The Holy Grail of Computational Law

H. Diedrich
hd@lexon.org
23 March 23

ABSTRACT

This paper describes an approach to realize a centuries-old goal of Computational Law, its implementation, how to use it, and how trustlessness can augment its usefulness. The basics of a plain-text programming language are explained; how to use its compiler to translate controlled English into programs; the utility of its programmable token, and the foundational capabilities it adds to trustless computing.

INDEX

INTRODUCTION

A method to *compute* legal texts has been searched for since Leibniz' 1666 *de arte combinatoria*.[1] While electronic *discovery* has become the norm since the 1970s, the hope for electronic *analysis* of legal texts – conceived already in the late 1940s – as the complementary tenet of Computational Law,[2] had so far not been realized.

This changes with the language *Lexon*, which makes it possible to make a computer 'understand'[3] the logic of a law or an agreement and *perform* it. Lexon provides what Leibniz was looking for: a way to *program law*,[4] and contracts – so transparently, that it is frequently called *no-code*. This empowers lawmakers and will reduce the cost of, and speed up access to justice by magnitudes. It creates a high synergy with blockchains, making smart contracts readable for all, providing a missing link to the paradigm of *trustlessness*[5] by alleviating the need to trust the programmers. And importantly, to enable the use of smart contracts in business, it makes smart contracts readable for judges. Yet, Lexon might find broader application in *trustful* [ibid. 5] settings and as a new form of legalese.

As a programming language, Lexon is the first of a new generation – arguably, the 6th and last before computers can *reliably*[6] read any human text. As an AI tool, Lexon complements machine learning: *intelligent agents* programmed in Lexon solve real-world problems, are *unbiased*, excel in *transparency* and provide unparalleled *agency* to users. Most consequentially, *digital contracts* written in Lexon elevate prose to a *speech-act* of *felicitous performative language*[7] when performed in a trustless environment: due to the unstoppable nature of the blockchain, these words become true by uttering

[1] Leibniz' thesis is regarded as the beginning of computer sciences. For more on the history of Lexon, and Computational Law, see *https://lexon.org* and the *Lexon* book, 2020 – *https://amazon.com/dp/169774768X*.

[2] See prof. M. Genesereth, 2021, *What is Computational Law? – https://law.stanford.edu/2021/03/10/what-is-computational-law/*.

[3] Concretely, the document's meaning is reflected in the *abstract syntax trees* (AST) that the Lexon compiler creates. See appx. *Example Abstract Syntax Tree*, pg. 29; cf. *Processing Meaning* in *Lexon*, ibid., pg. 89.

[4] Cf. Clack and Reyes, footnotes 24 and 25, pg. 3.

[5] In blockchain parlance, *trustless* means *secured by blockchain mechanics* – *trustful* means *without such technical guarantees*, *depending on trust in someone*.

[6] Note that 100% determinism – often translatable to accuracy – is required in many professional use cases, which is a well-known challenge for machine learning.

[7] J. L. Austin, 1955, *How to Do Things with Words*. First noted by David Bovil.

them; a power commonly associated with magic. And rightly so: In effect, such *illocution* [ibid. 7] needs neither judges nor litigators and will enable long-tail markets that now cannot exist because their margins could not sustain the cost of policing them. Seen as AI, an artificial judge is being built right into every digital contract: the computer will provide a deterministic result, as the case may be. This makes viable the very simple as well as the very complex.

The Lexon *compiler* (pg. 5) translates plain text that adheres to the Lexon *grammar* (pg. 4) into code that machines understand. The technical approach that the compiler implements has long been suspected to be a feasible path to give machines a handle on natural language but had so far successfully been applied only to first-order logic,[8] which typically does not suffice to express relevant programs.[9] Lexon, like most programming languages *and the language of law*,[10] is based on higher order logic.[11]

Programmable tokens (pg. 11) complement the power of plain-text programming, allowing for the expression of more fine-grained rules, as well as reacting to specific *events*. For example, to partially divert tokens the moment they come into an account; or to revert transfers within a pre-determined time window. Interventions like these are not attainable through *smart contracts* alone but must be anchored on a deeper level, at the level of the token implementation.[12]

Another contribution of the programmable tokens is *modularity*, allowing the building of a complex system in a more deliberate way with smart *accounts* interacting instead of smart contracts (pg. 14). As they generally cannot be changed much, once deployed, the functionality of smart contracts must be decided in its totality[13] before they are put on the chain. This hampers not only error correction but progress as a whole. The per-account extensions of the programmable tokens address this limitation, intro ducing to blockchains the Lego-block type looseness that enabled the growth of the internet.[14]

LANGUAGE

Lexon is a plain-text programming language. This means that it reads like natural English and *digital contracts* written in Lexon can be understood by anyone, without requiring any prior knowledge of programming. With moderate effort or guidance by commodity AI, everyone will be able to write them. Lexon is also understood by machines. Its grammar expresses the intersection of what both humans and machines can parse. Grammars and compilers will evolve to extend their reach into both domains.

```
LEX Escrow.

"Payer" is a person.
"Payee" is a person.
"Arbiter" is a person.
"Fee" is an amount.

The Payer pays an Amount into escrow,
appoints the Payee, appoints the Arbiter,
and fixes the Fee.

CLAUSE: Pay Out.
The Arbiter may pay from escrow the Fee to
themselves, and afterwards pay the
remainder of the escrow to the Payee.

CLAUSE: Pay Back.
The Arbiter may pay from escrow the Fee to
themselves, and afterwards return the
remainder of the escrow to the Payer.
```

Source 1 – Lexon digital contract example

Lexon allows for the articulation of unambiguous prose[15] and the *deterministic* computation of logical results from it. Its grammar overlays natural language and higher order logic, in the way that Wittgenstein[16] demanded. For *artificial* domains, this may complete the quest for an unambiguous, universal language for philosophy and pure thought as envisioned by Leibniz, Wilkins, Frege, Russel, or Carnap.

[8] *Attempto Controlled English* (ACE) stands out. It compiles to 1st order Discourse Representation Structures – *http://attempto.ifi.uzh.ch*

[9] Prolog and its heirs add a lot of fascinating math to their first-order logic clauses to make things work.

[10] See *Law and Logic,* the *Lexon* book, ibid., pg. 63.

[11] Lexon's stack is different; see *Lexon*, ibid., pg. 112. Essentially, code *and* natural language are parsed in the same step, with far-reaching consequences.

[12] Of course, the ERC20 implementation of a token is itself a special case of a smart contract. In the above, *smart contract* is to mean a program that does interesting things *with* tokens rather than the special case.

[13] A typical challenge in computer sciences that smart contracts share with other powerful paradigms requiring a holistic approach, e.g., *functional programming.*

[14] See *Decentralization of Logic*, pg. 15.

[15] The above example is really a template: The concrete contract will have digital or descriptive identifiers inserted for the parties.

[16] L. Wittgenstein, 1953, *Philosophical Investigations.* Asst. prof. Andrea Leiter first noted the connection.

Lexon achieves this differently than was long supposed to be the way.[17] It arguably developed in a blind spot caused by the focus on the meaning of *words* that emanated from analytical philosophy and informed the development of early, general artificial intelligence.[18] Instead of trying to define words out of context, all we might ever (need to) know is the context, or as the later Wittgenstein proposed: *"the meaning of a word may be defined by how the word can be used as an element of language."*[ibid. 16] Lexon focuses on the *use* and fundamentally abandons the notion that meaning is vested in nouns. In so far as this is a structuralist argument, it shifts the context from the language to the four corners of an agreement.[19]

The result is that in Lexon texts, nouns tend to be interchangeable, and meaning is transported instead *by the relationship between* the nouns that the text describes. What matters is that the same name, or noun, is used consistently to refer to the same entity throughout one digital contract. Their common meaning can contribute to readability – but not to the specific meaning of the document.

Lexon shares this feature with mathematical formulas and any programming language; it is in keeping with how in business contracts, nouns are promoted to proper names to increase clarity: uncoupling from the inert meaning of words, and instead putting them into the service of the context, as neutral markers. Preferably, meaningful markers, but to be ignored by a judge when discerning the meaning of a contract. To exaggerate, the one word Lexon actually[20] understands is *transfer*. Which is unsurprising as this is the only act computers can perform: to transfer bits from one register to another. This anchors Lexon texts; everything else is qualifiers. Again unsurprisingly, this approach covers many types of agreements, as the transfer of something is the common topic of contracts.

An elemental contribution of the Lexon approach is how it maps natural language to compiler building tools – intuitively convincing, and in line, too, with what the tools were designed for[21] – yet different from what computer sciences had gotten used to in the chase for ever faster compile times. Only a simple extension to an established meta-language (BNF[22]) was required to better describe natural language grammar, for Lexon to stand upon the shoulders of the giants who paved the way.

Because Lexon solves a long-standing question of Computational Law, it works for blockchain smart contracts, as well as off-line – and even off-machine. Transcending computers, it may[23] over time replace today's legalese as a more useful, less ambiguous, and more readable language of law and contracting. The work of professors of law and computer sciences regarding Lexon[24, 25] may serve as inspiration in imagining the progress that could be possible; also for a two-thousand-year-old industry that is doing just fine.

Lexon is for everyone, not only for lawmakers and programmers, and it enables the coming profession of the *legal engineer*. But for its advantages in transparency and accessibility, Lexon may become a mainstream programming language: new programming languages are successful when, to increase productivity, they can strengthen teamwork or reduce sources of errors. Lexon does both. Going beyond what *object-oriented programming* achieved for teamwork of programmers, Lexon includes non-programmer domain experts, expanding the concept of *team* to reach beyond the circle of coders. And while developers might see no reason to leave the current mainstay of 3rd generation programming languages behind, their employers will find it desirable to increase transparency, and to have legal, business, and domain experts verify the programmers' results first-hand.

---

[17] Cf. Wilkins, 1668, *https://archive.org/details/AnEssayTowardsARealCharacterAndAPhilosophicalLanguage* and *https://www.youtube.com/watch?v=TjdbrLxc3Ck*

[18] See *https://lexon.org* for the forthcoming paper on *Lexon Intelligent Agents* that elaborates on Lexon's role as a tool for general artificial intelligence.

[19] To make it *concrete* is a key philosophical demand.

[20] See *https://lexon.org/vocabulary* & the *Lexon BIBLE*, 2020, https://www.amazon.com/dp/1656262665

[21] Lexon uses *Generalized Left-to-right Rightmost* parsing (GLR), first implemented by Masaru Tomita for natural languages in 1984: *LR Parsers for natural languages*; first proposed for *extensible* languages by Bernard Lang: 1974, *Deterministic techniques for efficient non-deterministic parsers.*

[22] *Bachus-Naur form* (BNF) is a metasyntax notation to describe the grammar of computer languages, first used to describe the grammar of ALGOL in 1960.

[23] An expectation articulated by law scholars.

[24] Prof. Christopher C. Clack, 2021, *Languages for Smart and Computable Contracts – https://arxiv.org/ftp/arxiv/papers/2104/2104.03764.pdf*

[25] Asst. prof. Carla L. Reyes, 2021, *Creating Cryptolaw for the Uniform Commercial Code – https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3809901*

GRAMMAR

The Lexon approach is independent of a specific natural language and the Lexon *grammar compiler* allows for a multitude of natural languages to be implemented.[26]

Lexon Grammar Form

Lexon grammars are defined in Lexon Grammar Form (LGF),[27] which is similar to Backus-Naur Form (BNF),[ibid. 22] enhancing readability to better capture the complexity and redundancy of natural language. For example, LGF's square brackets resolve optional elements as expected:

```
sentence:
  subject [condition [","] [":"]] predicates separator
```

Source 2 – Lexon Grammar Form (LGF) example

The above rule is equivalent to:[28]

```
sentence:
  subject predicates separator
  or subject condition predicates separator
  or subject condition "," predicates separator
  or subject condition ":" predicates separator
  or subject condition "," ":" predicates separator (sic)
```

Sentence Structure

Lexon's English grammar realizes the English natural language sentence structure of *subject, predicate, object*. That Lexon reflects this core pattern of natural language [ibid. 3] sets it apart from other programming languages. Note how the object is included in the predicate:

```
sentence:    subject [condition [","] [":"]]
             predicates separator
predicates:  predicates "," ["and" ["also"]] predicate
                                    or predicate
predicate:   payment
...
payment:     pay expression preposition object
pay:         "pay" or "pays"
preposition: "to" or "into"
```

Source 3 – Lexon sentence grammar (detail)

The above rules are employed to parse a sentence like this *recital*:

```
The Payer pays an Amount into escrow, appoints
the Payee, appoints the Arbiter, and fixes the Fee.
```

Source 4 – Lexon code example sentence

Document Structure

Lexon's grammar includes the layout of the *document structure*. This makes it harder to write ambiguous agreements. It reflects a common sequence of the parts of a contract.

```
LEX Escrow.                                   Head
-------------------------------------------------
"Payer" is a person.
"Payee" is a person.
"Arbiter" is a person.                        Definitions
"Fee" is an amount.
-------------------------------------------------
The Payer pays an Amount into escrow, appoints
the Payee, appoints the Arbiter, and fixes the Fee.   Recital
-------------------------------------------------
CLAUSE: Pay Out.
The Arbiter may pay from escrow the Fee to
themselves, and afterwards pay the remainder of
the escrow to the Payee.                      Clause
- - - - - - - - - - - - - - - - - - - - - - - - - -
CLAUSE: Pay Back.
The Arbiter may pay from escrow the Fee to
themselves, and afterwards return the
remainder of the escrow to the Payer.         Clause
```

Source 5 – Lexon document structure

The internal representation that the compiler creates during the translation is shown in appendix *Example Abstract Syntax Tree*, pg. 29. It visualizes the binary relationships that the compiler actually 'understands' from the sentence in *Source 4*.

The reduced grammar of Lexon forces sentences to be written straightforwardly, even when nested and verbose. The fact that the grammar is parseable by a computer guarantees mathematical unambiguity even though many redundant ways of expressing the same meaning have been enabled. The grammar still provides a one-way funnel; the flexibility is not bidirectional: the same can be articulated in many different ways but each way has only one meaning. It is exactly this that is achieved by limiting English grammar to a *controlled grammar*.

[26] The Lexon approach has been tested for English, German, and Japanese. The indication is that it will work for most languages, with English being one of the least challenging cases. See *https://lexon.org*.

