https://blog.chain.link/solidity-vs-vyper/

Solidity vs. Vyper: Which Smart Contract Language Is Right for Me?

October 17, 2022  28 min read

Author: [Chainlink](https://blog.chain.link/author/chainlink/)

* [Solidity vs. Vyper: Which Smart Contract Language Is Right for Me?](https://blog.chain.link/solidity-vs-vyper/#post-title)
* [Which Smart Contract Language Should I Use?](https://blog.chain.link/solidity-vs-vyper/#which_smart_contract_language_should_i_use_)
* [EVM Programming Languages](https://blog.chain.link/solidity-vs-vyper/#evm_programming_languages)
* [What Is Solidity?](https://blog.chain.link/solidity-vs-vyper/#what_is_solidity_)
* [What Is Vyper?](https://blog.chain.link/solidity-vs-vyper/#what_is_vyper_)
* [The Current Landscape](https://blog.chain.link/solidity-vs-vyper/#the_current_landscape)
* [Comparing the Same Contract](https://blog.chain.link/solidity-vs-vyper/#comparing_the_same_contract)
* [Which Smart Contract Language Is More Gas-Optimized?](https://blog.chain.link/solidity-vs-vyper/#which_smart_contract_language_is_more_gas-optimized_)
* [Gas Cost Comparisons](https://blog.chain.link/solidity-vs-vyper/#gas_cost_comparisons)
* [Contract Creation Code](https://blog.chain.link/solidity-vs-vyper/#contract_creation_code)
* [Runtime Code](https://blog.chain.link/solidity-vs-vyper/#runtime_code)
* [Free Memory Pointer](https://blog.chain.link/solidity-vs-vyper/#free_memory_pointer)
* [Dynamic Arrays](https://blog.chain.link/solidity-vs-vyper/#dynamic_arrays)
* [Solidity vs. Yul vs. SolYul](https://blog.chain.link/solidity-vs-vyper/#solidity_vs._yul_vs._solyul)
* [Metadata](https://blog.chain.link/solidity-vs-vyper/#metadata)
* [Summary](https://blog.chain.link/solidity-vs-vyper/#summary)
* [Looking Forward](https://blog.chain.link/solidity-vs-vyper/#looking_forward)

*This article was written based on information from September 2022. Special thanks to*[*0xKitsune*](https://twitter.com/0xKitsune)*,*[*Hari*](https://twitter.com/_hrkrshnn)*,*[*Doggie B*](https://twitter.com/fubuloubu)*,*[*Alex Beregszaszi*](https://twitter.com/alexberegszaszi)*,*[*Zubin Pratap*](https://twitter.com/ZubinPratap)*, and*[*Sudo Init Vyper*](https://twitter.com/big_tech_sux)*.*

This article examines the following question—which smart contract language is better for me, Solidity or Vyper? Recently, there has been a lot of debate about which is “the best” smart contract language, with maxis on each side advocating for their chosen language.

I’m here to answer the main question at the bottom of this debate:

**Which Smart Contract Language Should I Use?**

To get to the bottom of the question, we’re going to talk about tooling and useability before considering one of the main questions for any smart contract developer: gas optimization. Specifically, we are going to look at four[EVM](https://ethereum.org/en/developers/docs/evm/) languages (the languages that work on chains like[Ethereum](https://ethereum.org/en/),[Avalanche](https://docs.avax.network/),[Polygon](https://polygon.technology/), etc.):[Solidity](https://docs.soliditylang.org/en/v0.8.16/),[Vyper](https://vyper.readthedocs.io/en/stable/index.html),[Huff](https://docs.huff.sh/), and[Yul](https://docs.soliditylang.org/en/latest/yul.html). Sorry[Rust](https://docs.solana.com/developing/on-chain-programs/developing-rust), you’ll have to wait your turn for an article covering non-EVM chains.

But first, a spoiler alert.

*Solidity, Vyper, Huff, and Yul are all great languages that get the job done. Solidity and Vyper are fantastic high-level languages that most people should use. If you’re interested in writing near-assembly code, Yul and Huff do the job.*

So guess what, if you’re stuck on picking one,[flip a coin](https://justflipacoin.com/): You’ll be successful whichever language you pick, I promise. If you’re new to smart contract programming languages, you can do great things with whichever languages you like best, or whichever one you choose at random.

Additionally, these languages change all the time, and you could easily cherry-pick smart contracts and data to make different languages seem better or worse. Keep this in mind when we get to the gas optimization comparisons. We’ve chosen a minimal contract for comparison, if you think you have a better example, we’d love to see it!

Now, if you’re a veteran of this space, let’s go deep under the hood of these languages. Prepare to geek out.

EVM Programming Languages

The four languages that we are going to be looking at are as follows:

* [Solidity](https://docs.soliditylang.org/en/v0.8.16/): Currently the most popular language by[DeFi TVL](https://impermax.medium.com/defi-explained-what-is-tvl-9800eda80b0b). High-level and similar to JavaScript.
* [Vyper](https://vyper.readthedocs.io/en/stable/index.html): Currently the second most popular language by DeFi TVL. High-level and similar to Python.
* [Huff](https://huff.sh/): A low-level language similar to Assembly.
* [Yul](https://docs.soliditylang.org/en/latest/yul.html): A low-level language similar to Assembly that comes built-in with Solidity (although some argue it’s [still too high level](https://github.com/ethereum/solidity/issues/13247)).

**Why These Four?**

We are using these four languages as they are all EVM-compatible, with Solidity and Vyper being the top two most popular languages by far. I’ve added Yul as it’s not fair to make gas optimization comparisons with Solidity without taking Yul into account. We’ve added Huff because we wanted to benchmark to a language that is nearly identical to writing in opcodes but isn’t Yul.

As far as the EVM goes, after Vyper and Solidity, third, fourth, and fifth are leaps and bounds down in popularity. Sorry to the languages that didn’t make this comparison; the adoption just isn’t there. However, there are many promising smart contract languages on the rise, and I’m looking forward to trying them out in the future.

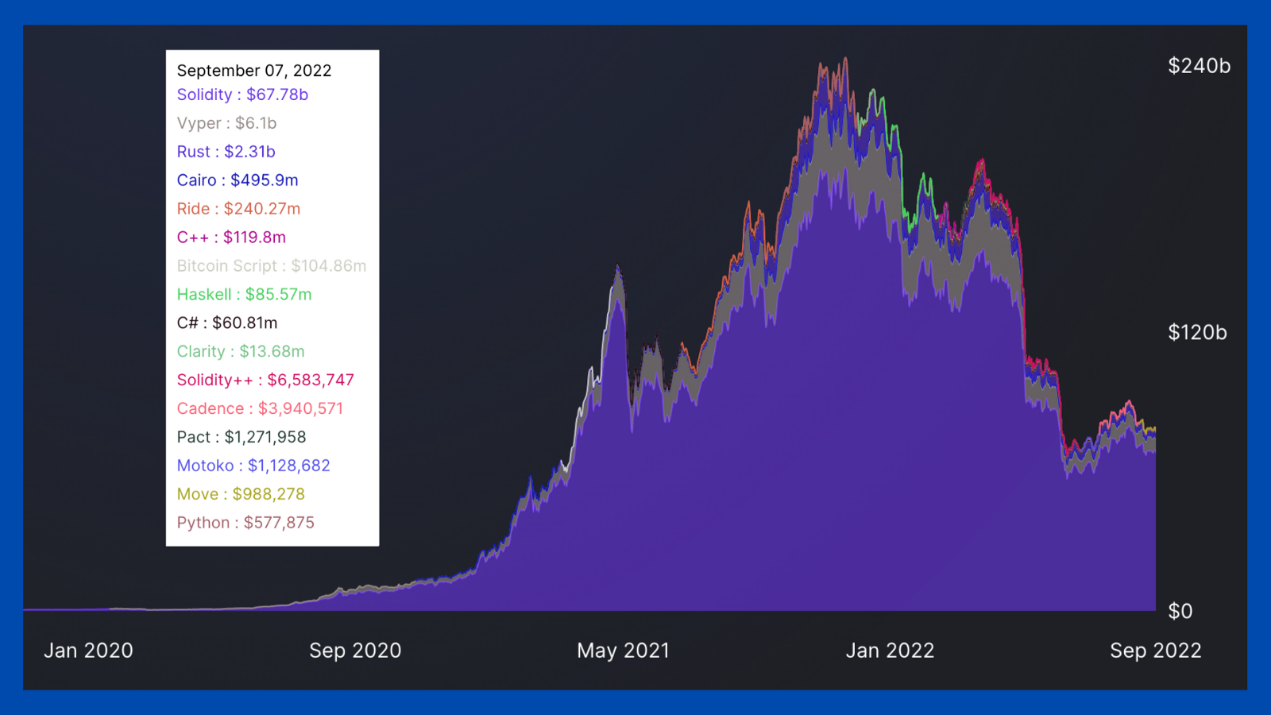
**What Is Solidity?**

Solidity is an object-oriented programming language for implementing smart contracts on Ethereum and other blockchains. Solidity is highly influenced by C++, Python, and JavaScript and has been designed for the EVM.

