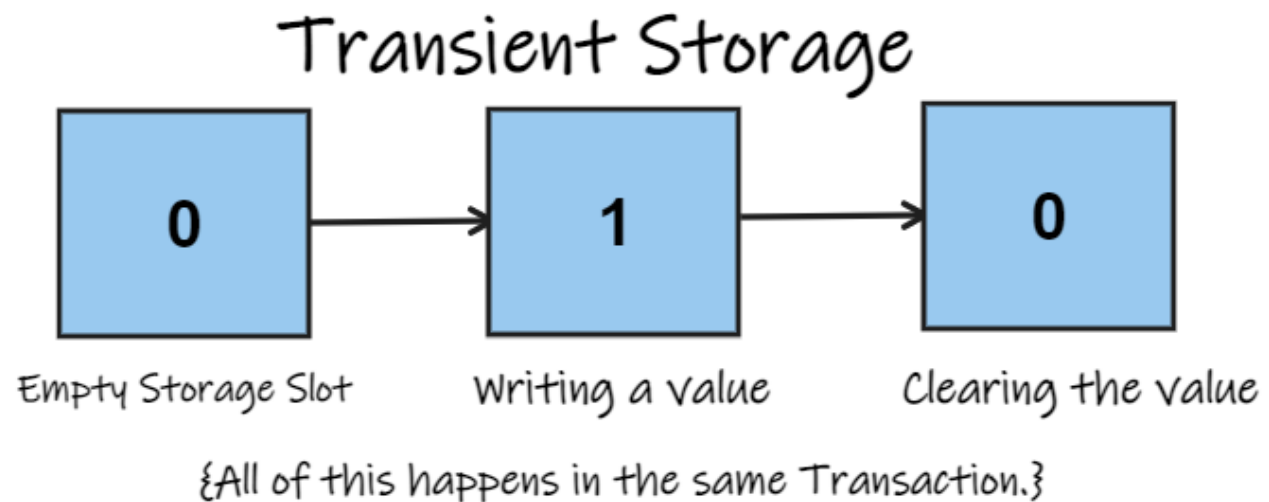
TransientStorage

Transient storage

See [EIP-1153](#)
and [discussion](#)

Add opcodes for manipulating state that behaves identically to storage but is discarded after every transaction.

See [overview article](#)



Because the blockchain doesn't have to store transient data after the transaction, nodes don't have to use the disk, making it much less expensive than storage.

So what is the difference between this and using memory ?

Transient storage is available when calling other contracts, for example with `DELEGATECALL`

Pros and Cons

Pros	Cons
It does not change the semantics of the existing operations.	It does not address transient usages of storage in existing contracts.
It will help future storage designs like Verkle trees.	It introduces new codes to the clients.
There is no need to clear storage slots after usage.	

Use cases

Re entrancy locks

Example from Uniswap V2

```

bool private locked;
modifier lock() {
    require(!locked, 'UniswapV2: LOCKED');
    locked = true;
    _;
    locked = false;
}

```

Non custodial flash loans

See [example](#)

```

interface IStartCallback {
    /// @notice Called on the `msg.sender` to hand over control to them.
    /// Expectation is that msg.sender#start will borrow tokens using
    NonCustodialFlashLoans#borrow,
    /// then return them to the original user before control is handed back to
    #start.
    function start() external;
}

contract NonCustodialFlashLoans {
    struct Borrow {
        uint256 lenderStartingBalance;
        address lender;
        IERC20 token;
    }

    // The full list of borrows that have occurred in the current transaction.
    Borrow[] public transient borrows;

    // The user borrowing. Borrower is able to call #borrow to release tokens.
    address public transient borrower;

    /// @notice Entry Point. Start borrowing from the users that have approved
    this contract.
    function startLoan() external {
        require(borrower == address(0)); // prevent reentrance

        // TSTORE it!
        borrower = msg.sender;

        /// Hand control to the caller so they can start borrowing tokens
        IStartCallback(msg.sender).start();

        // At this point `msg.sender` should have returned any tokens that
        // were borrowed to each lender. Check this and revert if not!
        for (uint256 i = 0; i < borrows.length; i++) {

```



```

        Borrow transient borrow = borrows[i]; // TLOAD!
        require(
            borrow.token.balanceOf(borrow.lender) >=
borrow.lenderStartingBalance,
            'You must pay back the person you borrowed from!'
        );
    }

    borrows.length = 0; // this doesn't actually work in recent solidity
versions for storage arrays, but we only need to set the length of the array, you
can also use TSTORE directly

    borrower = address(0); // clearing this allows it to be called again in
the same transaction
}

// Only callable by `borrower`. Used to borrow tokens.
function borrow(
    address from,
    IERC20 token,
    uint256 amount,
    address to
) external {
    require(msg.sender == borrower, 'Must be called from within the
IStartCallback#start');

    // TSTORE what has been borrowed
    borrows.push(Borrow({lenderStartingBalance: token.balanceOf(from),
lender: from, token: token}));

    token.transferFrom(from, to, amount);
}
}

```

The user calls `NonCustodialFlashLoans#startLoan`, and control is handed back to them using the `IStartCallback#start` callback.