[27] For more information on LGF see *https://lexon.org/lgf*

[28] Note the last rule that would not be correct English punctuation but is not ambiguous either.

COMPILER

The Lexon compiler[29, 30] accepts text adhering to the *controlled grammar* described above and transposes this natural-language code[36] to common 3rd generation programming languages like the ubiquitous, all-purpose Javascript or the preeminent blockchain language *Solidity*. Lexon Programmable Tokens[31] provide metered access to the online Lexon compiler.

Javascript is more interesting for Computational Law – the compilation process is the same as for Solidity but it is simpler to run – though it lacks a blockchain's facility to broadcast and transfer value. However, through a signed logging mechanism,[32] the Javascript programs produced by the compiler[33] can write a stand-alone, trustless *chain of log entries* that can be shared like a 'micro blockchain'[34] among the interested parties of (and third parties to) a contract.

OPERATION

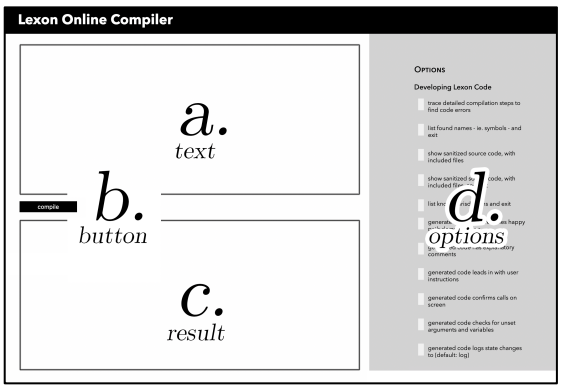

Figure 1 – Compiler screen at lexon.org/compiler

The online compiler is operated as follows:

a.	*text*	paste text into field ***a.***
b.	*compile*	click compile button ***b.***
c.	*result*	the resulting program code is shown in ***c.***
d.	*options*	to execute special functions, discussed below,[35] check boxes in list ***d.***

EXAMPLE

For example, the Lexon text given in *Source 1,* pg. 2, could be pasted into field ***a.*** Checking *barebones* in ***d.***, then clicking ***b.***, the Lexon compiler would translate the text in ***a.*** into this succinct Javascript code and show it in ***c.***:

```
module.exports = class Escrow {

   constructor(payer, amount, payee,
         arbiter, fee) {
      this.payer = payer;
      this.payee = payee;
      this.arbiter = arbiter;
      this.amount = amount;
      this.fee = fee;
      this.tx_(this.payer, 'escrow');
   }

   pay_out(caller)  {
      if(caller == this.arbiter) {
         this._pay('escrow',this.arbiter,
            this.fee);
         this.tx_('escrow',this.payee);
      } else {
         return 'not permitted.';
      }
   }

   pay_back(caller)  {
      if(caller == this.arbiter) {
         this.tx_('escrow',
            this.arbiter, this.fee);
         this.tx_('escrow', this.payer);
      } else {
         return 'not permitted.';
      }
   }

   tx_(from, to, amount) {
     console.log(`➡ system message:
        transfer ${amount} from ${from}
        to ${to}.`);
   }
}
```

Source 6 – Compilation example (Javascript, barebones)

The above code is optimized for demonstration purposes: it is short, not cluttered with comments, handling of fringe cases, nor extras like logging or state persistence. The settings ***d.*** controlling the output in ***c.*** are described below.[35] For a significantly more complex output from the *same* plain-text input, see appendix *Example Compilation*, from pg. 20. It adds all the elements that *barebones* tells the compiler to leave out, for a lot more code and comments.

[29] A compiler is basically a program that helps create other programs. It processes human-written files to create output that can be executed by a computer.

[30] Online at *https://lexon.org/compiler*

[31] See *Token*, from pg. 11.

[32] See *History*, pg. 8.

[33] Sic: the log is produced by the code that the Lexon compiler generates.

[34] See *The Micro Blockchain*, pg. 8.

[35] See *Options*, pg. 9.

From the same Lexon text,[36] *Source 1*, the Lexon compiler can also produce a Solidity smart contract for use on the Ethereum blockchain:

```
pragma solidity ^0.8.17;

contract Escrow {

   address payable payer;
   address payable payee;
   address payable arbiter;
   uint amount;
   uint fee;

   constructor(address payable _payee,
      address payable _arbiter, uint _fee)
      payable {
        payer = payable(msg.sender);
        payee = _payee;
        arbiter = _arbiter;
        fee = _fee;
   }

   function pay_out() public {
        require(msg.sender == arbiter,
           "not permitted");
        transfer_(arbiter, fee);
        transfer_(payee,
           address(this).balance);
   }

   function pay_back() public {
        require(msg.sender == arbiter,
           "not permitted");
        transfer_(arbiter, fee);
        transfer_(payer,
           address(this).balance);
   }

   function transfer_(address to_,
      uint amount_) internal {
         (bool success_, ) =
            to_.call{value:amount_}("");
         require(success_,
            "Transfer failed.");
   }
}
```

Source 7 – Compilation example (Solidity, barebones)

Compilation with the Ethereum Solidity compiler to deploy to an EVM-compatible blockchain is trivial.[37] A one-click deployment process from a Lexon text to the deployed smart contract has been demonstrated. Setups that include the generation of a user interface to interact with the contract on the chain are supported by the *ui info* option.[38] A *trustless Javascript* example is discussed next.

USE

Lexon's contribution is the translation from human language to the language of machines.[39] This section explores the machine angle, to help understand how Lexon's *results* can be embedded in a larger system; it is not concerned with Lexon text nor the Lexon compiler: it explains their *product*.

Focusing on Computational Law, the Javascript output is being discussed below, including a *trustless* mode, using signed logs. The use of the Solidity code is obvious.[ibid 37]

Running a resulting Javascript program manually requires beginner's programmer knowledge. It is valuable for research but is not the intended production use.[40] Embedding resulting code into a user interface, i.e., creating an app, is a routine task for a full-stack programmer.

Javascript output is executed using *node*.[41] The following uses a concrete example, again the *Source 1* (pg. 2), compiled with the option *all auxiliaries*.[42] The full resulting Javascript code is shown in appendix *Example Compilation* (pg. 20). The examples assume that the Javascript code resides in the file *./escrow.jsx*.

For an example terminal dialog of a live interaction with a Javascript digital contract, see appendix *Example Interaction*, pg. 25.

Prerequisites

When using the *all auxiliaries* option, the following node modules must be installed:[43]

```
$ npm install serialize-javascript
$ npm install tar
$ npm install nodemailer
$ npm install prompt-sync
```

Call Parameters

To execute examples, replace the parameters in angle brackets `< >` with literal values, e.g., `"Jane"` for `<<payer>>`, 10 for `<amount>`. The required *caller* is marked by double angle brackets `<< >>`. This designates the person whose

[36] Lexon *text*, *code* and *source* are used interchangeably.

[37] See example in appx. *Deploying to Ethereum*, pg. 24.

[38] See option *ui info*, pg. 10.

[39] Specifically, the *readability of the primary input text*, i.e., the transparency of the *core business logic*, to include non-programmers into its design and discussion.

[40] Lexon develops towards one-click deployment, to fully empower non-coders. Because it is *self-documenting*, and structured like a document, all information required for UI generation is present in a Lexon text.

[41] Node is a Javascript interpreter – *https://nodejs.org*

[42] See *all auxiliaries*, pg. 10.

[43] The required external modules are listed in the lead-in *instructions* comment section of the generated program code. Cf. *Source 12*, pg. 24. They vary, depending on Lexon code input and compiler options used.

passphrase will be required during the call if logs are signed[44] (see below).

If a role was not defined earlier, a call *makes the role be assigned* to the person named in the call: `"Jane"` becomes the `<<payer>>` by the first call that uses her name as `<<payer>>`. *If a role was defined earlier*, it can only be assigned to the same person[45] in subsequent calls: `"Jane"` must consistently be used for `<<payer>>` after having been used as payer once. Otherwise, the call will be rejected. This resembles how functional programming *binds* values, as well as how variables are understood in mathematics, and how natural language uses nouns to define roles implicitly. It is unusual only for 1st to 3rd generation programming languages.

The state for the internal checks is held by *node* between calls. State can be persisted.[46]

Initialization

The contract system is initialized by loading the module at the node console and instantiating it:

```
$ node
> contract = require(<code path>);
> escrow = new contract(<<payer>>,
  <amount>, <payee>, <arbiter>,<fee>);
```

For example, using these literal values:

```
$ node
> contract = require("./escrow.jsx");
> escrow = new contract("Jane", 10,
  "Joe", "Alice", 1);
```

Developing

Reset node's module cache each time you edit and recompile code, i.e., when experimenting:

```
> delete require.cache[
  require.resolve('./escrow.jsx')];
```

Core Functions

The main state progress functions that allow to interact with this example contract, are:

```
escrow.pay_out(<<arbiter>>)
escrow.pay_back(<<arbiter>>)
```

The function names are derived from the Lexon text, in the example from the clauses *Pay Out* and *Pay Back*. Different Lexon texts will result in different functions, parameter names, and arity.[47] Respecting scope and binding, parameters are deduced from the names that appear in the respective clauses. A larger example will have many core functions.[48] Their meaning is described by the Lexon text itself, e.g., the text of the clause *Pay Out* perfectly describes the core function `pay_out()`, because the latter was created from the former:

CLAUSE: Pay Out.
The Arbiter may pay from escrow the Fee to themselves, and afterwards pay the remainder of the escrow to the Payee.

The fact that the Lexon code precisely describes the Javascript code, contributes to the long-standing search in computer sciences for self-documenting code.[49, 50] It is of special value for the automated creation of UIs.

Trustless Contracting

The option *all auxiliaries* also triggers the creation of the following support functions that do not relate to individual Lexon clauses.

Most of them help to enable *trustless* operation of the Javascript code. To this end, user interactions with the contract are *logged* in a secure way; the state of the contract can be *persisted*; and log, state and contract code can be *bundled* to conveniently be archived and *sent* to a counterparty.

---

[44] This is a similar rhythm to how, e.g., Metamask helps users sign transactions in a blockchain setting.

[45] The focus of Computational Law is generally correctness. Javascript makes this example per se *trustful*: anyone could manipulate anything – there is just no gain in it, as a counterparty would immediately spot it. The *signed log*, however, cannot be manipulated and allows for *trustless* operation. Cf. *History*, pg. 8.

[46] See *Persistence*, pg. 8.

[47] The number of parameters of a function.

[48] Cf. the appendix *Example Compilation*, pg. 20, and *https://lexon.org/reyes.html* for a longer example, as well as more details on the code presented in *Creating Cryptolaw for the Uniform Commercial Code,* ibid.

[49] See *comments*, pg. 10.

[50] The reality is that most programmers are not fond of commenting and time pressure does not help, so that many developers call 'DRY' code the best, as it does not suffer from the irritations of bad – or worse – deprecated comments. But DRY stands for *Don't Repeat Yourself*, i.e., do not repeat in the comments what the code itself expresses. While this has virtue it also guarantees that today, in many important projects only programmers understand the code.

History

All state changes of the contract are written to a log[51] if the *log* option was selected during compilation. In our example, this includes actions performed by the neutral party called *Arbiter*. The log can be displayed with:

```
escrow.history()
```

The options *chaining* and *signatures* make the log *trustless*: an unforgeable trace of who did what, when. Each such log entry has this format (in one line):

```
⧉ <hash> ⏀ <timestamp> ✦ <role>
✓ <clause or noun> ❉ <signature>
```

Figure 2 – Hashed and signed log format

- ⧉ For every entry, the *hash* is the SHA-256 of the *entire* log file up to this point, from its first entry through the last signature.
- ⏀ The *timestamp* in plain text.
- ✦ The *role* name: can be a real name, an alias, a number, a public key or a hexadecimal Ethereum account id.
- ✓ The call's *clause* name as given in the Lexon code, or a *noun* being fixed
- ❉ The *signature* of the given *role* for all that came before ❉ in this entry.

A log entry looks like this:[52, cf. 102]

```
⧉ f3b21bde6076 ⏀ 3/22/2021, 1:34:22 AM
✦ FILER ✓ Collateral fixed ❉
6ff7ab169e49f2f574c7f13497a0c134eb5987476
a6fcc35515b60775cdaef75afa1bcdae06be79659
21cf36da228aec1b195f21b4696249d327a6799ef
ca1d5dd176ec95050407de40427dbdf5af8a6d4a6
e9eb88271717c51d7cded996fc931be7e1c932716
c26ee3cfbb2281579061342c9101e4bf66974ad85
e36c6dcca156fd1c6040f5f2925e4ae77e3b9b2c8
c7644020f86971d958600b8a17e2385f6d5d8c3c5
05f649d1a97852116869f2bca53fa172f63d05b88
eda1f312620bab5a90bf35334dc4a3890f737a7ad
950791e1c49eeabd5b64c51a3a6046cada2421e18
726643bbff3a7fe63ce18b15af0332972635caeca
c4bafc0659d4f71d3675
```

Source 8 – Log entry example

The Micro Blockchain

Hashes and signatures protect the integrity of the log [ibid. 52] like a blockchain would[53] – without a blockchain. A major difference is that someone could 'lose' the log whereas on a public blockchain, data is always accessible, and its consensus mechanism can help if two parties act at nearly the same time. Logs also do not support coins. But the essential upshots of a log-based *microchain* is that the contract state remains perfectly *private*, and the administrative effort is much lower: It can be magnitudes less costly in a business context to make a log accessible to all involved, than to operate a full-blown blockchain setup. Especially if the aim is to improve bookkeeping, i.e., when using tokens as a *unit of account* only and not as a store of value.[54] In business settings this will often be the more interesting use of tokens. A dedicated microchain, therefore, can be the more productive trustless solution. It offers guarantees similar to a blockchain, by the same method: re-iteratively hashing what came before and signing off on it.[55]

Persistence

The contract state[56] can be saved to disk and re-loaded at a later point in time, using a file that is literally called *state*. This serves to continue work after stopping and restarting node; and to allow for the sending of the entire contract system – state, code, and log – to a counterparty, who may perform the next step.

```
escrow.persist()
escrow.load()
```

Bundling

The contract code, state and log can be bundled into a tar archive, called *contract.tgz*, to more conveniently exchange or archive it.

```
escrow.bundle()
escrow.unbundle()
```

[51] This log is created by the program that the Lexon compiler generates, *not* the Lexon compiler itself. The compiler creates the code that creates the log.