**What Is Vyper?**

Vyper is a contract-oriented, pythonic programming language also designed for the EVM. Vyper was designed to improve upon Solidity by aiming to enhance readability and limit certain practices. On a high level, Vyper seeks to optimize the security and auditability of smart contracts.

The Current Landscape

Adapted from [DefiLlama’s languages tab](https://defillama.com/languages" \t "_blank).

According to[DefiLlama](https://defillama.com/languages), as of right now, in the DeFi space, Solidity smart contracts secure 87% of TVL, while Vyper smart contracts secure 8%.

So if pure popularity is what you’re looking for, you need not look farther than Solidity.

Comparing the Same Contract

Now let’s get into what each language looks like and then compare their gas performance.

Here are four *nearly*identical contracts written in each language. They all do approximately the same thing, They all:

1. Have a private number (uint256) at storage slot 0.
2. Have a function with the readNumber()[function signature](https://ethereum.stackexchange.com/q/135205/57451) that reads what’s at storage slot 0.
3. Allow you to update that number with a storeNumber(uint256) function signature.

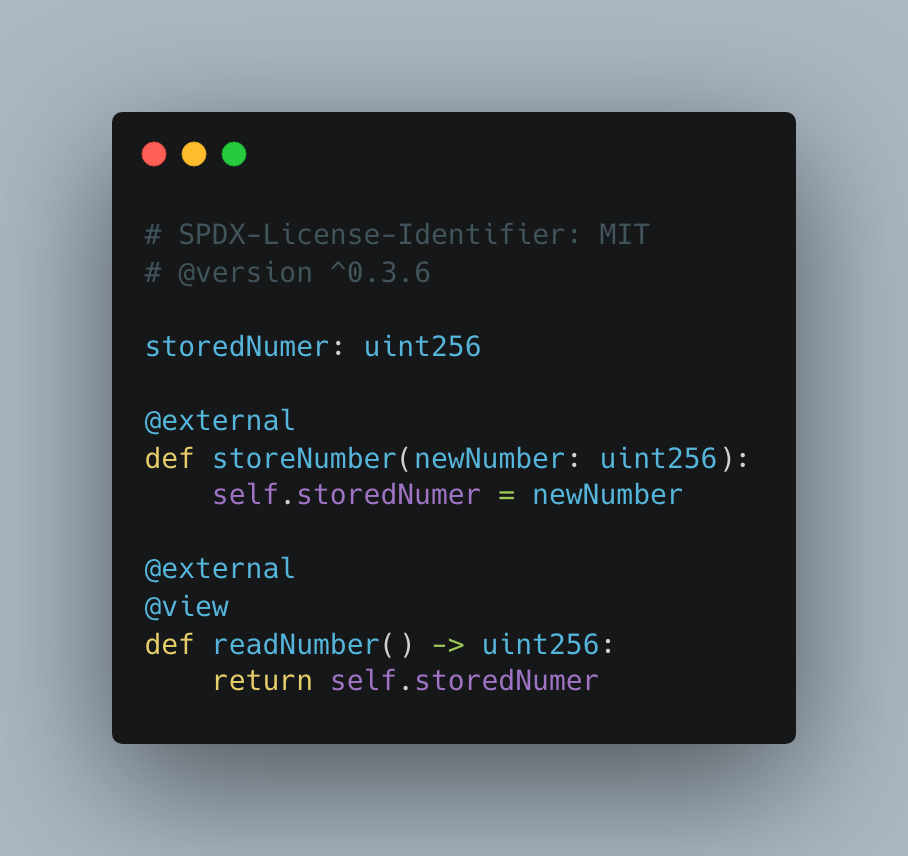
That’s it. Here are those contracts.

*All the code we’ve used to compare languages is located in*[*this GitHub repo*](https://github.com/PatrickAlphaC/sc-language-comparison)*.*

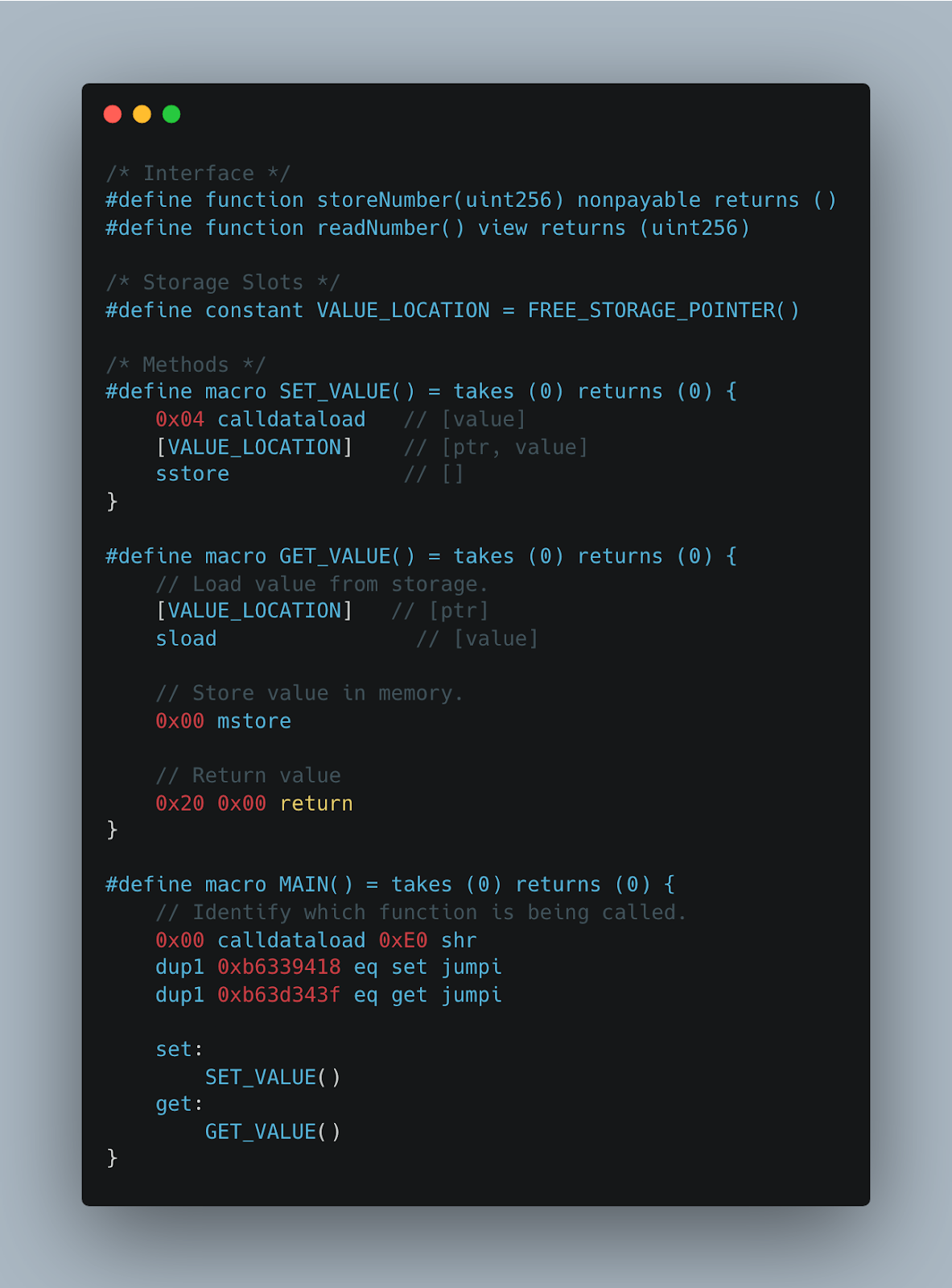
**Solidity**



**Vyper**



**♞ Huff**



**Yul**



**Developer Experience**

Just by looking at these four images, we can start to see a picture of how each language feels to write. As far as developer experience goes, **it’s substantially quicker to write Solidity and Vyper code.** This makes a lot of sense: those languages are higher level, while Yul and Huff are meant to be low-level code. For this reason alone, it’s easy to see why so many people have adopted Vyper and Solidity (as well as the fact that they have each been around much longer).