The user can then borrow as much of any token they want, from any user that has sent approval, within this callback by calling `NonCustodialFlashLoans#borrow`.

The contract uses transient storage to track what has been borrowed and check the tokens are returned at the end.

Alternately The SLOAD/SSTORE's make this pattern gas-infeasible without transient storage.

Huff

Repo

"Huff enables the construction of EVM assembly macros - blocks of bytecode that can be rigorously tested and evaluated. Macros can themselves be composed of Huff macros.

Huff doesn't hide the workings of the EVM behind syntactic sugar. In fact, Huff doesn't hide anything at all. Huff does not have variables, instead directly exposing the EVM's program stack to the developer to be directly manipulated."

Huff supports tables of jump destinations integrated directly into the contract bytecode. This is to enable efficient program execution flows by using jump tables instead of conditional branching.

A series of blog posts about [Huff](#)













Documentation is [here](#)



Series of [tutorials](#)

Getting started




Huff project template

See <https://docs.huff.sh/get-started/project-quickstart/#using-the-template>


 .github/workflows	Initial commit	now
 assets	Initial commit	now
 lib	Initial commit	now
 script	Initial commit	now
 src	Initial commit	now
 test	Initial commit	now
 .env.example	Initial commit	now
 .gitignore	Initial commit	now
 .gitmodules	Initial commit	now
 LICENSE	Initial commit	now
 README.md	Initial commit	now
 foundry.toml	Initial commit	now

 README.md 

huff-project-template •

 ci passing  license Unlicense  solidity ^0.8.15

Versatile Huff Project Template using Foundry.



Compiling the contract

see <https://docs.huff.sh/get-started/compiling/#compiling-contracts-with-the-huff-compiler>

Example approve

```
/// @notice Approve
/// @notice Approves the spender to use up to amount of the specified token up until the expiration
#define macro APPROVE() = takes (0) returns (0) {
    NON_PAYABLE() // []

    0x24 calldataload // [spenderAddress]
    0x04 calldataload // [tokenAddress, spenderAddress]
    caller // [caller, tokenAddress, spenderAddress]

    _ALLOWANCE() // [amount, expiration, nonce, packedAllowance]
    pop pop pop // [packedAllowance] let me have a clean stack, screw optimisation...

    // Clean packedAllowance data except nonce
    [UINT_48_MAX] and // [packedAllowance1]

    // Put expiration on the right part of packedAllowance
    0x64 calldataload // [expiration, packedAllowance1]
    dup1 iszero // [expiration == 0x00, expiration, packedAllowance1]
    dup1 timestamp mul // [timestamp * (expiration == 0x00), expiration == 0x00, expiration, packedAllowance1]
    swap1 not // [expiration != 0x00, timestamp * (expiration == 0x00), expiration, packedAllowance1]
    dup3 mul or // [(expiration * (expiration != 0x00)) | (timestamp * (expiration == 0x00)), expiration, packedAllowance1]
    swap1 pop // [newExpiration, packedAllowance1]
    [UINT_48_MAX] and [UINT_48_MAX_PLUS_1] mul // [newExpirationCooked, packedAllowance1]
    or // [packedAllowance2]

    // Put amount on the right part of packedAllowance
    0x44 calldataload // [newAmount, packedAllowance2]
    [UINT_160_MAX] and [UINT_160_MAX_PLUS_1] mul // [newAmountCooked, packedAllowance2]
    or // [packedAllowanceFinal]

    0x24 calldataload // [spenderAddress, packedAllowanceFinal]
    0x04 calldataload // [tokenAddress, spenderAddress, packedAllowanceFinal]
    caller // [caller, tokenAddress, spenderAddress, packedAllowanceFinal]
    [APPROVAL_SLOT] // [slot, caller, tokenAddress, spenderAddress, packedAllowanceFinal]

    STORE_ELEMENT_FROM_KEYS_3D(0x00) // []

    // Stop Execution
    stop // []
}
```

Example getting storage values


```
#define macro _ALLOWANCE() = takes (3) returns (4) {
    // Input stack: [ownerAddress, tokenAddress, spenderAddress]
    [APPROVAL_SLOT] // [slot, ownerAddress, tokenAddress, spenderAddress]
    LOAD_ELEMENT_FROM_KEYS_3D(0x00) // [packedAllowance] -> amount(uint160), expiration(uint48), nonce(uint48)

    // break bitpacking
    dup1 [UINT_48_MAX] and // [nonce, packedAllowance]
    dup2 0x30 shr [UINT_48_MAX] and // [expiration, nonce, packedAllowance]
    dup3 0x60 shr [UINT_160_MAX] and // [amount, expiration, nonce, packedAllowance]
}
```

Resources

Huffmate

Plugin for VSCode



Huff

v0.0.32Preview

huff-language | 479 | ★★★★★

VSCode syntax highlighting for huff programming language.

[Disable](#) [Uninstall](#) ⚙️

This extension is enabled globally.