[52] For a longer example see appx. *Example Log*, pg. 28.

[53] See instructions in the generated Javascript code, and the forthcoming paper *Lexon Microchain* at *lexon.org*.

[54] The three common functions of money are, to serve as store of value, as a unit of account, and as a medium of exchange. All three functions are expected from blockchain 'coins' but they can be utilized independently. Tokens do not have to be a store of value to be useful: They can help to understand how a business should be fairly settled – especially when parties trust each other or are fine with the legal system as backstop, e.g., for inter-department bookkeeping.

[55] See the original Bitcoin whitepaper: Nakamoto, 2008, *Bitcoin – https://bitcoin.org/bitcoin.pdf*.

[56] The term *state* means the current situation: the current internal variables of the contract. The *log* is the step-by-step list of events that led up to the current state. As such, a state is confirmed by its log.

The bundle contains the files *state*, and *log*, the source code, [57] and a file named INSTRUCTIONS.TXT that gives the receiver a first idea of what the bundle of files is about.

Email

The bundle can be sent to a counterparty. This can be done manually or by using the built-in:

```
escrow.send()
```

The function uses the email account configured in a *json* file called *config*:

```
{ email: {
    host: '<host>',
    port: <port>,
    user: '<email account user>',
    pass: '<email account password',
    from: '<email account address>',
    subject: '<subject line>',
    text: '<massage text>'
} }
```

Source 9 – Email configuration

The *host* and *port* entries can be summarily replaced by `service: 'gmail'` to utilize an existing gmail account.

Microchain Client

The easy to use, forthcoming *microclient* will streamline a fully private, decentralized, peer-to-peer setup, handling the log exchange directly. It is beyond the scope of this paper.[ibid. 53]

Keys

Keys for signing log entries are expected on-file, named after the actor, with the extension *.key*, e.g., *Joe.key*. For demonstration purposes, keys can be created using this utility function:

```
system.create_key(<name>, <passphrase>)
```

The resulting private key is written to disk, in the current directory, named `<name>.key`, e.g., *Joe.key*. The passphrase given in this call is not the private key but used to encrypt the private key in the file. It is this passphrase that is queried when running the contract.[58]

This convenience function is included to facilitate research; do not use it in production without assessing your risks first.

OPTIONS

Settings for the compilation process are made in the compiler screen at *https://lexon.org/compiler* (see Figure 1, pg. 5) by ticking boxes in screen area ***d.*** Not all options are interesting for everyone. Those more relevant to beginners are marked with an asterisk.*

Results shown in screen area ***c.*** (ibid.) will vary: some settings in ***d.*** cause information to be displayed in ***c.***, instead of code. In some instances, the contents of field ***a.*** will be ignored when button ***b.*** is clicked: e.g., when checking *version* in ***d.***, the version number of the compiler is displayed in ***c.***, no matter the contents of field ***a.*** When checking the option *names*, the list of all symbols (defined nouns) that are found in the Lexon code given in ***a.*** is listed in ***c.*** For some combinations of options, the output in ***c.*** will be a mix of code and other information.

Auxiliary Options

*version**

Display the compiler version information in ***c.***

echo source

List the Lexon source code that will be processed in ***c.***, but not the compilation result, to double check what input arrives at the compiler.

no result

No output of resulting code in ***c.***, to focus on other output, triggered by other options.

Developing Lexon Code

The following options can be helpful when writing Lexon texts. The online compiler serves as a convenient sounding board to find one's syntax errors and to explore what document structure will make sense for a task at hand.

*verbose**

Trace detailed compilation steps in ***c.***, to find errors in the Lexon text given in ***a.***

precompile

Show sanitized – *pre-compiled* – source code in ***c.*** and no compilation result. This shows the library[59] texts *included* in the source code, and the line numbering that error messages refer to. It also allows verification that definition and clause names are recognized as intended.

---

[57] At the command line, the compiler can learn the name of the source code file from the *-o* parameter. The code is generated accordingly. Online, a name is derived from the contract name following the LEX keyword.

[58] See appendix *Example Interaction*, pg. 25.

[*] option more likely of interest for beginners.

[59] Libraries contain text written to be used and re-used in multiple projects. It is inserted into the main text.

echo-precompile

Show precompiled Lexon source code in ***c.*** and also the compilation result.

*names**

List all names found in the Lexon code in ***c.*** As names can contain spaces, this list can help to check if they were parsed correctly.

*barebones**

The generated code is a simplistic 'happy path' for demonstration purposes. It does not have comments and does not catch errors or edge cases. This is a starting point to verify semantics and basic flow. It is an interesting learning device that *visually* surfaces the relationship between the Lexon text and the resulting program.

*comments**

The generated code embeds the Lexon text and generic comments to help the auditing of it.

*instructions**

The generated code has detailed instructions for use in its lead-in comments section. They resemble the discussion of the example code in chapter *Use* on pg. 6, and reflect the specific Lexon code at hand, listing all relevant core functions and their parameters.

*feedback**

The resulting Javascript code confirms calls onscreen. This is helpful for learning, experimenting and manual demonstrations from the console. The default – no feedback – is for production scenarios where the code is not supposed to talk back but a dedicated UI guides the user.

harden

The generated code checks for unset arguments and variables. This impacts readability of the output but is essential to catch user errors.

log

The generated code logs state changes to a file,[60] literally named *log*. This does not automatically activate hashes and signatures (see below): the log is by default trustful.

signatures

The generated code prompts users for a passphrase to sign log entries,[60] using the key in the file named like the caller,[61] with extension *.key*.

chaining

The generated code hash-chains log entries. This secures the log against manipulation, interconnecting its *entries* the same way that blockchain *blocks* are hashed and signed to build the eponymous chain of blocks.[ibid. 60]

persistence

The generated code can store state in a file, literally called *state*.

bundle

The generated code can tar the Lexon code, the Javascript code, the log, the contract state, and an instruction text into a file called, literally, *contract.tgz*.

all auxiliaries

The generated code features the options: *comments, instructions, feedback, harden, log, signatures, chaining, persistence,* and *bundle*.

Interfacing

This option produces the information needed for front-end generation for Lexon code:

ui info

Shows a JSON object encoding insights about the source code in area ***c.***

Developing Lexon Grammars

The following options support the development of new Lexon grammars, for different natural languages other than English.[62]

keywords

List in ***c.*** the keywords – the vocabulary – understood from an LGF[63] grammar provided in ***a.***

bnf

Produce BNF [ibid. 22] from an LGF grammar provided in ***a.*** This is useful to verify that optional terms in the LGF grammar spell out the intended individual BNF rules. The BNF is GNU Bison-compatible, which can help to create new targets, i.e., output in additional 3rd generation programming languages.

comments

Include the LGF rules in the BNF output as comments.

---

[60] See *Trustless Contracting*, pg. 7.

[61] See *Call Parameters*, pg. 6.

[62] See *https://lexon.org* on creating new grammars.

[63] See *Lexon Grammar Form*, pg. 4.

TOKEN

As legal agreements have a broader scope of concepts than covered by ERC20,[64] the token implementation adds functionality that allows for more powerful digital contracts. It also provides access to the Lexon online compiler.

UTILITY

The Lexon Programmable Token serves as a *standard library*[65] for the Lexon language and includes an *event-driven*[74] programming framework that allows the building of trustless logic under a new paradigm, *smart accounts*.[66] This introduces an edge that blockchains normally lack: arbitrary, programmable restrictions on *individual* accounts, making the core legal concept of *obligation*[ibid. 66] available to the otherwise strictly optional world of smart contracts. The token thus adds a foundational element that *increases the overlap of code and law* and demonstrates new, academically and commercially interesting features like *programmable money.*[67] It extends the breadth of the Lexon language, allowing for more *expressive*[68] digital contracts. In many cases, desired contract details would be impossible to realize without the new functions of the token, because what an underlying system does not provide, a *language* cannot support. Logically, the token enhances the *scope of the trustless membrane* that determines what functionalities can be performed in one coherently protected sequence without having to trust a third party for an intermediate step.[69]

Notably, the token also features conditional *reversibility* of transactions,[70] an essential attribute blockchains will have to offer to become relevant in traditional business settings.[71]

The token serves as subscription mechanism for the online compiler. It functions as a voucher to buy translations of Lexon texts into computer programs. The number of tokens held is the number of translations of Lexon texts that the owner can perform per month.[72]

The token is ERC20 and ERC2612-compatible[ibid. 64] and easily accessible through ERC20-compatible wallets. When used for *Deeds*,[73] an account switches to ERC721 (NFT)-compatibility, but all tokens remain accessible and usable.

ACCESSIBILITY

The token's features reflect its function as online voucher for using the Lexon compiler, and as enabler of the Lexon language. It also provides improved usability both for real-world business requirements and non-specialist users.

Engage *signatures pg. 33*

Accounts can be *engaged*, to use the Lexon compiler, or *unlocked* to allow for faster transactions. The default is *engaged*, unless the first tokens came in from an unlocked account. An *engaged* account can connect to the Lexon compiler and run one compilation per month[72] for every token in the account. The tokens are not consumed but remain in the account. To *unlock,* a delay may have to be respected.

To prevent defeat of the bookkeeping by account-hopping, an *engaged* account can only transfer tokens out to another account, one month[72] after the last transfer-in from any other account. The account can also be *unlocked* only one month[72] after the last transfer in, including purchase. An *unlocked* account cannot run compilations but can transfer out any amount at any time. It can be set *engaged* at any time, without delay, to use the compiler immediately.

Through this method, the Lexon tokens can be used and received without making an Ethereum transaction. The subscription mechanism is thus free of Ethereum transaction costs and cannot be impacted by Ethereum chain congestions. One can use the online compiler without owning or spending Ether.

---

[64] ERC20 is the main Ethereum token standard. See *https://eips.ethereum.org/EIPS/eip-20* and */eip-2612*.

[65] The *Solidity* function signatures are listed in appx. *Token Function Signatures*, pg. 32.

[66] See *Extensibility*, pg. 14.

[67] See P2P Financial Systems 2018, FED Cleveland – *https://lexon.org/programmable-money-2018.pdf* and appendix, *Programmable Money*, pg. 30.

[68] *Expressivity* in computer sciences is a measure of how much can be achieved with how many words. Programming languages differ by magnitudes in it, depending on the task at hand. Expressivity influences programmer productivity. And the more expressive a language is, the 'higher' it is often understood to be, as well as less costly to use, for safer and faster results.

[69] The innovation is informed by business needs that kept surfacing in Fortune500 consulting engagements.

[70] See *Reversibility*, pg. 13.

[71] Regarding the importance of reversibility, cf. *Lexon – Legal Smart Contracts*, 2017 – *https://lexon.org/lexon-whitepaper-2017.pdf*. The *implementation* presently described is a different approach to the same utility.

[72] A month is defined as exactly 30 days.

[73] See *Deeds*, pg. 13.

Sealing *signatures pg. 33*

The owner can *seal* an account to not accept transfers of tokens to it. If the account is also *engaged*, the seal disables all passive transfers out. If an attempt is made to transfer to or from a *sealed* account, it is reverted, unless the other account is on the *whitelist*. The *seal* can be made or dropped at any time. It helps protecting *engaged* accounts from being spammed, which could make it impossible to ever *unlock* them. The whitelist is per-account; the account owner can *whitelist* and *delist* other accounts at will. Incoming transfers from whitelisted accounts still trigger the delay described above.

Serial *signatures pg. 33*

Every account *in use* has a unique serial number, assigned at the time of the first reception of tokens to it. Token transfers can be made to the serial number, which is easier to remember and verify than the hexadecimal account address. Note that an account can 'exist' in terms of the public and private keys being in possession of the owner, and such account address – the hexadecimal Ethereum id – can be shared with third parties e.g., in anticipation of a future transfer to it. Such an account would not have a serial number yet if it never received tokens but it could be manually *registered* to obtain one. Serial numbers are spaced by an interval of 17 to reduce the likeliness of typos resulting into valid numbers.

Name *signatures pg. 33*

Accounts can be *labeled* with a unique name. Transactions can be sent to this name. A name cannot be changed once set, and once used, it can never be used by another account.

Account Abstraction *signatures pg. 33*

An account can be set to allow multiple keys to individually act like account owners. An additional key is first set to *anticipate* this role for an account, whose owner then *shares* access to it using the account's main key. The added key loses access to its original account, but it can *unshare* to revert back to normal operation, and the account's original key can be used to *revoke* the added key's role. An added key can *coshare* its power with a third key and subsequent *unshare* or *revoke* calls on the former do not cascade to the latter. An unlimited number of keys can share access to an account. The abstraction preserves compatibility with ERC20. Added keys can be used for all calls that require a signature, including direct transactions.

Multi-Signature *signatures pg. 33*

The account owner can *authorize* keys and set the number of signatures *demanded* for transactions. Multi-signatures are enforced only for direct *transfers*. The account owner must *sign* last. Signers can collectively *remove* a key.

Avatar *signatures pg. 33*

An account can be *marked* by an avatar image URL, which can be changed at any time.

Email *signatures pg. 33*

The owner can *publish* an email address to an account to be notified of relevant events. The entry is publicly readable and can be set and changed by the account owner at any time.

Subscribe & Feed *signatures pg. 33*

Accounts can *subscribe* to another account that they would like to hear *posts* from, e.g., via their email addresses.

Message *signatures pg. 33*

Accounts can receive short *messages*, which are stored in an append-only, otherwise stateless message-queue. A front-end can supply the messages to an account owner, keeping track of read-state and responses. Messages are public.

INTERACTION

Approval *signatures pg. 33*

The token features ERC20 [ibid. 64] *approval*, which allows for the definition of an amount of tokens that can be transferred out by another key.

Permit *signatures pg. 33*

ERC2612 [ibid. 64] *permits* can grant *approvals* so that accounts can be used that hold no Ether.