Focusing on Vyper and Solidity for a second, you can clearly see that Vyper draws inspiration from Python, and Solidity from JavaScript and Java. So if you like the feel of one of those languages, great—use it.

Vyper is meant to be a[minimalistic, easily audited programming language](https://vyper.readthedocs.io/en/stable/index.html#principles-and-goals), while Solidity is meant to be a general-purpose smart contract language. The experience of coding definitely feels like it too on a syntactic level, but I’ll let you make up your own mind on this subjective point.

I’m not going to go into tooling too much, as most of these languages have very similar tooling. Most of the main frameworks, including[Hardhat](https://hardhat.org/), ape, titanoboa,[Brownie](https://github.com/eth-brownie/brownie), and[Foundry](https://github.com/foundry-rs/foundry), have Vyper and Solidity support. Solidity has “priority citizenship” with most of these frameworks, whereas Vyper needs to[use a plugin](https://www.npmjs.com/package/@nomiclabs/hardhat-vyper) to work with tools like Hardhat. However, titanoboa is built to work specifically and only with Vyper, and most tooling is easy enough to use with either.

Which Smart Contract Language Is More Gas-Optimized?

Now for the main event. When comparing the gas performance of smart contracts, there are two main things to keep in mind:

1. Contract creation gas costs
2. Runtime gas costs

How you implement a smart contract can have a major impact on these factors. You could, for example, store a massive array in the code of your contract, making it expensive to deploy but cheaper to run a function with. Or, you could have your function generate the array on the fly, making the contract cheaper to deploy but more expensive to run.

So, let’s look at those four contracts and compare their contract creation gas costs against their runtime gas costs. You can find all my code on this, including the frameworks and tooling used to compare them, in my[sc-language-comparison repo.](https://github.com/PatrickAlphaC/sc-language-comparison)

**Gas Cost Comparisons Summary**

Here is how we compiled the contracts for this section:

