MIT Computational Law Report

Writing in Sign: Code as the Next Contract Language? ✍

Megan Ma

Published on: Aug 14, 2020

License: Creative Commons Attribution 4.0 International License (CC-BY 4.0)

Abstract

The use of formal logic for governance has a rich ancestry. From Liebniz to Boole, mathematical notation was conceivably dialogic; capable of expression and construction of arguments. With the rise of artificial legal intelligence, computable contracts – and more broadly, computable law – are making a powerful return. Contracts may be represented as computer data with terms made 'machine-readable' through a process of conversion: from descriptive natural language to consonant computer instruction. Conditions of agreements are not explained but listed as structured data records. However, despite the capacity to express contracts in an alternative computable form, no means exist for interpreting the code. Rather, the interpretation of contractual obligations is perceived as irrelevant. Should digital data inscription and processing be considered a form of legal writing? If so, would it change the character of law?

This paper considers the conundrum: what is the significance of the medium in contract drafting? *The paper is, therefore, a thought experiment on the translation of text to numbers by unpacking several formal programming languages used in computable contracts. In identifying the logic of these languages, the paper studies methods of legal writing. The hypothesis is that, by analyzing the components of both legal and programming languages, we can develop a richer dialogue on the sociological implications of translating law into algorithmic form. Furthermore, we can begin to consider what contextual information would be required to 'interpret' algorithmic contractual language.*

0. Background and Overture

Since the twelfth century, mathematical logicians allegedly used logical paradoxes to spot 'false' arguments in courts of law.[1] It was not, however, until the seventeenth century when Gottfried Leibniz proposed a mental alphabet[2] that thoughts could be represented as combinations of symbols and reasoning could be performed using statistical analysis. From Leibniz, George Boole's infamous treatise, *The Laws of Thought*, argued that algebra was a symbolic language capable of expression and construction of arguments.[3] By the end of the twentieth century, mathematical equations were conceivably dialogic–a form of discourse.

Boole's system successfully reduced complex thought into a solution of equations. Nevertheless, the most fundamental contribution of Boole's work was the isolation of notation from meaning.[4] That is, 'complexities' of the world filtered into the background as pure abstraction was brought into focus. Eventually, Boole's work would form the basis of the modern-day algorithm and expression in formal language.

ASCII, the acronym for the American Standard Code for Information Interchange, is an exemplary case. Computers are only capable of understanding numbers. In order for a computer to interpret natural language, characters must be translated from numbers. Using a binary numeral system, ASCII assigns a numerical value to a letter. In brief, by performing a mathematical calculation, ASCII translates a binary code of 0s and 1s into a letter. Early conceptual computing devices, such as the Turing machine, were born into existence as a direct product of Boolean algebra.

Christopher Markou and Simon Deakin point to the breakthroughs in natural language processing (NLP) as specifically contributing to the emergence of 'Legal Technology (Legal Tech).'[5] Markou and Deakin cite Noam Chomsky as inspiring early researchers of AI to design "hard-coded rules for capturing human knowledge."[6] Chomsky's work eventually contributed to advances in machine translation and language mapping. Known as expert systems, NLP applications "relied upon symbolic rules and templates using various grammatical and ontological constructs."[7] These achievements were then further enabled by Deep Learning[8] models, able to abstract and build representations of human language.

With the rise of artificial legal intelligence, computable contracts – and more broadly, computable law – are making a powerful return. Contracts may be represented as computer data with terms made 'machine-readable' through a process of conversion: from descriptive natural language to consonant computer instruction. Conditions of agreements are not explained but listed as structured data records. However, despite the capacity to express contracts in an alternative computable form, no means exist for interpreting the code. Instead, interpretation is perceived as irrelevant. Should digital data inscription and processing be considered a form of legal writing? If so, would it change the character of law?

This paper, therefore, considers the conundrum: what is the significance of the medium in contract drafting? This paper seeks to unpack several formal programming languages used in computable contracts. In identifying the logic of these languages, this paper studies methods of legal writing. This hypothesis is that by analyzing the components of both legal and programming languages, a richer dialogue on the sociological implications of translating law into algorithmic form may be formed. Furthermore, we can consider what contextual information would be required to 'interpret' algorithmic contractual language.

The paper will unfold as follows. Part I will open with the current challenges and state of Legal Tech. Part II embarks on a brief investigation of programming languages, analyzing sample translations of contracts from natural language to computer code. Part III will gather early observations. Part IV will suggest implications for contract law and further considerations. Finally, the author will conclude with a few remarks and possible next steps.

1. As it Stands

Kingsley Martin spoke of the two greatest barriers to legal technology: (1) adjudication and (2) language.[9] He teased at the subtlety and nuances of human communication. Meaning, he notes, could be changed with even the slightest adjustments to context. But beyond context, simple negations, "polysemy, synonymy, hyponymy and hypernymy"[10] are all functions of natural language that are obstacles for machines. He argues then that the general trend towards the simplification of language is rendering written legal documents, naturally, more machine-readable.[11]

Currently, expert systems and machine learning technology utilized for the revision of contracts seek to reduce the risk of human error. Eventually, contract analysis could manage, record, and standardize provisions that are 'proven favorable,'[12] in effect, perfecting contractual boilerplate. Boilerplate contracts are often regarded as a trade-off between customization and portability. With broader standardization, the 'burden' of interpretation is lifted.[13] Contractual boilerplate, therefore, relies heavily on formalistic drafting, where in form presides over meaning. For computable contracts, the migration of mediums – from descriptive natural language to mathematical form – generates data that identifies and signals the specific version of contracts that should be used in future cases.

a. Market Environment

Edilex, a Canadian legal tech start-up, is automating contract drafting by offering both AI-driven applications and downloadable legal document templates. Edilex's mission statement is expressed as: "The simplification of legal transactions and democratizing access to legal services." Genie AI is another fascinating legal tech start-up that offers AI-powered contract drafting. Using machine learning, the software recommends clauses to help legal practitioners "draft contracts faster."[14] Moreover, the technology marketed is focused on legal language, and as a product that is "suitable for lawyers."[15]

Evidently, the target demographic for each of the start-ups is rather different. The former is focused on the democratization of legal services, while the latter on enhancing the legal profession. Yet, both start-ups thrive on the notion of formalization — that a 'perfect' form can be achieved. Integrating AI into contract drafting necessarily pushes the legal profession away from static mediums of writing. These include Microsoft Word (MS Word) and Adobe PDF, the original technological artifacts that evolved from pen and paper. In either case, the technology is not described as a replacement of lawyers.[16] The purpose of these inventions is merely assistive.

b. Shifting Climates

Interestingly, the legal community is beginning to explore the problems associated with the use of static platforms like MS Word. Juro, for example, is a legal tech start-up that promotes contract

management on a dynamic platform.[17] In a recent paper, Michael Jeffrey critically examined the use of MS Word as the dominant and default form for writing and editing legal documents. He considers the inefficiencies of manual updating, drafting, and reviewing. MS Word has been a prized product for legal drafting, Jeffrey notes. Although interpreted as a static platform, MS Word, in actuality, "can be controlled through code."[18] In fact, MS Word has embedded in its software a number of templates modelled specifically for drafting legal documents. These templates contain automatic text entry, macros, and special formatting.[19] More recently, the startup Clause Logic has developed an extension that enhances MS Word's existing platform by automating clause creation and document assembly.[20]

Nevertheless, for long and complicated legal documents, Jeffrey argues that an integrated development environment (IDE) could "facilitate the authoring, compiling, and debugging" of contracts.[21] For programmers, the use of IDE provides several key features amenable to legal drafting. He notes options for increased readability using color-coded syntax highlighting, automatic error detection, and predictive auto-complete features to provide suggestions while drafting. These features, he claims, could improve the drafting process by reducing the risk of human error and increasing efficiency.

Yet, the most interesting perspective he offers is his subtle equation of linguistic concepts as inherently mathematical.[22] Jeffrey applies programming concepts to specific elements of legal drafting. The syntax, he notes, is "designed for drafting and document generation" and that the process would be "quite natural."[23] The underlying assumption is that the platforms of MS Word and an IDE have the same functional purpose. The differences lie in the added features for real-time changes. This speaks to a greater assertion: programming languages serve the same uses as natural language. But, the shift from pen and paper to MS Word did not fundamentally change the use of natural language for legal drafting. The use of IDEs, on the other hand, alters not only the platform, but also the method of execution.

Ultimately, the aforementioned start-ups, Edilex or Genie AI, are only a few of the growing number of legal tech companies committed to the 'betterment' of contract drafting. These contracts will be described as more efficient, precise, or 'smarter.' There is, nonetheless, a dearth of literature on the use of formal programming languages for legal writing. Albeit, formal languages for contract drafting not only exist but have proliferated in the past few years. Their ancestors evolved from logic programming in the 1970s.

2. So, Can You Code it?

Since the 1950s, long before the days of logic programming, the academic literature has identified logic-based strategies to contract drafting. In "Symbolic Logic: A Razor-Edged Tool for Drafting and Interpreting Legal Documents," Layman E. Allen proposes the use of mathematical notation for the

expression of contracts. He argues that its application will improve clarity, precision, and efficiency of analysis. He introduces six elementary logical connectives: implication, conjunction, co-implication, exclusive disjunction, inclusive disjunction and negation.[24] The most interesting connectives are implication and co-implication. These logical connectives are associated with the representation of causal relations, expressed as "if X then Y" statements. Allen labels this form of expression as "systematically-pulverized"[25] and the process of transforming a statement to this form requires two primary actions: (1) dividing the statement into its constituent elements; (2) and rearranging its elements to approximate a 'systematically pulverized' form. Co-implication enhances the equation by including logical equivalencies. In sum, Allen teases at the age-old use of syllogisms in legal writing and provides a foundation for contemporary study examining the ways programming languages can apply logic to legal drafting.

Two of the most broadly used programming languages, Python and Prolog, use opposing methods of operation; the former is procedural, while the latter is declarative. Procedural programs often specify *how* the problem is to be solved. That is, with procedural programs, clear instructions proscribe the behavior of the program. Akin to a cooking recipe, all terms of a procedural program are defined explicitly, and all rules laid out. Should a program, such as Python, find that it cannot proceed with a task, this is typically because the program is unable to recognize the syntax. Equally, Python is incredibly sensitive to changes in the code; even a misplaced comma or indent in the spacing could affect the overall outcome of the specified task. Procedural programs often include functions: self-contained modules of code capable of being manipulated and reused for innumerable tasks. Perhaps its most powerful operation, conditional statements allow Python to examine and decide actions on the basis of conditions. Moreover, Python simplifies work by being able to loop through the same tasks in a given list. Rather manually repeating a given task, Python is able to do so in a matter of seconds.

