University of Bergen
Department of informatics

Analyzing Solidity smart contracts

Sondre Aasemoen
Supervisor: Jan Arne Telle

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

2022

Abstract

Solidity is a smart contract programming language that is used on the Ethereum blockchain. Because smart contracts are immutable once deployed, bug fixes are impossible and authors need to ensure bugs are caught before being published. In this thesis we analyze the various language constructs in Solidity and compare them to other programming languages and previous, similar analyzes.

Acknowledgements

Thanks to my supervisor, Jan Arne Telle, for his patience and support while I fumbled and stumbled trying to write this thesis.

To my friends who helped me stay sane during my — our — final year, thanks for the long nights and fun we shared along the way.

And to my parents for their never ending support and faith in me, none of this would be possible without you.

Thank you.

Contents

— 1 —
Introduction

In this thesis we will implement a fully specification compliant parser and several analyzers for the smart contract language Solidity running on the Ethereum blockchain. We will analyze and compare how different language constructs are used and whether they are used in a similar or different manner to other programming languages?

Smart contracts are, more often than not, smaller programs that are written in niche programming languages targeting a very different runtime from most other languages; once written and published, the contracts are immutable and public. Once you have deployed your smart contract, there is nothing you can do if there is a serious bug with the verification of who can use it.

Because of this limitation — or advantage if you are so inclined — ensuring that everything is correct when the smart contract is published on the blockchain becomes very important. Any bug will exist forever, unless you've included a kill-switch in the contract, but the inclusion of one might scare away the users of it.

Blockchains are, depending on who you ask, either one of the biggest innovations in recent years or an unregulated mess where new currencies and scams are introduced and disappear with increasing frequency. The underlying technology that powers blockchains is nothing out of the ordinary, but its uses can and will have far-reaching consequences and possibilities.

These blockchains allow anyone to buy from and pay anyone; there is no regulatory oversight, no red tape, and no security should you fall for a scam. Where the "regular" finance sector is highly scrutinized and has to accommodate hundreds of countries various rules and regulations, cryptocurrencies work across any border as long as you have a wallet (an address you can deposit coins into).

Due to the inherent volatility of these cryptocurrency markets, how easy it is to create a new blockchain, and the propensity for market collapses, it is easy to view the world of blockchains as something akin to the American frontier of old: a large, unexplored, and lawless area of the Internet where you can just as likely to become a millionaire overnight as you are to lose your fortune.

It is therefore very important that the authors of smart contracts have made no errors, or more realistically, no serious logic bugs in their code to not lose all their money. The cases range from users who mistakenly sent the incorrect cryptocurrency to a contract [1], sending to the wrong address [2], to hackers who exploited a "stupid bug" in a smart contract that allowed them to steal $34 million USD [3].

In this thesis, we will compare how Solidity is used relative to more popular and mainstream languages like Java, C++, and others. We created a parser for the language using Haskell and built several analyzers on top of this parser to try to answer the question about complexity in Solidity smart contracts to help understand how and where bugs might occur in smart contracts.

Writing parsers for programming languages is a well-studied area of computer science, and we delve into the details and theory behind the grammars and parsers that are used to create meaning out of programs in chapter 2. Before looking at Ethereum, we briefly explain how blockchains and smart contracts work in chapter 3 and then look at Ethereum in more detail as this is where Solidity runs in chapter 4 before diving into the implementation of the parser and analyzers used in this thesis in chapter 5, and then concluding with the analysis in chapter 6.

— 2 —

Parsing and grammars

Before we dive into the technical aspects of blockchains and their smart contract programming languages, we will first provide some background on the theory, frameworks, and technology used in this master thesis. We will start with information about context-free grammars in section 2.1 before moving on to more practical topics with how to parse languages in section 2.2 and follow that up with our use of parser combinators for this thesis in section 2.3.

2.1 CONTEXT-FREE GRAMMARS

As we are working with parsing a programming language defined in EBNF notation, we start to touch on a class of languages (in the theory of formal languages, not programming languages) called context-free languages, languages that can be generated from context-free grammars; see definition 2.1 from [4].

Definition 2.1 (Context-free grammar)**.** A context-free grammar is a 4-tuple $G = (V, \Sigma, R, S)$ where

1. V is a finite set of nonterminal characters often called the variables,
2. Σ is a finite set, disjoint from V, called the terminals,
3. R is a finite set of productions, or rules, $V \times (V \cup \Sigma)^*$, and
4. $S \in V$ is the start variable.

As mentioned in section 5.1, the grammar for Solidity is written in ANTLR's dialect of Extended Backus-Naur Form notation, which in most cases can be converted to a regular BNF grammar, though it will be extremely verbose for most programming languages.

```
returnStatement: Return expression? Semicolon;
expression:
  expression (Inc | Dec)
  | expression (Equal | NotEqual) expression
  | literal
```

Listing 2.1: A select subset of the ANTLR grammar for Solidity.

The conversion from the ANTLR grammar in listing 2.1 to a BNF grammar can be seen in listing 2.2. From this select subset of all the production rules of Solidity, one can construct the return statement. It can be either an empty return statement with no value or some expression. A construction following the rules in the BNF grammar can be seen in listing 2.3.

$$
\begin{aligned}
\langle\text{syntax}\rangle &::= \langle\text{return}\rangle \\
\langle\text{return}\rangle &::= \texttt{return}\ \langle\text{opt-expression}\rangle\ \texttt{;} \\
\langle\text{expression}\rangle &::= \langle\text{expression}\rangle\texttt{++} \mid \langle\text{expression}\rangle\texttt{--} \mid \\
&\quad\ \ \langle\text{expression}\rangle\texttt{==}\langle\text{expression}\rangle \mid \\
&\quad\ \ \langle\text{expression}\rangle\texttt{!=}\langle\text{expression}\rangle \mid \\
&\quad\ \ \langle\text{literal}\rangle \\
\langle\text{literal}\rangle &::= \textit{quoted string} \mid \textit{number}
\end{aligned}
$$

Listing 2.2: ANTLR grammar in listing 2.1 converted to BNF.

One can either use these grammars to generate strings based on the productions of the grammar or go the other way and use these productions to *parse* some input string. All programming languages have rules and need to be parsed into some intermediary representation that is more suited for analysis, execution, or compilation. A parser creates a parse tree, a tree showing the derivation of the grammar from the input. An example parse tree for the last derivation in listing 2.3 can be seen in figure 2.1.

Context-free grammars can also contain what is called left-recursive rules [5], productions where the leftmost symbol is the same as the production itself. These can lead to troubles when writing recursive descent parsers or parser combinators for a language because the parser will endlessly loop attempting to parse the same production over and over again.

$$
\begin{aligned}
\langle\text{syntax}\rangle &\Rightarrow \langle\text{return}\rangle \\
&\Rightarrow \textit{return}\ \langle\text{opt-expression}\rangle\ ; \\
&\Rightarrow \textit{return}\ ; \\
\langle\text{syntax}\rangle &\Rightarrow \langle\text{return}\rangle \\
&\Rightarrow \textit{return}\ \langle\text{opt-expression}\rangle; \\
&\Rightarrow \textit{return}\ \langle\text{expression}\rangle\ !=\ \langle\text{expression}\rangle; \\
&\Rightarrow \textit{return}\ \langle\text{literal}\rangle\ !=\ \langle\text{literal}\rangle\ ; \\
&\Rightarrow \textit{return}\ \text{“hello”}\ !=\ 3.14\ ;
\end{aligned}
$$

Listing 2.3: Example derivations following the rules from listing 2.2.

As one might expect, right-recursive productions are also a problem one needs to be aware of when writing parsers, as these can often lead to problems when attempting to parse expressions that contain left-associative operators, such as plus or minus.

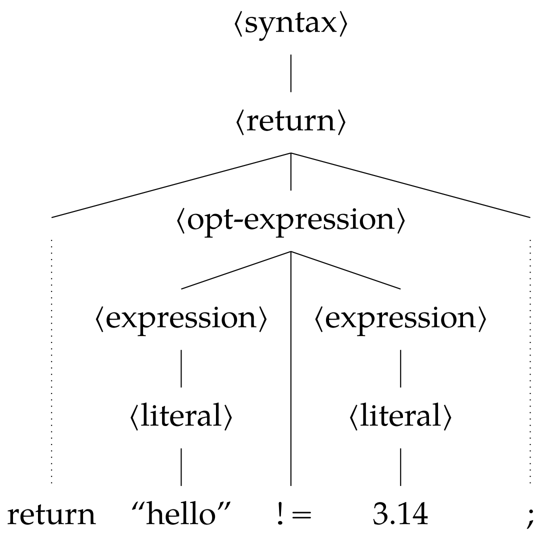


Figure 2.1: Parse tree for `return "hello" != 3.14` from listing 2.2.

Both of these problems can be solved by rewriting the production containing the left or right recursion, or by designing the parser in such a way that it can do lookahead to figure out what the next rule is (see section 2.2 for more). In practice, most parsing libraries or frameworks contain utilities to get around this, but it is still something that both beginners and experts alike can accidentally introduce to their grammar by overlooking or by being unaware of it.

In the case of the Solidity grammar, the expression grammar is a left-recursive production. If one is not careful when writing the expression parser, it can lead to infinite recursion.

Context-free grammars can also be ambiguous, which means that there are multiple ways to generate a string. Examples of these grammars abound, ranging from simple grammars for arithmetic where the string $2 + 1 \times 3$ can be derived in multiple different ways to the famous case of *dangling elses* [6]. Both the C and ALGOL grammars have this problem of undecidability where their grammar for **`if`**-statements and optional **`else`**-statements is ambiguous when they are nested, the grammar is written such that it cannot be decided which **`if`**-statement the **`else`** belongs to.

Definition 2.2 (Ambiguous grammars)**.** A context-free grammar is ambiguous if there exists a way to generate some string with more than one leftmost derivation (or parse tree).

To round out context-free languages, a topic of discussion that often comes up for students in computer science is whether programming languages are context-free. Many programming languages include a context-free grammar used for writing a parser for the language in question, but most programming languages are not purely syntactical languages, they are also semantic languages.

What do we mean by this distinction? The language that the context-free grammar recognizes can be semantically invalid; the parsed input is well-formed based on the grammar but is not a valid, executable program. Most programming languages are parsed in two steps, the first using some grammar to describe the syntax of the language, and then the result is once again parsed, and constraints are applied such that only valid programs are fully parsed.

Definition 2.3 (Syntax)**.** The syntax of a language is the grammar that defines well-structured programs (or strings of that language).

Definition 2.4 (Semantics)**.** The semantics of a language is the meaning behind the program (or string).

This leads to a notion of the difference between syntax and language semantics, even though a parser might parse some program does not mean it is a valid program for that language. The example program in listing 2.4

is a program that the parser will happily parse because it is syntactically valid, but since we never define the variable `one`, it is not semantically correct (the Solidity compiler will refuse to compile this program).

```
function sum() {
    int ten = 10;
    int total = ten + one;
}
```

Listing 2.4: Syntactically valid, but semantically invalid Solidity program.

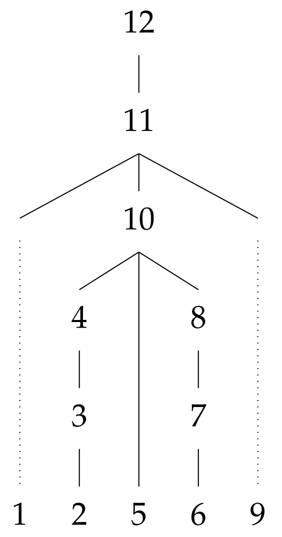
Thus, in the case of most programming languages, the grammar for the syntax of the language is often context-free, but for programs to be executable, it is more often than not *need* for context to validate them. Examples include, as above, that variables need to be declared before use, or that blocks of the program need to be indented correctly, or how some *keywords* are illegal in some places but not all (the **`break`** keyword can only be used in looping constructs, for example).