vyper src/vyper/VSimpleStorage.vy

huffc src/huff/HSimpleStorage.huff -**b**

solc --strict-assembly --optimize --optimize-runs 20000

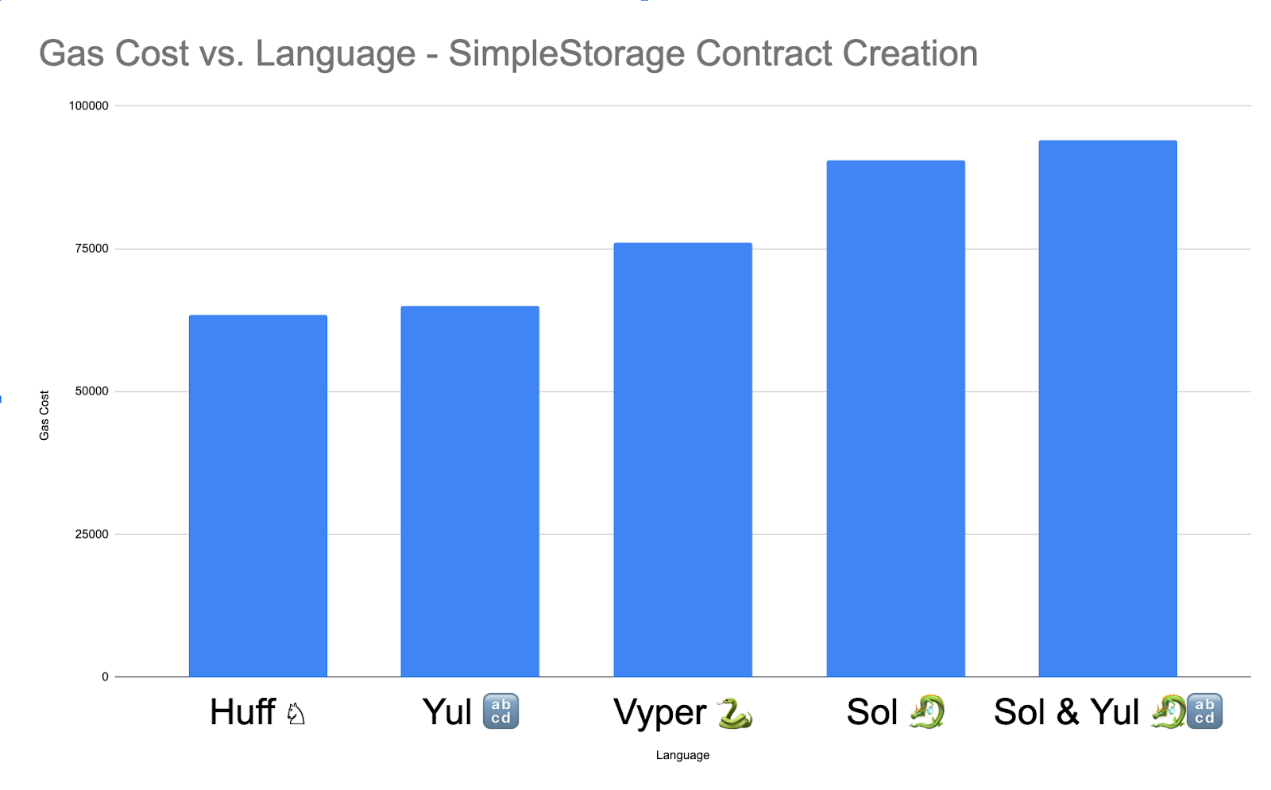
yul/YYSimpleStorage.yul --bin

solc --optimize --optimize-runs 20000 src/yulsol/YSimpleStorage.sol --bin

solc --optimize --optimize-runs 20000 src/solidity/SSimpleStorage.sol --bin

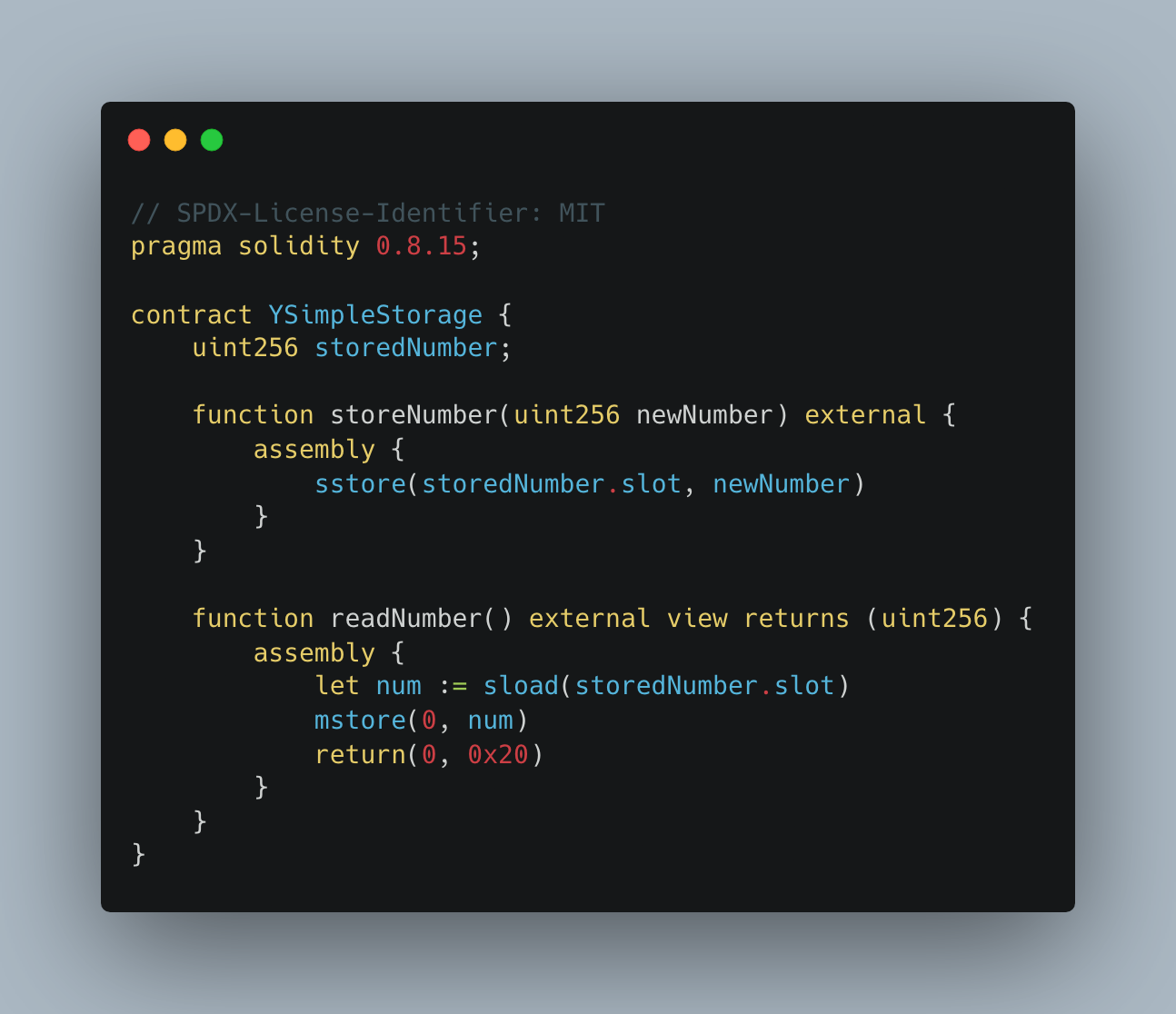
*Note: I could have used the –via-ir flag for the Solidity compilations as well. Also note, Vyper and Solidity add “Metadata” to the end of their contracts. This accounts for a small addition in overall gas costs, but not enough to change the rankings below. I’ll talk about this more in the metadata section.*

Results:

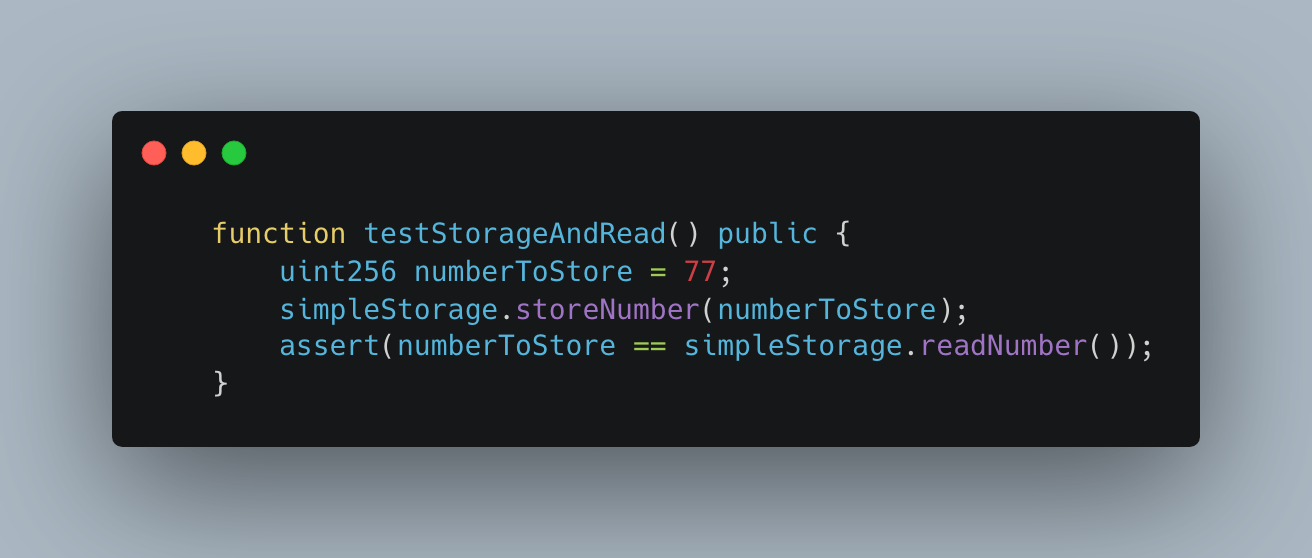
Contract creation gas costs across languages.

As we can see, lower-level languages like Huff and Yul are more gas efficient than Vyper and Solidity, but why is that? Vyper seems to be more efficient than Solidity, and we have this new “Sol and Yul” section. Well, that’s because you can actually write Yul *inside*Solidity. Yul was created as a language for Solidity developers to write in when they needed to get closer to machine code.

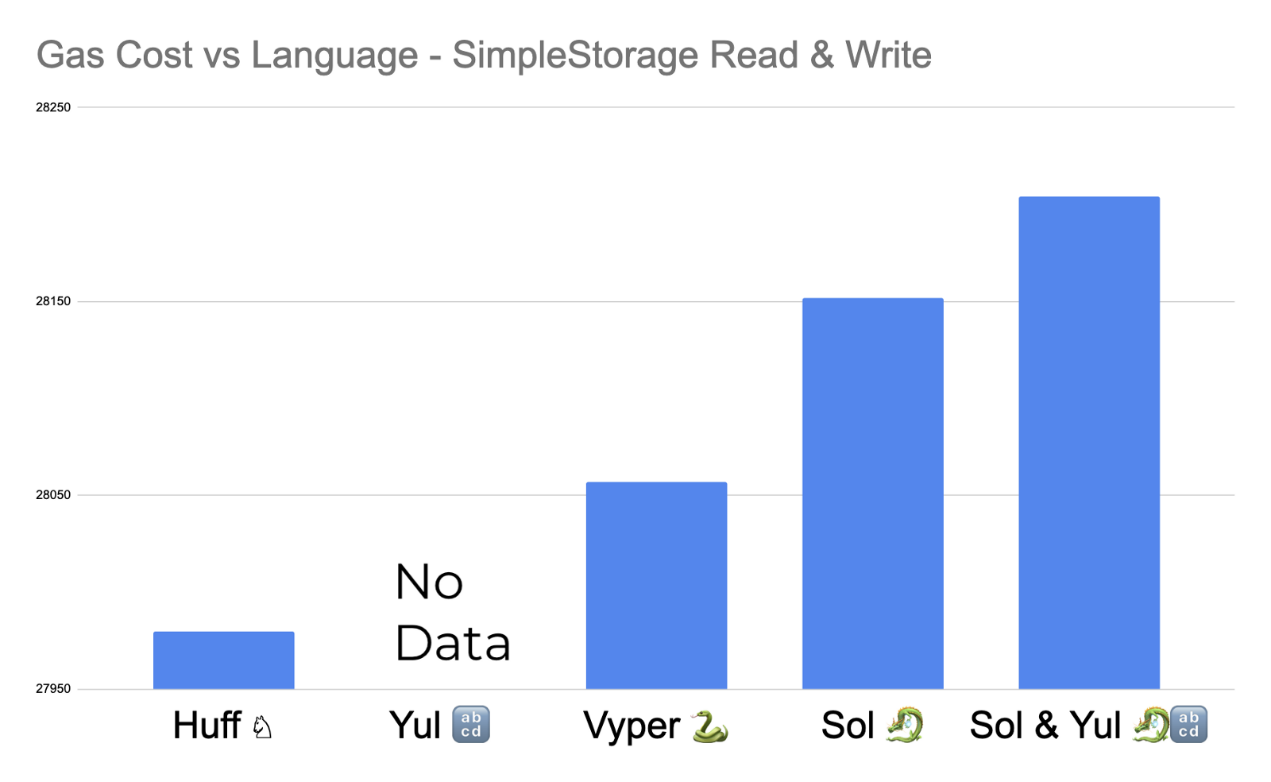
So in the chart above, we compare raw Yul, raw Solidity, and a Solidity-Yul mix. The Solidity-Yul version of our code looks like this:

Yul and Solidity combined.

