Lexon to Æternity

H. Diedrich
hd@lexon.org
2 June 23

ABSTRACT

This paper presents a workflow to create digital contracts. It describes the basics of the plain-text programming language Lexon; how to use the Lexon online compiler to translate controlled English into blockchain smart contracts; and the utility and sales mechanism of the Lexon Æternity Token, LÆX.

INDEX

Introduction ... 1
LANGUAGE ... 2
Approach ... 2
Application ... 3
Grammar ... 4
COMPILER ... 5
Operation ... 5
Example ... 5
Options ... 6
TOKEN ... 7
Utility ... 7
Sale ... 7
CONCLUSION ... 8
DISCLAIMERS ... 9
LICENSE ... 9
APPENDIX ... 10
Example Compilation ... 10
Lexon for Law ... 12
Abstract Syntax Tree ... 18
Robotic Laws ... 19
INDICES ... 20

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, central 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. Lexon provides what Leibniz was looking for: a way to *program law*,[4] and contracts. This empowers lawmakers and will reduce the cost of access to justice. It is a beautiful match 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. More importantly, to enable the use of smart contracts in business, Lexon makes them readable for judges. Yet, it might find broad 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 – the penultimate developmental step before computers can *reliably*[6] read any human text: *intelligent agents* programmed in Lexon solve real-world problems, are *unbiased*, excel in *transparency* and provide unparalleled *agency* to users – the well-known weaknesses of machine learning. *Digital contracts* written in Lexon elevate prose to a *speech-act* of *felicitous performative language*[7] when executed in a trustless environment: because of the unstoppable nature of the blockchain, these words become true by the act of uttering 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. From an AI point of view, an artificial judge is being built right into every digital contract; because a computer will provide a deterministic result, as the case may be. This long-sought device makes viable the very simple as well as the very complex.

---

[1] Leibniz' thesis is regarded as the beginning of computer sciences. For more on the history of Lexon, and Computational Law: *https://lexon.org/intro* & 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 *abstract syntax tree* (AST) that the Lexon compiler creates is Lexon's internal model of the meaning. See appx. *Abstract Syntax Tree*, pg. 18; cf. *Processing Meaning* in *Lexon*, ibid., pg. 89.

[4] Cf. Clack and Reyes, footnotes 21 and 22, pg. 3 and appendix *Lexon for Law*, pg. 12

[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 known challenge for machine learning.

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

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 really 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

The Lexon approach 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]

APPROACH

Lexon allows for the articulation of unambiguous prose[12] and the *deterministic* computation of logical results from it. Its grammar overlays natural language and higher order logic, in the way that Wittgenstein[13] demanded. For *artificial* domains – like law, finance, programming, or entertainment – this contributes to the quest for unambiguous, universal languages for philosophy and pure thought as envisioned by Leibniz, Wilkins, Frege, Russel, or Carnap.

Lexon achieves its result differently than was long supposed to be the way.[14] It arguably developed in a blind spot caused by the focus on the meaning of *words* that emanated from analytical philosophy and informed – and maybe hampered – the development of early, general artificial intelligence.[15] 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. 13]

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.[16]

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. A noun's common meaning can contribute to readability – but not to the specific meaning of the document. This may be surprising only because it does not conform to a naïve take on linguistics. But dropping the inherent meaning of nouns is not unusual:

---

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

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

[14] Cf. Wilkins 1668 proposal for a better way to write words – *https://archive.org/details/AnEssayTowardsARealCharacterAndAPhilosophicalLanguage* and *https://www.youtube.com/watch?v=TjdbrLxc3Ck*

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

[16] To make it concrete is a philosophical demand, too. Cf. W. James 'vicious abstractionism' in *The Meaning of Truth,* 1909, pg. 135.

Lexon shares this feature with mathematical formulas and any programming language where variable names are interchangeable; 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[17] understands is *transfer*. Which is unsurprising as this is the only act computers can perform: to transfer bits from one register to another. This verb anchors Lexon texts; everything else is qualifiers. Again unsurprisingly, this design 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[18] – 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[19]) was required to better describe natural language grammar, for Lexon to stand upon the shoulders of the giants who paved the way.