Commitment *signatures pg. 34*

Commitments resemble ERC20 *approvals*, but the designated receiver cannot actively pull the tokens to their account. The commitment is made by an account owner over a specified amount to a specified receiver for a specified time. When a commitment is made, no tokens are transferred yet, but the committed amount is blocked: it cannot be transferred out to a third party, neither by direct transfer nor by other mechanisms. The receiver can *release* the commitment and it can time out. A commitment is fulfilled by transferring the committed amount to the designated receiver. An account can have only one commitment at a time and must be *unlocked*. The rules as described can be enhanced by the use of *gates*.[ibid. 66]

Promise *signatures pg. 34*

A transfer to another account can be *promised*. As with *commitments*, no tokens are transferred at the time. The giver of the promise does not *approve* of nor *commit* to a transfer. *Promises* are a separate, weaker, and simpler concept that can be useful especially in connection with *gates*.[ibid. 66] Promises do not have to be covered at the time they are made and no amount is blocked. A promise is *made* for a specific amount, to a specific receiver, who can *forgive* it. They cannot be changed by the giver and do not time out. Any transfer from the giver to the receiver reduces the remaining amount. Multiple promises can be made for one account to multiple receivers. The rules as described can be extended by *gates*.

Cheques *signatures pg. 34*

An account owner can make an arbitrary number into a *cheque*. *Writing* a cheque blocks a given amount of tokens that is not available for transfers or any other purpose until the cheque is *deposit*ed. A cheque can be deposited only by the designated receiver, by providing the cheque number. Only an *unlocked* account can create checks. It cannot be switched to *engaged* until all its cheques are deposited. A cheque number can be any number but it can only be used once per issuing account. An unlimited number of cheques can be created, also to the same receivers, and be outstanding at the same time. Because the receiver is associated with the cheque and cannot be changed, the cheque number does not have to be a secret.

Escrow *signatures pg. 34*

An account holder can set a price in Ether and an amount of tokens to be sold for that price. Any account can act as buyer and will receive the set amount of tokens on sending the set amount of Ether. The offer can be changed or taken back at any time. The sale will fail if there are not enough tokens in the seller's account.

An account can also fix an amount of tokens, an amount of Ether, a specific buyer account, and a deadline. The buyer will receive the amount of tokens on sending the set amount of Ether before the deadline, provided that the supply of tokens in the selling account suffices.

Burning *signatures pg. 34*

Tokens can be *burn*ed, with proof, so they could be re-minted in controlled fashion on another chain. An account must be *unlocked* to burn.

Reversibility *signatures pg. 34*

This mechanism allows for transfers that can be reversed, at the discretion of the owner of a designated *forum* key, e.g., a court, arbitration service, notary or neutral third party. The transferred amount is locked-in, on the receiver's account, until it is reversed, the set time has elapsed, the sender makes the transfer final, or it is sent back to the sender by the receiver.

The owner of an account can *accede* to another account, the *grantee*, a specific time window and a public key, called *forum*, that can be used to trigger the *reversal* of any transfers *to* the *acceding* account from the *grantee*, from that point on. Any such transfer is marked as *pending* at the receiving account and cannot be used in any way, except sent back in whole or in part. The pending amount is initially zero. It grows with every transfer from the grantee and is reduced by any transfer back to the grantee. After the time out, the transfers are final and not pending anymore. The reversal can be triggered by the forum but *not* by the grantee. It transfers all *pending* tokens back to the grantee. The timeout can be set to any length, including indefinite. The grantee can end the pending state unilaterally. Pending funds count as *unlocked*.

Deeds *signatures pg. 34*

An account can be linked to one or multiple virtual assets by URL or hash. The assets are *transferrable* to other accounts following the ERC721 NFT standard. Methods of the same name and arity lose their ERC20 binding when an account is used for *deeds* but the tokens in the account remain accessible. The combination of fungible and non-fungible token functionality provides accessibility and extension features such as *serials*, *names*, *abstraction*, *reversibility, messaging* and *gates* to both use cases for every account, and enables DAOs.

DAO *signatures pg. 34*

An account can be shared by *members* who make a token contribution when *joining* and receive a pro rata share of the account's then-current token balance when *leaving*. The account can be set to be *decentralized* to disable direct access to it by the original owner. It is then not possible to initiate standard transfers. Only leaving members can receive tokens out of *decentralized* accounts, unless *gates* [ibid. 66] are employed. There are no functionalities beyond the basic mechanism of joining and leaving but *gates* can be used to add arbitrary rules, e.g., complex governance or asset handling.

EXTENSIBILITY

Gates

signatures pg. 35

Gates are a new concept that upgrade an account to a *smart account* that reacts directly to *events.*[74] A *gate* is a voluntary, per-account restriction on the use of the account's tokens.[75] It can be used to express *obligations*, which smart contracts generally cannot. Technically, gates are separate, re-usable callback function collections, tied-in through special event hooks in basic token functions, like transfers.

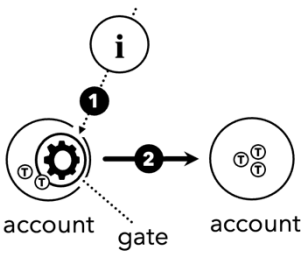

Figure 3 – two-step gate transfer

The trigger ❶ in *Figure 3* can be an out-of-band fact, communicated by an oracle; a special event like a *promise* or a *commitment*[76] being made or changed; or an incoming transfer. The gate can be programmed to intervene in any of these cases, e.g., to stop the transaction or to make a token transfer to a third account.[77]

The gate restrictions allow others to trust that a smart account's owner *must* honor the obligation expressed in its rules. Like with the general blockchain paradigm, the power of gates lies in the unbreakable promise they allow to be made. Some uses of gates are *trustless*, e.g., in connection with pending *commitments* or *reversible* transfers. Other useful setups start out *trustfully*, relying on out-of-band incentives that make sure that a smart account – and thus its gate – is actually used, and not sidestepped to escape the gate's restrictions. Such scenarios fit business settings where there exists some trust – if in the judicial system as backstop; and for friendly interactions that use tokens as a unit of account only,[ibid. 54] where there is no gain in open, traceable sabotage of the bookkeeping.

A gate is like a smart contract built *into* an account (cf. the cog icon in *Figure 3*), instead of existing 'between' accounts (*Figure 4*). It is more efficient than the typical smart contract flow, where a transfer often takes three steps (❶ pay in, ❷ trigger, ❸ pay out), and tokens get locked in at the smart contract's internal escrow between step ❶ and ❷, i.e., before anything really happens:

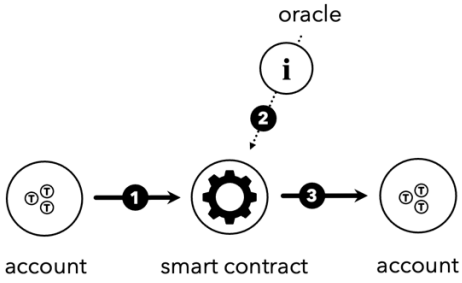

Figure 4 – common three-step smart contract transfer

Because a gate shifts the logic *into the account*, one step is eliminated. There is no preparatory step ❶ for the rules to kick in,[78] and the tokens are not locked-in while the transaction is pending.

Gates tie into the described mechanisms of *approval*, *commitment*, *reversibility*, *promises,* and *deeds*[79] and can extend them. They basically provide functions that are called at the time of specific *events*, for example, when a transfer comes in, a promise is *forgiven,* or a transfer *reversed.* They can consist of a single one-line function or be a complex, stateful object. Gates have temporary access to incoming tokens.

A gate can provide its rules to an unlimited number of accounts. It is *set* or *prepared* by an account owner, who elects the specific gate to become the unshakeable extension to the token functionality of the account. The owner can also assign a *keeper* of the gate. It is then *activated* by the keeper, confirming the settings, and

[74] *Event handling* is a staple paradigm of system programming that facilitates the composability of systems. It allows for different parts of a system to be created and deployed at different times, by different parties, without knowledge of each other. Gates are essentially stateful event handlers, expanding on Ethereum's original trajectory of adding state to Bitcoin's stateless transactions.

[75] This is different from prior proposals that extended the functionality of *all* tokens of a specific denomination equally. Cf. *operators* and *hooks* of EIP 777, *https://eips.ethereum.org/EIPS/eip-777.* Such efforts focused on allowing token creators to augment existing protocols at the *time of the creation* of a new token, affecting all tokens and accounts of the implementation. Gates, however, allow individual account owners to accept voluntary restrictions – the very nature of contracts – for one account only, at any time.

[76] See *Interaction*, pg. 12.

[77] See *Gate Interface*, pg. 34.

[78] There is a preparational step to set up the gate, once.

[79] See *Interaction*, pg. 12.

paying the fee if there is any. An account can only be *gated* with consent of both account owner and gate keeper, if there is one.

Whenever a transfer is initiated *from* the gated account, the gate smart contract is called first with the information from whom to whom and on what amount the transfer is to be. The gate contract can then veto the transaction. When a gated account *receives* funds, the gate is called and can cancel the transaction or draw part or all of the received amount from the gated account. An internal locking mechanism prevents re-entrance circularity.

Gates can be programmed to be permanent, or to *close* according to specified rules. The keeper, or the gate itself, can close a gate and thus restore a smart account to its default, non-smart state. The keeper can also *update* parameters of the gate contract, if there are any. What parameters control a gate depends on its design. The owner of a smart account will have had insight into it before accepting the gate. Both the owner of a smart account and its gate keeper can retrieve *information* about the gate's status, which can be of any complexity.

Gates can be re-used, which allows for standard implementations to emerge. Gates increase the cost of a transfer, depending on the intricacy of their event handling.

Decentralization of Logic

Conceptually, gates are a more decentralized administration of logic than smart contracts, with the expected advantages. The relationship topology of a complex smart contract typically has a star shape, revealing its centralizing effect. This is what graphical blockchain browsers often show:[80]

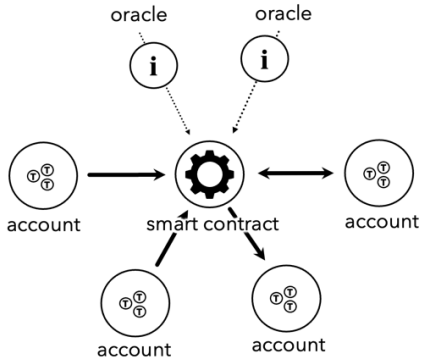

Figure 5 – Virtual centralization by smart contracts

In contrast, the gate mechanism, being per-account, replaces this pattern with a web of direct connections, introducing intricate peer-to-peer interaction between accounts:

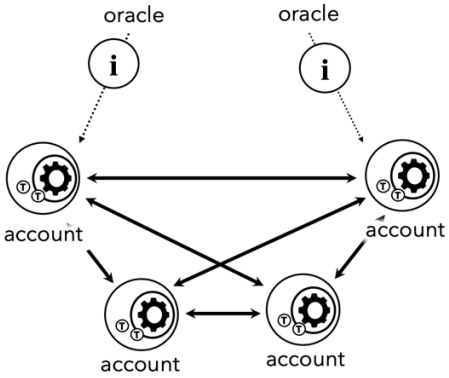

Figure 6 – Decentralization through gates

This is more robust and better suited for growth than a star shape, as it allows for incremental progress in the development of relationships; rather than requiring the limiting, holistic and even dictatorial approach that characterizes the design of bulky smart contracts,[ibid. 13] Each account can be fitted with custom logic independently and the cascade of algorithms that interact to implement complex business logic can be built up in successive steps. Notably, the individual building blocks are malleable – correctible and improvable – with authority to change accruing to exactly those parties who would stand to lose from a change. Even the most complex governance mechanisms could but approximate this modularity for a central smart contract.

Dynamic Stacking of Logic

A main advantage of gates is the more dynamic flow: tokens need not be moved out first, and the application of rules across participating accounts can be stacked,[81] for high complexity. The cascading transaction cost can be shared.

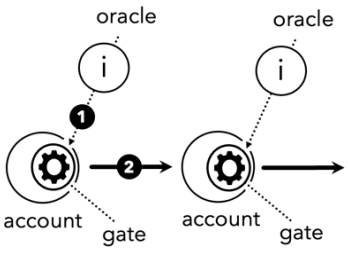

Figure 7 – Loosely stacked gates

The interaction between the account-specific rules is ad-hoc, for any specific transaction.

[80] The author developed a graphical Ethereum transaction browser for IBM where these shapes emerged as the prevalent patterns of transactions.

[81] A trustless custody mechanism prevents spill-over effects between gates, namely that a deeper gate could veto (revert) a transfer for everyone involved.

The two gates that may be involved are independent and oblivious of each other. In sum though, they form a loosely coupled emergent system. Such dynamic interaction was an original hope for blockchain smart contracts. It is being held back by the lack of API definitions, now addressed by the community-driven EIP process that does not aspire to be fast. Gates are magnitudes simpler, individual decisions. They work without intra-contract standards because their tie-in is not *between* accounts. Gates may affect each other but need not talk to each other.

Proofing Programmable Money

Gates demonstrate a mechanism to securely track hierarchic channels of supply chain cash flow[82] to speed up disbursement and reduce risks of trade financing, or to implement complex bookkeeping, e.g., for staggered media royalties. An ultimate employer does not have to lock-in the entire budget of a contract into a smart contract but merely commit to be using a specific account to trigger a pre-agreed, unstoppable distribution mechanism. After that first trigger, the transfer mechanism can be trustless. If the ultimate employer desires that subcontractors – who do the actual work – get paid, incentives are aligned.

Gates thus demonstrate the possibility of *programmable money*, as proposed at the Cleveland Federal Reserve in 2018:[83] For a specific budget to reliably and continuously be subject to specific rules, a system architecture must provide for the programmability of individual accounts. This enables the metaphor of the funds themselves being programmable. Neither traditional blockchain smart contracts nor traditional automated banking mechanisms are sufficiently subtle for this purpose. A shortfall inflicts significant economic cost, most commonly in the form of late or denied payment that punishes the productive party to a contract. To address this well-researched power abuse, some of the relevant logic must be anchored in the layer of the code that implements the token,[84] rather than at the higher layer of the contract logic.

SALE

Purchase *signature pg. 34*

The token can be purchased for Ether at *https://lexon.org/tokens* or by sending Ether to *0x3761cd40e07eb5948150202b484647fD595E674F,* from a whitelisted address. The smart contract at this location returns the purchased amount of Lexon tokens at the correct total price. Sending Ether from a whitelisted wallet to this address, results in the wallet receiving the Lexon tokens.

Promotion

First-time visitors have 10 compilations free. A purchase of tokens is offered automatically after the 10^{th} compiler run. Professors and students of law, computer sciences, linguistics, political sciences, philosophy, and related fields can apply for a drop at *https://lexon.org/faculty.*

Use

Accounts are *engaged*[85] by default and can immediately be used with the compiler but transferred out only after 30 days; except when the first transfer in came from an *unlocked* account.