On the other hand, declarative programs specify *what* the problem is and ask the system, instead, to solve it. Declarative languages are founded on either relationships (1) between objects or (2) between objects and their properties. These relationships may be defined implicitly through rules or explicitly through facts. Facts describe relationships, while rules qualify them. The purpose of Prolog, therefore, is to form a fixed dataset that would derive answers to future queries about a relationship or set of relationships based on the inputted information. In contrast, the purpose of Python is to complete a particular task. While it can certainly account for prospective changes to the data, every step is explicitly expressed.[26]

Advancing forward several decades, Python and Prolog have become inspirations for a new era of programming languages currently being prototyped to draft computable contracts. This paper will explore a number of these formal languages, including Ergo, Sophia, Solidity, Lexon, Blawx, and OpenLaw. While they certainly do not account for all the languages that are being workshopped, they

are among the most broadly discussed in the legal tech sphere. Each language is built from different models. Ergo is a programming language modelled on execution logic for legal writing. It belongs to the suite of resources offered by The Accord Project.[27] Sophia and Solidity were both influenced by the Python syntax, but were created specifically for smart contract implementation.[28] Lexon and Blawx, on the other hand, are non-coding options with the former developed on declarative logic and the latter derived from linguistic modelling.[29]

Finally, OpenLaw is more complicated to characterize. OpenLaw stems from neither Python nor Prolog. OpenLaw instead runs on Javascript[30] and uses a markup language to "transform natural language agreements into machine-readable objects with relevant variables and logic defined within a given document."[31] These documents are then compiled together to act as contracts. Interestingly, the markup language allows for legal agreements to be enabled on the blockchain, but with natural language qualifiers.[32]

Prior to delving into the mechanics, there are a few disclaimers. First, the author does not distinguish between machine-readable and machine-executable contracts. Rather than bifurcating the two architectural forms, the analysis focuses broadly on Smart Legal Contracts.[33] Next, to understand how formal languages may be used to draft contracts, the author refers to extracts of legal documents translated from natural language to code. These translations are originals of each programming language, unedited and taken directly from their technical documentation. They were included as examples of how contracts may be drafted in the select language. The translations are, therefore, presumed to be manually written by each language's programmers and thereby exhibit respective design choices. As well, the formal languages analyzed are understandably evolving in their capacities. Consequently, this study is current only to the time of this analysis. Finally, as there are, to date, no quantitative metrics to evaluate the existing pitfalls of contracts drafted in natural language, this study is limited to qualitative perspectives on formal languages as a medium for legal drafting.

a. Ergo

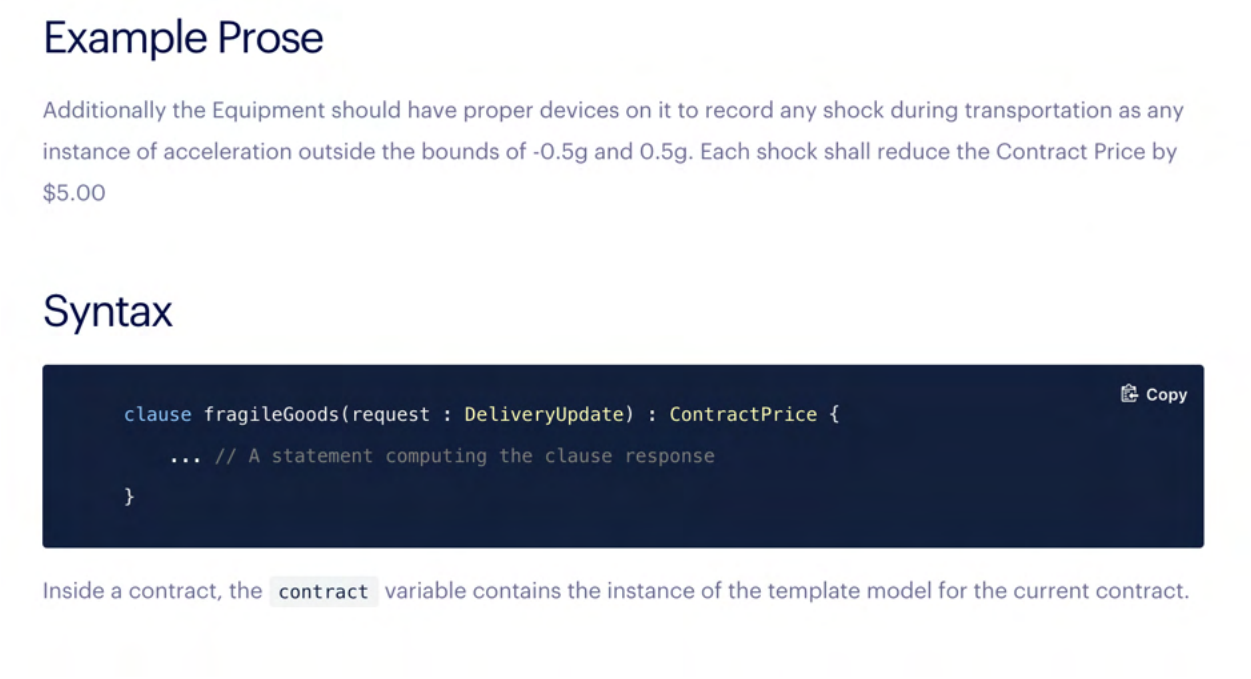
To begin, Ergo follows a more traditional form of procedural programming and is largely function-based. This means that its language is predicated on the performance of the contract. Contractual clauses are typed but are constrained to a pragmatic, functional core. That is, the design of the language is such that descriptions qualifying the terms of the contract are superfluous. Descriptive clauses may only be programmed as comments, or annotated notes, and are non-operational.[34] Though computer engineers may be able to view them, the comments neither contribute nor enhance the actual execution of the contract. More importantly, Ergo is intentionally limited in its expressiveness.[35] Ergo is best used for contracts with 'unidirectional relationships,' such as straightforward sales

transactions or procurement contracts. However, contracts that specify details of performance are not currently possible.

Consider the following contractual clause translated from descriptive natural language to Ergo. The original provision, in prose, states:

> *Additionally, the Equipment should have proper devices on it to record any shock during transportation as any instance of acceleration outside the bounds of -0.5g and 0.5g. Each shock shall reduce the Contract Price by $5.00.*

The clause, in code, reads:



The translation is rather striking. At first glance, several omissions warrant concern. First, any mention of recording devices that determine changes in the weight of the Equipment is excluded from the code. Moreover, fluctuations in the Contract Price are excluded. Instead, the syntax is interpreted as a "Delivery Update." A closer reading of the natural language clause would immediately raise attention about the interpretation of performance in this particular procurement contract. The natural language clause states that a device should be installed to record *any* shock during transportation. In the Ergo language, this requirement was merely coded as an update without further qualifying statements, specifications, or clarifications. Offering an update upon delivery, versus the explicit use of a recording device, are distinct contractual guarantees. Arguably, merely providing an update would not have sufficiently discharged the party from its contractual obligation. From a legal perspective, the

exclusion of the qualifier – the recording device – has effectively changed the meaning of the clause. This suggests that Ergo interprets contracts within a narrow scope of functionality.

b. Sophia and Solidity

Sophia is a language customized for smart contracts[36] on the Aeternity Blockchain.[37] The main unit of the code structure focuses on the performance of the contract. As the code is limited to contract implementation, the syntax of the language is again purely functional.[38] Prior to delving into the translation, it may be important to define a few key terms. First, the "state" is understood as the objects of the contract. The "entrypoints" are the actions to be taken pursuant to the contract. If the contract stipulates modifying the state, entrypoints are annotated with the "stateful" keyword.[39] The inclusion of stateful is the dividing line between transactions and calls in smart contracts. The former requires modification; the latter does not. For example, a procurement contract requests notice upon delivery. As the notice does not require modifying the state, a simple entrypoint would suffice. The actual delivery, on the other hand, would require the stateful qualifier. Structurally, Sophia applies a Python-style syntactic structure with minor changes to the notation.

Consider the sample purchase agreement written in Sophia:

2. **BuyerContract** implements following functions:

- `deposit_to_seller_contract(price : int, key : address)` - the passing arguments are the price of the item and seller's contract address. There are functions implemented in Transport and Seller contracts, that we can call from Buyer contract, to check status of the item:
- `check_courier_status(transport_contract_address : address)` - will return the status of the order.
- `check_courier_location(transport_contract_address : address)` - will return the current location of the order.
- `check_courier_timestamp(transport_contract_address : address)` - will return the timestamp of last update.

```
contract SellerInterface =
  entrypoint received_item : () => bool
  entrypoint seller_contract_balance : () => int
  entrypoint check_item_status : () => string

contract TransportInterface =
  entrypoint check_courier_status : () => string
  entrypoint check_courier_location : () => string
  entrypoint check_courier_timestamp : () => int

contract Buyer =
  stateful entrypoint deposit_to_seller_contract(price : int, key : address) : () =
    Chain.spend(key, price)

  entrypoint received_item(remote : SellerInterface) : bool =
    remote.received_item()

  entrypoint seller_contract_balance(remote : SellerInterface) : int =
    remote.seller_contract_balance()

  entrypoint check_item_status(remote : SellerInterface) : string =
    remote.check_item_status()

  entrypoint check_courier_status(remote : TransportInterface) : string =
    remote.check_courier_status()

  entrypoint check_courier_location(remote : TransportInterface) : string =
    remote.check_courier_location()

  entrypoint check_courier_timestamp(remote : TransportInterface) : int =
    remote.check_courier_timestamp()
```

The purchase agreement is remarkably direct. In the above contract, the terms of the agreement have been reduced to a mere 19 lines of code. The remainder of the agreement serves to provide notifications regarding delivery and updates on courier status. Notably, the contracts apply existing functions that have been pre-programmed, thereby rendering performance automatic. Most purchase agreements are templates easily found with a quick search on the Internet. The programmed functions within Sophia are akin to the use of these agreement templates. Placeholders within contract templates are replaced by the dynamic variables of Sophia. Clauses that indicate qualitative expectations of the product for purchase (i.e. the condition of the good) remain as annotations outside of the contract.