Compiler built in rust

huff-rs

Foundry x Huff

CI passing License Apache 2.0 chat 154 online

A [foundry](#) library for working with [huff](#) contracts. Take a look at our [project template](#) to see an example project that uses this library.



Library

Install with

```
curl -L get.huff.sh | bash
forge install huff-language/foundry-huff
```

Huff Macros

There are only two fundamental building blocks to a Huff program:

- Macros
- Jump tables (and packed jump tables)

Example Macros

```
template <p1,p2>
#define macro POINT_DOUBLE = takes(3) returns(3) {
    <p1> dup3 callvalue shl
    swap3 dup4 mulmod
    <p2> dup2 callvalue shl
    dup2 dup1 dup1 dup4 dup10
    mulmod dup2 sub swap8
    dup1 mulmod 0x03 mul
    dup2 dup2 dup1
    mulmod dup9 callvalue shl add swap8
    dup9 add mulmod swap3 mulmod add swap2
    <p2> swap2 mulmod <p1> sub
}
```

Huff Jump tables

See [Documentation](#)

From docs :

Jump Tables are a convenient way to create switch cases in your Huff contracts. Each jump table consists of jumpdest program counters (PCs), and it is written to your contract's bytecode. These jumpdest PCs can be codecopied into memory, and the case can be chosen by finding a jumpdest at a particular memory pointer (i.e. 0×00 = case 1, 0×20 = case 2, etc.). This allows for a single jump rather than many conditional jumps.

There are two different kinds of Jump Tables in Huff: `Regular` and `Packed`. Regular Jump Tables store jumpdest PCs as full 32 byte words, and packed Jump Tables store them each as

2 bytes. Therefore, packed jumptables are cheaper to copy into memory, but they are more expensive to pull a PC out of due to the bitshifting required. The opposite is true for Regular Jump Tables.

There are two builtin functions related to jumptables.

`__tablestart(TABLE)`

Pushes the program counter (PC) of the start of the table passed to the stack.

`__tablesize(TABLE)`

Pushes the code size of the table passed to the stack.

Example

```
// Define a function
#define function switchTest(uint256) pure returns (uint256)

// Define a jump table containing 4 pcs
#define jumptable SWITCH_TABLE {
    jump_one jump_two jump_three jump_four
}

#define macro SWITCH_TEST() = takes (0) returns (0) {
    // Codecopy jump table into memory @ 0x00
    __tablesize(SWITCH_TABLE)    // [table_size]
    __tablestart(SWITCH_TABLE)  // [table_start, table_size]
    0x00
    codecopy

    0x04 calldataload            // [input_num]

    // Revert if input_num is not in the bounds of [0, 3]
    dup1                        // [input_num, input_num]
    0x03 lt                      // [3 < input_num, input_num]
    err jumpi

    // Regular jumptables store the jumpdest PCs as full words,
    // so we simply multiply the input number by 32 to determine
    // which label to jump to.
    0x20 mul                    // [0x20 * input_num]
    mload                       // [pc]
    jump                        // []

    jump_one:
        0x100 0x00 mstore
        0x20 0x00 return
    jump_two:
        0x200 0x00 mstore
        0x20 0x00 return
    jump_three:
        0x300 0x00 mstore
        0x20 0x00 return
```

```

    jump_four:
        0x400 0x00 mstore
        0x20 0x00 return
    err:
        0x00 0x00 revert
}

#define macro MAIN() = takes (0) returns (0) {
    // Identify which function is being called.
    0x00 calldataload 0xE0 shr
    dup1 __FUNC_SIG(switchTest) eq switch_test jumpi

    // Revert if no function matches
    0x00 0x00 revert

    switch_test:
        SWITCH_TEST()
}

```

Constants in Huff

```

#define constant NUM = 0x420
#define constant HELLO_WORLD = 0x48656c6c6f2c20576f726c6421
#define constant FREE_STORAGE = FREE_STORAGE_POINTER()

```

In order to push a constant to the stack, use bracket notation: `[CONSTANT]`

Custom Errors

Custom errors can be defined and used by the `__ERROR` builtin to push the left-padded 4 byte error selector to the stack.

Jump Labels

Jump Labels are a simple abstraction included into the language to make defining and referring to `JUMPDEST`s more simple for the developer.

```

#define macro MAIN() = takes (0) returns (0) {
    // Store "Hello, World!" in memory
    0x48656c6c6f2c20576f726c6421
    0x00 mstore // ["Hello, World!"]

    // Jump to success label, skipping the revert statement
    success      // [success_label_pc, "Hello, World!"]
    jump         // ["Hello, World!"]

    // Revert if this point is reached
    0x00 0x00 revert

    // Labels are defined within macros or functions, and are designated
    // by a word followed by a colon. Note that while it may appear as if

```

```
// labels are scoped code blocks due to the indentation, they are simply
// destinations to jump to in the bytecode. If operations exist below a
label,
// they will be executed unless the program counter is altered or execution
is
// halted by a `revert`, `return`, `stop`, or `selfdestruct` opcode.
success:
    0x00 mstore
    0x20 0x00 return
}
```