2.2 PARSERS AND PARSING

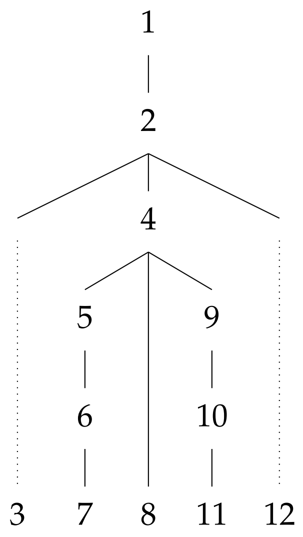
Once we have a well-formed grammar with (or without) left-recursive productions, one can move on to writing a parser for this grammar. Parsing is one of the most studied areas of computer science, as it is fundamental to many areas of research, from natural language processing, to programming languages, and more.

There are many ways to build parsers, all with their different trade-offs and strengths. A parser can be built as a multi-pass parser, where it needs more than one pass over the input to fully build an abstract syntax tree, or as single-pass parsers requiring only one pass. Parsers can work in two stages with a lexer (or tokenizer/scanner) first building a list of tokens that are then parsed into an abstract syntax tree, or by bypassing this step and directly building the abstract syntax tree from the input.

Furthermore, parsers are often divided into two main algorithmic groups; top-down parsers and bottom-up parsers. Although there exist many techniques for writing parsers, most boil down to these two techniques [7]. The top-down parser works, as one would expect from the name, by beginning to parse from the top. Starting from the start production, the parser builds the syntax tree by following the productions from left to right until it reaches the end of the string. A bottom-up parser works, also as expected, by attempting to find the most basic productions and then working its way up from there.



((a)) Bottom-up parser



((b)) Top-down parser

Figure 2.2: Parse steps for parser based on figure 2.1.

Finally, one needs to distinguish parsers by the order in which they build their derivations. A parser can generate either a leftmost derivation or a rightmost derivation. In general, top-down parsers tend to generate leftmost derivations, while bottom-up parsers tend to generate rightmost derivations. For a bottom-up parser to generate a leftmost derivation, it needs to keep track of the entire string and parse it from the right end instead of recognizing smaller pieces of it, one at a time. An example of the order of the two different techniques can be seen in figure 2.2.

The aforementioned kinds of parsers usually break down into two groupings, LL parsers and recursive descent parsers in one and LR parsers in the other. The names LL and LR might appear cryptic, but are quite obvious in hindsight; LL parsers are **L**eft-to-right, **L**eftmost derivation parsers and LR parsers are then **L**eft-to-right, **R**ightmost derivation parsers.

Both LL and LR parsers can use lookahead, in cases where the parser needs to peek ahead a certain distance to determine how to parse its current token. These parsers are occasionally denoted LL(k) and LR(k), with k being the amount of lookahead it can do. Often in the cases where the lookahead is one token (e.g. LL(1)), the amount is omitted, as one token of lookahead is by far the most common value for *k*.

Recursive descent parsers and LL parsers share the property that both can parse, perhaps unsurprisingly, because of their grouping, LL grammars [8]. Recursive descent parsers are top-down parsers where the productions of the grammar are their own functions or procedures, defined mutually recursively. Often, one function is a production of the grammar, which makes these parsers very easy to write and follow. Depending on the grammar and implementation, the parser might backtrack. Backtracking occurs when a parser has followed a set of productions down a path where it is unable to continue parsing the input. In this case, it might backtrack, moving backwards up the parse tree, and attempt following a different path instead. If the parser does not implement backtracking, it is called a predictive parser.

When choosing the technique to use to write your parser, there are many requirements, constraints, and trade-offs that must be taken into account. Recursive descent parsers with backtracking are easy to write, but if they backtrack excessively, the time to parse might be exponential, if it even terminates. LR parsers require that the grammar be unambiguous, something that an LL parser does not require. LL parsers are easier to debug, since the programmer can follow the parser from the top using their debugging tool of choice, while an LR parser is much harder to follow.

Finally, a restriction for LL and recursive descent parsers is that they cannot parse languages that have left-recursive rules in them naïvely. However, by using techniques such as backtracking, lookahead, or memoization [9], this can be worked around, although care must still be taken to avoid blowing up the runtime.

Often, programmers will opt out of writing their own parser by hand and instead lean on *parser generators*, such as ANTLR, LALR (lookahead, LR parser) [7] or PEG (Parsing Expression Grammar) [10]. These tools work by reading a grammar and generating a parser for that language. Since they work from a defined grammar, they can, for example, warn the programmer of syntax ambiguity at compile time instead of runtime, and are often written in a domain-specific language quite similar to Backur-Naur form.

Parsing is an enormous topic, and there is much more that could be covered, like precedence parsing, recursive ascent parsers, shift-reduce variations for LR parsers, Earley parsers, and much, much more. This section should give the reader enough background to understand the next topic needed for this thesis: parser combinators.

2.3 PARSER COMBINATORS

Parser combinators are a generalization of recursive descent parsers that are often found in programming languages where functions are first-class and thus support higher-order functions (functions that take functions as arguments). The *combinator* part of the name comes from the fact that you build your parser as a series and combination of parsers (or functions) that when combined form a full parser for your language of choice. This technique for parsing is often called combinatory parsing.

Due to the nature of composition of the parsers, it is mostly functional programming languages where parser combinators see the most popularity [11]. Haskell is a if not the prime language of choice for programmers looking to write parser combinators because it is a lazy, functional, and strongly typed programming language. The most famous monadic parser [12] library is called Parsec [13].

Being a kind of recursive descent parser, parser combinators also inherit some potential shortcomings of them, chief among them being performance. It is very easy to write parsers that exhibit exponential time usage by allowing excessive recursion and backtracking by not being careful with the combination of parsers. When considering how the parser will traverse the input, one can write efficient parser combinators, but as Kurš, Vraný, Ghafari, *et al.* [14] shows, they still cannot compete with efficient parser generators or optimized handwritten parsers.

```haskell
type Parser a = Parsec ErrorType Text a
```

Listing 2.5: Parser combinator type definition.

As this thesis is written using Haskell, the rest of the thesis will talk about parser combinators in the context of the features that Haskell provides; specifically monadic parsers that are polymorphic. Almost everyone who learns Haskell at some point stumbles upon the word monad and wonders what it is; many blog posts have been written attempting to answer this question. Ranging from using burritos [15] as the metaphor of choice to “a monad is a monoid in the category of endofunctors [...]” [16]. A rough explanation is that a monad is some structure that wraps functions and their return values, allowing additional work to happen behind the scenes or allowing effects to happen.

In the case of monadic parsers, the **`Parser`** type is a type synonym for the **`Parsec`** monad transformer (which again is a type synonym for the **`ParsecT`** monad transformer). In the specific case of monadic parsers, the **`Parser`** monad transformer keeps all the state of the parser hidden away from the functions, allowing the programmer to focus on the individual parsers without the mental overhead of explicit error handling, handling backtracking, the current position in the input, and more. The use of monad transformer here simply means a type constructor that takes a monad and returns a monad.

This allows for very terse and to the point parsers in the languages that support these monadic parser combinators. In section 5.3 we will see how this is implemented in practice.

— 3 —
Blockchains and smart contracts

3.1 BLOCKCHAINS

A blockchain (figure 3.1) is a series of data, blocks or records, where each one is calculated by hashing a ledger of transactions and its predecessor using cryptographically secure hash functions to create a "chain" of records [17]. Different blockchains will implement this with different hash functions, transaction ledgers, etc., but this description is correct in the general case.

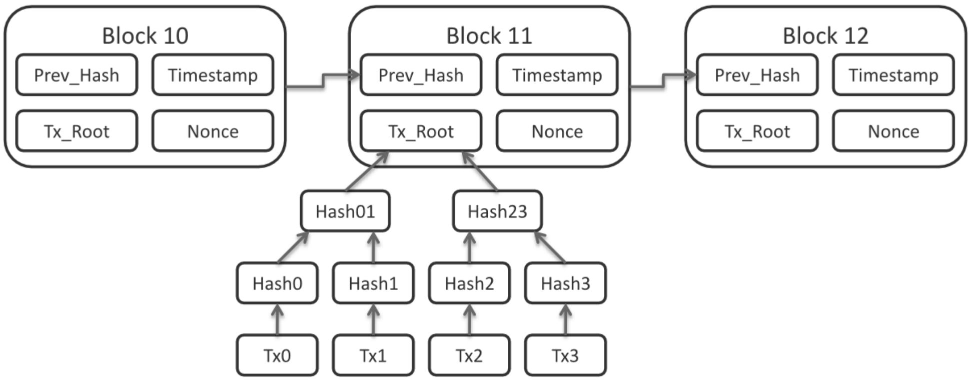


Figure 3.1: Example of a blockchain [18]

One can think of blockchains as a kind of state transition system, where the currency in circulation moves from one account to another. Similarly to how regular banks need a ledger to keep track of all transactions that have passed through their systems, a blockchain needs this to keep track of the state of the system in total. For most blockchains, the state of the system is the coins (or currency), with transactions with denominations and an owner going to one or more addresses (or wallets, as they are often called).

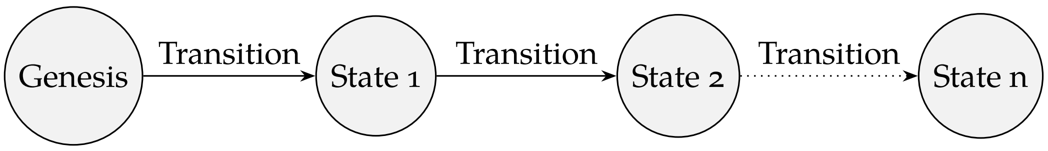


Figure 3.2: Example blockchain state transition

As a rough overview, one can imagine the transition function (see algorithm 3.1) between states as a function that accepts the current state, a transaction, and either transitions to a new state or returns an error if the transition is illegal. Each state is reached when a transition is valid and the network of nodes mining validates and reaches consensus on it.

The diagram in figure 3.2 and the corresponding pseudocode in algorithm 3.1 describe in a very high level what occurs between each state in the blockchain. These transitions are not the same as the blocks in the blockchain; the individual transitions are a collection of transactions done between users via smart contracts, direct payments, money transfers, etc. Once a node on the blockchain correctly calculates the magic value that is required to create a new block, all the transactions that have happened between the previous block and this cut-off point are collected into a ledger and forever immutably stored in the blockchain.

Algorithm 3.1: Example state transition function

```
Input: Current state, list of transactions
Output: Next state or error
for transaction in transactions do
    if referenced coin not in state then
        return error
    end
    if signature not equal to owner of coin then
        return error
    end
end
if sum of coins in transactions $\geq$ sum of coins in state then
    return error
end
return New state with coins removed from owners, added to receivers
```

Bitcoin and its enigmatic creator Satoshi Nakamoto [19] is where the popular usage of the term blockchain emerges and has since become a household name. Blockchains see many uses around the world, but is nowadays primarily centered around cryptocurrencies and smart contracts. Other use cases where it has been proposed range from elections [20] to supply chains [21] and more.

There are many proponents and opponents, with an equal amount of arguments in favor and against using a blockchain technology for common use cases. Although there have been multiple attempts at bringing this technology to the mainstream, most attempts still remain niche or have never materialized.

Let us examine blockchains in more detail, as previously mentioned, and, as the name implies, there is a chain of blocks. The blocks themselves are ledgers of transactions from the inception of the block until some point in time, when some form of consensus was reached among all the nodes partaking in this. The exact way to determine this cut-off point varies between different implementations, for example, both Bitcoin and Ethereum use proof-of-work [22] as their consensus algorithm.

What this means is that all the nodes in the blockchain network are all working towards verifying some amount of work has occurred. All these nodes read and verify all the transactions happening at once, ensuring that everything is in order. However, this happens without any leader that ensures consensus; the nodes themselves have to reach consensus.

The way this works for Ethereum specifically is that each block has a nonce that the various miners (mining is what miners do to create new blocks) have a race to find. As you can see in figure 3.1, each block contains the hash of the previous block, a set of transactions, and a nonce that creates a unique hash that every node in the network can easily verify to be correct. The miners therefore has to continuously download the entire history of all transactions on the blockchain, get the latest transactions not in a block and attempt to find a nonce that together with the previous hash and a difficulty (defines how many possible valid hashes there are) creates a hash that is valid for a set of these constraints. Once a miner has found a nonce that gives a correct hash, it is very easy for the other miners in the blockchain to verify this value and once this happens and consensus is reached, a new block is minted, and the miner that found the nonce is rewarded for their work.