APPLICATION

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[20] 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[21, 22] 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*. For its advantages in transparency and accessibility, Lexon may become a mainstream programming language. Because 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.

But Lexon's home game are *digital contracts* for everyone, i.e., simple blockchain smart contracts that are legally enforceable agreements. They reach beyond Computational Law and add the unique feature of unbreakability to contracting, which in due time will have tremendous economic impact across all walks of life.

As a match to Lexon in the crypto world, the Æternity blockchain stands out, because like Lexon, it is designed and implemented with a focus on sound engineering and reliability; its motto 'for the masses' is reflected in the economic transaction costs that allow for low-cost, DIY Lexon contracting. And different from many other projects, the Æternity blockchain is a true, decentralized and common good. Finally, its fast blocktimes – thanks to its microblocks consensus mechanisms – carry web3 programming over the threshold where wait times are short enough that users can seamlessly interact directly with the chain. Æternity's speed and scalability make it a tool of choice for AI for an additional reason: because AI suffers from the trash-in-trash-out syndrome, blockchains are understood to play a central role in future AI architectures as reliable shared data stores.

---

[17] Lexon's vocabulary is out of the scope of this paper. A playful interactive device to inspect it can be found at *https://lexon.org/vocabulary*. Also see the forthcoming 2nd edition of the *Lexon Bible*, Amazon.

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

[19] *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.

[20] An expectation articulated by law scholars.

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

[22] 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.[23]

Lexon Grammar Form

Lexon grammars are defined in Lexon Grammar Form (LGF),[24] which is similar to Backus-Naur Form (BNF),[ibid. 19] 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:[25]

```
sentence:
  subject predicates separator
  or subject condition predicates separator
  or subject condition "," predicates separator
  or subject condition ":" predicates separator
  or subject condition "," ":" predicates separator
```

Sentence Structure

Lexon's grammar realizes the English natural language sentence structure of *subject, predicate, object*. That Lexon's internal model reflects this 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*.

LEX Escrow.	**Head**
"Payer" is a person. "Payee" is a person. "Arbiter" is a person. "Fee" is an amount.	**Definitions**
The Payer pays an Am the Payee, appoints the	**Recital**
CLAUSE: Pay Out. The Arbiter may pay fi themselves, and afterwa the escrow to the Payee	**Clause**
CLAUSE: Pay Back. The Arbiter may pay fi themselves, and afterwa remainder of the escrov	**Clause**

Source 5 – Lexon document structure

This order makes it harder to write ambiguous agreements. It reflects a common sequence of the parts of a paper contract.

The internal model that the compiler creates during the translation is shown in appendix *Abstract Syntax Tree*, pg. 18. It visualizes the relationships that the compiler actually 'understands' from the sentence in *Source 4*, expressing a linguistic structure as a binary tree.

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.*

[23] 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/intro*.

[24] For more on LGF see *https://lexon.org/intro*.

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

COMPILER

The Lexon compiler[26, 27] accepts text adhering to the *controlled grammar* described above and transposes this natural-language code to the functional 3rd generation blockchain programming language *Sophia*. Lexon Æternity Tokens[28] provide metered access to the online Lexon compiler.

ÆTERNITY

Æternity is a layer-1 blockchain that is particularly well-crafted and economic to use.[29] The Lexon online compiler is a web3 æpp interacting with the Æternity blockchain to help creating new web3 æpps for this chain. Its payment mechanism, the LÆX token, is implemented as a smart contract running on Æternity.

SOPHIA

Sophia[30] is the language that smart contracts are programmed in for the Æternity blockchain. It is designed to be as clear and safe as possible. Lexon users however, do not need to learn Sophia to be able to create smart contracts.