Similarly, Solidity is another language used for the implementation of smart contracts. Solidity draws influence from Python and is an object-oriented language.[40] As opposed to the Aeternity Blockchain, Solidity is a language customized for the Ethereum Blockchain.[41] In lieu of states and entrypoints, Solidity uses the syntax of variables and functions, creating a 'Python-ese' language. For example: rather than using 'stateful' as the performative, Solidity uses 'modifier.' Simply put, the uses of Solidity parallel those of Sophia. Solidity, however, offers more options in qualifying contracting

parties. Structs and Enums are syntactical operations that better classify the types of users engaged in the contract.[42]

Consider the sample purchase agreement written in Solidity.

```
contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }
    // The state variable has a default value of the first member, `State.created`
    State public state;

    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
        require(
            msg.sender == buyer,
            "Only buyer can call this."
        );
        _;
    }

    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    modifier inState(State _state) {
        require(
            state == _state,
            "Invalid state."
        );
        _;
    }
```

```
    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();
    event SellerRefunded();

    // Ensure that `msg.value` is an even number.
    // Division will truncate if it is an odd number.
    // Check via multiplication that it wasn't an odd number.
    constructor() public payable {
        seller = msg.sender;
        value = msg.value / 2;
        require((2 * value) == msg.value, "Value has to be even.");
    }

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        public
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        // We use transfer here directly. It is
        // reentrancy-safe, because it is the
        // last call in this function and we
        // already changed the state.
        seller.transfer(address(this).balance);
    }

    /// Confirm the purchase as buyer.
    /// Transaction has to include `2 * value` ether.
    /// The ether will be locked until confirmReceived
    /// is called.
    function confirmPurchase()
        public
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }
```

```
    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Release;

        buyer.transfer(value);
    }

    /// This function refunds the seller, i.e.
    /// pays back the locked funds of the seller.
    function refundSeller()
        public
        onlySeller
        inState(State.Release)
    {
        emit SellerRefunded();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Inactive;

        seller.transfer(3 * value);
    }
}
```

Again, the drafting of the purchase agreement is highly procedural and direct. There are no terms and conditions qualifying the object for purchase. Instead, there are only 'code-ified' limitations, measures to verify the identities of the contracting parties and confirm the purchase. All operations within the code serve only to facilitate performance of the contract.

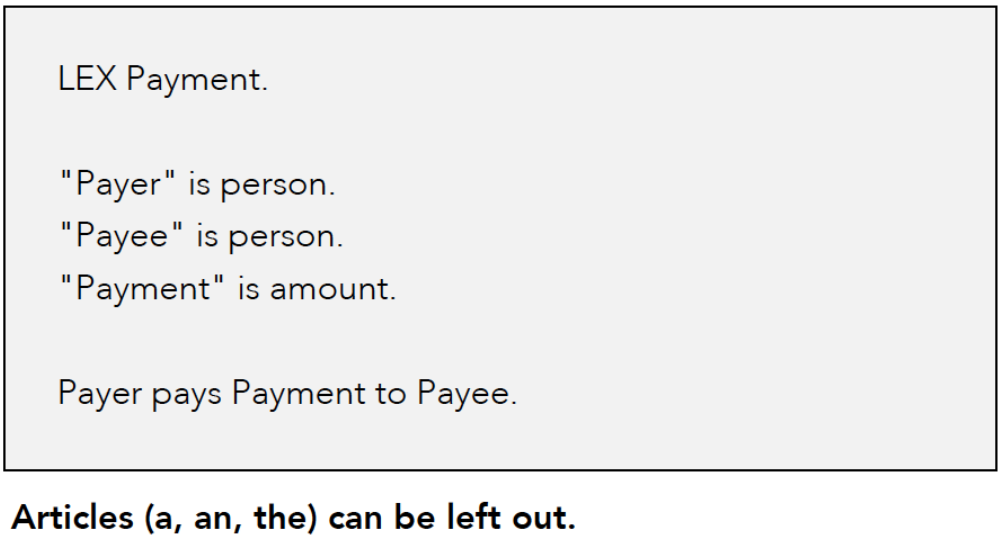
In both Sophia and Solidity, the agreements are not translated from natural language to code. Rather, there are merely examples of contracts drafted in the formal language. That is, natural language contracts are reimagined in code at their creation. The translation process is internalized and configured to the parameters of the programming language. The purchase agreements 'speak the language'[43] of smart contracts. Certainly, the future potential of smart contracts extends beyond purchase agreements. However, the current implementations of the technology are narrow and typically do not require qualitative accounts.[44] The issue perhaps is the conflation of other use cases with contracts in particular. For programmers well-versed in Solidity or Sophia, the identifiable problem is determining whether the purchased item has arrived at the buyer's address. *How* the good

arrived is never of material concern. However, in eliminating the question of "how", there is a risk of reducing contracts to a Boolean binary.

c. Lexon

Lexon is a peculiar alternative to the programming languages previously studied. Unlike others, Lexon bases its foundation on linguistic structures and is designed to reason in natural language. Lexon reduces vocabulary and grammar to rule sets. The base vocabulary of the language consists of definable 'names' which are used to designate objects and clauses. Just as one would draft sentences in natural language with a subject and predicate, Lexon operates in a similar fashion. There is, however, an important difference: articles are considered superfluous 'filler' words.

Below is a sample contract drafted in Lexon:



For an agreement at this level of simplicity, articles may not seem necessary to clarify the meaning of contractual terms. Nevertheless, party obligations do occasionally hinge on articles, potentially affecting the performance of a contract. It is not inconceivable that specifying a particular object as opposed to a general one could be materially relevant, especially in certain procurement and sales contracts. Lexon argues that the primary role of articles is to improve text readability. Yet, Lexon concedes that articles can "fundamentally change the meaning of a contract" and that this may be an area ripe for abuse.[45]

Further complicating the narrative, Lexon is not concerned about semantics altogether. The startup's creator, Henning Diedrich, acknowledges the inherent ambiguity of natural language that renders interpretation challenging, but argues that the Lexon language should not be used to clarify nor complete contracts. Instead, the role of Lexon is to bridge the gap between formal programming and

natural languages. Like other formal languages, Lexon cannot understand the 'meaning' of its terms. Its structural design only accounts for functionality. Lexon uses Context Free Grammars (CFG). First theorized by Chomsky, CFG does not depend on context; its rules operate independently of the objects in question. Chomsky had originally developed CFG in an effort to formalize natural language rule structures. While this was largely unsuccessful in linguistics, it has since been popularized in computer science. Consequently, Lexon applies the model to create a programming language that is both expressible in natural language and readable by machines.

Diedrich contends that this meaning could not be attained. Meaning is regarded as something that is unextractable yet is certainly conceivable and describable.[46] The Lexon language is structured in a manner reflective of these underlying assumptions. That is, rather than dwelling on the interpretation of the specific word or phrase in natural language, Lexon limits meaning to function. Diedrich states, "the actual functionality of the contract is the better description of… the list of the actual rights and obligations of that person without relying on the original meaning of the word."[47] By framing functionality as a proxy for party obligations, Lexon inadvertently reframes the basis of contract theory from party autonomy to contract performance.

d. Blawx

The Blawx language is structured with declarative logic. Perhaps the most interesting element of this language is its user interface. The code visually appears as puzzle pieces, or blocks, with certain notches that allow certain pieces to fit together only when certain conditions are met. Blawx was inspired by the program "Scratch," created at MIT as an educational assistant for children learning how to code. As the 'blocks' literally connect with one another, they visually capture the relationships between objects and their properties. Moreover, this approach has the additional benefit of limiting the potential for error, as the 'pieces' of code would not physically fit together should the contract be drafted incorrectly.

Much like Prolog, Blawx operates on sets of facts and rules. Facts represent objects, or things, known to be true *in the code*. Rules are coded statements composed of both conditions and conclusions. Both elements are required in order for a rule to exist. Unlike other programming languages, Blawx functions according to the premise of declarative rules such that "conclusions are true if conditions are true." This may seem no different than traditional 'if…, then…' statements. This is surprisingly false. In programming, the 'if conditions, then conclusions' framework operates temporally. For machines, this means that conditions only apply to the specific task at hand and do not apply globally to the program.[48] In the case of Blawx, rules are encoded in a declarative manner to help form the particular program's 'universe of knowledge.' Once the 'universe' of facts and rules have been established, the program can accurately provide answers to queries. Queries are fact-based and binary.

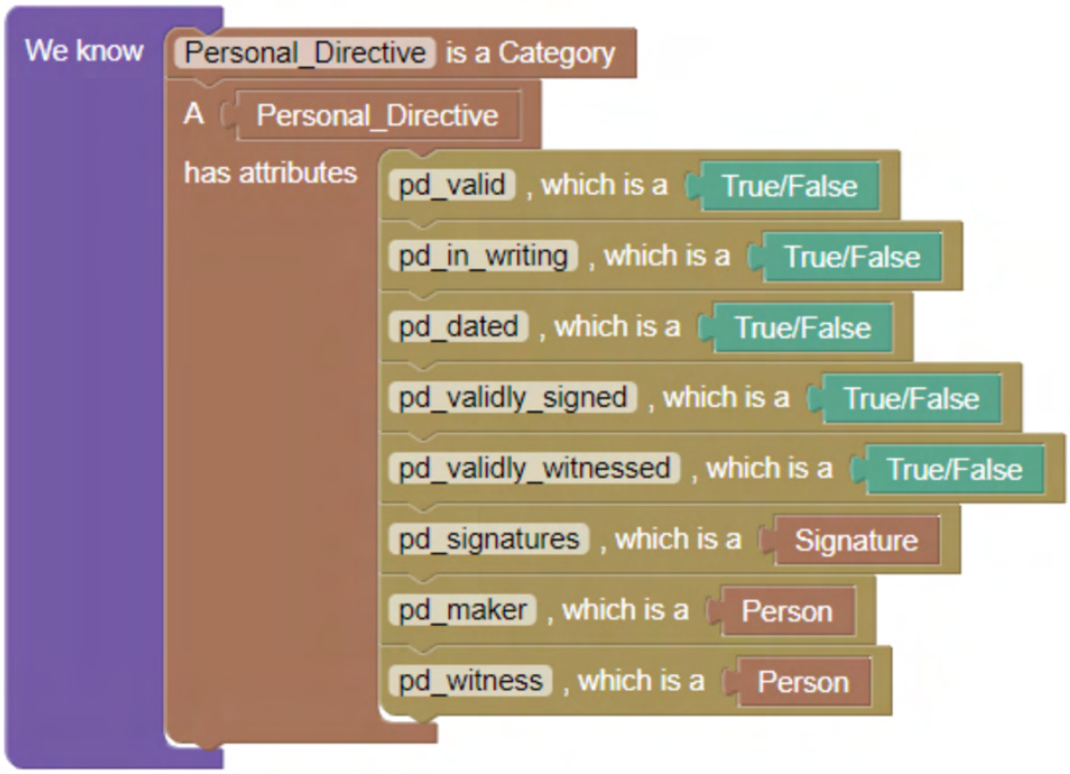
Blawx aims to transform legal documents into queryable databases. In practice, this would suggest that contracts may be encoded using the aforementioned logic of the program. Ultimately, the goal is for parties to be able to reason about contractual agreements through simple binary queries of the application. The encoding of facts and rules allows parties to move from legal reasoning to legal information extraction. Interpretation, then, is no longer required since the solutions are presumed to be directly retrievable.