This process is very time and compute intensive, and has in recent times been a large focus due to the environmental damage cryptocurrencies therefore cause. In terms of both the growing problems of electronic waste [23] and also with how much electricity is consumed in total [24]. As de Vries, Gallersdörfer, Klaaßen, *et al.* estimates, Bitcoin is responsible for 65.4 megatonnes of CO2 (MtCO2) per year. More recent estimates show that Bitcoin may use as much energy as Thailand does [25].

3.2 SMART CONTRACTS

Smart contracts are computer programs running on blockchains that allow anyone with the ability to create contracts without the need for intermediate arbiters or agents to execute them faithfully.

These differ from smart legal contracts in which the contract is still written in a normal language supplemented with terms written in machine understandable code. In this case, one can use contract provers to verify properties about the contents and terms, whereas this is near impossible for smart contracts.

While blockchains entered the public mind with the release and popularization of Bitcoin, smart contracts came to fruition with later blockchain technologies. Bitcoin is still the largest cryptocurrency around, and Ethereum is the largest smart contract blockchain around. In fact, Ethereum has other blockchains with their own cryptocurrencies and ecosystems running on it.

According to Cryptorank [26] — a web application for crowd sourced analysis of blockchains — there are several hundred blockchains running on top of Ethereum.

A common kind of example used to illustrate how and why smart contracts are useful is based on bets that are unfulfilled because one of the members of said bet refuses to finalize what was previously agreed on. Say two friends Bob and Alice bet on who can get the best grade for a course, with Bob being utterly certain that there is no way for Alice to outperform him. However, when the results are published, it turns out that Alice *did* beat him. Bob refuses to honor their agreement, Alice must have cheated to get a better grade.

In this example, the non-smart contract is never fulfilled even though all requirements are met. Proponents of smart contracts would argue that this would not happen for a smart contract, as you remove the need for interpersonal trust between agents agreeing to the contract.

If, however, Bob and Alice instead codified this bet as a smart contract, there is no way for either of them to back out once it is deployed to the blockchain. Bob and Alice craft a smart contract that specifies that after the date when grades are released, the contract will automatically pay out to whoever has the best grade. In this specific example, unless the contract itself can automatically fetch and compare their grades, they must somehow send them to the contract. The example starts to break down at this point, but the point is that bets or agreements can be automated, predictable, and without worry of interference once deployed.

As a final example, consider an insurance company based on smart contracts. Instead of having humans be the arbiters of when and how much money will be paid out, users can instead sign contracts based on events that are fully automatable and happen without human oversight. Say that you want to protect your crops from pests, floods, or similar disasters. The contract could work by querying the governments' database of outbreaks based on location and simply pay out an agreed-upon amount whenever this happens.

Other advantages of smart contracts are that they are public, yet private. Alice can verify that the contract exists, check that the contract matches what was agreed upon by verifying its source, and track transactions to see where money is moved to and from. The privacy of users is protected since addresses are not tied to identities, and Ethereum is a pseudoanonymous (everything is public, but there is no identity tied to anything) network, so you can interact with these contracts without fear of others finding out who is who.

— 4 —

Ethereum and Solidity

4.1 ETHEREUM

Ethereum is one of many popular blockchains that have emerged in recent years. Its popularity makes it the second most used cryptocurrency in the world, after Bitcoin. Ethereum is formally defined in its "Yellow Paper" [27], as opposed to the more common used white paper.

Ether is the cryptocurrency that is used on Ethereum, which can be used as a regular cryptocurrency but can also interact with immutable programs running on the blockchain; smart contracts. A smart contract is a program written in a language that targets the Ethereum Virtual Machine with which users of the blockchain can interact, using it to lend Ether, purchase goods, and more without the need for intermediaries such as brokerages and exchanges.

These smart contract applications running on the Ethereum Virtual Machine allow for a runtime environment with enough opcodes that makes it Turing complete, and these programs are distributed out onto the blockchain on thousands of nodes executing the programs using transactions.

The two most popular programming languages for Ethereum are Solidity and Vyper, the former a C-like, object-oriented statically typed programming language and the latter is closer to Python but is still strongly typed. Other languages exist but are far less popular, but as the virtual machine is based on a limited number of opcodes and executes using a fairly simple stack machine, it is not hard to create either new languages or interpreters for the Ethereum Virtual Machine.

Smart contracts act as immutable applications stored in an immutable ledger, where any user can interact with its public functions. A smart contract can be many things, from contracts between users, auctions between unknown parties, voting systems, and more. Since the languages used to write these programs are Turing complete, in theory, you could create any program as a smart contract, but the ecosystem surrounding smart contracts focuses mostly on transaction-based applications.

4.2 ETHEREUM AND OTHER BLOCKCHAINS

The world of cryptocurrencies and the number of available blockchains have grown dramatically in the last few years. From what started with essentially Bitcoin has grown into a large market of available options for users. Ranging from cryptocurrencies created to mock others like Dogecoin or Garlicoin, to stable coins like Tether or USDCoin to regular currencies like Bitcoin or Ethereum. What separates these currencies?

The most obvious difference is popularity; Bitcoin and Ethereum are by far the two most used, with the largest market caps and usage. Blockchain is still the largest cryptocurrency on the market, with over 400 billion USD in total market capitalization compared to Ethereum's roughly 200 billion.

Stablecoins are another interesting kind of cryptocurrency, where the value of the coin itself is not dependent on circulation and speculation, but is instead pegged against some other asset as a reference. The most common asset to use as a peg is the US dollar, and cryptocurrencies such as Tether are pegged at a 1:1 valuation to it. The primary benefit is a currency that is not as volatile as other currencies; Bitcoin has had periods of intense swings in evaluation, which can be detrimental if you want to use the currency to purchase something that doubles in price in the blink of an eye.

Blockchains also have trade-offs that one needs to take into account when evaluating them, some are popular but slow – like Bitcoin – while others are not so popular but have a very high rate of transactions. Some use expensive operations to create its blocks, often using a so-called proof-of-work method to mint new coins. This leads to some cryptocurrencies using enormous amounts of energy to essentially crack difficult math equations, such as Bitcoin and Ethereum.

A more recent development in the cryptocurrency space is the market for non-fungible tokens, mostly referred to as NFTs. These immutable and unique tokens allow users to purchase a digital token to represent ownership over some asset, whether it is a digital photograph or some physical object. Nearly all blockchains that allow for smart contracts also allow for non-fungible tokens.

Ethereum finds itself in the sweet spot of being the second most popular blockchain while also having smart contracts, something Bitcoin lacks. This means that while many other cryptocurrencies make other trade-offs compared to Ethereum, like not using proof-of-work to build its blocks or peg their value against some other asset, it still finds itself with a thriving community and developers.

4.3 SOLIDITY

Solidity is the primary programming language targeting the Ethereum Virtual Machine, and is a statically typed, object-oriented programming language in the C-family of languages.

In listing 4.1 we create an example contract that allows the creator of the contract to deposit coins into the contract and see how much is stored in the contract. This is a contrived example, as only the creator can deposit and nobody can withdraw the coins, only view who owns the contract and how much is stored in it.

```
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    address public owner;
    uint storedData;

    constructor() { owner = msg.sender; }

    function set(uint x) public {
        require(msg.sender == owner);
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Listing 4.1: Example Solidity contract

Via language pragmas, a way to provide the compiler with additional information such as compiler versions, language extensions, or experimental features, Solidity also enables developers to target specific versions of the language to ensure compatibility with the compiler and runtime, but also to enable experimental features. As of this writing, there is also an intermediate language called Yul that one can use with the same compiler as Solidity, but it is targeted at advanced users who require or want more optimizations for their programs.

Solidity has no official specification such as C++ [28] or Scheme [29]; by this, we mean documentation that lays out how other users can implement the language for themselves; the grammar, runtime behavior, its standard library, and so on. Solidity has extensive end-user documentation for how it works and how to use it, and has an actual specification for the contract ABI (Application Binary Interface) to define how to interact with contracts and an EBNF grammar. Many modern languages like Python and Rust also do not have an official specification but instead rely on what is called a reference implementation; there exists a single implementation of the programming language that is the authoritative source on grammar and behavior, and if other users want to implement the language themselves, they have to feature-for-feature and bug-for-bug follow this implementation.

As far as programming language features go, Solidity is superficially quite similar to other object-oriented languages like Java. It has support for defining interfaces to which contracts must adhere should they choose to implement them, and it supports abstract contracts similar to how abstract classes exist in similar languages.

Unlike most mainstream programming languages, Solidity has no repository of user-submitted third-party libraries that can easily be downloaded and imported (in contrast to Maven for JVM languages, Crates for Rust, RubyGems for Ruby, and so on). Instead, Solidity has a language construct called libraries.

Libraries are very similar to contracts in that they can define functions, structs, and most things like contracts, but they have no state. Instead, contracts can call libraries and their associated functions, and the library acts as if it is a part of the contract itself, allowing the state from the contract to be changed via the library.

Libraries can be embedded alongside the contract or called externally if the library contains public functions. In this case, the contract needs to know the address of this library so that it can know where to call it. This is the only mechanism that enables code reuse, where you can call third-party code without including it. However, due to the immutability of programs deployed to Ethereum, bugs are forever, and therefore most libraries are downloaded and compiled together with the contract. Many contracts, for example, include OpenZeppelin [30] libraries, as these are battle tested, open source and easy to install, allowing developers to easily add overflow-safe arithmetic or role-based permissions.

While there are many superficial similarities between Solidity and object-oriented languages, one must keep in mind that Solidity is closer to a specialized embedded programming language than a general higher-level language. Generics is an example of a feature that most modern popular languages support which Solidity neither has nor plans on supporting.

4.4 THE ETHEREUM VIRTUAL MACHINE

The fact that Ethereum is a blockchain that allows users to publish and deploy smart contracts is all well and good; but how do these programs written by users that are deployed to a decentralized network of independent computers and nodes work?

As an analogy, instead of thinking of Ethereum as a distributed ledger of transactions, but as we illustrated in section 3.1 as a distributed state machine. Compared to blockchain which is solely a distributed ledger as it has no concept such as smart contracts, Ethereum needs not only to move currencies between accounts, but also execute programs.

On Ethereum, the state of the network and the blocks in its chain is not only the transactions; but also the state of all the programs and how they have changed between blocks. This happens via the virtual machine that underpins all smart contracts: the Ethereum Virtual Machine (or EVM for short). From one block to the other, the states of all the programs on the blockchain are updated according to the rules of this machine. A way to think of how this works is to imagine that the entire state of the blockchain is one gigantic, distributed finite-state machine.

The simplified state function in figure 3.2 and algorithm 3.1 is in essence the virtual machine. This machine is responsible for maintaining the immutable state of all accounts and smart contracts and defines the rules that move the network from one block to another. The actual state function for Ethereum does need to do a lot more bookkeeping to ensure both consensus and correctness, but works roughly the same. The entire state of the Ethereum blockchain is called the world state and maps between all the addresses in the network and their associated account states.

To be able to track all this state efficiently, Ethereum uses a Merkle tree variant called Patricia Merkle Tree [31]. This tree is used to store four different states per block in tries, (1) state (2) storage (3) transactions (4) receipts. The primary reason for the use of this Merkle tree is efficiency and to be a cryptographically secure way to store data. The implementation in Ethereum manages to get $O(\log n)$ insertion, deletion and lookup.

A Merkle Tree – or a binary hash tree as it is also called – is, as the name suggests, a tree data structure that allows for secure and efficient storage of large amounts of data. As blockchains handle thousands upon thousands of transactions between many accounts, efficient storage of this data is of utmost importance. They also allow blockchains to validate the integrity of the data and can be transmitted across the network in small chunks instead, which means that users do not need the entire transaction tree, for example, to make a small transfer between accounts.

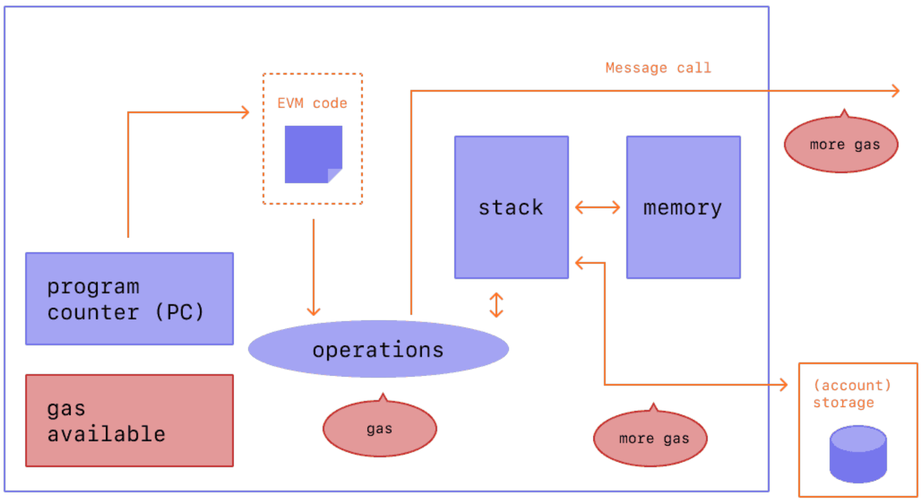


Figure 4.1: Illustration of EVM execution [32].

Before being executed, smart contracts need to be compiled to something the Ethereum Virtual Machine understands, a series of opcodes. These opcodes are what tells the virtual machine whether to add two numbers together or perform specific operations for use cases on the blockchain. These opcodes are not free; they require a certain amount of gas to execute.

Gas in Ethereum is very similar to the fuel required to power a vehicle; it is a way to specify how expensive an operation is. When executing a smart contract, each operation accrues a cost, along with other operations, such as transactions. Gas in Ethereum is paid in the cryptocurrency ether, using a unit called gwei (a gwei is 10^{-9} ether). This gas is a way to pay miners for their work; you can even include a tip on your transactions if you want miners to prioritize yours over others.

The EVM is designed as a stack machine, a fairly simple design to reduce the number of required registers required where the machine works by pushing values and operations onto a stack and popping them as required. In the case of Ethereum, the depth of the stack is 1024 items, where each item is a 256 bit word, for ease of use with 256 bit cryptography.

When a smart contract is being executed, it has a short-lived temporary memory allocated while it is running, but this is deleted as soon as the transaction is finished. This is not to say that contracts cannot store data; as mentioned above, contracts are able to store data in the storage trie.

— 5 —

Implementation of thesis

5.1 THE SOLIDITY GRAMMAR

To parse the Solidity programming language, we implemented a tokenizer and parser in Haskell, written using parser combinators, a way to write efficient and easy-to-develop recursive descent parsers. Solidity has comprehensive grammar documentation, both as a ANTLR [33] grammar [34] and syntax diagrams (railroad diagrams, see figure 5.1) [35]. This thesis targets the 0.8 release of the Solidity language (version 0.8.12 as of this writing).

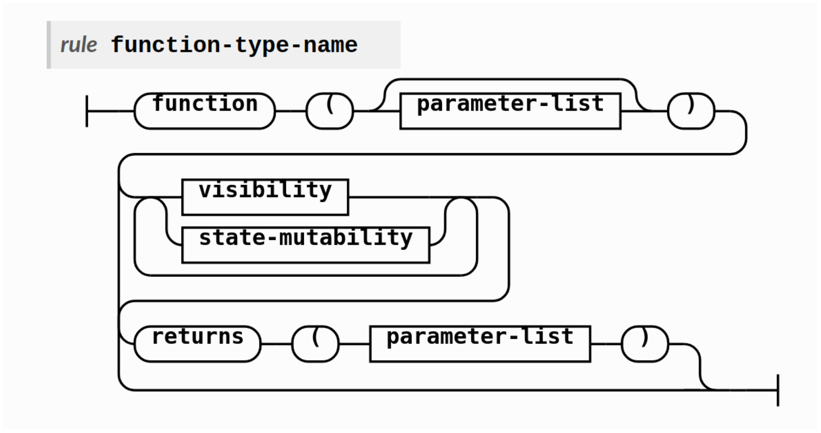


Figure 5.1: Railroad diagram for the function name type rule.

The grammar for the language before the move to an ANTLR-based parser generator was based on a simple **`grammar.txt`** file, where the grammar was very informally specified in what appears to be an attempt at something akin to what a Backus normal form grammar ought to look like. Fortunately, the ANTLR grammar that replaced it is more robust and has been kept up-to-date with the language as it has evolved. ANTLR is a parser generator (tools that generate parser from grammar) that reads an Extended Backus-Naur Form (EBNF) grammar and creates a parser from it, and it can generate parser for several languages, such as Java, C++ and more.

This allows for easy use of the grammar in other projects by generating a parser from the official grammar, so that one can ensure that the parser is the same as the compiler itself. This makes it easy to build tools on top of the generated parser, such as static analyzers, control flow visualizations, and more.