You’ll see an example later where this inline-Yul makes a **major**difference in gas costs. We’ll take a look at why these gas differences exist later, but now let’s look at the gas costs associated with a single test in[Foundry](https://github.com/foundry-rs/foundry).

Our testing function.

This will test the gas cost of storing the number 77 in storage, and then reading the number from storage. Here are the results of running this test.

SimpleStorage read and write gas comparisons.

We didn’t have data for Yul since we’d have to make a Yul-Foundry plugin, which I didn’t want to have to do — and I’d bet the results would be similar to Huff anyway. Keep in mind that this is the gas cost of running the whole testing function, not just the individual functions.

Gas Cost Comparisons

Ok, let’s analyze this data. The first question we need to answer is: Why is Huff and Yul contract creation so much more gas efficient than Vyper and Solidity? Well, we can find that answer by looking directly at the bytecode of these contracts.

When you compile a contract, it is usually split into two or three different parts.

1. Contract creation code
2. Runtime code
3. Metadata (optional)

For this section, it’s important to understand the basics of opcodes. OpenZeppelin’s blog on[deconstructing a contract](https://blog.openzeppelin.com/deconstructing-a-solidity-contract-part-i-introduction-832efd2d7737/) is a great starting point.

Contract Creation Code

The contract creation code is the first section of the bytecode that tells the EVM to stick that contract on-chain. You can typically find it by looking for the CODECOPY opcode (39) in the resulting binary, and then finding where it sticks it on-chain and returns with the RETURN opcode (f3) and ends the call.

Huff:

602f8060093d393df3

Yul:

603e80600c6000396000f3fe

Vyper:

61006b61000f60003961006b6000f3

Solidity:

6080604052348015600f57600080fd5b5060ac8061001e6000396000f3fe

Solidity-Yul:

608060405234801561001057600080fd5b5060bc8061001f6000396000f3fe

You’ll also notice a lot of fe opcodes, which is the INVALID opcode. Solidity adds these as markers to show the difference between runtime, contract creation, and metadata code. f3�3 is the RETURN������ opcode, and is usually the end of a function or context.

You might think that because Yul-Solidity has the largest contract creation bytecode and Huff has the smallest, that’s the reason why Huff is the cheapest and Yul-Solidity is the most expensive. But when you copy the entire codebase and stick it on-chain, the size of the codebase makes a big difference and is the main determining factor. However, this contract creation code does give us an eye into how each of these compilers think, and will give us good insight into how they are going to compile our contracts.

**How to Read the Opcodes and Stack**

Now, the EVM is a[stack-based machine](https://www.evm.codes/about), meaning that the majority of “stuff” you do is pushing and pulling stuff off a[stack](https://www.geeksforgeeks.org/stack-data-structure/). You’ll see to the left we have the[opcodes](https://www.evm.codes/), and to the right we have two slashes (//) notating that they are a comment, and what the stack looks like after executing the opcode on the same line, with the top of the stack to the left, and the bottom of the stack to the right.

**Huff Explained**

The Huff contract creation does just about the most minimal stuff it can do. It grabs the code you wrote, and returns it on-chain.

**PUSH** 0x2f        // [2f]

DUP1             // [2f, 2f]

**PUSH** 0x09        // [09, 2f, 2f]

RETURNDATASIZE   // [0, 09, 2f, 2f]

CODECOPY         // [2f]

RETURNDATASIZE   // [0, 2f]

**RETURN**           // []

**Yul Explained**

Yul does the same, it uses a few different opcodes, but essentially, it just puts your code on-chain with as few opcodes as possible, and one INVALID opcode.

**PUSH** 0x3e  // [3e]

DUP1       // [3e, 3e]

**PUSH** 0x0c  // [0c, 3e, 3e]

**PUSH** 0x0   // [0, 0c, 3e, 3e]

CODECOPY   // [3e]

**PUSH** 0x0   // [0, e3]

**RETURN**     // []

INVALID    // []

**Vyper Explained**

Vyper is pretty much the same.

PUSH2 0x06B  // [06B]

PUSH2 0x0F   // [0F, 06B]

PUSH1 0x0    // [0, 0F, 06B]

CODECOPY     // []

PUSH2 0x06B  // [06B]

PUSH1 0x0    // [0, 06B]

RETURN       // []

**Solidity Explained**

Now let’s take a look at the Solidity opcodes.

// Free Memory Pointer

PUSH1 0x80   // [80]

PUSH1 0x40   // [40]

MSTORE       // []

// Check msg.value

CALLVALUE    // [msg.value]

DUP1         // [msg.value, msg.value]

ISZERO       // [msg.value == 0, msg.value]

PUSH1 0xF    // [F, msg.value == 0, msg.value]

JUMPI        // [msg.value] Jump to JUMPDEST if value is not sent

// We only reach this part if msg.value has value

PUSH1 0x0    // [0, msg.value]

DUP1         // [0, 0, msg.value]

REVERT       // [msg.value]

// Finally, put our code on-chain

JUMPDEST     // [msg.value]

POP          // []

PUSH1 0xAC   // [AC]

DUP1         // [AC, AC]

PUSH2 0x1E   // [1E, AC, AC]

PUSH1 0x0    // [0, 1E, AC, AC]

CODECOPY     // [AC]

PUSH1 0x0    // [0, AC]

RETURN       // []

INVALID      // []

Solidity does a lot more stuff. The first thing Solidity does is create what’s called a[Free Memory Pointer](https://docs.soliditylang.org/en/v0.8.16/assembly.html?highlight=free%20memory%20pointer#memory-management). In order to create dynamic arrays in memory, you need to keep track of which parts of your memory are free to use. We don’t use this free memory pointer in our contract construction code, but it’s one of the first things it always does. This is one of the first main differences between languages we’ve uncovered: memory management. Each language handles memory differently.

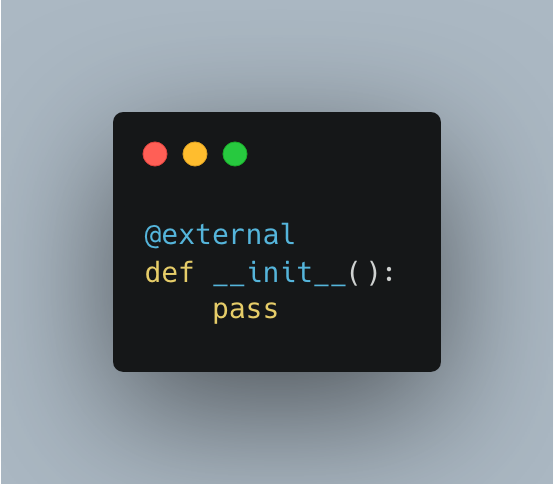
Next, the Solidity compiler looks at your code and notices that you didn’t specify a constructor to be payable. So to make sure you don’t shoot yourself in the foot and accidentally send ETH with your contract creation, it uses the CALLVALUE opcode and starts a check to make sure you don’t send any tokens with your contract creation. This brings us to the second main difference between languages: They each have different checks and protections against common issues.

Finally, Solidity does what the rest of the languages do: It sticks your contract on-chain.

We’re going to skip Solidity-Yul this works in a similar way to Solidity on its own.

**Checks and Protections**

It looks like Solidity is “safer” in this sense since it has many more protections than the rest of the languages. However, if you were to add a constructor to your Vyper code and then re-compile, you’d notice something different.

Vyper Constructor

Compile this and your contract creation code starts to look more like Solidity’s.

// First, we check the callvalue, and jump to a JUMPDEST much later in the opcodes

CALLVALUE

PUSH2 0x080

JUMPI

// This part is identical to the original compilation

PUSH2 0x06B

PUSH2 0x014

PUSH1 0x0

CODECOPY

PUSH2 0x06B

PUSH1 0x0

RETURN

It still doesn’t have the memory management that Solidity has, but you’ll see that it does a check for callvalue with a constructor. If you make the constructor payable and re-compile, that check will then again go away.

So we can come to two conclusions just by looking at these contract creation setups:

1. In Huff and Yul, you’ll need to be explicit about checks and write them yourself.
2. Solidity and Vyper will do checks for you, with Solidity potentially doing more out of the box.