Consider the sample translation of a legislative act from descriptive natural language to Blawx. The article states:

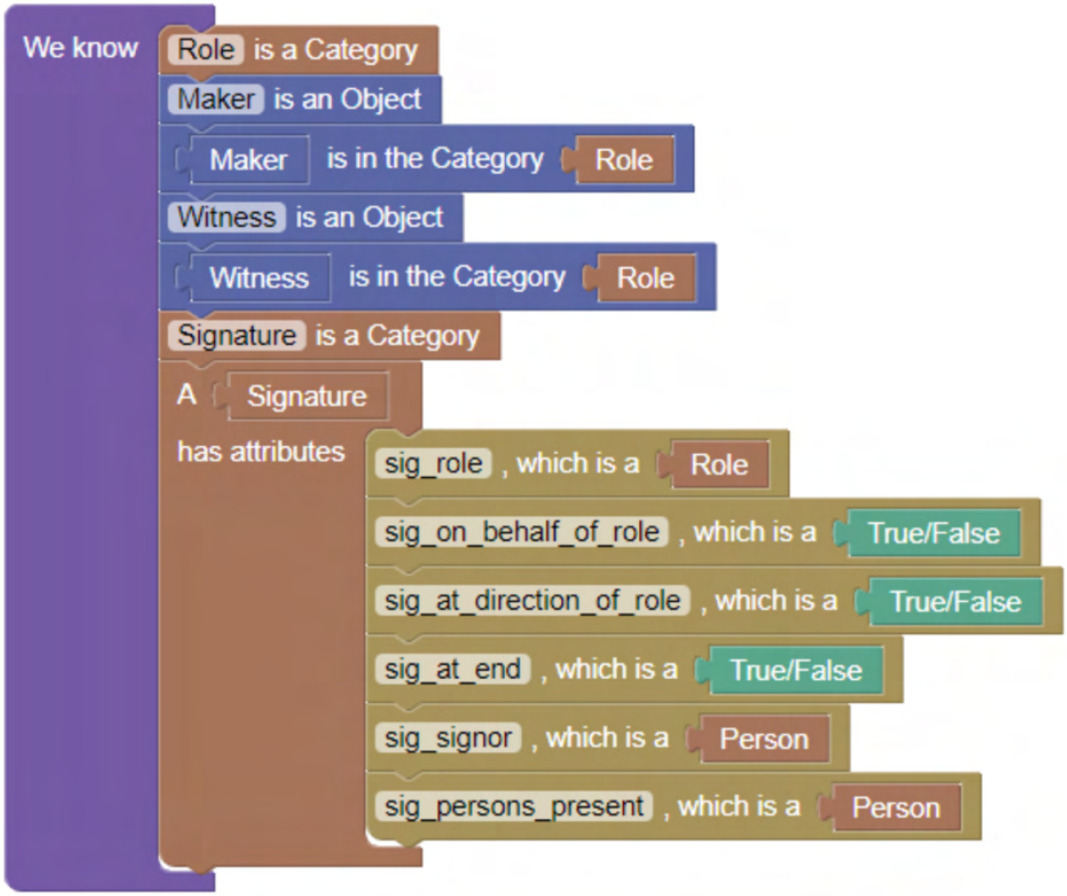
> ***5(1)**: A personal directive must*
>
> > *(a) Be in writing,*
> >
> > *(b) Be dated*
> >
> > *(c) Be signed at the end*
> >
> > > *i. by the maker in the presence of a witness, or*
> > >
> > > *ii. If the maker is physically unable to sign the directive, by another person on behalf of the maker, at the maker's direction and in the presence of both the maker and a witness,*
> > >
> > > *and*
> >
> > *(d) Be signed by the witness referred to in clause (c) in the presence of the maker.*

The provision, in Blawx, reads:

And this is the ontology for Personal Directives.



Here is what our new ontological elements look like. First, the Roles, and Signatures.



Without familiarity with the technology, this translation is a difficult read. First, the 'block' appearance of the language may be troubling for those who are not tactile learners. The programming language forces the reader to focus on the conceptual components of the rules as opposed to the clause. The logic of the program necessitates a substantive breakdown of the legislation into its ontological elements. Simply put, it reduces the law only to its relevant actors and their obligations. In this case, these elements are (1) the roles (actors); and (2) the signatures (obligations).

More importantly, the process of converting natural language to Blawx creates significant challenges with interpretation.[49] Coding the legislation required reframing the meaning of "personal directive"[50] into a binary, either as an object or an action. Fundamentally, this is a reconfiguration of the law into a function. Rather than "what are the requirements of a personal directive," the legal question becomes

"what actions must be taken in order for the personal directive to have legal effect?" These questions, asked *de facto*, incidentally have the same meaning. However, the subtle difference crucially points to an implicit recognition of the legal effect of the document in natural language. Notably, a personal directive could only exist if the stated requirements are met. Otherwise, it would simply be a piece of paper. This prerequisite was raised as a note on the translation. Blawx introduced the concept of "validity" as a new condition for personal directives[51] because there was no form of classification for a document that was not a personal directive. These interpretational challenges suggest that in the context of computable contracts, the Blawx language – like Ergo – would perhaps work best for contracts having explicit objectives and unidirectional relationships.

e. OpenLaw

The final programming language reviewed in this paper perhaps poses the starkest contrast to the other formal languages studied.[52] For OpenLaw, the aim is not to translate natural language agreements in their entirety. Instead, the language acts as a hybrid, an integration of machine-readable code with clauses drafted in natural language.[53] The intention is to generate variables and logic to be imported and incorporated into future contracts of a specified type. For example, a non-disclosure agreement (NDA) typically includes the names of contracting parties and transforms them as dynamic variables. If the variable requires further description, additional string[54] text could be used to qualify the term. Boolean logic is a feature of OpenLaw's programming language. The "conditionals" function embeds logic into a legal agreement, reconstructing contractual terms into binary questions. Clauses are interpreted as "embedded template[s]."[55] The goal is to reduce drafting work by storing boilerplate clauses as data that may be added to contracts.

Below is an excerpt of an advisor agreement written in OpenLaw:

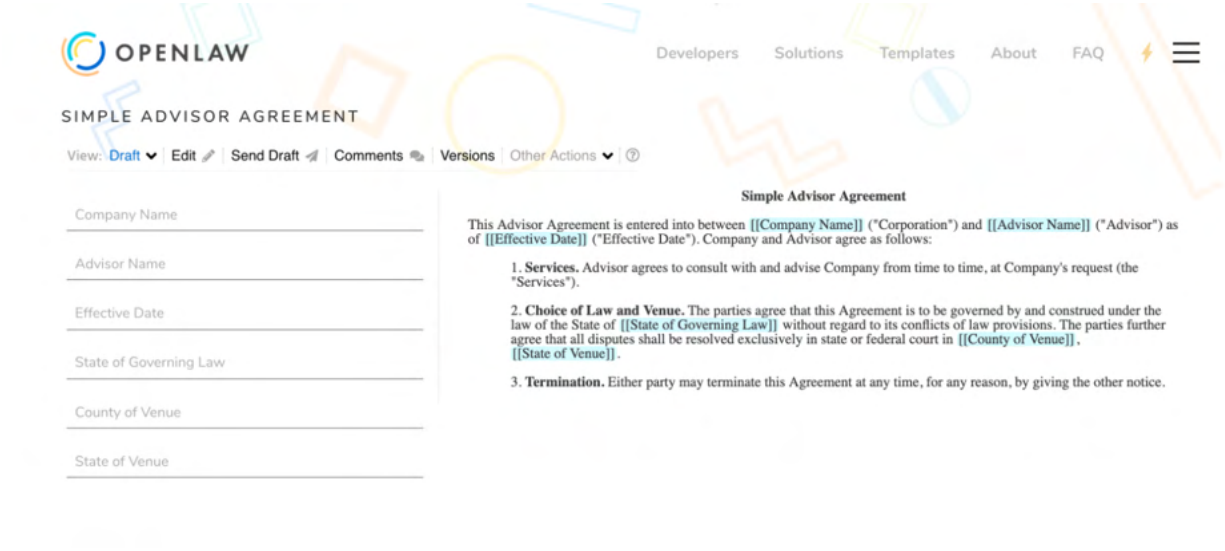
```
\centered **Simple Advisor Agreement**

This Advisor Agreement is entered into between [[Company Name]] ("Corporation")
and [[Advisor Name]] ("Advisor") as of [[Effective Date: Date]] ("Effective Date
Company and Advisor agree as follows:

^**Services.** Advisor agrees to consult with and advise Company from time to ti
at Company's request (the "Services").

^[[Choice of Law Insert: Clause("Choice of Law and Venue Clause")]]

^**Termination.** Either party may terminate this Agreement at any time, for any
reason, by giving the other notice.
```



The excerpt of the agreement is presented in two forms: (1) in code and (2) in OpenLaw's drafting editor. In either arrangement, the natural and formal language are woven together seamlessly. At first glance, it may be difficult to determine whether the natural language is ever translated to OpenLaw. The enduring presence of the natural language in OpenLaw and the structural consistency of the contract, suggests the integrity of the agreement remains intact. Yet, the incorporation of code with natural language offers a dynamic interpretation of legal agreements. It mirrors the notion that certain elements of contracts are reproducible and calculable, while others require human intervention. The drafting process, however, is left rather unchanged. OpenLaw's hybrid approach simplifies the process of contract drafting, identifying portions of the agreement that are quantifiable.

The question becomes: what are the risks of simplification? Is this 'hybridization' an additional form of translation?