```
functionTypeName
locals [boolean visibilitySet = false, boolean mutabilitySet = false]
:
  Function LParen (arguments=parameterList)? RParen
  (
    {!$visibilitySet}? visibility {$visibilitySet = true;}
    | {!$mutabilitySet}? stateMutability {$mutabilitySet = true;}
  )*
  (Returns LParen returnParameters=parameterList RParen)?;
```

Listing 5.1: ANTLR grammar used to generate figure 5.1.

The railroad diagrams generated from the grammar make it easy to follow the parser around, if a node is a parsing rule, one can click on it to move to that rule to see how to parse it. This enables one to quickly build an intuition for the parsing of the language, and if needed, one can dive into the actual grammar if the diagram is ambiguous about how to parse something.

The code in listing 5.1 is what generates the diagram in figure 5.1, with some modifiers attached that are not visible in the diagram itself. For example, to create a parser for function type names, one can either use the railroad diagram in figure 5.1 or the grammar in listing 5.1, the resulting parser to parses it in the following way:

(1) Read the keyword `function`

(2) An optional parameter list contained by parenthesis

(3) Parse function modifiers

    (a) Optionally parse function visibility

    (b) Optionally parse state mutability

(4) Optionally, parse what the function returns

    (a) Read the keyword `returns`

    (b) A required parameter list contained by parenthesis

(5) The rule ends without any terminals, the semicolon in listing 5.1 is the end of the grammar rule and not the literal itself.

As we can see, the rule for a function type must declare itself as a function and include its parameters, but steps (3) and (4) are optional, and the parser must therefore account for the possibility that they may be missing. By combining these rules, our parser takes shape by following the possible paths starting from the source root until the parser completes, successfully or not.

The parser builds a complete abstract syntax tree from its input, based on the grammar from Solidity. The grammar in many ways defines a hierarchy of units that can be parsed, starting with zero or more **`SourceUnit`**s, each of which contains zero or more **`Statement`**s which in turn has zero or **`Expression`**s and so on.

5.2 ABSTRACT SYNTAX TREES

As mentioned previously, for this thesis a complete parser was written for Solidity, and we therefore need to be able to accurately express the parsed program. In our case, the parser emits an abstract syntax tree directly without any intermediate representations like concrete syntax trees (or parse trees as they are sometimes called). This means that while it can be used to create most tools for a programming language such as an interpreter or compiler or an analyzer and so on, it would be a poor fit for a formatter, for example, because it has no information about whitespace or comments in the syntax tree, and therefore cannot transform the parsed tree back to its text representation without loss of information.

Haskell was specifically chosen as the implementation language for the parser due to its long history of being used to write parsers in and because of its many great features, like the very advanced type system alongside an ecosystem of tooling for building parsers. Because of features like algebraic data types, pattern matching, and monads, one can effortlessly create type safe and easily composable abstractions for representation and usability.

In listing 5.2, we can see a subset of the full representation of the syntax tree as a sum type `SourceUnit` being the highest unit in the syntax tree, where each `SourceUnit` has a record (a Haskell type similar to structs or simple classes in other languages) containing the required metadata about its representation.

```haskell
data ContractDefinition = ContractDefinition
  { abstract :: Bool,
    name :: Identifier,
    inheritance :: Maybe Text,
    body :: [SourceUnit]
  }

data SourceUnit
  = Pragma PragmaDefinition
  | Contract ContractDefinition
  ...
  | Library LibraryDefinition
```

Listing 5.2: Haskell source unit type definitions.

By repeatedly combining these sum types, one can easily build trees or any large structure. And in our case, abstract syntax trees. In the above listing, each possible top-level unit is represented in the `SourceUnit` sum type: pragmas, contracts, imports, etc. These, in turn, can contain other units, such as the statements in listing 5.3 that are contained in the bodies of functions, constructors, and so on. For example, the body of a contract contains a subset of source units, such as inline structs or functions, that then contain struct members or statements.

One can easily see that these definitions themselves build a tree from their constituent parts, for example: a Solidity file is zero or more contracts that can contain zero or more functions, which themselves contain zero or more statements that contain zero or more expressions.... At the end of parsing the file, we have a syntax tree.

On a very simple contract for Solidity, see listing 5.4, you get a simplified abstract syntax tree like in listing 5.5. For the full abstract syntax tree, see appendix A.1.

```
data IfStatement = IfStatement
  { trueStmt :: [Statement],
    elseStmt :: Maybe [Statement]
  }

data Statement
  = While WhileStatement
  ...
  | If IfStatement
```

Listing 5.3: Haskell statement type definitions.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

      /* This is a comment */
    function get() public view returns (uint) {
        return storedData;
    }
}
```

Listing 5.4: Example Solidity program

```
Pragma [solidity >=0.4.16 <0.9.0]
Contract [SimpleStorage]
  uint storedData =
  Function [set]
      storedData = x
  Function [get]
      return storedData
```

Listing 5.5: Abstract syntax tree

5.3 THE MONADIC PARSER COMBINATOR

As was mentioned in the introduction, the parser was implemented in the Haskell programming language, using a monadic parser combinator library called Megaparsec — a fork of the Parsec parsing library — to create our parser. Megaparsec has been used to build parsers for many programming languages, such as Dhall and Idris [36]. The main difference between Megaparsec and Parsec is in how it handles error messages, an area where parser combinators historically have been rather poor.

The parser itself clocks in at about 900 lines, while all the type definitions required for the full language of Solidity is about 500, plus another 100 for the type definitions for Yul. The ANTLR grammar files from the Solidity compiler uses about 350 lines for the lexer and nearly 600 for the parser. Omitting the type definitions, the total lines of code for the parser and the grammar are surprisingly close, both at just about 900 lines of code. As mentioned in section 2.3, parser combinators allow one to write a single function per production, and in our case the total line count is nearly the same as the grammar.

As a final comparison between the handwritten parser for this thesis and one generated by a tool like ANTLR — especially when using Haskell — is the flexibility, power, and control one has of the parser and its output. The parser is defined in the same language as the type definitions; one does not have to learn to use a different tooling ecosystem, new grammar and language, and remember to update the generated files whenever you want to change something.

Let us then continue by giving an example of how one can build parsers using parser combinators, consider the parser in listing 5.6 for parsing a contract definition. As mentioned previously, the **`Parser`** type is generic and we can use it here to return a single type **`ContractDefinition`** that is used in the sum type **`SourceUnit`**.

To explain without getting bogged down by syntax and Haskell terms, we will go line-by-line through the function and talk about what each line does at a high level. The first two lines are the declaration, telling us it is a parser for a contract definition, and the start of the function.

(3) Parse and set the optional `abstract` property based on if it was successfully parsed

(4) Parse but discard the **`contract`** keyword

```haskell
parseContract :: Parser ContractDefinition
parseContract = do
  abstract <- isJust <$> (optional (try $ keyword "abstract") <?> "abstract")
  void (keyword "contract") <?> "contract"
  name <- parseIdentifier <?> "name"
  inheritance <- try . optional $ keyword "is" *> sepEndBy parseInheritance comma
  body <- braces parseContractBody
  pure $ ContractDefinition {abstract, name, inheritance, body}
```

Listing 5.6: A Haskell parser for function visibility

(5) Parse the identifier for the contract

(6) Optionally parse what the contract inherits

(7) Parse the contract body contained inside curly braces

(8) Return the parsed contract

The largest contract in the corpus is about 6500 lines of code in total, called **`TrustedNotifierManager`**. As a quick benchmark to measure performance, this file was run through the parser approximately 200 times using a tool called *hyperfine* [37] to give a very rough estimate of how well the parser performs. On the benchmark machine without background programs running, with an Intel i5-9400H processor with 16GB of RAM, the average running time was about 11ms to parse the entire file.