That is going to be one of the biggest tradeoffs between languages: What checks are they performing under the hood? Writing in Huff and Yul will be more effective since both languages are not meant to do anything under the hood. So of course your code will be more gas efficient, but it’ll be tougher for you to keep track of everything that is going on.

Runtime Code

Now that we have some familiarity with what’s going on under the hood, we can look at how the different functions of the contracts perform, and why they perform the way they do.

Let’s look at calling the storeNumber() function, with the value 77 for each language. I’m getting the opcode by walking through the[Forge debug feature](https://book.getfoundry.sh/forge/debugger) with a command like forge test –debug “testStorageAndReadSol”. I also used the[Huff VSCode extension](https://github.com/huff-language/vscode-huff).

**Huff Explained**

// First, we get the function **selector** of the call **and** jump to the code for our storeNumber function

**PUSH** 0x0         // [0]

CALLDATALOAD     // [**b6339418]** The function **selector** for storing

**PUSH** 0xe         // [e, **b6339418]**

SHR              // [**b6339418]**

DUP1             // [**b6339418, b6339418]**

**PUSH** 0xb6339418  // [**b6339418, b6339418, b6339418]**

EQ               // [true, **b6339418]**

**PUSH** 0x1c        // [1c, true, **b6339418]**

JUMPI            // [**b6339418]**

// We skip a **bunch** of opcodes since we jumped

// We place the 77 in storage, **and** end the call

JUMPDEST         // [**b6339418]**

**PUSH** 0x4         // [4, **b6339418]**

CALLDATALOAD     // [4d, **b6339418]** We load 77 from the calldata

**PUSH** 0x0         // [0, 4d, **b6339418]**

SSTORE           // [**b6339418]** Place the 77 in storage

STOP             // [**b6339418]** End call

Interestingly, if we didn’t have the STOP opcode, our Huff code would have actually added a group of opcodes to return the value we just stored, making it more expensive than our Vyper code. But this code still seems very straightforward, so let’s look at how Vyper does it. We are going to skip Yul for now as the results would be pretty similar.

**Vyper Explained**

// First, we do a check on the calldata size to make sure we have at least 4 bytes for a function selector

**PUSH** 0x3        // [3]

CALLDATASIZE    // [3, 24]

GT              // [true]

**PUSH** 0x000c     // [000c, true]

JUMPI           // []

// Then, we jump to our location, and get the function selector

JUMPDEST

**PUSH** 0x0        // [0]

CALLDATALOAD    // [b6339418]

**PUSH** 0xe        // [e, b6339418]

SHR             // [b6339418]

// And we do a check for sending value

CALLVALUE       // [0, b6339418]

**PUSH** 0x0059     // [59, 0, b6339418]

JUMPI           // [b6339418]

// Value looks good, so we compare selectors, and jump if the selector is something else

**PUSH** 0xb6339418 // [b6339418, b6339418]

DUP2            // [b6339418, b6339418, b6339418]

XOR             // [0, b6339418]

**PUSH** 0x0032     // [32, 0, b6339418]

JUMPI           // [b6339418]

// We do a check to make sure the calldata size is big enough for a function selector and a uint256

**PUSH** 0x24       // [24, b6339418]

CALLDATASIZE    // [24, 24, b6339418]

XOR             // [0, b6339418]

**PUSH** 0x0059     // [59, 0, b6339418]

JUMPI           // [b6339418]

// Then, we store the variable and end the call

**PUSH** 0x04       // [4, b6339418]

CALLDATALOAD    // [4d, b6339418]

**PUSH** 0x0        // [0, 4d, b6339418]

SSTORE          // [b6339418]

**STOP**

We can see that we did a few checks along with storing the value:

1. Does the calldata have enough bytes for a function selector?
2. Is their value sent with the call?
3. Is the calldata size a function selector + uint256 sized?

All these checks add gas to our computation, but they also mean that we have a greater chance of not shooting ourselves in the foot.

**Solidity Explained**

// Free Memory Pointer

**PUSH** 0x80        // [80]

**PUSH** 0x40        // [40,80]

MSTORE           // []

// msg.value check, jump to function, revert otherwise

CALLVALUE        // [0]

DUP1             // [0,0]

ISZERO           // [true, 0]

**PUSH** 0x0f        // [0f, true, 0]

JUMPI            // [0]

// Skip reverting code

// We do a check to make sure the calldata size is big enough for a function selector and a uint256

JUMPDEST         // [0]

**POP**              // []

**PUSH** 0x04        // [4]

CALLDATASIZE     // [24, 4]

LT               // [false]

**PUSH** 0x32        // [32, false]

JUMPI            // []

// Find the function selector and jump to it's code

**PUSH** 0x00        // [0]

CALLDATALOAD     // [b6339418]

**PUSH** 0xe0        // [e0, b6339418]

SHR              // [b6339418]

DUP1             // [b6339418, b6339418]

**PUSH** 0xb6339418  // [b6339418, b6339418, b6339418]

EQ               // [true, b6339418]

**PUSH** 0x37        // [37, true, b6339418]

JUMPI            // [b6339418]

// Setup the function by checking the calldata size, and setup the stack for the function

JUMPDEST

**PUSH** 0x47        // [47, b6339418]

**PUSH** 0x42        // [42, 47, b6339418]

CALLDATASIZE     // [24, 42, 47, b6339418]

**PUSH** 0x04        // [4, 24, 42, 47, b6339418]

**PUSH** 0x5e        // [5e, 4, 24, 42, 47, b6339418]

JUMP             // [4, 24, 42, 47, b6339418]

JUMPDEST         // [4, 24, 42, 47, b6339418]

**PUSH** 0x00        // [0, 4, 24, 42, 47, b6339418]

**PUSH** 0x20        // [20, 0, 4, 24, 42, 47, b6339418]

DUP3             // [4, 20, 0, 4, 24, 42, 47, b6339418]

DUP5             // [24, 4, 20, 0, 4, 24, 42, 47, b6339418]

SUB              // [20, 20, 0, 4, 24, 42, 47, b6339418]

// See if the calldatasize minus the function selector size is smaller than 32 bytes

SLT              // [false(0), 0, 4, 24, 42, 47, b6339418]

ISZERO           // [true, 0, 4, 24, 42, 47, b6339418]

**PUSH** 0x6f        // [6f, true, 0, 4, 24, 42, 47, b6339418]

JUMPI            // [0, 4, 24, 42, 47, b6339418]

// Get the 77 value, and jump to the function selector code

JUMPDEST

**POP**              // [24, 42, 47, b6339418]

CALLDATALOAD     // [4d, 24, 42, 47, b6339418]

SWAP2            // [42, 24, 4d, 47, b6339418]

SWAP1            // [24, 42, 4d, 47, b6339418]

**POP**              // [42, 4d, 47, b6339418]

JUMP             // [4d, 47, b6339418]

JUMPDEST         // [4d, 47, b6339418]

// Store our 77 value to storage and end the function call

**PUSH** 0x00        // [0, 4d, 47, b6339418]

SSTORE           // [47, b6339418]

JUMP             // [b6339418]

JUMPDEST         // [b6339418]

**STOP**

There is a lot to unpack here. What are some of the main differences between this and the Huff code?

1. We set up a free memory pointer.
2. We did a check on the value being sent.
3. We did a check on calldata size for the function selector.
4. We did a check on the size of the uint256.

What about the main differences between Solidity and Vyper?