In examining these various programming languages, current legal technologies are observably limited. Namely, contracts drafted in these languages govern only simple transactions. Nonetheless, they expose conflicting interpretations of contract theory. More specifically, a commonality across all formal languages is the treatment of contracts as mere predicates to performance. Consequently, all of the languages studied were largely function-based. The principle of party autonomy, expressed often as details in contract terms, is secondary to the actual execution of the transaction. Rather than focusing on the specific terms the parties have agreed upon and *how* they must fulfill their obligations, the legal question becomes solely dependent on *whether* the obligation has been completed. Negotiated contracts represent a 'meeting of the minds.' Using programming languages, technologists run the risk of reconfiguring basic doctrines of contracts, conflating the principles of "consideration as offer" and "acceptance as obligation." The exception, of course, is OpenLaw. Its hybrid approach raises provocative questions on whether the use of embedded code in legal drafting can aid practitioners while retaining the requisite complexities of natural language.

3. Observations and Analysis

With the increasing normalization of smart contracts, computer code could foreseeably become a language in which contracts and other legal instruments are drafted. But should programming languages be recognized as a form of legal language? The following section will analyze the observations taken from the study against existing literature. As previously discussed, functional operations are paramount to computable contracts. Formal programming languages help demonstrate the indeterminacy of natural language, suggesting that a migration away from semantics to syntax could resolve challenges relevant to contract interpretation.

This was the impetus behind another innovative start-up, – cleverly named – Legalese. L4, their marketed programming language, is a domain-specific language (DSL) designed to "capture the particularities of law, its semantics, deontics, and logic."[56] Unlike other formal languages, their 'logic' draws influence from Prolog, but has been developed for the sole intention of expressing laws.[57] The purpose of L4 extends beyond the general application of programming languages to legal language. Rather, L4 produces formally verified 'smart' contracts that could equally be transformed into PDFs written in natural language. The idea is that the 'legalese' of contractual terms is a seamless translation between code and natural language. As Legalese co-founder, Alexis Chun, states, "legal as a utility, not a consultation."[58] This may well have served as the mission statements of the other programming languages studied in this review.

The idea of L4 sprung from a programmer seeking to 'decipher' an investment contract written in 'legalese.'[59]

1.2.1 If the investment for the purpose of the Series B Funding is valued at not more than $32.5 Million, then the investors in the Note shall be entitled to convert the Note into Shares at a fixed valuation of $27.5 million.

1.2.2 If the investment for the purpose of the Series B Funding is valued at less than $40 million but not below $32.5 million, investors in the Convertible Note will be entitled to convert the Note into Shares at a 15% discount over the valuation of the Series B Funding (for instance, if the series B Funding is at a valuation of $35 million, then the investors in the Note shall be entitled to convert at a valuation of 35M less 15% discount);

1.2.3 If the investment for the purpose of the Series B Funding is valued at not less than $40 million but less than $47.06 million, investors in the Convertible Note will be entitled to convert the Note into Shares at a 15% discount over the valuation of the Series B Funding (for instance, if the series B Funding is at a valuation of $47.06 million, then the investors in the Note shall be entitled to convert at a pre-money valuation of 40M i.e. $47.06 million less 15% discount);

1.2.4 If the investment for the purpose of the Series B Funding is valued at not less than $47.06 million but less than $80 million, investors in the Convertible Note will be entitled to convert the Note into Shares at a fixed pre-money valuation of $40 million;

1.2.5 If the investment for the purpose of the Series B Funding is valued at not less than $80M but less than $100M, investors in the Convertible Note will be entitled to convert the Note into Shares at a fixed pre-money valuation of $45M; and

1.2.6 If the investment for the purpose of the Series B Funding is valued at not less than $100 million, investors in the Convertible Note will be entitled to convert the Note into Shares at a fixed pre-money valuation of $50 million.

The programmer then drafted a translation of the investment contract. It read as follows:

```
if( seriesB < 32.5 ) { conversion = 27.5 }
else if( seriesB < 40 ) { conversion = seriesB * 0.85 }
else if( seriesB < 47.06 ) { conversion = seriesB * 0.85 } // then why the above line?
else if( seriesB < 80 ) { conversion = 40 }
else if(seriesB < 100 ) { conversion = 45 }
else { conversion = 50 }
```

Evidently, the translation consists of a specific excerpt of the contract — a markedly quantifiable section. Nevertheless, this translation highlights the monotony of certain contractual clauses. Every provision follows a similar phrasal structure. In effect, the programmer is pointing to the innate formalism that exists in select legal language. Though drafted in natural language, the repetition of noun phrases in the aforementioned excerpt divorces contextual knowledge from interpretation. The result? The ability to distil and transform natural language into clear computable form.

a. Early Inspirations

In another fascinating analysis, Layman E. Allen reflects on ambiguity in legal writing stemming from syntactic uncertainties. Allen considers alternative constructions to manage logical issues found 'between sentences' of legal documents.[60] He first engages in an exercise to deconstruct an American patent statute, and immediately notices a difficulty with the word 'unless.' He asks whether the

inclusion of 'unless' asserts a unidirectional condition or a bidirectional condition.[61] That is, does the clause mean (a) if not *x* then *y*, or (b) if not *x* then *y* and if *x* then not *y*?

Though nuanced, Allen exposes an ambiguity that obfuscates the legal force of the statute. Interpreting 'unless' bidirectionally requires us to consider what "not *y*" could mean. In this particular case, "not *y*" could affect whether exceptions might exist in determining eligibility for a patent. He later acknowledges that both sections immediately preceding and following the statute provide sufficient context. Nevertheless, he maintains that language must have a clear structure. Though conceding that semantic uncertainties are often deliberate, structural uncertainties are often inadvertent.[62] Drawing inspiration from computer science, Allen argues that drafting legal provisions requires replacing the use of imprecise terms (i.e. 'unless') and, instead, constructing sentences that use "lowest common denominators of structural discourse."[63] These common denominators include 'and,' 'or,' 'not,' 'if…then,' and 'if and only if…then.' The similarities with formal language are stark, begging the question: how does reducing language to 'lowest common denominators' affect the complexity and richness of legal language?

In "Self-Driving Contracts," Casey and Niblett consider the inconsistencies in contract theory owed to the ambiguity of natural language. They argue that, currently, natural language as a medium of legal expression allows contracts to be both intentionally and unintentionally incomplete.[64] Intentional incompleteness is interesting because it implies that general language circumvents the *ex ante* costs of decision-making and creates a space for changes in conditions. However, this often leads to issues with enforceability, such as disputes about the definitions of "reasonable" and "material."[65] A 'self-driving contract' would use machine learning algorithms and expert systems to remove these questions of enforceability.

Much like 'self-driving' contracts, the aforementioned programming languages help automate the processes of contract creation and interpretation. As observed in the study, interpretation is internalized by the technical bounds of the programming language. That is, contractual clauses are constructed to reason functionally.

b. Ergo[66]

For Ergo, the translation of the natural language required the removal of certain specifications such as the recording device. Instead, the contractual obligation was redefined as a delivery notification. Ergo reconceptualized the clause by broadening the scope of the obligation, reclassifying specifications from conditions to warranties. Effectively, Ergo fixes the meaning of contractual terms and renders interpretation irrelevant. In ordinary negotiations of a contract drafted in natural language, disputes may arise over mutual assent and performance, perhaps related to agreement on finer details of the

contract.[67] With programming languages like Ergo, mutual assent is automatic and indisputable. Semantic ambiguities no longer exist. Perhaps illustrative of the design, Ergo contracts only address "consideration, mutuality of obligation, competency and capacity."[68] Offer and acceptance are assumed. Nevertheless, the reconceptualization of consideration remains problematic. Contractual negotiation would be limited to operations, with little concern given to parties' preferences. This could potentially invoke a behavioral change between organizations as contracts would become primarily functional in nature. This could also conceivably lead to a simplification of contracts and a convergence towards contractual boilerplate.

c. Lexon

Lexon's language poses a similar puzzle. Readable in natural language, Lexon's verbs are coded such that they coincide with the performance of the transaction. Diedrich's formulation of meaning parallels Ludwig Wittgenstein's writings. Wittgenstein argues that language, as used presently, extends beyond names and "dry dictionary entries with their definitions."[69] Actions derived from words are effectively married to their meanings. It is conceivable then that a language could be formalized with no more than a list of orders and classifications. It follows that abiding by the rules of association is to accept the inherent authority of its practice. Meaning is found in the performance of the word, not in the understanding of it.

Diedrich claims that Lexon neither translates nor transforms thought.[70] Instead, Lexon preserves the natural language construction of 'meaning,' by placing constraints on its rules. That is, Lexon uses a subset of natural language grammar as the programming language of the legal contract.[71] This approach is known as "controlled natural language." Rather than processing *all* of natural language, a machine needs only to process an assigned vocabulary and grammar. The assigned set becomes the operatives of the language game. Additionally, Lexon embraces the legacy of Chomskyan formal semantics, whereby the syntactic structure is both a projection and a vessel of its function. Interpretation is again performed internally by "mapping…symbols to a reference structure."[72]

d. Blawx

Blawx requires that the actions of contractual parties be defined in advance. Again, its code internalizes interpretation as a preliminary step. Using declarative logic, Blawx first sets the parameters of its dataset. On several occasions,[73] the code required defining new categories and forming different classifications in order to be amenable to translation. This involved making explicit the relationship between legal objects and their properties. Interestingly, legal questions, particularly those seemingly accommodating to mathematical configuration, presented challenges in the Blawx language. For example, validating a personal directive could easily be structured as a binary question. Still, it was necessary to create a definition of an object that did not fulfill the requirements of a

personal directive. This subsequently provoked a deeper question on the implicit recognition of legal documents.