The smart contract file used in this benchmark weighs in at 240KiB. The performance of the parsers is often calculated by converting the speed to megabits per second (Mb/s), or by converting it to number of lines per second.

$$Perf = \frac{240\text{KiB}}{11\text{ms}} = 178.7\text{Mb/s}$$

This could potentially be slightly misleading as the file is very small, so to give the parser more work to do we concatenated the benchmarking file together 100 times, resulting in a file that is 671000 lines long and is 49MiB in size. On this file, the parser uses on average 328ms to parse the entire file.

$$Perf = \frac{49\text{MiB}}{328\text{ms}} = 156.6\text{Mb/s}$$

The parser performance in lines per second then clocks out to around 200000 lines per second, which ought to be more than enough for most applications of this parser as very few files in any programming language will ever reach sizes of several hundred thousand lines and are instead split up into smaller files. Concrete numbers for other parsers are hard to come by, little comparative research has been done with most benchmark passing as word of mouth through blogs and websites.

Finally, parsing is just a very small part of what interpreters, compilers and analyzers does. Comparisons can be done by running formatters for different languages against one another, but here the dominating factor wouldn't be the parsing but the analysis and calculations required to format the code.

5.4 ANALYZING SOLIDITY CONTRACTS

For the analysis part of the program, we can again leverage Haskell's type system and pattern matching while recursing down the abstract syntax tree to reach the data that we want. In other languages that lack such a rich and powerful level of abstraction one often has to resort to using other kinds of patterns to traverse trees. In Java for example the most common way to traverse graphs or trees is the visitor pattern [38].

Ten analyzers were written in Haskell, for a total of 650 lines of code, plus one manual analysis done by hand. Each of these parsers traverse the abstract syntax tree generated by the parser, allowing us to ask and answer questions about smart contracts written in Solidity. One could also leverage the parser to build other tools, like code formatters, linting tools or control flow visualizations and more

For example, a simple program to count all the functions (see 6.2 for results) in a parsed Solidity file and its corresponding abstract syntax tree can be implemented as in listing 5.7.

As we can see from this listing, the program is written in a very straight forward and intuitive way. Instead of having some abstract interface that we iteratively call to get nodes from a tree, we simply pattern match against the required constructs and recursively walk the syntax tree.

The function ***`countFns`*** is our entry point, and already here we see some of the power of pattern matching; if the body of a Solidity program is empty, we simply return an empty list, no need to check inside the function body when we can match at the function call level.

```
countFns :: Solidity -> [(SourceUnit, Int)]
countFns (Solidity []) = []
countFns (Solidity xs) = mapMaybe recurse xs

recurse :: SourceUnit -> Maybe (SourceUnit, Int)
recurse (Contract def) = Just (Contract, foldl' (\acc e ->
 ↪  acc + count e) 0 (def ^. #body))
recurse (Interface def) = Just (Interface, foldl' (\acc e
 ↪  -> acc + count e) 0 (def ^. #body))
recurse (Library def) = Just (Library, foldl' (\acc e ->
 ↪  acc + count e) 0 (def ^. #body))
recurse _ = Nothing

count :: ContractBodyElement -> Int
count (Constructor _) = 1
count (CFunction _) = 1
count (CFallbackFunction _) = 1
count (CReceiveFunction _) = 1
count _ = 0
```

Listing 5.7: Haskell program to count functions

Next, we leverage the `Maybe` type — a way to signal to the reader and program that the return value can have some value (`Just`) or be empty (`Nothing`) — to ignore any element in a source file that is not a contract, interface or library implementation. The *mapMaybe* function (`(a -> Maybe b) -> [a] -> [b]`) maps a function that returns an optional value onto a list of some type and returns the `Just` values.

We can expand on the simple implementation in listing 5.7 to build more sophisticated analyzers, but they all in essence work the same.

— 6 —
Analysis

Perhaps the biggest cause of underlying issues and bugs in software is the ever-increasing complexity of the software being written. By evaluating the complexity of individual programs, we can empirically say something about the underlying complexity of the program.

One could argue that simply parsing programs and assigning numerical values for their inherent complexity might not accurately represent the actual complexity. Complexity comes in many forms and shapes, ranging from programming languages like APL [39], where a single line might have more complexity than the same function in Python due to its use of glyphs over conventional words, to working with inherently complex fields, such as attempting to program software for the Large Hadron Collider.

Where more popular and often-used languages like C++ and Java have had studies done about their inherent complexity, less so has been done on smart contract and Solidity in particular.

6.1 GETTING CONTRACTS

As Solidity is a compiled language, the source code for published contracts exists only as bytecode on the blockchain, which, while useful, is not the best target to analyze the code on. However, authors can upload the source for published contracts to Etherscan [40], a website for exploring and analyzing Ethereum, where the bytecode is compared to the newly compiled bytecode that enables the verification of contracts. This means that users who want to interact with contracts know that the contract at a specific address matches the source provided by the author of said contract, and they can therefore make sure everything is in order and not have to take the author at face value.

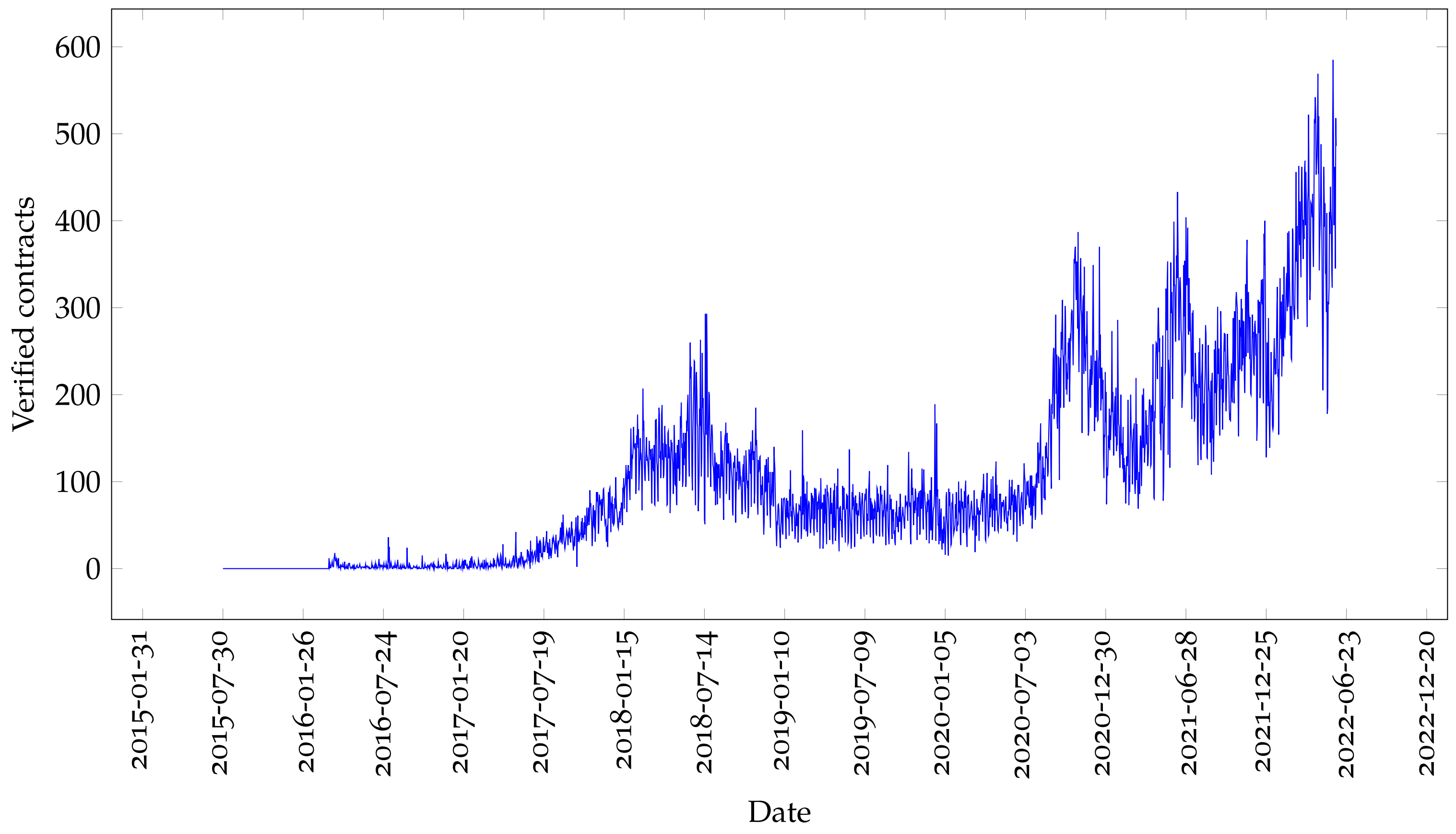


Figure 6.1: Number of verified contracts per day on Etherscan.

We will compare and verify the results that Hegedüs [41] reported, as well as some new metrics. However, between this thesis and the latter paper, it seems that Etherscan has changed their API as it is now impossible to find all verified contracts. One can either download all verified contracts with a valid open-source license or scrape the latest 500 verified contracts. The former yields 4678 contracts that can be downloaded, while the latter obviously restricts us to 500.

Files	Lines	Code	Comments	Blanks
11884	5288673	2628995	1888483	771195

Table 6.1: Statistics on downloaded contracts

A Python script was developed that reads the CSV file that is available from Etherscan and attempts to download all these contracts and extract them to their own files without conflicts and errors. Although about 4500 contracts yield a significant amount of files, it might not be an accurate representation of all contracts, as these are the ones where the author explicitly allows reuse and distribution via a license choice. It is nonetheless the best option for downloading a large corpus of contracts.

After running the Python script and expanding all files, we end up with the data in table 6.1. Of these 11884, 11555 were parseable by our developed parser, the files that were not parseable were nearly all files that were written in older versions of Solidity where the syntax grammar was slightly different from the version we targeted.

6.2 THE AVERAGE SOLIDITY FILE

Based on the data from table 6.3 and 6.1, we get the averages seen in table 6.2. The average Solidity file that was downloaded is 445 lines long, and based on the number from table 6.1, contains about 220 lines of code, 160 lines of comments, and 65 blank lines.

Type	Average per file	Average per file from [41]
Contract	2.01	4.44
Interface	1.64	0.04
Library	0.81	0.13

Table 6.2: Statistics on downloaded contracts

As an initial analysis of the programs downloaded from Etherscan, we will look at the top-level defined items in the source files. This includes things like contracts, interfaces, errors, and more. From 11555 successfully parsed files containing 97482 defined top-level elements we get the following data in table 6.3.

As one would expect, the vast majority of defined top-level elements are contracts, interfaces, and libraries with pragmas slightly behind. Interestingly, there is a large decrease in the number of contracts per file, but instead a large increase in interfaces and libraries. This might come from the growing ecosystem, where users are able to import or copy existing code that performs dangerous operations or are often misused features over coding these themselves.