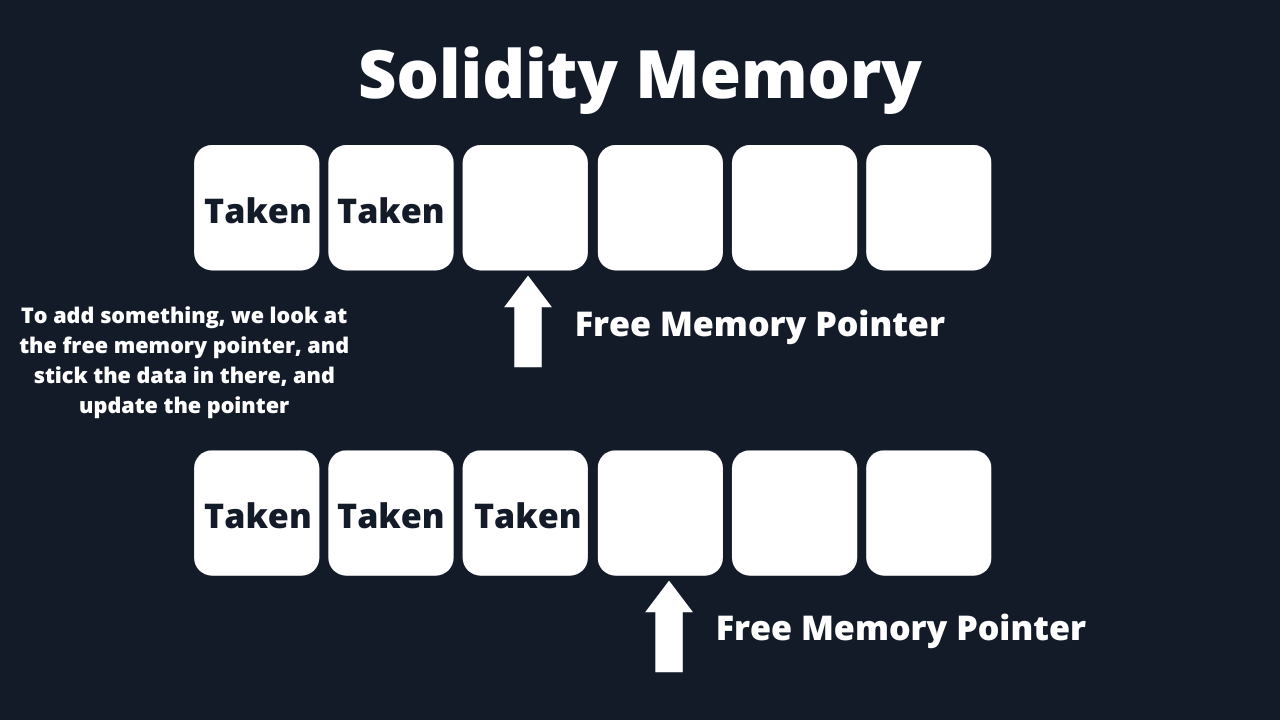
1. Free memory pointer setup.
2. Stack was much deeper at some points.

The two of these combined seem to be the rationale behind Vyper being cheaper than Solidity. It’s also interesting that Solidity uses ISZERO for its checks and Vyper uses XOR; both seem to need about the same gas though. It’s these little design differences that make all the difference!

So we can now see why Huff and Yul are cheaper in gas: they are very specific about doing exactly what you tell them to do, nothing more, whereas Vyper and Solidity try to protect you from doing something silly.

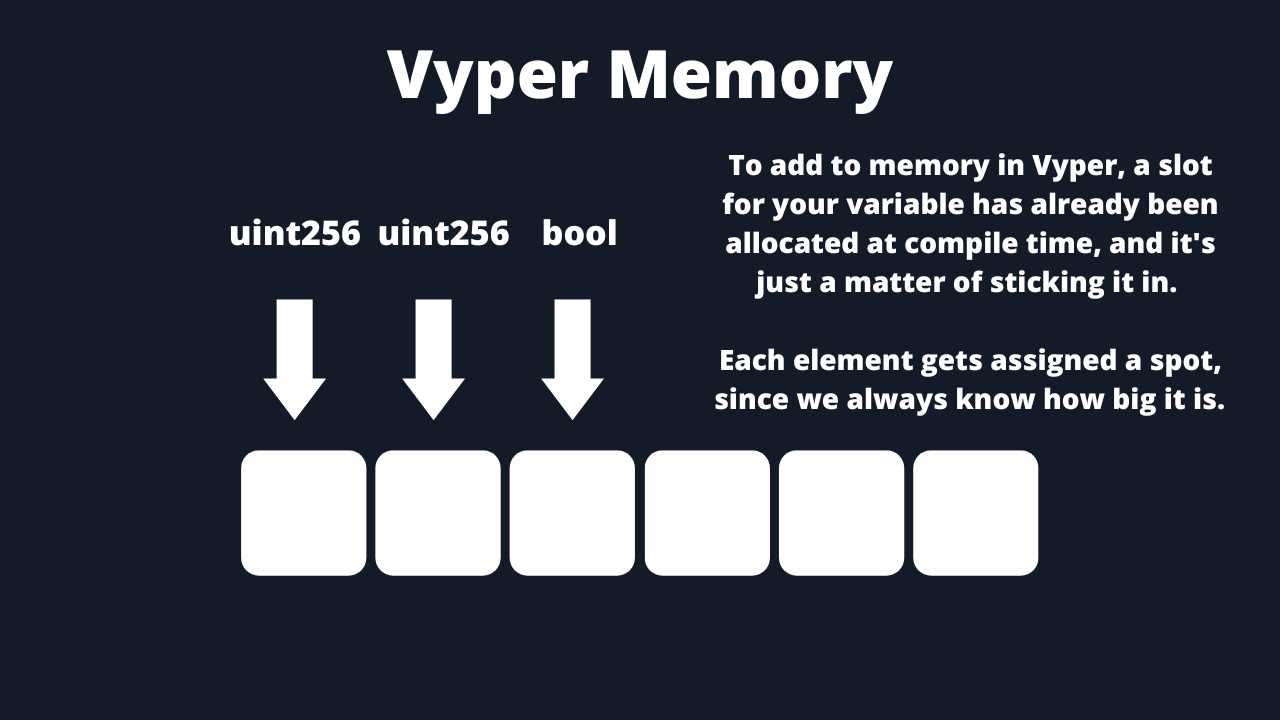
Free Memory Pointer

So what’s the deal with this free memory pointer? It seems to create a big difference in gas consumption regarding Solidity versus Vyper. The free memory pointer is a feature that controls memory management—anytime you add something to your memory array, your free memory pointer just points to the end of it, like so:



This is great since there are data structures like dynamic arrays that we may have to load into memory. With a dynamic array, we don’t know how big it will be, so we will need to know where memory ends.

In Vyper, there are no dynamic data structures, you are forced to say exactly how big an object like an array will be. Knowing this, Vyper can allocate memory at compile time and not have a free memory pointer.



This means that Vyper can be more gas optimized than Solidity when it comes to memory management. The downside is that with Vyper you need to explicitly state the size of your data structures and can’t have dynamic memory. However, the Vyper team actually looks at this as a plus.

Dynamic Arrays

Setting aside the memory stuff for a minute, it’s true that with Vyper you have to declare the bounds of an array. In Solidity, you can declare an array without a size. In Vyper, you can have a dynamic array, but it has to be “bounded”.

This could be seen as frustrating for the developer experience, however, in Web3 this could also be seen as protection against denial-of-service attacks and prevent massive gas costs in your functions.

If you have an array that grows too large in size and you iterate over it, it can cost a ton in gas. However, if you explicitly state the bounds of the array, you’ll know exactly what the worst-case performance of your smart contract would be.