Holding

Tokens can be managed with ERC20-compatible Ethereum wallets. Tokens can be used with the compiler even when air-gapped, in cold storage, or *sealed*[86] because no transactions and no use of keys are required for compile runs.

Transacting *signatures pg. 33*

Tokens can be transferred using ERC20-compatible Ethereum wallets. They can be passively transferred without owning any *Ether,* via ERC2612-permits.[87] Other specific token mechanisms described above[88] – e.g., ERC20 approval – can move tokens, even *if in cold storage*, but not when *engaged*[85] and *sealed.*[86]

Cap

The supply is capped at 200 million tokens. The sale can be paused, effecting a temporary soft cap. The first soft cap will be at around 10M tokens issued to balance compiler capacity.

[82] See the appendix, *Programmable Money*, pg. 30.

[83] The essential example for programmable money is given in a supply chain where the main contractor's funds are restricted to distribution to subcontractors only, without possibility to unduly withhold or divert. See appendix, *Programmable Money*, pg. 30.

[84] A comparable context can be found in locking mechanisms of databases. It is impossible to implement protections for data integrity during transactions on the application level if the data storage layer does not provide the required basic, atomic features. E.g., locking to prevent near-parallel changes from overwriting each other in a concurrent system.

[85] See *Engage*, pg. 11.

[86] See *Sealing*, pg. 12.

[87] See *Permit*, pg. 12.

[88] See *Interaction*, pg. 12.

Price

signature pg. 34

The price for Lexon Programmable Tokens increases with the amount of tokens issued.[89] This serves as load protection for the online compiler.

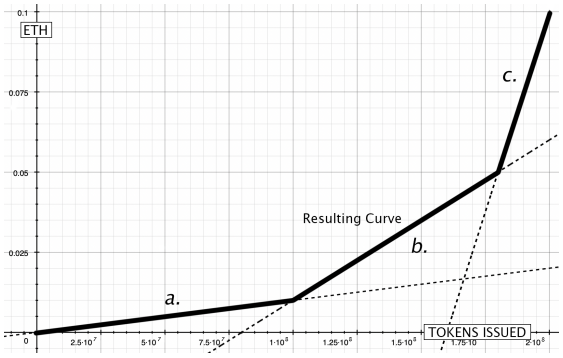

Figure 8 – Token sale price based on tokens issued[89]

Current Price

The current price, in Ether, can be learned at *https://lexon.org/tokens*. The page lists the price for the *next* token sold and allows the querying of the total price for a planned purchase, e.g., the amount of tokens one would receive for 0.01 Ether.

Price Formula

The token price is calculated by a formula *p = (issued – k) / m ± offset*. This has a logarithmic effect in terms of *purchasing power*: The increase is steepest in the beginning, relative to Ether spent, because the same amount of Ether buys progressively fewer tokens, which drives the price progressively to a lesser degree. Dampening the effect, the initial price increase rate (*Figure 8, a.*) grows steeper after 100M tokens have been issued (*b.*) and again after 180M (*c.*). For the respective partial curves, *a.*, *b.*, *c.*, the formulae are:

Price Point Formula

issued[89]	Price	Curve
$< 100\text{M}$	$\frac{\text{issued}}{10\text{B}}$	*a.*
$\geq 100\text{M}$	$\frac{\text{issued} - 80\text{M}}{9\text{B}}$	*b.*
$\geq 180\text{M}$	$\frac{\text{issued} - 160\text{M}}{480\text{M}}$	*c.*

Table 1 – Token price formula

The *offset* serves as protection against imbalances from outside the sales mechanism.

Price Points

Some resulting price points are as follows. E.g., at exactly 10 million tokens issued, the price for the next token is 0.001 Ether:

Select Price Points

issued[89]	Price
1M	0.0001 Eth
10M	0.001 Eth
100M	0.01 Eth
200M	0.1 Eth

Table 2 – Token price points

Effective Rebate

For an individual purchase, ten price points are established to calculate the total price. This effects a rebate, the steeper the higher the amount purchased. It will therefore at any point be more economic to buy in one transaction, instead of spreading a purchase across multiple transactions.

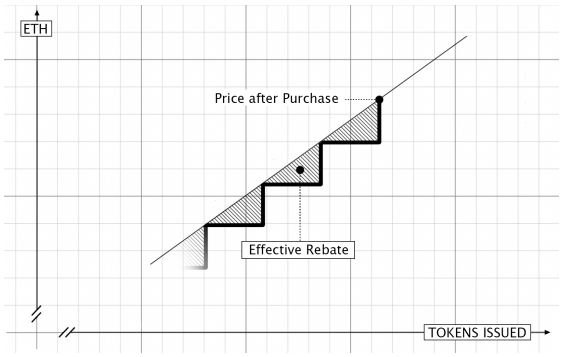

Figure 9 – Effective rebate (schematic)

Sponsoring

signatures pg. 34

Schools can *sponsor* their students, supplying students' accounts with tokens that can be used for the online compiler but not transferred in any way. The sponsor can eventually *reward* select receivers by giving them full control over their tokens. Or, where not, *collect* the tokens back to the sponsoring account, where they also revert back to being normal tokens. A receiver can *dropout* at any time and send all tokens back to the sponsor, upon which the sponsored account becomes a normal account again. Sponsoring can be initiated from any account at *https://lexon.org/sponsoring*. For the initial supply, see *Promotion*, pg. 16.

[89] Drops and locked-in sales can be exempted.

CONCLUSION

Lexon's real-world impact is broad and sustained. It unites developments in computational law, cryptography, computer sciences, AI[90] and linguistics to achieve long-sought milestones in each field: digital contract analysis, legally enforceable smart contracts, self-documenting code, deterministic language processing, and an executable human language. The resulting accessibility and agency drive a productivity increase set to transform commerce, finance, and governance. It opens new ways even to *think* about some of the more intractable-looking challenges of our times, and solve them.

Lexon's contribution is unique, a result of original research. It starts with compiler technology, built on industry standards for scalability and robustness, to enable a language design that achieves perfect readability, and a bridge between law and coding. Accordingly, Lexon has been called the *"Holy Grail of Computational Law"* and the co-inventor of the AI language Prolog, Robert Kowalski, named Lexon among the *"next biggest changes."*[91]

Lexon addresses a burning platform issue considered an almost hopeless cause: to lower the cost of access to justice, to the level needed to heal our societies. It will level the playing field in business, protecting creativity and merit against the deep pockets of incumbents, and regulatory capture. Because Lexon is up to a million times cheaper, and a billion times faster,[92] the difference it makes is a qualitative one. Over time, it will fundamentally change how business, law and politics work.

But Lexon can be used to write law, too. An official proposal for U.C.C. model law [ibid. 25] has been presented to the reform committee appointed by the American Law Institute. Eventually, Lexon will be the language that the real Robotic Laws[93] will be articulated in, to embed reliable and unambiguous limitations into autonomous machines. This will be plaintext code, written by elected lawmakers, approved in the democratic process.

Lexon even works purely as a language, entirely 'off-machine.' Because of its readability and unambiguity, lawyers call it a new form of legalese. With the Lexon compiler as a sui generis test tool.

Being 'human-readable,' Lexon is a catalyst for trustless technology. Its *digital contracts* are at the same time legally enforceable agreements and unbreakable blockchain smart contracts. This solves the question whether *code is law*. It makes contract programs – like those on blockchains – admissible in court and will close the digital divide between the legal profession and the numerous black box automations that 'administer justice' today.

Informed by real-world scenarios, the Lexon Programmable Tokens improve the expressiveness [ibid. 6 8] of digital contracts and significantly increase the usefulness of tokens for business, providing missing features like reversibility and modularity – options whose lack has been identified as a major inhibitor of blockchain utilization in traditional commerce, and which must be implemented at the lowest technical layer – 'inside' the token.

The token's *event-handling* framework upgrades Ethereum's single-thread object-oriented approach with a state-of-the-art, *composable* paradigm that emphasizes the role of accounts as the actual *objects* of the system. It introduces a new, modular growth model on the system level, allowing for more powerful digital contracts that can interact more flexibly, preparing the ground for larger patterns of interactivity on Ethereum.

Smart accounts enable the metaphor of programmable money,[94] as well as a more distributed logical topology of interaction between blockchain accounts, overcoming the bottlenecking 'star' pattern of smart contracts.

Lexon's far-reaching consequence is a merging of the legal and the IT space into a perplexing new reality that may appear unexpected but has been envisioned, and worked towards, from the beginning of the computer sciences.[95] Its transparency and ease will unleash enormous power for good, pulling law back to a semblance of equal justice – a notion as urgently

[90] Machine learning is complementary to Lexon, its romp the perfect fit for the preparatory phase of writing.

[91] Prof. Robert Kowalski, 2021 FutureLaw, Stanford – Together with *Blawx* and Kowalski's *Logical English*: *https://law.stanford.edu/press/new-codex-prize-awarded-to-computational-law-pioneers-during-9th-annual-codex-futurelaw-conference/* – regarding the differences between Lexon and Logical English, see *https://lexon.org#logical-english*

[92] These are not exaggerations, see the *Lexon* book, ibid.

[93] See appx. *Robotic Laws*, pg. 31.

[94] See appx. *Programmable Money*, pg. 30.

[95] Leibniz' first idea of what should be programmed – in 1666 – was a thousand years old, Roman contract law.

necessary as it sounds naïve – and drive the overdue digital reform of democratic governance, strengthening participation and representation in the way that many intuit should be possible with present-day means. For fairer global commerce, Lexon will help to provide new rails that are safe, low-cost, and transparent for every participant – in the course of which, stopping the descent of programming into a gatekeeping, dark art of the powerful.

An economic and social quantum leap is what the world needs, according to the assessment of the secretary-general of the UN:

"Something is fundamentally wrong with our economic and financial system," António Guterres told the general assembly,[96] reporting increasing poverty, hunger and burdens of debt.

"It needs a radical transformation."

The trustless technology for commerce, law, and governance that Lexon enables can provide the make-over the secretary-general calls for. This is no co-incidence but the result of focused research that has been going on since the 1980s, not only into how the power of computers can be used for good, but into what could be done to counter the rampant *abuse* of digital innovation in all walks of life.[97] Lexon brings together deep tech that emerged from these passionate efforts and makes it accessible.

But importantly, Lexon is *backwards-compatible*: As it is difficult to see how the beneficiaries of the status quo will be incentivized to help with meaningful change, the most powerful transformational aspect of technology is that it *just works*. Lexon can drive change, by incremental improvements, because – looping back to its very essence – it is compatible with what exists: viz., *readable by judges*. It was made to strengthen our most powerful interface, lurid cyborg dreams aside: *language*.

The key to creating Lexon programs is the Lexon compiler. It can be used online without installation at *https://lexon.org/compiler*.

Payment for its use is by subscription, expressed in Lexon Programmable Tokens hedl. The tokens can be purchased at *https://lexon.org/tokens*.

DISCLAIMERS

The information provided in this paper is strictly for educational purposes. There are no warranties, express or implied. Any use of this information is at your own risk. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption. See *https://lexon.org/disclaimer*.

Lexon is not an all-purpose human language. An unambiguous language is desirable for programming and lawmaking but less so for other purposes of human communication.[98]

Lexon compiler output must be audited before using it in production. There is no warranty for fitness for any purpose, nor any other warranty, for the compiler output or the token functionalities. See the license information at *https://lexon.org/license*.

Secret key usage examples in this paper are simplified for educational purposes. Do not create or store keys in production as shown in the examples.[99]

The described tokens are not for investment; they may not work as a store of value. There is no secondary market for the tokens, and none is planned. The token is not bought back by the issuer. The token does not represent a share in a company or IP. It does not make eligible for any payment.

LICENSE

[96] A. Guterres, *Briefing to the General Assembly on Priorities for 2023 – https://www.un.org/sg/en/content/sg/speeches/2023-02-06/secretary-generals-briefing-the-general-assembly-priorities-for-2023*

[97] See the *Lexon* book, ibid., Appendix II, *Blockchains & Smart Contracts* on the history of the Cypherpunks.

[98] Cf. appx. *The Principles of Newspeak* in G. Orwell, 1949, *Nineteen-Eighty-Four*.

[99] Key management is a field in its own right. Lexon, per se, works completely without keys.

[100] *https://creativecommons.org/licenses/by-sa/4.0/*

[101] *https://www.gnu.org/licenses/agpl-3.0-standalone.html*

APPENDIX

EXAMPLE COMPILATION

For the reader's convenience, the two boxes on this page are a repeat from pages 2 and 5.

LEX Escrow.

"Payer" is a person.
"Payee" is a person.
"Arbiter" is a person.
"Fee" is an amount.

The Payer pays an Amount into escrow, appoints the Payee, appoints the Arbiter, and also fixes the Fee.

CLAUSE: Pay Out.
The Arbiter may pay from escrow the Fee to themselves, and afterwards pay the remainder of the escrow to the Payee.

CLAUSE: Pay Back.
The Arbiter may pay from escrow the Fee to themselves, and afterwards return the remainder of the escrow to the Payer.

Source 10 – Lexon code example (escrow)

Using the *barebones* option, the Lexon compiler translates the above Lexon code into this Javascript:

```
module.exports = class Escrow {

    constructor(payer, amount, payee, arbiter, fee) {
        this.payer = payer;
        this.payee = payee;
        this.arbiter = arbiter;
        this.amount = amount;
        this.fee = fee;
        this._pay(this.payer, 'escrow', this.amount);
    }

    pay_out(caller)  {
        if(caller == this.arbiter) {
            this._pay('escrow', this.arbiter, this.fee);
            this._pay('escrow', this.payee, this.amount);
        } else {
            return 'not permitted.';
        }
    }

    pay_back(caller)  {
        if(caller == this.arbiter) {
            this._pay('escrow', this.arbiter, this.fee);
            this._pay('escrow', this.payer, this.amount);
        } else {
            return 'not permitted.';
        }
    }

    _pay(from, to, amount) {
        console.log(`➡ system message: pay ${amount} from ${from} to ${to}.`);
    }
}
```

Source 11 – Lexon compilation example (barebones)

Using the *all auxiliaries* option, the Lexon compiler translates the Lexon code from the previous page into the following Javascript program. Its core functionality is identical to the *barebones* version, but it has additional features as described in *Trustless Contracting* from pg. 7. In part, to make up for the lack of *trustlessness* that comes for free for a program running on a blockchain.