Element	Count	Percentage
Pragma	31068	31.87%
Contract	23137	23.73%
Interface	18925	19.41%
Import	10142	10.40%
Library	9356	9.60%
Error	4320	4.43%
ConstVar	341	0.35%
Struct	173	0.18%
Enum	14	0.01%
Function	6	0.01%
UserDefinedType	0	0%

Table 6.3: Counted top-level elements

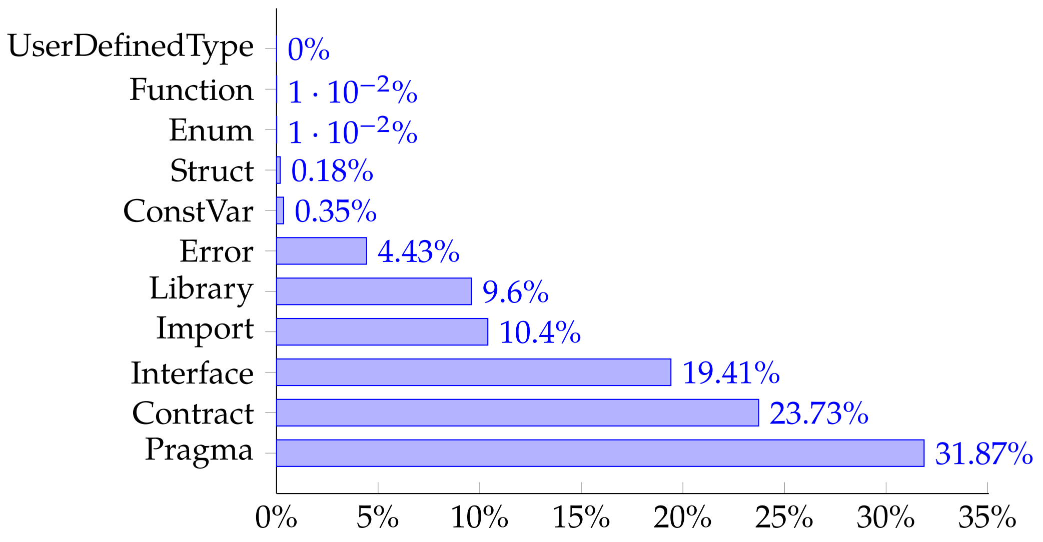


Figure 6.2: Histogram of top level element occurrence

6.3 LICENSES

Another soft requirement, similar to the pragmas in section 6.4, all Solidity programs should include a header comment denoting the license used for the smart contract. Of the 11884 downloaded files, 11070 contains a license.

Name	Count	Percentage
MIT License	8662	78.62%
None	887	8.05%
GNU General Public License v3.0 only	544	4.94%
GNU Affero General Public License v3.0 only	330	3.00%
GNU General Public License v3.0 or later	175	1.59%
Apache License 2.0	120	1.09%
GNU Affero General Public License v3.0 or later	117	1.06%
The Unlicense	70	0.64%
GNU General Public License v2.0 or later	36	0.33%
GNU Lesser General Public License v3.0 or later	31	0.28%
Do What The F*ck You Want To Public License	21	0.19%
GNU General Public License v2.0 only	13	0.12%
Simple Public License 2.0	11	0.10%

Table 6.4: Double digit licenses

6.4 PRAGMAS

Solidity has language pragmas used to specify compiler version targets and optionally the ABI (Application Binary Interface) encoder and decoder as well as a few select experimental features. In some ways, it is slightly similar to how language pragmas work in Haskell, where one can use them to enable language features on a per file/project basis, which works the same in Solidity. Files can specify different requirements for pragmas, even when importing files that use a different one.

The version ranges in Solidity follow the very popular Semantic Versioning specification [42]. Without going into too much detail about it, essentially it means that versions with a ^ can be compiled with any version starting from the one specified up to, but not including, the next minor or major version (that is, `^0.8.0` means any version from 0.8.0

Pragma	Count	Percentage
`solidity ^0.8.0`	22553	72.59%
`solidity ^0.8.4`	946	3.04%
`solidity ^0.8.1`	791	2.54%
`solidity >=0.7.0 <0.9.0`	700	2.25%
`solidity >=0.6.0 <0.8.0`	415	1.33%
`solidity 0.8.9`	353	1.13%
`solidity ^0.8.9`	345	1.11%
`solidity 0.6.12`	301	0.97%
`experimental ABIEncoderV2`	299	0.96%
`solidity ^0.8.7`	291	0.94%

Table 6.5: Top 10 language pragmas

to 0.9.0, exclusive). Versions without any kind of range symbol must be compiled with the specific version.

As we can see from table 6.5, the vast majority of contracts that we were able to download are targeting the 0.8 release of Solidity, released in December 2020. Of 160 unique version ranges, in the top ten almost 84% of the files are compiled against the latest Solidity release.

6.5 STATEMENT AND EXPRESSION FREQUENCY ANALYSIS

Another way to analyze programming languages and their usage is to look at how users use the language. In [43], the authors analyze the frequency of statements for Java, C++, and C which allows us to compare usage between different languages.

It is mostly impossible to compare languages directly, as there are subtle differences not just in semantics, but also in available features, how the source code is expressed, and whether the languages allow for both expressions and statements or have just one or the other. In our case, the parser and syntax for Solidity separate statements and expressions, which means that some data points will not map directly to, for example, Java.

Statement and expression data

In table 6.6 and its corresponding histogram 6.3, you can see the total and percentage usage for all statements in our corpus. Similarly, in table 6.7

and its corresponding histogram 6.4, the data for total and percentage usage for all expressions in our corpus are shown.

Statement	Count	Percentage
ExpressionStatement	432884	50.19%
Return	162303	18.82%
VariableStatement	123884	14.36%
If	72565	8.41%
Emit	33709	3.91%
For	11571	1.34%
Revert	8117	0.94%
While	7386	0.86%
Assembly	6911	0.80%
Try	2257	0.26%
Break	485	0.06%
Continue	174	0.02%
DoWhile	171	0.02%

Table 6.6: Counted statements

As can be seen in the data for statements, expressions are the most used kind of statements. This should not come as a surprise, as expressions are where a lot of the heavy lifting is done; to do any operation on variables and parameters, one needs to use expressions. We will look at these in more detail in the next part of this section.

The next two most commonly used statements are the return keyword and declarations of variables. Interestingly, as we look further down the statement data, we see that the usage of **`continue`** and **`break`** is very low.

Looking at table 6.7 – for expressions – we see that there is a dominating item: identifier expressions. These are the expressions for looking up variables in the scope of a function, method, constructor, etc. Without this one could not do much in any language, and as such it makes sense that it is very frequently used. The next item is the function call expression, which, as the name suggests, is any and all cases of calling a function like `foo(1, 2)`.

The next most commonly used afterward are binary expressions; these are all expressions in which we perform a binary operation on two items, summing them, comparing them, and so on. An expression literal is the usage of literals in the language; these are numbers, strings, and booleans.

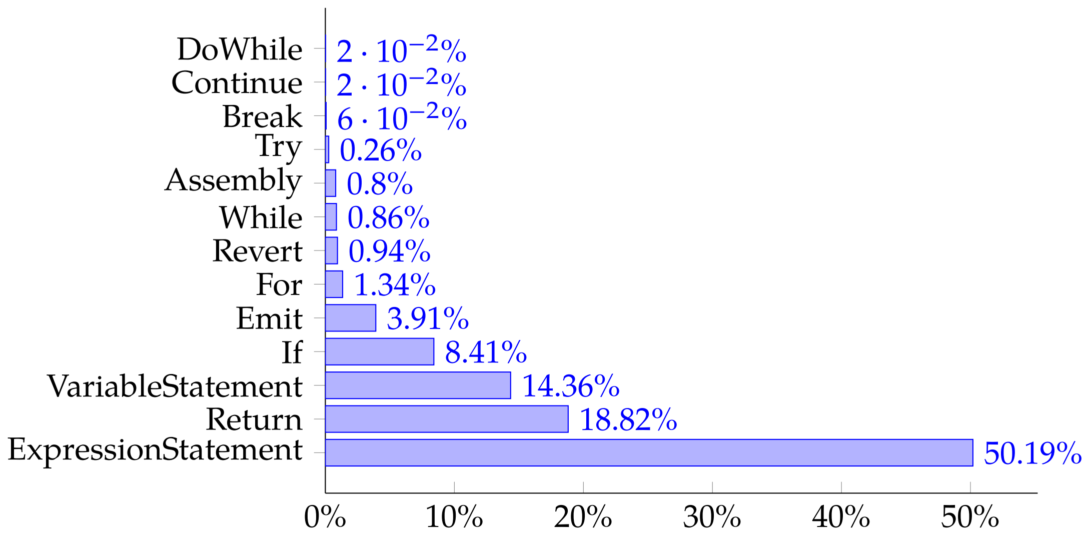


Figure 6.3: Histogram of statement occurrence

Expression	Count	Percentage
IdentifierExpression	2064345	46.42%
FunctionCall	698499	15.71%
BinaryExpression	499075	11.22%
ExpressionLiteral	448926	10.09%
MemberAccess	290447	6.53%
IndexExpression	181386	4.08%
ElementaryTypeExpression	132095	2.97%
UnaryExpression	43658	0.98%
TupleExpression	32646	0.73%
NewType	28989	0.65%
FunctionCallOptions	8938	0.20%
MetaType	8809	0.20%
ConditionalExpression	4907	0.11%
PayableConversion	4681	0.11%
InlineArrayExpression	74	0.00%

Table 6.7: Counted expressions

Member access and index expressions are used to look up identifiers associated with a class or to index into arrays. The final thing we will mention here is the elementary type expression, used to specify the types of variables, arrays, and such; it includes the literal `bool`, `bytes32`, `uint`, and so on.

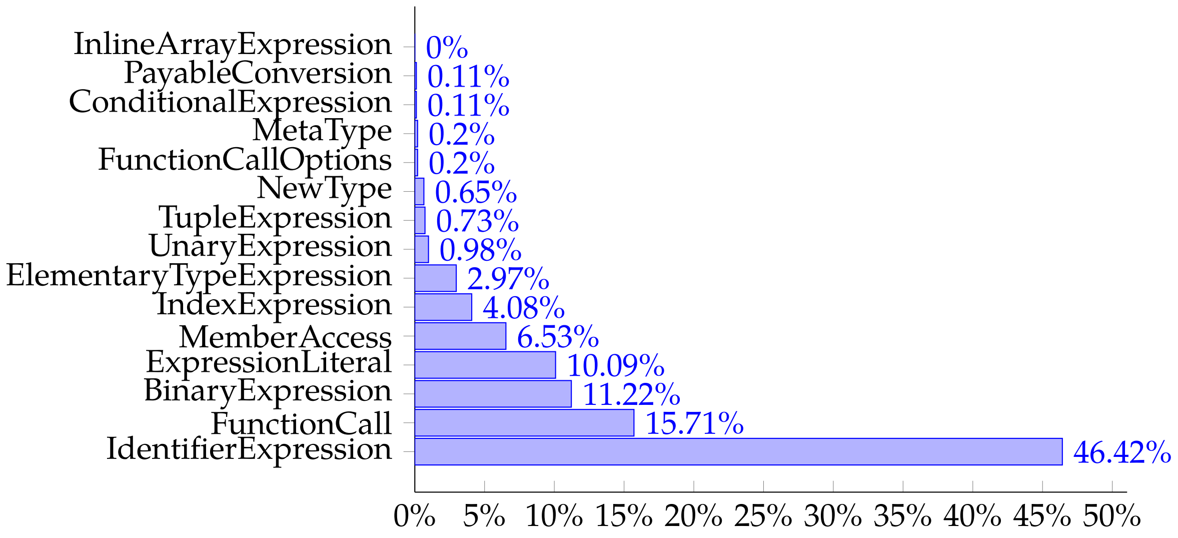


Figure 6.4: Histogram of expression occurrence

Frequency analysis

In [43], the authors calculate the frequency of statements on a per-line basis, meaning that the frequency of, say, function calls is how many function calls there are per line of the source file. This way of calculating frequencies is different from what we have done, as the parser implemented for this thesis omits this information from its AST. However, trends and similarities will still be apparent.

In table 6.9, we see that there is a difference between solidity and the other languages compared to it. While the other follow the same order for the top four most used statements, Solidity differs quite a bit. It is worth noting, however, that the most used expression in Solidity is a function call, so one could argue that it is the most used, but it is not a statement alone in Solidity.

Interestingly, as we can see in the data in table 6.8, Solidity uses `return` much more than the other languages and assigns and creates variables

Statement	Solidity	Java (median)	C++ (median)	C (median)
FunctionCall	7.88%	0.39	0.40	0.22
Variable	14.36%	0.19	0.17	0.19
If	8.41%	0.014	0.013	0.013
Continue	0.02%	0.0009	0.0008	0.0012
Break	0.06%	0.004	0.007	0.001
Return	18.82%	0.058	0.045	0.037
Switch	0%	0.0008	0.0014	0.0021
Try	0.26%	0.0072	0	N/A

Table 6.8: Frequencies of statements with data from [43]

Order	Solidity	Java	C++	C
1	Expression	FunctionCall	Functioncall	FunctionCall
2	Return	Variable	Variable	Variable
3	Variable	If	If	If
4	If	Return	Return	Return

Table 6.9: Four most used statements with data from [43]

more frequently than them. Although it is difficult to analyze the data when comparing percentages with median values that are also calculated differently, we can use the data from [44] in table 6.10 to get a more direct comparison.