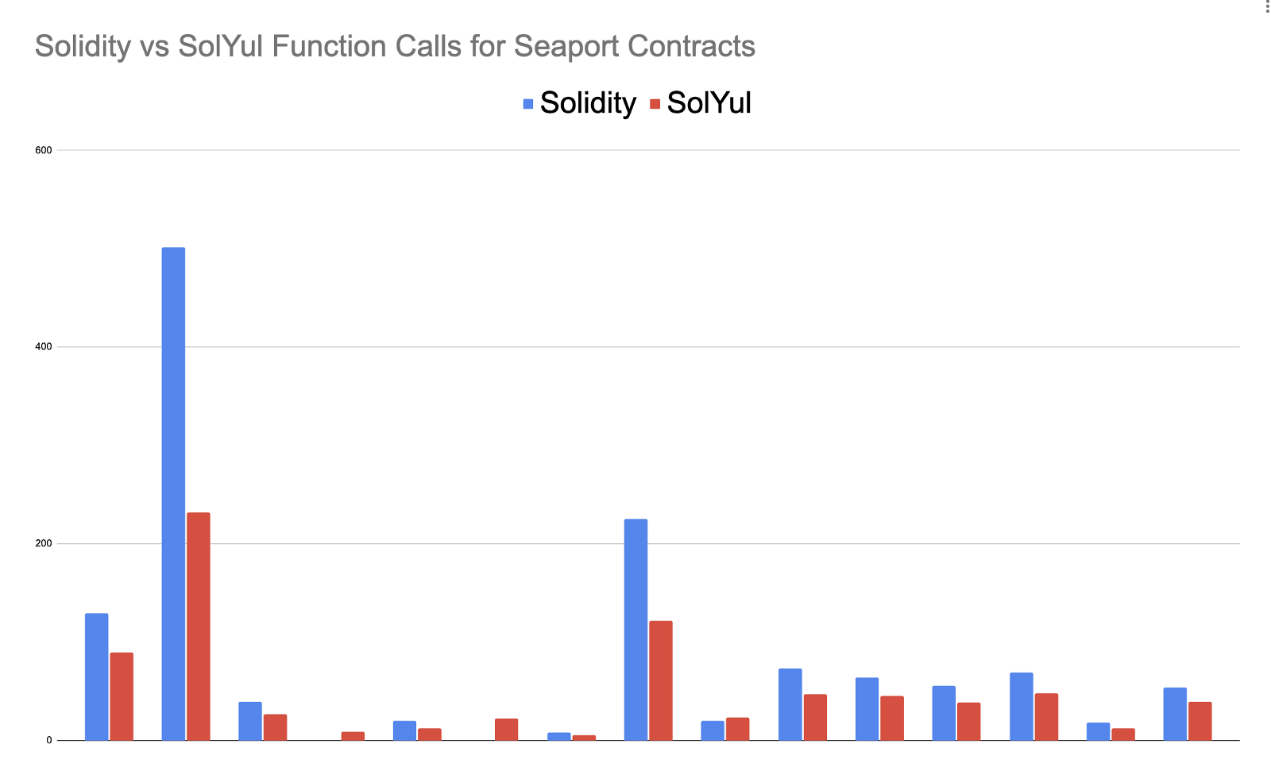
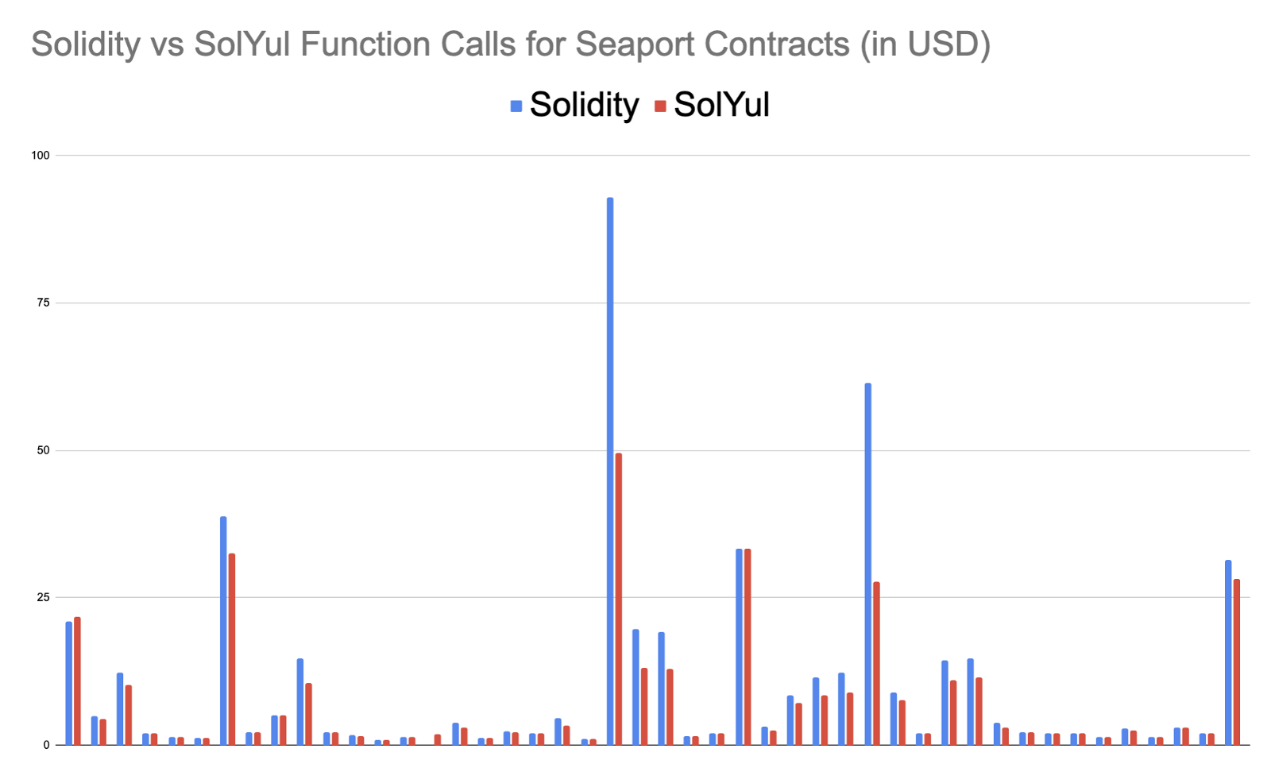
Solidity vs. Yul vs. SolYul

Looking at my chart above, working with Solidity and Yul seems like the worst option since the contract creation code is so much more expensive. This might be the case for smaller projects since Solidity does a few gymnastics to get the Yul going, but what about at scale?

One of the most popular projects to be written in a Solidity version and then a SolYul version is the[Seaport](https://github.com/ProjectOpenSea/seaport) project.

[Seaport Project Logo](https://github.com/ProjectOpenSea/seaport).

One of the best aspects of using these languages is that you can run commands to test out the gas effectiveness of each contract directly from the source code. We added a[pull request](https://github.com/ProjectOpenSea/seaport/pull/646/files) to aid in the command for testing the gas costs of the pure Solidity contracts, as the Sol-Yul contracts have the tests already. The result of this was pretty staggering, and you can see all the data in[gas-report.txt](https://github.com/PatrickAlphaC/seaport/blob/added-gas-profiling-for-reference-contracts/gas-report-reference.txt) and[gas-report-reference.txt](https://github.com/PatrickAlphaC/seaport/blob/added-gas-profiling-for-reference-contracts/gas-report-reference.txt).

Contract creation gas cost differences in Seaport.Function calls gas cost differences in Seaport.

On average, the function calls performed 25% better on the SolYul versions, and the contract creation performed 40% better.

That’s a lot of gas savings. I wonder how much they could have saved in pure Yul? I wonder how much they would have saved in Vyper vs. Sol-Yul?

Metadata

And finally, metadata. Vyper and Solidity both append some extra “metadata” to the end of their contracts. It’s such a small amount though, that we are going to be basically ignoring it for the comparisons here. You can always manually chop it off (and adjust the marker for how long your Solidity code is), but the Solidity team is also working on a PR where [you can remove it at compile time.](https://github.com/ethereum/solidity/pull/13265)

Summary

Here’s my opinion on these languages:

1. If you’re coding smart contracts, use Vyper or Solidity. They are both high-level languages that will protect you from shooting yourself in the foot by looking at call data size and whether you accidentally sent ETH when you shouldn’t have. They are both great languages, so pick whichever and have fun.
2. Yul and Huff are fantastic learning resources or tools to use if you need super specifically performant code. I don’t recommend most people write in these languages, but I think they are both fantastic to learn and understand. They will each give you a better understanding of the EVM.
3. One of the main differences in gas costs between Solidity and Vyper is the free memory pointer in Solidity — keep this in mind once you get to an advanced level and are looking to understand one of the underlying differences between the tools.

Looking Forward

These languages will continue to evolve, and we will likely see more languages pop up too, like the[Reach programming language](https://docs.reach.sh/) [and fe](https://fe-lang.org/).

The Solidity and Vyper teams have worked on an intermediate representation compilation step. The Solidity team has a –via−ir–���-�� flag in production that will help optimize Solidity code, and the Vyper team has their venom����� intermediate representation as well.

Whichever language you choose, you’ll be able to write some awesome smart contracts. Happy coding!

*The opinions expressed within this post are solely the author’s and do not reflect the opinions and beliefs of the Chainlink Foundation or Chainlink Labs.*