Simply put, Blawx exposed the tacit force of law. Reflecting on H.L.A. Hart, the underlying assumption of "power-conferring rules [...] exist not in virtue of some further law-making act, but in virtue of a fundamental rule of recognition implicit in the practice of law-applying officials."[74] Similarly, J.L. Austin contemplated the performative effect of 'utterances.' Austin uses the act of marriage to demonstrate how uttering a few certain words effectuates their meaning.[75] Austin suggests that legal and moral obligations are relative to public specification; utterances necessarily correspond with particular procedures situated within social contexts. Their mis-performance leads to a nullification or voidance of the act.[76]

In the case of Blawx, the meaninglessness and inability to articulate the 'inverse' of a legal document (i.e. a missing witness signature on an otherwise would-be personal directive) points to the implicit dimension of the law.[77] The dividing line between a document having or not having legal force speaks to the inherent authority of legal rules. Just as marriage could only be recognized within a specific circumstance, it was necessary for Blawx to acknowledge the deeper context of the translated legislation by considering "how is legal recognition being defined?" Blawx then applied a purposive interpretation, classifying legal recognition as validity. While the translation is rather sound – and validity is often a proxy for determining legal effect – the questions asked are distinct. The inquiries "is it legal" to "is it valid" are necessarily distinguishable in contract law. A contract may be valid but legally unenforceable. Therefore, interpreting legal force as validity subverts existing contract theory and, again, narrows interpretation to seemingly functional equivalents. Casey and Niblett are correct in noting that there will be an attempt to "pigeonhole [computable contracts] into existing frameworks of thought."[78] For Blawx, implementing the language into practice would likely require changes to existing contract doctrines.

The challenge of using programming languages centers on interpretation. Drafting contracts in formal programming languages highlights the ambiguity of the original source. The task of translating contracts from descriptive natural language to code reveals underlying assumptions of legal authority and reevaluates party autonomy in contract theory. In nearly all cases studied in this paper, interpretation was performed *ex ante*; the contract's legal effect was established in direct parallel to performance.

4. Implications and Further Considerations

As mentioned, formal programming languages have the impact of unifying legal concepts such as mutual assent with performance, effectively reinvigorating arguments associated with contractual boilerplate.[79] Conversely, formal programming languages also help demonstrate how to break down

legal concepts into atomic, executable tasks and conceptual components in such a way that provides richer insight into the vast landscape of contracts. In either case, contract doctrines are reframed by the new mediums in which they are able to be represented.

How, exactly, these impacts will be felt remains to be seen. Derrida comes to mind: is the use of computer code for legal writing beyond 'convenient abbreviation'?[80] Hofstadter would argue that computer code cannot be devoid of meaning and would indeed imprint its effect on the system. Hofstadter states "[w]hen a system of 'meaningless' symbols has patterns in it that accurately track, or mirror, various phenomena in the world, then that tracking, or mirroring imbues the symbols with some degree of meaning…"[81] Structure cannot be divorced from meaning.

Interestingly, Duncan Kennedy tested the relationship between structure, or symbols, and meaning by deconstructing arguments into a system of 'argument-bites.' Argument-bites form the basic unit of an argument and such bites often appear in opposing pairs. Operations performed on argument-bites constitute and build legal arguments. Such operations diagnose and assume the circumstances, or relationships, in which the argument-bite is to be manipulated and 'deployed.'[82] Such importation of structural linguistics conceptualizes law and arguments as systematically formulaic, "a product of the logic of operations."[83] Perhaps most interesting about Kennedy's theory is his idea of 'nesting.' Kennedy describes nesting as the act of argument 'reproduction' or the "reappearance of [argument-bites] when we have to resolve gaps, conflicts or ambiguities that emerge [from]…our initial solution to the doctrinal problem."[84] This creates an issue where language may be applied to law in a mechanical fashion; however, the process of reducing a legal argument into a system of operations raises considerations on the act of labelling and the power in its performance. That is – and as Kennedy rightfully notes – "language seems to be 'speaking the subject,' rather than the reverse."[85]

Kennedy's thought exercise is precisely analogous to the use of formal programming languages for legal drafting. Perhaps the question is not whether programming languages *should* be a legal language, but *how* they could be amenable to the demands of contract law. Are these demands to create more complete contracts, or to limit ambiguity and ensure contract enforcement? Thus far, this paper has sought to raise a number of concerns relevant to the use of programming languages, particularly in the translation of contracts from natural language to code. These concerns speak to whether the effort to complete contracts or disambiguate contractual terms could resolve inherent tensions of contract interpretation and enforceability.

a. The Spectrum

Modularity theory in contract design has made a triumphant return in recent scholarship.[86] Recalling Smith, "natural language comes in varieties that are more or less formal."[87] As seen in Legalese's example of the investment contract, contractual clauses undeniably span a range of formality, wherein

some clauses are formal than others. The trade-off, Smith claims, between context-dependence and formalism relies on the "amount of information conveyed"[88] within a particular provision. The amenability of the clause to a wider variety of situations and audiences then determines the appropriate degree of formalism.[89] In other words, genericism mirrors formalism.

This appears to be the approach taken by OpenLaw. The method of integrating code with natural language suggests that a latent assessment of formalism should be applied to contracts. In the sample advisor agreement, the Choice of Law and Venue clause was determined to be highly reproducible. Leaning on Smith, the provision was likely to be rather broad and contained language generic to most advisor agreements. The clause would, therefore, satisfy the test; it is amenable to translation. The OpenLaw method has seen adoption by the legal industry. Perhaps acknowledging the limitations of contracts drafted entirely in a programming language,[90] King and Wood Mallesons (KWM) have piloted a hybrid 'architecture' that combines "computational code and human discretion to produce a single contract[...]"[91] In the "ordinary life cycle of [a] contract" where there is "nothing unpredictable,"[92] performance is paramount, and such performance could be easily automated. Yet, complexity in the market renders prediction impossible; human judgment is required to assess the "extraordinary range of possibilities...facts which are far beyond the scope of any contract."[93] The solution, KWM recommends, is a 'seamless bond' between terms drafted in computational and natural language. The contract should be designed considering both languages together to avoid the risk of "complicating the legal framework through inconsistent terms."[94]

KWM's project is remarkable and touches on the significance of legal design. The question then becomes one of operation. Using programming languages to draft contracts could pose challenges akin to incorporating contractual boilerplate into new contracts. As Richard Posner argues, clauses "transposed to a new context may make an imperfect fit with the other clauses in the contract."[95] KWM seeks to overcome Posner's objection by actively acknowledging the significance of the legal relationship which forms the basis of contract law. But, drafting contractual clauses in tandem with code and natural language is a difficult task. The underlying purpose of code is efficiency by reducing redundancy. Recalling Jeffrey, IDEs are conducive to contract 'reusability,' fostering an increase in 'base documents' and the importance of boilerplate clauses.[96] It may be unavoidable that clauses drafted in formal language become standard boilerplate, easily reusable in a number of agreements. Consequently, there remains the risk of a conceptual mismatch.

Drafting contracts both in natural language and code at inception is perhaps optimistic. To preserve the integrity and consistency of their contracts, KWM would be obligated to determine whether (a) the clause should be importable to other agreements, or (b) hybrid contracts should act as unique templates of their own. A workaround may be to create standards for contractual clauses which are

conducive for 'code-ification,' as opposed to co-drafting or drafting in a combinatory manner. Regardless, the possibility of reframing contract doctrine altogether is foreseeable.

b. The Code is Mightier than the Pen?

Though efficient, standardizing legal language has the potential of shifting the dynamics of contract negotiation drafting. Consider Legalese's L4. The difficulty with the customized language, intended for legal writing, is that its default function has already translated legal language into code. The language embodies specific assumptions of the law in its descriptive state. Parties using the L4 language then inherit such assumptions, changing their interpretation of contractual obligations and post-agreement behavior.

Perhaps Stanley Fish describes it best, "language carries obligations and commitments that were once undertaken but eventually assumed; thereby rendering inseparable its original intentions at its core."[97] As a result, inherent philosophical and moral concepts are 'built into'[98] the language; over time, the interpretative exercise is forgotten and the concepts are accepted as fact. Similarly, Smith states "…there are many [contractual] phrases requiring the assignment of an interpretation and the interpretations can interact in ways that are sometimes hard to foresee."[99] With the use of programming languages to draft contracts, the forthcoming challenges would be to ensure that the interpretative exercise is not forgotten, that meaning must remain a continuum. Interpretation should allow for responsiveness to changing environments.

Frank Pasquale reflects on interpretation by drawing on the "elective affinities between poets and lawyers."[100] He argues, "[t]he law is a rich source of metaphor for poetry"[101] that extends beyond technical expertise in its drafting. Pasquale warns of the "reductive demands of technology,"[102] whereby its competencies are limited to sets of commands and series of directives. Rather, the poetic construction of legal rules embodies a sensibility and sensitivity to circumstance that is necessary in legal writing.[103] As a result, the space for quantification and simplification of language stands in opposition to the inherent art of legal drafting. If polysemy is an integral feature of natural language that cannot be rid, how then could programming languages find its place in legal language?

c. Party Reactions

Understanding party behavior may be helpful. Zev Eigen reflects on contracts "in action."[104] Contracts hold the impression of legal constraints,[105] making specificity in language a critical matter in the formation of a contract. In an empirical study, Eigen identifies two key propositions on behavior around contracts. He sees contracts as products of how drafters and signers interpret the law.[106] This suggests that, again, *how* contracts are written frame the behavior of parties and that drafting necessarily influences performance. Contracts that are negotiated tend to be less specific and have

more room for interpretation. Performance is less likely to be exact. Yet, performance is not compromised despite the 'incompleteness' of the contract. Instead, the contract's incompleteness signals trust between the parties.[107] In contrast, standardized legal language is authoritative in character. The drafters' interpretation of the law is reflected, not the signers. In this case, programming languages risk eliminating a part of that relationship between drafters and signers altogether.[108]

Online form-contracts[109] may be a revealing ancestor. In another study, Eigen tests the extent of party compliance with online form-contracts. The paper empirically examines contract enforcement on individuals, relative to the framing of obligations and participation in drafting.[110] His findings note that the option to modify terms and conditions positively impacts the eventual performance of contractual obligations. To participate in the formation of the contract importantly distinguishes an individual's interpretation of contractual obligations. Participation transforms meaningless instructions into promises. Eigen states "[p]romise creates obligation, whereas consent tolerates limits on what is being passively imposed or, [...] on rights surrendered."[111] The outcome of Eigen's experiment reveals that even the slightest involvement in the contracting process is sufficient to demonstrate the fundamental doctrine of contracts: the will of contracting parties.

5. Concluding Remarks and Next Steps

For programming languages to act as a legal language, party autonomy cannot be compromised. While the intention of programming languages is presumably not to place limitations on contract formation, "law has language at its core."[112] Consequently, the functional nature of most programming languages inadvertently transforms legal writing and the character of contract law. Correcting this problem requires an untangling performance from mutual assent. For programming languages such as Ergo, Solidity, or Sophia, an easy fix may be to assign legal effect to the annotations.[113] This would immediately reaffirm the weight of a contract's details, ensuring their role as understandably *prescriptive* as opposed to *descriptive*. Moreover, unintended transformations of contractual terms from conditions into warranties could be avoided.