Statement	Solidity	Java
Break	0.06%	0.2212%
Continue	0.02%	0.0430%
DoWhile	0.02%	0.0115%
For	1.34%	0.2583%
If	8.41%	2.6462%
Return	18.82%	2.8310%
Switch	0%	0.0677%
Try	0.26%	0.3645%
Variable	14.36%	3.7576%
While	0.86%	0.1077%

Table 6.10: Usage of statements with data from [44]

As we can see in this table, where similar constructs exist, some have very different usage percentages. The **`break`** keyword is used much more frequently in Java than in Solidity, while a for loop is much more common in Solidity compared to Java.

However, the two biggest outliers are the **`return`** keyword and the variable declarations. For the first one, one might surmise that this is because in object-oriented programming, a lot of the structure of the programs is around internal mutation of variables declared with classes; thus not as many functions or methods return values. Another reason might be that Solidity leans more on smaller functions that one chains together to build functionality since it is lacking in constructs like classes.

For the difference in the variable declaration, a probable reason is differences in how the variable declarations are represented in the Java syntax. Looking at table 3 in [44], we see that there is also a variable declaration expression, something that Solidity does not have.

Expression	Solidity	Java
Array access	2.48%	0.7587%
Conditional expression	0.06%	0.1718%
Field access	3.27%	1.1060%
Binary expression	5.62%	4.9584%
Unary expression	0.49%	1.09%

Table 6.11: Usage of expressions with data from [44]

Next, we have the usage of expressions. Where Zhu, Whitehead Jr., Sadowski, *et al.* only looked at statements in Java, C++ and C, Qiu, Li, Barr, *et al.* also looked at expression usage. However, most of the expressions in Java have no direct mapping to a similar construct in Solidity, so we took the ones with which we had a direct comparison. Of note here is that we combined the prefix and postfix expressions into a single unary expression row, as Solidity does not separate these at a syntax level.

Interestingly, the usage of binary expressions does not differ dramatically as others. This means that, even though the two languages have very different applications, the number of binary expressions in both languages should be about the same for a given source file in either language. Interestingly, array accessing is much more common in Solidity compared to Java.

Declaration	Solidity (total)	Java (per file)
Class/contract declaration	23.73%	87.5%
Enum declaration	0.01%	2.7%
Import declaration	10.40%	86.4%
Interface declaration	19.41%	11.6%

Table 6.12: Usage of top-level declarations with data from [44]

And finally, a quick look at top-level declarations in Java and Solidity. The numbers here mean slightly different things, as in the Solidity case it is the percentage of a construct in total, while Java is how many files contain the given construct. As such, we cannot meaningfully compare the numbers directly, but it gives a view of the frequency of top-level usage.

Other comparisons to Java

Function arity

In [45], Grechanik, McMillan, DeFerrari, *et al.* find that the average Java method (since Java does not have freestanding functions, we can compare them directly) takes 1.5 arguments with a median of 1 with the largest arity of a method they found was 30. For comparison, we found that in Solidity the average function accepts 1.45 parameters, has a median of 1.3 and a maximum arity of 10. Hegedüs does a similar analysis and found that the average function in Solidity has 1.5 parameters with a median of 1.5. The largest arity found in their case was 12.

Functions per class or similar construct

Grechanik, McMillan, DeFerrari, *et al.* find that the average class has 3.5 methods with a median of 4. For interfaces, they have an average of 3.4 methods with a median of 3. When we compare this with Solidity, we find that the average contract (the most similar construct to a class) has on average 9.1 functions contained in it with a median of 5. For an interface, the average is 5.2 with a median of 3 while a library has an average of 8.3 with a median of 6.

In [41], the author makes this comparison, but does not look at individual constructs, but rather at files as a whole. In it they find that the average contract/interface/library has 4.94 functions in it with a median of 3.

The class with the largest number of methods found in [45] contained 1175 methods. Conversely, the largest contract in Solidity had 93 functions. For interfaces, the largest one found had 68 methods, while in Solidity it had 380. The library with the most functions in Solidity had 380. And in [41], the construct with the must functions has 127 functions in it.

The difference between an interface and a library in Solidity is that an interface is a contract that specifies functions that must be implemented, while a library is a free-standing construct that one can import and use functions from. It is most commonly used to provide common or security-related functionality that users do not want to write themselves, allowing users to import functionality at will.

6.6 COMPLEXITY ANALYSIS

Halstead complexity measures

Halstead complexity measures were introduced by Halstead [46] as a way of quantifying and identifying software properties. By traversing the AST and analyzing the operators and operands used in these programs, we can calculate a set of measurements such as how difficult it is to understand the code.

What is defined as an operand and an operator? In the case of most programming languages, and indeed in our case too, the operands are the identifiers, constants and type names used in a program, while operators are more or less the rest of the program. Anything that operates on operands are counted as operators. This means that functions, keywords (**`return`**), operators (`!=`, ++), control flow operators, and so on are counted among these.

Halstead defines a few different measures, from program vocabulary to estimated program length, and from difficulty to an estimation of the number of delivered bugs. Although this has been used to analyze complexity [47], the accuracy of the metrics may not be as accurate as originally depicted [48]. We chose to focus on the difficulty and estimated bugs delivered.

When traversing the code, we collect these operands and operators into their distinct and total counts and use these to calculate our results. By parsing our corpus, we find that the average Solidity file has 106 operands and 182 operators on average. The file with the largest number of operands contained 3176 of them, while for operators it contained 4765.

The formula to obtain the difficulty of a program is as seen in 6.1, while the formula for bugs is in 6.2.

$$D = \frac{\text{distinct operators}}{2} \times \frac{\text{total operands}}{\text{distinct operands}} \tag{6.1}$$

$$\begin{aligned} n &= \text{total operators} + \text{total operands} \\ N &= \text{distinct operators} + \text{distinct operands} \\ V &= n \times \log_2 N \\ E &= D \times V \\ B &= \frac{\mathrm{E}^{\frac{2}{3}}}{3000} \end{aligned} \tag{6.2}$$

From these equations, we find that the average difficulty of a Solidity program is 21.6, the least difficult program being 1.5 and the most difficult one being 142.4. For bugs, the average is 0.43 with the least buggy program getting 1.7×10^{-3} and the most buggy gets 7.62. Sadly, we could not find any similar analysis of other programming languages on a scale similar to our experiment, a small evaluation [49] of using Halstead complexity measures for Python, Java, and C++ was performed, but is not representative of real-world program analysis.

McCabe's Cyclomatic Complexity

Cyclomatic complexity is a measure that is used to calculate the complexity of programs by analyzing their control flow. Normally, this is done by creating a control flow graph from the analyzed program and then traversing this graph and calculating the complexity based on the branching within this graph.

However, Hegedüs [41] has done this analysis previously but did it differently simply by calculating the sum of control flow statements in a program, so for comparison, we will do the same. Their analysis was done in 2018 while ours was done in 2022, so we can do comparisons across two points in time for the Solidity ecosystem.

Paper	Min	Avg	Med	Max
Hegedüs	0	1.15	1	43
Ours	0	1.16	1	15.3

Table 6.13: Results of cyclomatic complexity.

As can be seen in table 6.13, the only major difference is that in our corpus there is no function nearly as complex as the most complex function found in theirs. In [41] a comparison is made between object-oriented languages and Solidity based on the collected metrics, and based on this we find that the cyclomatic complexity for Solidity is still low compared to languages like Java or C++.

In [50], an analysis was performed on a large corpus of Java and C programs: nearly two million Java files and almost five hundred thousand C programs. From this we get table 6.14 as a comparison between Solidity, Java and C.

Language	Min	Avg	Med	Max
Java	1	2.33	1	4377
C	1	5.98	3	18320
Solidity	0	1.16	1	15.3

Table 6.14: Results of cyclomatic complexity.

Nesting

Another way to analyze complexity in programs is to see how much nesting happens in the codebase. While Halstead and McCabe attempt to quantify complexity based on control flow or formula, nesting is much more straightforward and intuitive.

A program with deep nesting would be considered by many developers to be a code smell. High indentation means that you have nested control flow and iteration deeply without extracting it to auxiliary helper functions and methods to be able to give them names, comments, etc.

From table 6.15, we can see that again the only major difference between us and Hegedüs is that the maximum found in [41] is much larger than any found in our corpus.

Paper	Min	Avg	Med	Max
Hegedüs	0	0.14	0	17.86
Ours	0	0.13	0	9

Table 6.15: Results of nesting analysis.

Unfortunately, no similar analysis was found for other languages. Although Hegedüs claims that these values are low compared to object-oriented languages, we have not found data to support this. In our opinion, this claim is likely true, but it needs data to be verified.

6.7 SMALLER ANALYZES

Finally, we will look at some smaller data points and analyzes that do not have any direct comparisons, but were done out of interest of the author.

Operator analysis

A full analysis of all the programs was done to find and count the usage of all the operators in our corpus. The closest comparison the author could find to this data was in [45], where the authors found 25523 occurrences of the increment operator and 2005 occurrences of the decrement operator. In comparison, we found 17229 occurrences of the increment operator and 1937 occurrences of the decrement operator.

Element visibility

As a final analysis, we looked at the visibility of constructs in contracts. This was primarily to show how the frequency and usage of the visibility modifiers, but also because they can inadvertently become a security concern. Public statue variables can be a large concern, though we luckily found none. Interestingly, the community has landed on always explicitly defining a visibility modifier on state variables, but our analysis found quite a lot missing this.

Name	Operator	Count	Percentage
Assignment	=	151877	27.98%
Equal	==	57548	10.60%
NotEqual	!=	52098	9.60%
GreaterThan	>	28303	5.21%
Add	+	25388	4.68%
Mul	*	21123	3.89%
And	&&	20288	3.74%
Not	!	19297	3.56%
LessThan	<	19254	3.55%
GreaterEqual	>=	18487	3.41%

Table 6.16: Ten most frequently used operators

Construct	Public	Internal	External	Private	None
Functions	98885	49947	28486	21807	6
Constructors	631	189	0	0	11487
State variables	0	51507	39037	4552	3653

Table 6.17: Construct visibility count

— 7 —

Conclusion

In this thesis, we created a fully compliant parser of the Solidity smart contract programming language for Ethereum. Using this parser, we fetched a large corpus of publicly available smart contracts and analyzed their structure for similarities to other popular programming languages.

We found that the most used version of Solidity across all downloaded smart contracts was version 0.8 and above, having above 75% usage, showing that nearly all users of the language are on a very recent version of the compiler for it.

Based on statement and expression analysis, we found and compared similar constructs to Java, C++, and C and found that while there are similarities, certain statements and expressions are used with very different frequencies across these languages. While Java, C++, and C have the same four most used statements, Solidity differs in having a separate expression type but also using the **`return`** keyword and variable assignments a lot more frequently than them.

When comparing function arity, the averages were very close between Java and Solidity. We compared the number of methods/functions in classes, contracts, and interfaces and found that Solidity contracts have on average double the number of methods compared to a Java class. For interfaces, the difference is not as large, but Solidity interfaces on average contain two more methods than a Java interface.

Finally, we performed an assortment of complexity analyzes on our corpus using three different metrics. We first analyzed the programs using Halstead's complexity measure and followed it up with McCabe's cyclomatic complexity analysis before ending with a quick look at the average nesting of statements. We found that the average cyclomatic complexity for Solidity was much lower than C and more than half that of

Java.

Through our analyses, we show that while there are similarities between object-oriented languages like Java and Solidity, the language and its use case means that it does not have the same average complexity. Where Java is used to build large programs with many users, Solidity is meant to create digital contracts between entities on the Ethereum blockchain.

Based on the analysis performed in this thesis, we can draw some conclusions about the intended usage of Solidity compared to more mainstream programming languages. In the cases where we have direct comparisons, we find that Solidity is less complex than Java on average with a much lower maximum complexity.