Instructions assume that the result was saved into file *escrow.jsx*.

```
/* Lexon-generated Javascript

   code:            Escrow

   file:            escrow.lex

   compiler:        lexon 0.3 alpha 85

   grammar:         0.2.20 / subset 0.3.8 alpha 79 - English / Reyes

   backend:         javascript 0.3.82

   target:          node 14.1+

   parameters:      --javascript  --all-auxiliaries


   INSTRUCTIONS FOR USE:

   Execute this program using node. Replace the <parameters> with literal values.

   Running this program as-is requires beginners programmer knowledge. This phase is
   yet not covered by lexon's mission to make code readable and useful for non-coders.
   In the future, an interface will be generated to complete this last mile. However,
   embedding this code into a self-explanatory user interface is a straight forward
   task for a full-stack programmer.

   Note that the instructions below reflect your lexon code as well as the parameters
   used during compilation of the code: different functions and parameters will result
   from different input. Some functions are 'built-in' but only appear when needed as
   per compiled-in features – a list of which is available with lexon -h. The functions
   are not given in a specific order of execution but as listed in the lexon source.

   These node modules have to be installed once:

   $ npm install serialize-javascript
   $ npm install tar
   $ npm install nodemailer
   $ npm install prompt-sync

   Parameters below are marked with double angle brackets << >> for the respective
   required caller. If the role is defined earlier, it can only be performed by this
   person. (But remember that this entire setup is trustful: anyone can manipulate
   anything about this contract. Though they cannot sign it or change the signed log.)
   If the role is not defined earlier, the call makes the role be assigned to the
   person named for the call. Some functions can be called without naming a caller.
   Some clauses of the original lexon source will not appear below. Namely, those
   that have no permission phrase, wherefore they are regarded as internal.

   The main contract system is initialized by loading the module and instantiating:

   $ node
   > contract = require("./escrow.jsx");
   > escrow = new contract(<<payer>>, <amount>, <payee>, <arbiter>, <fee>);

   Remember to reset node's module cache each time you edit and recompile your code:

   > delete require.cache[require.resolve('./escrow.jsx')];

   These are the state progress functions that allow to interact with the contract:

   > escrow.pay_out(<<arbiter>>)
   > escrow.pay_back(<<arbiter>>)

   state changes of the contract can be listed, e.g. actions performed by
   a party to it, or agents who are assigned privileges. In case hash chains
   or signatures are used, they are visible in this log. The log is stored in
   in the file 'log'.

   > escrow.history()

   The complete contract state can be saved to disk and re-loaded at a
   later point in time. This serves to continue work after stopping and
```

```
   restarting node; or to send the entire contract system and its current
   state - which can include hashes and signatures - to another party,
   who may perform the next steps.

   > escrow.persist()
   > escrow.load()

   The contract code, state and log can be bundled into one file to exchange
   or archive it:

   > escrow.bundle()
   > escrow.unbundle()

   The contract code, state and log can be sent to a counterparty. This
   requires configuring an email account in the file 'config'.

   > escrow.send()

   Keys for signing log entries are expected on-file, by default named after
   the actor, with the extension .key. For demo purposes, key files can be
   created using this utility function:

   > escrow.create_key(name, passphrase)
*/

var fs = require('fs');
var crypto = require('crypto');
var serialize = require('serialize-javascript');
var prompt = require('prompt-sync')();
var tar = require('tar');
var nodemailer = require('nodemailer');
var last_caller;
var last_passphrase;

/**
 **
 ** Main Escrow contract system
 **
 **/

module.exports = class Escrow {

    constructor(payer, amount, payee, arbiter, fee) {

        let main = this;

        /* object members: skip for restoring serialized object */
        if(typeof payer !== 'undefined') {
            this._escrow = 0;
            this.payer = payer;
            this.payee = payee;
            this.arbiter = arbiter;
            this.amount = amount;
            this.fee = fee;
            this.logname = 'log';

            /* start log - overwrites previous by same name */
            fs.writeFileSync(this.logname, "Lexon log " + (new Date).toLocaleString('en-US') +
"\n", ()=>{});
            this._pay(caller, this.payer, 'escrow', this.amount);
            this.log(payer, "✓ Payee appointed");
            this.log(payer, "✓ Arbiter appointed");
            this.log(payer, "✓ Fee fixed");
        }

        /* restore object from file - must be below class definition */
        if(typeof payer === 'undefined') {
            console.log("> restore from file 'state'");
            var data = fs.readFileSync('state', ()=>{});
            var live = eval('(' + data + ')');
            Object.assign(this, live);
        }
    }

    /* Pay Out clause */
    pay_out(caller)  {
        if(caller == this.arbiter) {
            this._pay(caller, 'escrow', this.arbiter, this.fee);
            this._pay(caller, 'escrow', this.payee, this._escrow);
        } else {
            return 'not permitted.';
        }
        return 'done.';
```

```
    }

    /* Pay Back clause */
    pay_back(caller)  {
        if(caller == this.arbiter) {
            this._pay(caller, 'escrow', this.arbiter, this.fee);
            this._pay(caller, 'escrow', this.payer, this._escrow);
        } else {
            return 'not permitted.';
        }
        return 'done.';
    }

    /* built-in convenience function to view state change log. */
    history() {
        fs.readFile(this.logname, (e,d)=>{console.log(d.toString())});
    }

    /* built-in serialization and storage of entire contract system state. */
    persist() {
        console.log('> persisting');
        var data = serialize(this, {space: 4});
        fs.writeFileSync('state', data, ()=>{});
    }

    /* re-instate entire contract system from serialized file store */
    static load() {
        return new Escrow();
    }

    /* built-in tar-balling of code, log and state. */
    bundle() {
        console.log('> bundling into contract.tgz');
        tar.create({gzip:true, file:'contract.tgz'},
          ['escrow.lex', 'escrow.jsx', 'state', 'log', 'INSTRUCTIONS.TXT']);
    }

    /* built-in untar-balling of code, log and state. */
    static unbundle() {
        console.log('> unbundling contract.tgz');
        tar.extract('contract.tgz');
    }

    /* built-in email sending of code, log and state. */
    send() {

        this.persist();
        this.bundle();

        console.log('> sending via email');
        var receiver = prompt('enter receiver address: ');

        var config = fs.readFileSync('config', ()=>{});
        var email = eval('(' + config + ')').email;
        console.log(email);

        var transporter = nodemailer.createTransport({
            service: email.service,
            auth: { user: email.user, pass: email.pass }});

        var mailOptions = {
            from: email.from,
            to: receiver,
            subject: email.subject,
            text: email.text,
            attachments: { path: './contract.tgz', contentType: 'application/gzip' }};

        transporter.sendMail(mailOptions, function(error, info){
            if (error) {
                console.log(error);
            } else {
                console.log('> email sent: ' + info.response); }});
    }

    /* built-in logging of state changes. */
    log(caller, msg) {
        console.log(msg);
        let stamp = (new Date()).toLocaleString('en-US');
        var entry = `⏀  ${stamp} ✦ ${caller} ${msg}`;
        var passphrase = this.sync_passphrase(caller);
        var pem = fs.readFileSync(caller + '.key');
        var key = pem.toString('ascii');
        var sign = crypto.createSign('RSA-SHA256');
        sign.update(entry);
```

```
        var sig = sign.sign({ key: key, passphrase: passphrase }, 'hex');
        fs.appendFileSync(this.logname, `${entry} ❇ ${sig}\n`);
        let pay = fs.readFileSync(this.logname);
        let hash = crypto.createHash('sha256').update(pay);
        fs.appendFileSync(this.logname, '⧉ ' + hash.digest('hex').substr(0, 12) + " ");
    }

    /* built-in password query for private key file, with cache. */
    sync_passphrase(caller) {
        if(!caller) process.exit('no caller information');
        if(caller == last_caller) return last_passphrase;
        last_caller = caller;
        return last_passphrase = prompt('enter pass phrase for ' + caller + ': ', {echo: ''});
    }

    /* built-in convenience function to create keys for users. */
    static create_key(name, passphrase) {
        const { publicKey, privateKey } =
            crypto.generateKeyPairSync('rsa',
                { modulusLength: 2048,
                    publicKeyEncoding: { type: 'spki', format: 'pem' },
                    privateKeyEncoding: { type: 'pkcs8', format: 'pem', cipher: 'aes-256-cbc',
                       passphrase: passphrase }});

        fs.writeFileSync(name+'.key', privateKey);
        fs.writeFileSync(name+'.pub', publicKey);
        return true;
    }

    /* built-in pay message */
    _pay(caller, from, to, amount) {
        this.log(caller, `➡ system message: pay ${amount} from ${from} to ${to}.`);
        if(from == 'escrow') main._escrow -= amount;
        if(to == 'escrow') main._escrow += amount;
    }
}

/* end */
```

Source 12 – Lexon compilation example (Javascript, all auxiliaries)

DEPLOYING TO ETHEREUM

Deployment to Ethereum goerli:

- Obtain an <API KEY> from *https://www.alchemy.com/*.
- Receive testnet Ether from *https://goerlifaucet.com* to the address of <PRIVATE KEY>.
- Solidity code is expected in the clipboard on Mac.
- Alternatively, use *./lexon --sol -o contracts/Escrow.sol escrow.lex*

Hardhat is used and (public!) accounts 0 and 1 of the hardhat node are given as parameters to deploy().

```
$ mkdir escrow; cd escrow
$ npm install --save-dev hardhat @nomicfoundation/hardhat-toolbox
$ mkdir contracts; pbpaste > contracts/Escrow.sol
$ echo 'require("@nomicfoundation/hardhat-toolbox");
  module.exports = { solidity: "0.8.17", networks: { goerli: {
  url: "https://eth-goerli.g.alchemy.com/v2/<API KEY>",
  accounts: ["<PRIVATE KEY>"] } } };' > hardhat.config.js
$ mkdir scripts; echo 'async function main() {
  const Escrow = await hre.ethers.getContractFactory("Escrow");
  const escrow = await Escrow.deploy("0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266",
  "0x70997970C51812dc3A010C7d01b50e0d17dc79C8", 10n**18n); await escrow.deployed();
  console.log(`deployed to ${escrow.address}`); } main();' > scripts/deploy.js
$ npx hardhat --network goerli run scripts/deploy.js
```

EXAMPLE INTERACTION

The live interaction with Javascript code created from a Lexon text[102] can look like follows. Note that the manual calls of the functions in the terminal is intended to demonstrate how these functions can be used by a frontend. It is also helpful for testing and research but not a production scenario.

• Start node and instantiate the Financing Statement.

```
$ node
>> contract = require("./statement.jsx");
[Function: UCCFinancingStatement]
```

• Instantiate the Financing Statement. Roles are given generic names in this example, *FILER* for the filer etc. The names are relevant subsequently for the role to identify itself when initiating an action. The name is then used to find the private key file that is used to sign log entries.

```
> statement = new contract("FILER", "OFFICE", "DEBTOR", "BANK", "TRACTOR");
✓ Filing Office fixed
```

• The program will ask for a pass phrase to read the private key for the *FILER* expected in *FILER.key*:

```
enter pass phrase for FILER:
✓ Debtor fixed
✓ Secured Party fixed
✓ Collateral fixed
```

• It then dumps the created state:

```
UCCFinancingStatement {
  financing_statement: null,
  file_number: null,
  initial_statement_date: null,
  filer: 'FILER',
  debtor: 'DEBTOR',
  secured_party: 'BANK',
  filing_office: 'OFFICE',
  collateral: 'TRACTOR',
  digital_asset_collateral: null,
  reminder_fee: null,
  continuation_window_start: null,
  continuation_statement_date: null,
  continuation_statement_filing_number: null,
  lapse_date: null,
  default_: null,
  continuation_statement: null,
  termination_statement: null,
  termination_statement_time: null,
  notification_statement: null,
  logname: 'log'
}
```

• A new statement, named *FN-890*, is being certified. The role impersonated in this instance is *OFFICE*. The program prompts for the passphrase to decrypt the private key found in file *OFFICE.key:*

```
> statement.certify("OFFICE", "FN-890");
✓ File Number certified
enter pass phrase for OFFICE:
'done.'
```

[102] The *U.C.C. Statement* example discussed in *Reyes*, ibid. For more context, see *https://lexon.org/reyes.html*

• The *filing date* is set by the *OFFICE*. The passphrase is re-used implicitly.

```
> statement.set_file_date("OFFICE");
✓ Initial Statement Date fixed
'done.'
```

• Ditto for the *lapse date* and the start of the *continuation*.

```
> statement.set_lapse("OFFICE", new Date("4/1/25"));
✓ Lapse Date fixed
'done.'
> statement.set_continuation_start("OFFICE", new Date("4/1/24"));
✓ Continuation Window Start fixed
'done.'
```

• Now the *BANK* certifies that it is paying a *fee*. Being a *trustful* example, the program does not facilitate a transaction itself but merely prompts the real world to make this transfer.

```
> statement.pay_fee("BANK", 2000);
➡ system message: pay 2000 from BANK to escrow.
enter pass phrase for BANK:
'done.'
```

• The *OFFICE* sets the language of the notification statement. As there was a switch in roles, the passphrase is queried again.

```
> statement.notice("OFFICE", "be notified!");
✓ Notification Statement fixed
enter pass phrase for OFFICE:
'done.'
```

• The *OFFICE* now *sends* a notification. Note that we are operating on one concrete instance of the Financing Statement that handles one form. The notification is going out to the role set as the *FILER*. Because this is a trustful program, it prompts the user with the action point, writing it to screen.

```
> statement.notify("OFFICE");
➡ system message: send message «be notified!» from OFFICE to DEBTOR.
'done.'
```

• The *DEBTOR* makes a payment, at least attests that this is so.

```
> statement.pay_escrow_in("DEBTOR", 1000000);
➡ system message: pay 1000000 from DEBTOR to escrow.
enter pass phrase for DEBTOR:
'done.'
```

• In an alternate scenario, we skip forward in time and have the *BANK* assert that there was a failure to pay. In the intended logic of the Financing Statement, this announcement is all that it takes. The bank does not have to prove it immediately for the default mechanism to kick in.

```
> statement.fail_to_pay("BANK");
✓ Default declared
enter pass phrase for BANK:
'done.'
```