As discussed, OpenLaw poses challenges around the simplification of legal writing. A hybrid language enables parallel drafting in both natural language and code. This could lead to new working guidelines that encourage articles and provisions that are (1) invariant to context and (2) adaptable for a variety of contracts. Smith's 'modular boilerplate' could be an excellent start, specifically with regard to the combined considerations of the remoteness of the audience and risk of the transaction.[114] Still, contracts must be "tailored to the parties' needs,"[115] and integrating standard 'reusable' code could occasionally lead to an improper fit. Therefore, determining which clauses may be 'code-ified' becomes

paramount to maintaining the integrity of the contract. For natural language clauses and code to have equivalent legal effect, execution must mirror the qualitative description.

In the case of Blawx and Lexon, the question becomes more complex as these projects intentionally reconfigure language rules, categories, and framing. Blawx and Lexon are predicated upon a shift in the performance of the law, bringing to light the translation of legal concepts. The adage "the medium is the message," is particularly relevant for these languages.[116] Both Blawx and Lexon express their own conceptual framework, redefining and asserting the meaning of existing legal interpretations. This further speaks to the inherent limitations of the law[117] and the difficulty with demarcating legal concepts.

Lessons of methodological transplant may be insightful. Katja Langenbucher engages with a theory of knowledge transfer that occurs between fields, creating import and inherit concepts. Langenbucher notes that the integration of economics, for example, into 'legalese' offers: (1) 'tested predictions,' (2) clear questions and precise methodology, and (3) a common language.[118] However, the difficulty of transplant, she suggests, is the superficiality of the import. This is typically owed to a misalignment between *assumptions* about the discipline, not the method itself.

Similarly, programming languages such as Blawx and Lexon promise comparable promises of clarity and precision. Their current state, however, risks undercutting contract doctrine by forcibly fitting the language onto predefined rules instead of existing legal principles. For Blawx, the conflation of validity with enforceability is problematic. Lexon, on the other hand, constructs barriers for contracting parties by limiting the available vocabulary and grammar. Again, legal language must be sufficiently agile to accommodate the possibility of unpredictable circumstances. Ultimately, contracts are about regulating the future through transactions.[119] Legal agreements allow performance "to unfold over time without either party being at the mercy of the other."[120] By confining the operational space, these 'mediums' inadvertently tie the hands of the parties.

To conclude, the purpose of this paper is not to suggest that programming languages cannot possibly aid legal writing. In fact, formal languages could provoke a more transparent discussion of party obligations and expectations within the dynamics of contractual negotiations.[121] Yet, the mechanics of current programming languages illustrate that there is still work required for code to become a legal language. As Geoffrey Samuel states, the "true meaning of a legal text is hidden within the language employed."[122] Reflecting on using programming languages as a medium for contract drafting has revealed that language indeed could alter the function of contract law. Further discussion is required on *how* programming languages could better navigate and shape the legal landscape. For now, perhaps it is best understood that the tool is an extension of the craft, and not simply a means for its effectuation.

Megan Ma is a PhD Candidate in Law and Lecturer at Sciences Po.

Footnotes

1. Keith Devlin, *Goodbye Descartes: The End of Logic and The Search for a New Cosmology of the Mind* 54 (1997). ↩
2. *Id.* at 62. ↩
3. George Boole, *The Laws of Thought* Chapter 1 (1854). ↩
4. Devlin, *supra* 1 at 77. ↩
5. Christopher Markou and Simon Deakin, *Ex Machina Lex: The Limits of Legal Computability*, Working Paper (2019), available at SSRN: https://ssrn.com/abstract=3407856. ↩
6. *Id.* See also E Brill and RJ Mooney, *Empirical Natural Language Processing*, 18 AI Magazine 4 (1997). ↩
7. *Id.* at 11-15. ↩
8. Deep Learning is a subset of machine learning that involves artificial neural networks and the assigning of numerical weights on input variables. For further explanation, see *id* at 10-12. ↩
9. Kingsley Martin, "Legal Technology Barriers – Understanding Language and Exercising Judgment," Legal Executive Institute (September 24, 2015), available at: https://www.legalexecutiveinstitute.com/legal-technology-barriers-understanding-language-and-exercising-judgement/. ↩
10. *Id.* ↩
11. *Id.* ↩
12. Beverly Rich, "How AI is Changing Contracts," Harvard Business Review (February 12, 2018), https://hbr.org/2018/02/how-ai-is-changing-contracts. *See* also white paper "How Professional Services Are Using Kira," Kira Machine Learning Contract Analysis (accessed February 2019) available at: https://cdn2.hubspot.net/hubfs/465399/04-resources/whitepapers/KiraSystemsWhitePaper-HowProfessionalServicesFirmsAreUsingKira.pdf. ↩
13. Henry E. Smith, *Modularity in Contracts: Boilerplate and Information Flow*, 10 Mich. L. Rev. 1175, 1176 (2006). ↩

14. "Super Drafter," Genie AI (accessed February 2020), available at: https://genieai.co/home. ↩
15. See *id.* Genie equally advertises smart filters and an automatic knowledge base. ↩
16. These examples follow existing literature which views technology as being only complementary to the law. *See* Frank Pasquale, *A Rule of Persons, Not Machines: The Limits of Legal Automation*, 87 Geo. Wash. L. Rev. 2, 6 (2019). *See* also Neil M. Richards and William D. Smart, "How should the law think about robots?" in Ryan Calo, A Michael Froomkin, Ian Kerr, eds., *Robot Law* 16-18 (2018). Their chapter argues how law hinges on social and political relationships and metaphors that require a *latent* understanding of temporal social constructs (emphasis added). ↩
17. Based in London, Juro translates contracts drafted in natural language to machine-readable form. The platform allows contracts to be built in a text-based format that is also language independent (i.e. JSON). The contracts, thereby, exist as code. *See* Juro's whitepaper, Richard Mabey and Pavel Kovalevich, "Machine-readable contracts: a new paradigm for legal documentation," Juro Resources (accessed February 2020), available at: https://info.juro.com/machine-learning?hsCtaTracking=60e75e06-22bb-4980-a584-186124e645b3%7C6a7d3770-289d-4c97-bcfb-c9f47afec77f. ↩
18. Michael Jeffrey, *What Would an Integrated Development Environment for Law look like*?, MIT Computational Law Report Release 1.1 (2020), available at: https://law.mit.edu/pub/whatwouldanintegrateddevelopmentenvironmentforlawlooklike. ↩
19. "MS Word for Lawyers: Document Templates," Tech for Lawyering Competencies: Research & Writing (accessed May 2020), available at: https://law-hawaii.libguides.com/TLC_Research_Writing/WordTemplates. ↩
20. "Our Technology," Clause Logic (accessed May 2020), available at: https://www.clauselogic.com/. ↩
21. Jeffrey, *supra* 18. ↩
22. Jeffrey notes, "For legal drafting…the focus is linguistic – rather than mathematical – but the core concepts are the same." See *id.* ↩
23. *Id.* ↩
24. Layman E. Allen, *Symbolic Logic: A Razor-Edged Tool for Drafting and Interpreting Legal Documents*, 66 Yale L. J. 833 (1957). ↩
25. *Id.* at 836. ↩

26. The author acknowledges that Python is able to work in adaptive environments and does not have a fixed data set. The comment is directed at the explicit expression of a given task. ↩
27. The Accord Project also offers Cicero and Concerto. The former is a contract template generator that helps build agreements embedded with machine executable components. The latter is a program that enables the data of computable contracts to be manipulated and modelled. The Accord Project also offers a trial template editor to build and test out smart agreements. For more information, see "What is the Accord Project," Accord Project (accessed February 2020), available at: https://www.accordproject.org/about. ↩
28. "The Sophia Language," Github Aeternity Docs (accessed April 2020), available at: https://github.com/aeternity/aesophia/blob/lima/docs/sophia.md#stateful-functions. *See* also "Solidity," Solidity (accessed April 2020) https://solidity.readthedocs.io/en/v0.6.6/. ↩
29. Lexon qualifies its model as designed with the intention of reasoning in natural language and uses formal linguistic structure. *See* Henning Diedrich, *Lexon: Digital Contracts* (2020). ↩
30. Defined as a programming language with a code structure to build commands that perform actions. "Code Structure," The JavaScript Language (accessed April 2020), available at: https://javascript.info/. ↩
31. "Markup Language," OpenLaw (accessed April 2020), available at: https://docs.openlaw.io/markup-language/#variables. ↩
32. *Id.* ↩
33. The author relies on the definition of Smart Legal Contracts as legal agreements that include digital components. These components allow the document to be interpreted and executed by computers. Both machine-readable and machine-executable contracts tie legal text to code. ↩
34. "Ergo Overview," Accord Project (accessed February 2020), available at: https://docs.accordproject.org/docs/logic-ergo.html. ↩
35. The goal is for conditional and bounded iteration. This presumably contributes to the reusability of contractual clauses. See *id.* ↩
36. Sophia defines smart contracts as contracts limited to the enforcement of relationships through cryptographic code. *See* "How do Ethereum Smart Contracts Work?," Coindesk (March 30, 2017), available at: https://www.coindesk.com/learn/ethereum-101/ethereum-smart-contracts-work. ↩

37. Aeternity describes its blockchain as a scalable platform for executing smart contracts. *See* "Why aeternity is so innovative?," Aeternity (accessed April 2020), available at: https://aeternity.com/. ↩
38. "The Sophia Language," *supra* 28. ↩
39. *Id.* ↩
40. "Solidity," *supra* 28. ↩
41. The author has elected to refrain from extensive review of blockchain mechanics. This reference simply clarifies that these languages, while similar, operate on different smart contracts platforms. ↩
42. "Structure of a Contract," Solidity (accessed April 2020), available at: https://solidity.readthedocs.io/en/v0.6.7/structure-of-a-contract.html. ↩
43. Recall Mireille Hildebrandt noting the shift to computation as one from reason to statistics. *See* Mireille Hildebrandt, "Law *as* computation in the era of artificial intelligence: Speaking law to the power of statistics," Draft for Special Issue U. Toronto L.J., 13 (2019). ↩
44. Smart contracts have been developed for blockchain use cases such as trading cryptocurrencies, voting, or even blind auctions. *See* "Solidity by Example," Solidity (accessed April 2020), available at: https://solidity.readthedocs.io/en/v0.6.7/solidity-by-example.html. *See* also Gideon Greenspan, "Why Many Smart Contract Use Cases Are Simply Impossible," Coindesk (April 17, 2016), available at: https://www.coindesk.com/three-smart-contract-misconceptions. ↩
45. Lexon has noted that future tools would account for the possibility of such abuse. *See* Diedrich *supra* 29 at 33. ↩
46. *Id.* at 107. ↩
47. *Id.* at 106. ↩
48. This is described as "if right now the conditions are true, then next the computer should do conclusions." See "Facts, Rules, and Queries," Blawx.com (accessed February 2020), available at: https://www.blawx.com/2019/09/facts-rules-and-queries/#page-content. ↩
49. Blawx commentary repeatedly discusses the difficulty of interpreting natural language when converting to a binary. "Example: Using Blawx for Rules as Code," Blawx.com (accessed February 2020), available at: https://www.blawx.com/2020/01/example-using-blawx-for-rules-as-code/#page-content. ↩
50. Here, a personal directive is understood to be a testator's 'living will.' ↩