It would be interesting to answer the question if Solidity is a better language for writing smart contracts than other general purpose languages, though this would require more data and a different set of analyzes; and would likely be much harder to answer as one would need to quantify question like "how is a language better suited than another", "how many bugs happen due to language constructs and not programmer error" and others.

— A —

Appendix

A.1 SOLIDITY EXAMPLE AST

```
[ Pragma
    ( PragmaDefinition
        { pragma = "solidity >=0.4.16 <0.9.0" }
    )
, Contract
    ( ContractDefinition
        { abstract = False
        , name = Identifier "SimpleStorage"
        , inheritance = Nothing
        , body =
            [ CStateVariableDec
                ( StateVariableDec
                    { kind = ElementaryType (
 ↪  UnsignedInteger Nothing )
                    , modifiers = Just []
                    , ident = Identifier "storedData"
                    , expr = Nothing
                    }
                )
            , CFunction
                ( FunctionDefinition
                    { name = Identifier "set"
                    , params =
                        [ Parameter
                            { kind = ElementaryType (
 ↪  UnsignedInteger Nothing )
                            , location = Nothing
                            , ident = Just
                                ( Identifier "x" )
                            }
```

```
                          ]
                      , restrictions = [ FuncVisibility
↪  FuncPublic ]
                      , returns = Nothing
                      , body = BlockStatement
                          [ ExpressionStatement
                              ( BinaryExpression Assign
                                  ( IdentifierExpression
                                      ( Identifier
↪  "storedData" )
                                  )
                                  ( IdentifierExpression
                                      ( Identifier "x" )
                                  )
                              )
                          ]
                      }
                  )
              , CFunction
                  ( FunctionDefinition
                      { name = Identifier "get"
                      , params = []
                      , restrictions =
                          [ FuncVisibility FuncPublic
                          , FuncMutability View
                          ]
                      , returns = Just
                          [ Parameter
                              { kind = ElementaryType (
↪  UnsignedInteger Nothing )
                              , location = Nothing
                              , ident = Nothing
                              }
                          ]
                      , body = BlockStatement
                          [ Return
                              ( Just
                                  ( IdentifierExpression
                                      ( Identifier
↪  "storedData" )
                                  )
                              )
                          ]
                      }
                  )
              ]
          }
```

```
        )
]
```

Bibliography

[1] I. Frost. "A crypto user sent $50,000 to a smart contract. it's gone forever." (2020), [Online]. Available: https://decrypt.co/51692/a-crypto-user-sent-50000-to-a-smart-contract-its-gone-forever (visited on 2022-05-25).

[2] L. Frost. "Mistake costs user $1.1 million in aave." (2020), [Online]. Available: https://decrypt.co/45537/user-loses-1-1-million-in-aave-due-to-mistake (visited on 2022-05-25).

[3] C. Goodin. "Really stupid "smart contract" bug let hackers steal $31 million in digital coin." (2021), [Online]. Available: https://arstechnica.com/information-technology/2021/12/hackers-drain-31-million-from-cryptocurrency-service-monox-finance/ (visited on 2022-05-25).

[4] M. Sipser, *Introduction to the Theory of Computation*, eng, 3rd ed. Australia: Cengage learning, 2013, ISBN: 9781133187813.

[5] A. V. Aho, *Compilers: principles, techniques, and tools* (Pearson custom library), eng, Second edition, Pearson new international edition. Harlow, Essex: Pearson, 2014, ISBN: 9781292024349.

[6] P. W. Abrahams, "A final solution to the dangling else of algol 60 and related languages," *Commun. ACM*, vol. 9, no. 9, pp. 679–682, 1966-09, ISSN: 0001-0782. DOI: 10.1145/365813.365821. [Online]. Available: https://doi.org/10.1145/365813.365821.

[7] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide* (Monographs in Computer Science), eng. New York, NY: Springer New York, 2007, ISBN: 9780387202488.

[8] T. Parr and K. Fisher, "Ll(*): The foundation of the antlr parser generator," *SIGPLAN Not.*, vol. 46, no. 6, pp. 425–436, 2011-06, ISSN: 0362-1340. DOI: 10.1145/1993316.1993548. [Online]. Available: https://doi.org/10.1145/1993316.1993548.

[9] R. A. Frost and R. Hafiz, "A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time," *ACM SIGPLAN Notices*, vol. 41, pp. 46–54, 2006.

[10] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," *SIGPLAN Not.*, vol. 39, no. 1, pp. 111–122, 2004-01, ISSN: 0362-1340. DOI: 10.1145/982962.964011. [Online]. Available: https://doi.org/10.1145/982962.964011.

[11] G. Hutton, "Higher-order functions for parsing," *Journal of Functional Programming*, vol. 2, no. 3, pp. 323–343, 1992. DOI: 10.1017/S0956796800000411.

[12] G. Hutton and E. Meijer, "Monadic Parser Combinators," Department of Computer Science, University of Nottingham, Technical Report NOTTCS-TR-96-4, 1996.

[13] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," 2001-12.

[14] J. Kurš, J. Vraný, M. Ghafari, M. Lungu, and O. Nierstrasz, "Efficient parsing with parser combinators," *Science of Computer Programming*, vol. 161, pp. 57–88, 2018, Advances in Dynamic Languages, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2017.12.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642317302654.

[15] M. Dominus. "Monads are like burritos." (2009), [Online]. Available: https://blog.plover.com/prog/burritos.html (visited on 2022-05-27).

[16] J. Iry. "A brief, incomplete, and mostly wrong history of programming languages." (2009), [Online]. Available: http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html (visited on 2022-05-27).

[17] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, "Blockchain," English, *Business And Information Systems Engineering*, vol. 59, no. 3, pp. 183–187, 2017-06.

[18] M. Wander. Licensed under Creative Commons Attribution-ShareAlike 3.0 Unported. (2022), [Online]. Available: https://commons.wikimedia.org/wiki/File:Bitcoin_Block_Data.png (visited on 2022-02-16).

[19] "Satoshi nakamoto; brain scan," eng, *The Economist (London)*, vol. 428, no. 9107, p. 6, 2018, ISSN: 0013-0613.

[20] B. Wang, J. Sun, Y. He, D. Pang, and N. Lu, "Large-scale election based on blockchain," eng, *Procedia computer science*, vol. 129, pp. 234–237, 2018, ISSN: 1877-0509.

[21] M. Montecchi, K. Plangger, and M. Etter, "It's real, trust me! establishing supply chain provenance using blockchain," *Business Horizons*, vol. 62, no. 3, pp. 283–293, 2019, ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2019.01.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007681319300084.

[22] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols(extended abstract)," in *Secure Information Networks: Communications and Multimedia Security IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99) September 20–21, 1999, Leuven, Belgium*, B. Preneel, Ed. Boston, MA: Springer US, 1999, pp. 258–272, ISBN: 978-0-387-35568-9. DOI: 10.1007/978-0-387-35568-9_18. [Online]. Available: https://doi.org/10.1007/978-0-387-35568-9_18.

[23] A. de Vries and C. Stoll, "Bitcoin's growing e-waste problem," *Resources, Conservation and Recycling*, vol. 175, p. 105 901, 2021, ISSN: 0921-3449. DOI: https://doi.org/10.1016/j.resconrec.2021.105901. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921344921005103.

[24] A. de Vries, U. Gallersdörfer, L. Klaaßen, and C. Stoll, "Revisiting bitcoin's carbon footprint," *Joule*, vol. 6, no. 3, pp. 498–502, 2022, ISSN: 2542-4351. DOI: https://doi.org/10.1016/j.joule.2022.02.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542435122000861.

[25] A. de Vries, "Bitcoin energy consumption index," 2022. [Online]. Available: https://digiconomist.net/bitcoin-energy-consumption/ (visited on 2022-05-12).

[26] "Cryptorank." (2022), [Online]. Available: https://cryptorank.io/ (visited on 2022-04-28).

[27] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2022.

[28] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2012-02, 1338 (est.) [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber%20=50372.

[29] N. I. Adams, D. H. Bartley, G. Brooks, *et al.*, "Revised<sup>5</sup> report on the algorithmic language scheme," *SIGPLAN Not.*, vol. 33, no. 9, pp. 26–76, 1998-09, ISSN: 0362-1340. DOI: 10.1145/290229.290234. [Online]. Available: https://doi.org/10.1145/290229.290234.

[30] "Openzeppelin." (2022), [Online]. Available: https://openzeppelin.com/contracts/ (visited on 2022-04-28).

[31] "Patricia merkle trees." (2021), [Online]. Available: https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/ (visited on 2022-05-25).

[32] T. Tani, *Ethereum evm illustrated*, 2022. [Online]. Available: https://github.com/takenobu-hs/ethereum-evm-illustrated (visited on 2022-05-28).

[33] T. Parr and K. Fisher, "LL(*): The Foundation of the ANTLR Parser Generator," in *LL(*): The Foundation of the ANTLR Parser Generator*, Retrieved 2021-06-22, 2011-06. [Online]. Available: https://www.antlr.org/papers/LL-star-PLDI11.pdf (visited on 2021-06-22).

[34] "Solidity antlr grammar file." (2022), [Online]. Available: https://github.com/ethereum/solidity/blob/develop/docs/grammar/SolidityParser.g4 (visited on 2022-02-11).

[35] "Solidity language grammar." (2022), [Online]. Available: https://docs.soliditylang.org/en/latest/grammar.html (visited on 2022-02-11).

[36] M. Karpov, *Megaparsec*, 2022. [Online]. Available: https://github.com/mrkkrp/megaparsec (visited on 2022-05-20).

[37] D. Peter, *Hyperfine*, 2022. [Online]. Available: https://github.com/sharkdp/hyperfine (visited on 2022-05-20).

[38] B. C. Oliveira, M. Wang, and J. Gibbons, "The visitor pattern as a reusable, generic, type-safe component," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, ser. OOPSLA '08, Nashville, TN, USA: Association for Computing Machinery, 2008, pp. 439–456, ISBN: 9781605582153. DOI: 10.1145/1449764.1449799. [Online]. Available: https://doi.org/10.1145/1449764.1449799.

[39] K. E. Iverson, *A programming language*, eng, New York, 1962.

[40] "Etherscan." (2022), [Online]. Available: https://etherscan.io/ (visited on 2022-04-08).

[41] P. Hegedüs, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 35–39, ISBN: 9781450357265. DOI: 10.1145/3194113.3194119. [Online]. Available: https://doi.org/10.1145/3194113.3194119.

[42] "Semantic versioning." (2022), [Online]. Available: https://semver.org/ (visited on 2022-05-10).

[43] X. Zhu, E. J. Whitehead Jr., C. Sadowski, and Q. Song, "An analysis of programming language statement frequency in c, c++, and java source code," *Software: Practice and Experience*, vol. 45, no. 11, pp. 1479–1495, 2015. DOI: https://doi.org/10.1002/spe.2298. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2298. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2298.

[44] D. Qiu, B. Li, E. T. Barr, and Z. Su, "Understanding the syntactic rule usage in java," *Journal of Systems and Software*, vol. 123, pp. 160–172, 2017, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.10.017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216302126.

[45] M. Grechanik, C. McMillan, L. DeFerrari, *et al.*, "An empirical investigation into a large-scale java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10, Bolzano-Bozen, Italy: Association for Computing Machinery, 2010, ISBN:

9781450300391. DOI: 10.1145/1852786.1852801. [Online]. Available: https://doi.org/10.1145/1852786.1852801.

[46] M. H. Halstead, *Elements of software science*, eng, New York, 1977.

[47] M. Alfadel, A. Kobilica, and J. Hassine, "Evaluation of halstead and cyclomatic complexity metrics in measuring defect density," in *2017 9th IEEE-GCC Conference and Exhibition (GCCCE)*, 2017, pp. 1–9. DOI: 10.1109/IEEEGCC.2017.8447959.

[48] R. Al-Qutaish and A. Abran, "An analysis of the design and definitions of halstead's metrics," 2005-09.

[49] S. Abdulkareem and A. Abboud, "Evaluating python, c++, javascript and java programming languages based on software complexity calculator (halstead metrics)," 2021-02.

[50] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju, "Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions," *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 589–618, 2016. DOI: https://doi.org/10.1002/smr.1760. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1760. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1760.