• Upon the declaration of the bank, the *OFFICE* can sign off on the bank's desire to take possession of the posted collateral:

```
> statement.take_possession("OFFICE");
➡ system message: pay 1000000 from OFFICE to BANK.
enter pass phrase for OFFICE:
'done.'
```

• In yet another scenario, the bank must file for continuation after the required time has passed:

```
> statement.file_continuation("BANK", "continue!");
✓ Continuation Statement filed
enter pass phrase for BANK:
'done.'
```

• The *OFFICE* can declare when the statement will lapse:

```
> statement.set_continuation_lapse("OFFICE", new Date("4/1/23"));
✓ Continuation Statement Date fixed
enter pass phrase for OFFICE:
'done.'
```

• The *BANK* terminates the Financial Statement when the loan has been repaid (out-of-band):

```
> statement.file_termination("BANK", "terminate!");
✓ Termination Statement filed
enter pass phrase for BANK:
✓ Termination Statement Time certified
'done.'
```

• The *OFFICE* then releases the escrow to the *DEBTOR* ...

```
> statement.release_escrow("OFFICE");
➡ system message: pay 1000000 from OFFICE to DEBTOR.
enter pass phrase for OFFICE:
'done.'
```

... releases the reminder fee to the *BANK* ...

```
> statement.release_reminder_fee("OFFICE");
➡ system message: pay 2000 from OFFICE to BANK.
'done.'
```

... and finally terminates the statement.

```
> statement.terminate_and_clear("OFFICE");
'done.'
```

For an in-depth legal discussion of this example, see asst. prof. Carla L. Reyes, 2021, *Creating Cryptolaw for the Uniform Commercial Code – https://papers.ssrn.com/sol3/papers.cfm?abstract_ id=3809901*

For technical background and updates, *https://lexon.org/reyes.html*

EXAMPLE LOG

The log forms the basis for *microchains*.[103] Each input creates a log entry in the format:[104]

```
⧉ <hash> ⌽ <timestamp> ✦ <role> ✓ <action> ❉ <signature>
```

After the above test, the signed and hashed log of transactions will look like this:

```
Lexon log 3/22/2021, 1:34:19 AM

⌽ 3/22/2021, 1:34:19 AM ✦ FILER ✓ Filing Office fixed ❉
5a22046438c33aa138fd44486c137655c37d46f3b6bbdac166d6daae7cf3abf6fcc94f030dbc6f3f95ce8a5c5609202d
d3728676b84e538c9bfd47fe3c0e595dd26f0e0ac1c3f1691ada598cef4e299d0b60191c9128ca74aa66594e6acba5ff
e57016798ccb9ad177c666199dbd1707f0b18a3fa2777f66538596f28bedfb05539baf2e4f72302958b5557d42c030cc
1111fb799c2bae2fe3326d98479f5fda1465a87a1d7bcc2792142d49aedc4ea7eb354d5c07af89821d54d3af163358b7
0765b55e187cd9c15102b2ebffef1b234f3e9776c4c0b367992a112ee6fd3ee4c650c9bc80b423cd25dbc0b2ac0ada01
c18b8c972ea5807ccc4821463bbf421b

⧉ ced930bc47be ⌽ 3/22/2021, 1:34:22 AM ✦ FILER ✓ Debtor fixed ❉
098c119ecde9f2d87ef2e78ba2fb6dfd2a35234b7be8c7f1caa3d2398e0a78f88a9a138b3ecf6b3bd002710cc949168c
edc5054485dd0c8c473ac879d1cb9fdc1528a8120edb8f1dafa3d4bc945f18bccf8f2d5d4fcbdce3d47c68b51509e9c2
9e5f772343d0b54087e4045d1f9a03da2cad56e0bd4427ed54e30b59aced9371d30f99bee4980d54df40b97fa64465b0
c46e471e280795b61de937d8c5af9e93f961f2ecb0e3588abad12db1b2e7aac73a13e919325f595563089b1b615df0d4
a78643d01ebe4968f195f61191737e7fb7af6a7f06297ead727bcd9251fa4985a978d9a02df047192e6ca7671157907e
29265e4337102982945714930O1df5e1

⧉ eb5c28b69b4b ⌽ 3/22/2021, 1:34:22 AM ✦ FILER ✓ Secured Party fixed ❉
5b89a88bfec5fcac46cbfb3b541a408ad4160c52e95d5dd51006cbdb3f0603d1850a1cc0853dcb4245b047e626ae4b99
704fd0a75c09c1bc4c539b0b631f5862a3275599006e4436e65f76a013b62204b63d5747882180faa98884b5b1a0a893
0bd0e6a5339be7b5fcd148d690c840e18c60d8092c88ed8e0387cff5cb0a25cb0ed8ea90cca7fed2425aa830add7b4c8
f4164476fd0f19cbadee4d7dd7b0d2c76bf5330232981432820e43a9e6af14a5e11c69812e78cad9e43d53d58f0281c2a
8e180dd2c3b6ae3a851e38dded02c1be6c144a40399d3beb9d66ae4d8a0654bc1c4d94243fcf347a675fdfebab024d0e
7951407817f8678c87c42c612804c57f

⧉ f3b21bde6076 ⌽ 3/22/2021, 1:34:22 AM ✦ FILER ✓ Collateral fixed ❉
6ff7ab169e49f2f574c7f13497a0c134eb5987476a6fcc35515b60775cdaef75afa1bcdae06be7965921cf36da228aec
1b195f21b4696249d327a6799efca1d5dd176ec95050407de40427dbdf5af8a6d4a6e9eb88271717c51d7cded996fc93
1be7e1c932716c26ee3cfbb2281579061342c9101e4bf66974ad85e36c6dcca156fd1c6040f5f2925e4ae77e3b9b2c8c
7644020f86971d958600b8a17e2385f6d5d8c3c505f649d1a97852116869f2bca53fa172f63d05b88eda1f312620bab5
a90bf35334dc4a3890f737a7ad950791e1c49eeabd5b64c51a3a6046cada2421e18726643bbff3a7fe63ce18b15af033
2972635caecac4bafc0659d4f71d3675

⧉ 448d9a8e7ba5 ⌽ 3/22/2021, 1:35:12 AM ✦ OFFICE ✓ File Number certified ❉
6d41057ea47910a576a0c2a686fdf73855b581199476309d90a207fdc3a8da70ac9bb71ab3bc5e2f9a2fc368305a6f4b
b14dee00e590e7dd5265c0ef847e8f9f5a9f7352f55eed0eb2a9a62365d344df646240cf4cfcde1cc75c85b8a26b2d18
b66908089372e3f5ba8e09117d4aa07cc54d105ffa37a8f623814040b145821530c75cdea45a440e00960bbc5f118751
32d1d603723fa28fa0415cb709bee6d0c75b1b390d07545614abdb111434970b5ebb43c974ebcfaf840a7424d6109c5b
b1905e4bb0faf4ebdb5a98ee93f5783f6732fb2e720dd52a9b6095c8547570224c57128ba2487c5443a3b888b6e03d63
c771250ecbc0b09f64bb47de04f90499

⧉ caaa3095898b ⌽ 3/22/2021, 1:35:28 AM ✦ OFFICE ✓ Initial Statement Date fixed ❉
3c444e5d473349ebe33471a0409d87567eff9fa4eae1171e8fa24dd2fb0708f2275a2706a141c16970aaed96b9d6adef
16dbc70c81708b2b4d16035e77bf550d6f93d12d4367b3724b4d81ce66f519a351b15a9b656c176b2abba539b277f0dc
747544b59397d01f7301327252e22a298a3cd22a6d31073b762e243d6a4332249384ea3c492dfbacef5be0efe34f2641
3876f2977f4c2a3249c5a44cd11ad62814dda2ab365413fe4d0483d1f069a6e27bf661f2a123b20470bd0f0bcbde699a
292b90e8beffc557ba391f8cebc5b7ef851bbb4dced5364a573fd0ceed306a05cf742fe7492297365b61304fa511f7d0
69b0be6f3bc046cfb71a151b726f680a

⧉ cf770ab8947d ⌽ 3/22/2021, 1:35:49 AM ✦ OFFICE ✓ Lapse Date fixed ❉
6107eff31587d8a1d75b0923ed71469bba40181bd8852703ea8f237d2185acbdee5b3052716537cd3b8c1a7382ec1717
83c324a3467e2ebc937b580d41fbffbb78ab0e100c29afabdccd7dda0a3985159f74cee3387c50d0834a801d82a93e64
8a91aac1203cd7a4ab9b45b4f3c5b21313a9199d75b160f371ea4fdf1a577d411859c2dc33355af1f0544906d679b41a
989b90bd2248a6c81758dc4a345f6fdd08449c44b0666e721b5948bbe770e9c31a8574d3a1fb50959452fdff90989dca
3c44e0ff6526926099e70af07cca72f840c2ba01d3d36f894d5ca7af491d0a0a3169c50f95fb4438ed17871c9034a275
139c64574c3528b8e54e66e5d6e67547
...
```

[103] Cf. *The Micro Blockchain*, pg. 8.

[104] When used with the options *signed* and *hashed*, cf. *History*, pg. 8.

EXAMPLE ABSTRACT SYNTAX TREE

This is a part of the *abstract syntax tree* (AST) that the compiler creates internally when processing the grammar and text discussed in chapter *Grammar*, pg. 4. It reflects natural language grammar rather than programming logic. Such a tree can be created from any Lexon text using the *flat tree* options.

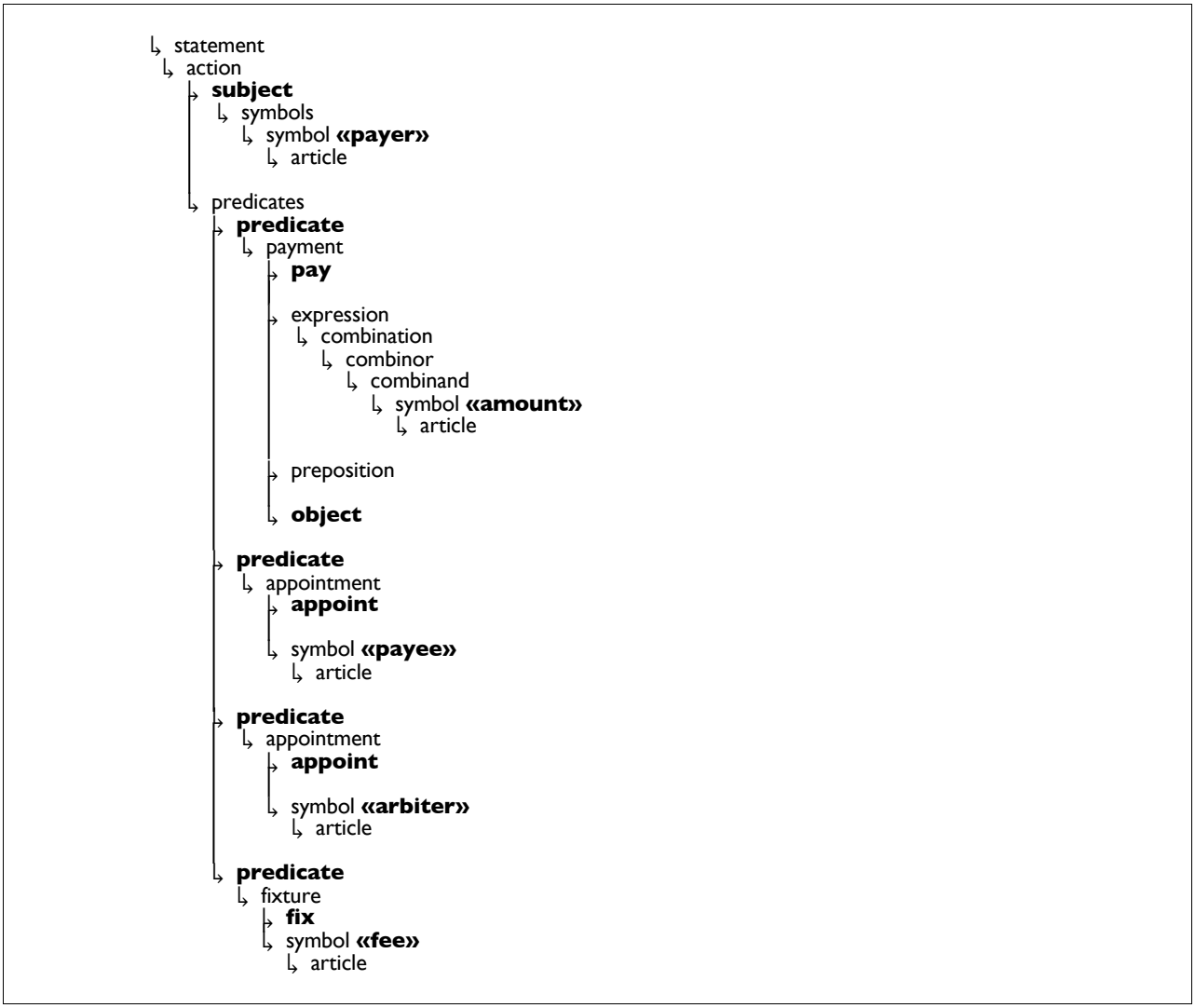

Figure 10 – Partial example of a Lexon abstract syntax tree

To create such a tree for your own Lexon text, at *https://lexon.org/compiler* paste it into **a.** (see *Figure 1*, pg. 5), check options *flat* and *tree* in ***d.***, click the compile button ***b.*** for the tree to appear in ***c.***

There are fine-grained options for highlighting specific elements of the tree: *color*, *highlight* etc.

ELECTRONIC BILLS OF EXCHANGE

Lexon gates and account abstraction can be used to implement electronic instruments compliant to the 2022 updates of the UCC. The Uniform Commercial Code (UCC) is the U.S. model trade law that states can adopt, and mostly do, with the goal of harmonizing the laws of sales and other commercial transactions across the United States.[105] *"The 2022 amendments to the Uniform Commercial Code address emerging technologies, providing updated rules for commercial transactions involving virtual currencies, distributed ledger technologies (including blockchain), artificial intelligence, and other technological developments. The amendments ... add a new Article 12 addressing certain types of digital assets defined as "Controllable Electronic Records" (CERs). The amendments provide new default rules to govern transactions involving these new technologies ..."*[106]

[105] Cf. *https://en.wikipedia.org/wiki/Uniform_ Commercial_ Code*

[106] *https://www.uniformlaws.org/committees/community-home?CommunityKey=1457c422-ddb7-40b0-8c76-39a1991651ac*

PROGRAMMABLE MONEY

This is an excerpt from the 2018 presentation *Programmable Money* at the *Cleveland Federal Reserve*, with a future head of the SEC in attendance. It explains a blockchain-based concept of *programmable money*. The Lexon Token achieves programmability through *smart accounts* (See *Extensibility,* pg. 14).