51. Following the formula of declarative logic, the validity condition creates the rule statement "this is a personal directive (conclusion) if it is valid (condition)." Blawx, *supra* 48. ↩
52. To clarify, the Accord Project also seeks to develop legal templates with associated computing logic. Nevertheless, while the Accord Project offers a similar form, the study focuses on the independent application of the Ergo language. *See* "Overview," Accord Project (accessed February 2020), available at: https://docs.accordproject.org/docs/accordproject.html#what-is-a-smart-legal-contract. ↩
53. *See* "Markup Language," *supra* 31. ↩
54. In computer programming, a string is defined as a sequence of characters which represent text. *See* "String," TechTerms (accessed February 2020), available at: https://techterms.com/definition/string. ↩
55. "Markup Language," *supra* 31. ↩
56. "What is Legalese?," About our Company (accessed April 2020), available at: https://legalese.com/aboutus.html#innovation-premise. ↩
57. *Id.* ↩
58. "AL Interview: Software is Eating Law – Legalese.com," Artificial Lawyer (July 29, 2016), available at: https://www.artificiallawyer.com/2016/07/29/al-interview-software-is-eating-law-legalese-com/. ↩
59. "Why Computational Law?," Legalese (accessed April 2020), available at: https://legalese.com/computational-law.html. ↩
60. Layman E. Allen, "Language, Law, and Logic: Plain Legal Drafting for the Electronic Age," B. Niblett (ed.) *Computer Science and Law* 76 (1980). ↩
61. *Id.* at 77. ↩
62. *Id.* at 96. ↩
63. *Id.* at 99. ↩
64. Anthony J. Casey and Anthony Niblett, *Self-Driving Contracts*, 43 J. of Corp. Law. 101, 112-117 (2017). ↩
65. *Id.* at 113. ↩

66. As Ergo, Solidity, and Sophia all raise similar challenges, only the observations found with Ergo will be described. ↩

67. One could consider *Chartbrook Ltd. v. Persimmon Homes Ltd.* [2009] UKHL 38, the infamous English contracts case regarding the interpretation of contractual terms. The dispute concerned the amount Persimmon Homes was contractually obliged to pay Chartbrook. The Court of Appeals ruled that the natural meaning of the language more closely matched Chartbrook's interpretation. Upon appeal, the House of Lords unanimously ruled in favor of Persimmon Homes, stating that Chartbrook's interpretation was not commercially sensible. This case is a fascinating example regarding the express intention of contracting parties. Although the Court ruled on the basis of meaning, negotiations preceding contract formation could inform the interpretation. ↩

68. "Declarations," Accord Project (accessed February 2020), available at: https://docs.accordproject.org/docs/logic-decl.html. ↩

69. Sheila Jasanoff, *Can Science Make Sense of Life?* 117 (2019). Wittgenstein considered language as a form of life; and thereby, linguistic expression constructs its own being. *See* also Ludwig Wittgenstein, *Philosophical Investigations* 19 (1958). ↩

70. Diedrich, *supra* 29 at 104. ↩

71. *Id.* ↩

72. Giosuè Baggio, *Meaning in the Brain* 62 (2018). ↩

73. Blawx encountered difficulty interpreting the natural language of the legislation. Blawx recognized that it took 'creative liberties' in converting the statute to Blawx language. *See* Blawx, *supra* 48. ↩

74. H.L.A. Hart, *The Concept of Law* Chapters 4, 6 (1961). ↩

75. John L. Austin, *How to Do Things with Words* 7 (1975). ↩

76. *Id.* at 16. ↩

77. Gerald J. Postema, *Implicit Law*, 13 Law and Philosophy 361 (1994). Recall the difficulty of interpreting what is "not y" *See* Allen, *supra* 60. One could argue that Blawx may not be the right programming language for particular types of law (i.e. legislation). Procedural languages could potentially be a better option. Python, a procedural language, could construct a personal directive on the basis that its requirements are fundamentally conditional. A deeper investigation may be

warranted to determine if certain programming languages are more conducive to specific types of contracts. ↩

78. Casey and Niblett, *supra* 64. ↩

79. Recall the concept of boilerplate contracts as lifting the burden of interpretation and ensuring enforcement. Computable law borrows and extends the characteristics of contractual boilerplate in the name of increased precision, efficiency, and certainty. Smith, *supra* 13. ↩

80. Jacques Derrida questioned natural language and the medium of writing as the accepted form of communication. His argument strikes an interesting parallel between the uses of written and descriptive language in law. Derrida considers how writing is perceived as the original form of technology, believing "the history of writing will conform to a law of mechanical economy." Ultimately, writing was a means to conserve time and space and was independent of structure and meaning. *See* Jacques Derrida, *Limited Inc*. 4 (1988). ↩

81. Douglas Hofstader, *Gödel, Escher, Bach* preface-3 (Twentieth-anniversary ed. 1999). ↩

82. Kennedy describes relating argument-bites to one another by such operations as a means of confronting legal problems. *See* Duncan Kennedy, *A Semiotics of Legal Argument*, 3 Collected Courses of the Academy of European Law 317, 351 (1994). ↩

83. *Id.* at 343. ↩

84. *Id.* at 346. ↩

85. *Id.* at 350. ↩

86. *See* Smith, *supra* 13. *See* also George Triantis, *Improving Contract Quality: Modularity, Technology, and Innovation in Contract Design*, Stanford Law and Economics Olin Working Paper No. 450 (2013); and Matthew Jennejohn, *The Architecture of Contract Innovation*, 59 B.C.L. Rev 71 (2018). ↩

87. Smith, *id* at 1204. ↩

88. *Id.* ↩

89. *Id.* at 1206. ↩

90. The firm comments on the uniqueness of the project from other 'code-fication' ventures. The project avoids mere replication and enforcement of existing legal agreements. *See* "DnA Contracts," Github King and Wood Mallesons (accessed May 2020), available at: https://github.com/KingandWoodMallesonsAU/Project-DnA. ↩

91. *Id.* ↩
92. *Id.* ↩
93. *Id.* ↩
94. *Id.* ↩
95. Richard A. Posner, *The Law and Economics of Contract Interpretation*, 83 Texas L. Rev. 1581, 1587 (2005). ↩
96. Jeffrey, *supra* 18. ↩
97. Stanley Fish, *Is there a text in this class? The Authority of Interpretive Communities* 108 (1980). ↩
98. *Id.* at 107. ↩
99. Smith, *supra* 13 at 1206. ↩
100. Frank Pasquale, *The Substance of Poetic Procedure: Law & Humanity in the Work of Lawrence Joseph*, 32 Law & Literature 1, 7 (2020). *See* also Pasquale's references to the similarities between lawyers and poets found in David Kader and Michael Stanford, *Poetry of the Law: From Chaucer to the Present* (2010). ↩
101. *Id.* ↩
102. *Id.* at 33. ↩
103. *Id.* at 34. ↩
104. Zev J. Eigen, *Empirical Studies of Contract*, Faculty Working Paper 204 (2012), available at: https://scholarlycommons.law.northwestern.edu/cgi/viewcontent.cgi?article=1203&context=facultyworkingpapers. ↩
105. *Id.* at 16. ↩
106. *Id.* at 7. ↩
107. *Id.* at 17. Eigen references the study by Chou, Halevy and Murninghan. *See* Eileen Y. Chou et. al, *The Relational Costs of Complete Contracts*, IACM 24th Annual Conference Paper (2011), available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1872569. ↩
108. Consider Lawrence Lessig and the conceptualization of code as law. Lessig draws attention to code as a form of control, stating "code writers are increasingly lawmakers." *See* Lawrence Lessig,

Code 2.0 79 (2006). ↩

109. Zev Eigen defines online form-contracts as "contracts unilaterally drafted." *See* Zev J. Eigen, *When and Why Individuals Obey Contracts: Experimental Evidence of Consent, Compliance, Promise, and Performance*, 41 J. of Legal Studies 67 (2012). ↩

110. *Id.* at 68. ↩

111. *Id.* at 90. ↩

112. See Markou and Deakin, *supra* 5 at 3; and "...the central place of language in law" described in Pasquale, *supra* 101 at 31. *See* also Frank E. Cooper, *Effective Legal Writing* (1953) and his introduction, Law is Language. ↩

113. This, however, depends on the technical competency of lawyers to verify that annotations have been performed. Such an approach has been suggested by Shaanan Cohney and David Hoffman in *Transactional Scripts in Contract Stacks*, U of Penn Inst for Law & Econ Research Paper No. 20-08, 40-60. ↩

114. Smith, *supra* 13, at 1209-1210. ↩

115. *Id.* at 1210. ↩

116. See Marshall McLuhan, *Understanding Media: The Extensions of Man*, MIT Press (1994). Available at: https://mitpress.mit.edu/books/understanding-media. ↩

117. Namely, the exercise of distinguishing certain principles and standards being included or excluded from the legal system. *See* Joseph Raz, *Legal Principles and the Limits of Law*, 81 Yale L. J. 823 (1972). ↩

118. Katja Langenbucher, *Economic Transplants: On Lawmaking for Corporations and Capital Markets* 8-9 (2017). ↩

119. Geoffrey Samuel, *The Reality of Contract in English Law*, 13 Tulsa L. J. 508, 523 (2013). ↩

120. Posner, *supra* 95 at 1582. ↩

121. Recall the discussion on modularity; *see also* Smith, *supra* 13. ↩

122. Geoffrey Samuel, *Is Legal Reasoning like Medical Reasoning?*, 35 Legal Studies 323, 334 (2015). ↩