Towards Programmable Money

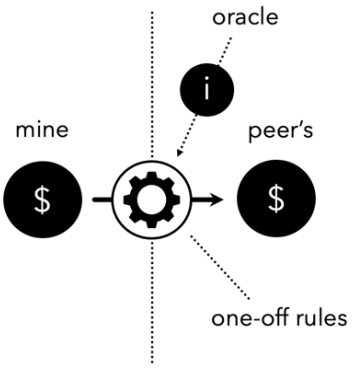

Figure 11 – One-off transfer rules

With programmable money the idea is that the rules do not only control my outlet, one-off, at exactly the time and place – and only then – that money leaves me (or not).

That could be (and often is) implemented locally centralized, even if the *transport* was P2P.

You don't need a blockchain for that. Even if the rules are triggered from the outside, which in blockchain scenarios is called an *oracle*.

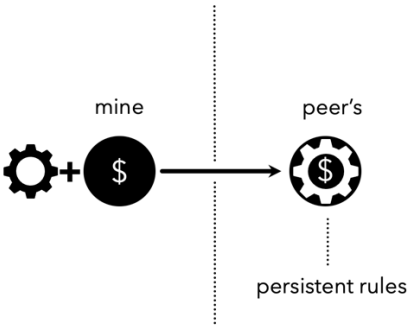

Figure 12 – Persisting, 'baked-in' rules

... instead, programmable money has rules 'baked into the coin.' Rules that I built in and that travel with the money even after I have given it away.

Rules that can identify peers and oracles, 'have memory' of past facts and can become arbitrarily complex.

And of course, they are still simply smart contracts.

The Concept

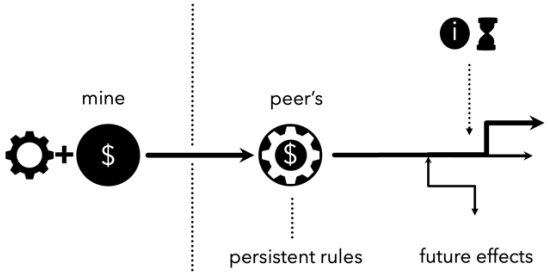

Figure 13 – Future effect of persistent rules

... the major difference is that information that arrives later (after my release of the coin), can still be included in the execution of the rules. This makes the rules programs.

And my rules can determine the way of the coin long after I paid it out.

Supply Chain: Safe Contracting

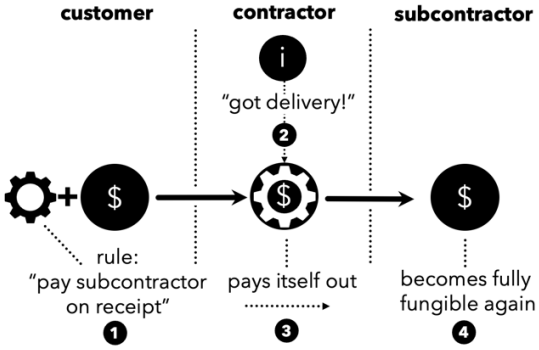

Figure 14 – Safe subcontracting

For supply chains, programmable money can protect subcontractors against being squeezed out or suffering from delayed payments. When the contractor receives a delivery from the subcontractor, the receipt triggers the payment.

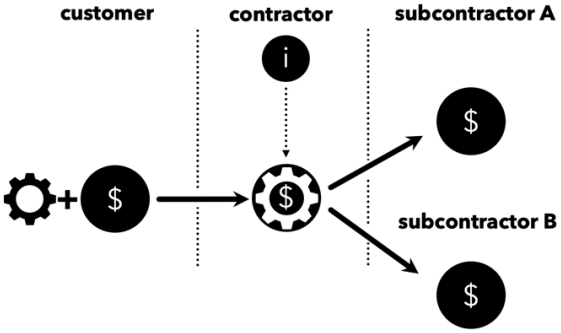

Figure 15 – Complex persistent rules

Supply Chain Payment Hierarchies

The payout can be more interesting than just a binary signal of 'pay' or 'not'.

It can let money flow to different parties, or back to the customer on arbitrarily complex conditions.

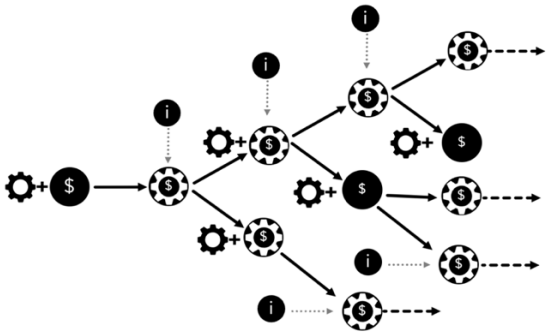

Figure 16 – Hierarchy of persistent rules

... the logic can be nested, programmed by different parties, layered across different concerns 'into the same coin.'

Eventually it can automate, speed up, reduce frictions and costs, and opportunities to seek rent, across the entire supply chain.

Cost Reductions for Supply Chains

- Legal cost for contracting
- Cost of suits (to company)
- Cost of legal system (to public)
- Losses through defaulted claims
- Cost of regulatory compliance
- Cost of regulating
- Cost of money transfer
- Cost of financing
- Cost of corruption
- Cost of auditing

For the original, longer presentation, see *https://lexon.org/programmable-money-2018.pdf*

✧

ROBOTIC LAWS

The science fiction author Isaac Asimov coined the term *robotic laws*[107] in the 1940s for the science fiction universe over-arching his short stories and novels. He evolved them over time and showed how easily they can become self-contradictory or exploitable by a rogue machine.

The Laws are so often quoted and well known in nerd culture that they will have informed many discussions about consequential, real-world decision-making algorithms. They are cited here to indicate one direction in which lawmaking will have to think – and write – in Lexon, to create laws and regulations that can be directly and verbatimly implemented into autonomous machines.

This is not science fiction but both possible today and indispensable tomorrow. Because Lexon is based on symbolic AI, it complements generative AI, adding agency, transparency, and reliability.

First Law	*A robot may not injure a human being or, through inaction, allow a human being to come to harm.*
Second Law	*A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.*
Third Law	*A robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.*

[107] Isaac Asimov, 1950, *I, Robot*, pg. 40.

TOKEN FUNCTION SIGNATURES

The token is implemented in Solidity. For summary descriptions see from pg. 11, for function details including log emissions (Solidity *events*), see *https://lexon.org/token-api*.

Basic Transfer

availableBalance(address a) public view returns (uint)
transfer_to_serial(uint serial, uint amount) external returns (bool)
transfer_to_name(string calldata name, uint amount) external returns (bool)
funds(address from, address to) public view returns (uint)
balanceOf(address account) external view returns (uint)
transfer(address recipient, uint amount) public returns (bool)

Approval

allowance(address owner, address spender) public view returns (uint)
approve(address spender, uint amount) public returns (bool)
transferFrom(address sender, address recipient, uint amount) public returns (bool)
increaseAllowance(address spender, uint addedValue) public returns (bool)
decreaseAllowance(address spender, uint subtractedValue) public returns (bool)
permit(address owner, address spender, uint256 value, uint256 deadline, uint8 v, bytes32 r, bytes32 s) public
nonces(address owner) public view returns (uint256)

Engage

engage() external
unlock() external
lockwait() public view returns (uint)
lockwaitOf(address a) public view returns (uint)

Sealing

seal() external
unseal() external
whitelist(address entry) external
delist(address entry) external

Serial & Name

register(address account) public
label(string calldata name) external

Multi-Signature

demand(uint requirement) external
add(address signer) external
remove(address signer) external
sign(address account, address to, uint amount) external
retract(address account) external

Account Abstraction

anticipate(address anticipated) external
share(address secondary) external
coshare(address secondary) external
unshare() external
revoke(address secondary) external

Avatar

mark(string calldata url) external returns (bool)

Email

publish(string calldata email) external returns (bool)

Subscribe & Feed

subscribe(address poster) external returns (bool)
post(string calldata message, string calldata attachment) public returns (bool)
unsubscribe(uint index) external

Message

message(address addressee, string calldata message, string calldata attachment) public returns (bool)

Commitment

commit(address committee, uint commitment, address arbiter, uint expiration) public
prolong(uint expiration) external
lower(address committer, uint reduction) external
raise(uint increase) external
end(address committer) external

Promise

make(address promissee, uint amount) external
forgive(address promissor) external

Cheques

form(address receiver, uint number) external pure returns (bytes32)
write(bytes32 hash, uint amount) external
good(address signer, bytes32 hash) external view returns (uint)
deposit(address signer, uint number) external

Voucher

voucher(uint nonce, address issuer, address claimant, uint tokens, uint deadline, uint lock_in)
    public view returns(bytes32)
convert(uint nonce, address issuer, address convertant, uint class, uint deadline, uint lock_in, bool force,
    bytes calldata signature) public returns (uint)
claim(bytes32 voucher) external

Reversibility

accede(address protected, address forum, uint deadline) external
extend(uint deadline) external
reduce(address revertee, uint reduction) external
release(address revertee) external
reverse(address revertee) external

Escrow

establish(address prospect, uint offer, uint ask) external
place(uint offer, uint tag, uint minimum) external
complete(address seller, uint amount) external payable returns (bool)
agree(address seller, uint amount, uint tag) external payable returns (bool)

Sponsoring

collect(address [ ] calldata protege) external
sponsor(address [ ] calldata receivers, uint amount) external
dropout() external
reward(address [ ] calldata protege) external returns (uint)

Burning

burn(uint amount) public
burnFrom(address account, uint amount) public

Digital Deeds

approve(address to, uint token_id) public
balanceOf(address owner) public view returns (uint)
getApproved(uint token_id) public view returns (address)
isApprovedForAll(address owner, address operator) public view returns (bool)
ownerOf(uint token_id) public view returns (address)
safeTransferFrom(address from, address to, uint token_id) public
safeTransferFrom(address from, address to, uint token_id, bytes memory data) public
setApprovalForAll(address operator, bool approved) public
supportsInterface(bytes4 interface_id) public pure returns (bool)
tokenByIndex(uint index) public view returns (uint)
tokenOfOwnerByIndex(address owner, uint index) public view returns (uint)
tokenURI(uint token_id) public view returns (string memory)
totalSupply() public view returns (uint)
transferFrom(address from, address to, uint token_id) public

Sale

price(uint total, uint offset, uint adjust) public pure returns (uint)
purchase() external payable

DAO

daofy(uint value) external
decentralize() external
join(address dao, uint pay) payable external
leave(address dao, uint pay) payable external
nominate(address candidate) external
perform(address proposal) external
propose(address lib0, address lib1, address lib2, address lib3, address lib4, address lib5) external
affirm(address proposal) external
count(address candidate) external

Gates

gate_set(Gate _gate, address keeper, bytes32 filter, uint obligation) payable external
gate_prepare(Gate gate, address keeper, bytes32 filter, uint obligation, uint fee, payable receiver) payable external
gate_activate(Gate gate, address gatee, bytes32 filter, uint obligation, uint fee, payable receiver) payable external
gate_close(address gatee, uint fee, payable receiver) payable external
gate_access(address gatee, uint cmd, address addr1, address addr2, uint n1, uint n2, uint n3) payable external
gate_info(address gatee, uint cmd, address addr1, address addr2, uint n1, uint n2, uint n3)
payable external view returns (string memory)

Table 3 – Token function signatures

GATE INTERFACE

See *https://lexon.org* for a forthcoming paper on gate programming and *https://lexon.org/gate-api*.

Operation

enter(address gatee, address keeper, bytes32 filter, uint obligation, uint fee, uint receiver)
payable external returns (bool)
bar(address gatee, uint fee, payable receiver) payable external returns (bool)
close(address gatee, uint fee, payable receiver) payable public returns (bool)
access(address gatee, address keeper, uint cmd, address addr1, address addr2, uint int1, uint int2, uint int3)
payable external returns (bool)
query(address gatee, address keeper, uint cmd, address addr1, address addr2, uint int1, uint int2, uint int3)
payable external view returns (string memory)
info(uint value) payable public view returns (string memory)

Events

onSending(address signer, address from, address to, uint amount) external returns (bool)
onReceiving(address signer, address from, address to, uint amount) external returns (bool)
onWriting(address signer, address account, bytes32 hash, uint amount) external returns (bool)
onSetting(address signer, address account, string calldata email) external returns (bool)
onMessaging(address signer, address account, address addressee, string calldata message) external returns (bool)
onPosting(address signer, address account, string calldata message) external returns (bool)
onCommitting(address signer, address account, address committee, uint commitment, address arbiter,
uint expiration) external returns (bool)
onProlonging(address signer, address account, uint expiration) external returns (bool)
onRaising(address signer, address account, uint increase) external returns (bool)
onLowering(address signer, address account, address committer, uint reduction) external returns (bool)
onEnding(address signer, address account, address committer) external returns (bool)
onMaking(address signer, address account, address promissee, uint amount) external returns (bool)
onForgiving(address signer, address account, address promissor) external returns (bool)
onAcceding(address signer, address account, address forum, address protected, uint deadline) external returns (bool)
onExtending(address signer, address account, uint deadline) external returns (bool)
onReleasing(address signer, address account, address revertee) external returns (bool)
onReducing(address signer, address account, address revertee, uint reduction) external returns (bool)
onReversing(address signer, address account, address revertee) external returns (bool)
onBurning(address signer, address account, uint amount) external returns (bool)
onTransferring(address signer, address from, address to, uint id, uint price) external returns (bool)
onAccepting(address signer, address from, address to, uint id, uint price) external returns (bool)
onLetting(address signer, address from, address to, uint id, uint fee) external returns (bool)
onReturning(address signer, address from, address to, uint id, uint fee) external returns (bool)
onJoining(address signer, address account, address dao, uint contribution) external returns (bool)
onLeaving(address signer, address account, address dao, uint take) external returns (bool)

Table 4 – Gate interface

INDICES

INDEX OF FIGURES

INDEX OF TABLES

INDEX OF SOURCES