OPERATION

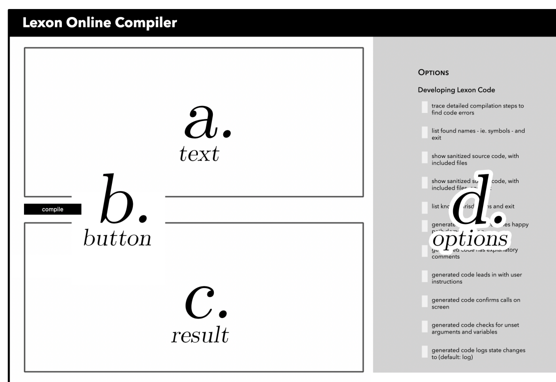

Figure 1 – Compiler screen at lexon.org/sophia

The online compiler is operated as follows:

a.	*text*	paste Lexon text into ***a.***
b.	*compile*	click compile button ***b.***
c.	*result*	the resulting Sophia code is shown in ***c.***
d.	*options*	to execute special functions, discussed below,[31] 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 Lexon text in ***a.*** into this Sophia code and show it in ***c.***:

```
@compiler >=6

main contract Escrow =

  record state = {
           payer       : address,
           payee       : address,
           arbiter     : address,
           amount      : int,
           fee         : int
       }

  entrypoint init(payee : address,
    arbiter : address, fee : int) = {
            payer = Call.caller,
            payee = payee,
            arbiter = arbiter,
            amount = Call.value,
            fee = fee
       }

  stateful function transfer(to : address,
    amount : int) =
      Chain.spend(to, amount)

  function permit(authorized : address) =
    require(Call.caller == authorized,
      "no access")

  stateful entrypoint pay_out() =
    permit(state.arbiter)
    transfer(state.arbiter, state.fee)
    transfer(state.payee,
      Contract.balance)

  stateful entrypoint pay_back() =
     permit(state.arbiter)
     transfer(state.arbiter, state.fee)
     transfer(state.payer,
       Contract.balance)
```

Source 6 – Compilation example (barebones)

The options ***d.*** controlling the output in ***c.*** are described below.[31]

The above code can be deployed to the Æternity blockchain. It is optimized for demonstration purposes: it is short, not cluttered with comments, handling of fringe cases, nor extras like logging to the chain receipt log. For a more production-ready compiler output from the *same* plain-text input, see appendix *Example Compilation*, pg. 10. It adds all the elements that *barebones* tells the compiler to leave out.

---

[26] 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.

[27] Online at *http://lexon.org/sophia*

[28] See *Token*, from pg. 7, and *http://lexon.org/laex*

[29] See *https://aeternity.com/aeternity-101*

[30] See *https://aeternity.com/#sophia*

[31] See *Options*, pg. 6.

© 2023 Henning Diedrich

OPTIONS

Settings for the compilation process are made in the compiler screen at *https://lexon.org/sophia* (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.

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.

*version**

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

*verbose**

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

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.

precompile

Show sanitized – *pre-compiled* – source code in ***c.*** and no compilation result. This shows the library[32] 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.

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.***

*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 Sophia.

*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 reflect the specific Lexon code at hand, listing all relevant core functions and their parameters.

harden

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

log

Write events to the global Aeternity receipts log.

all auxiliaries

The generated code features the options: *comments, instructions, harden* and *log.*

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.[33]

keywords

List in ***c.*** the keywords – the vocabulary – understood from an LGF[34] grammar provided in ***a.***

bnf

Produce BNF [ibid. 19] 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.

* option more likely of interest for beginners.

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

[33] See *http://lexon.org/intro* on creating grammars.

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

TOKEN

The Lexon Æternity Token, LÆX, provides access to the Lexon online compiler.

UTILITY

The token functions as prepaid voucher for the online compiler. It buys one translation of a Lexon text of arbitrary length into the Æternity blockchain language Sophia.[35]

The token is AEX-9-compatible[36] and easily accessible through AEX-9-compatible wallets like *AirGap.*[37]

SALE

Purchase

The token can be purchased for Æ at *https://lexon.org/laex.*

Use

Tokens can immediately be used with the compiler but transferred out only after 30 days.

Transacting

Tokens can be transferred using AEX-9-compatible Æternity wallets. Other specific token mechanisms – e.g., compilation, AEX-9 *approval* – can move tokens, *even if in cold storage.*

Promotion

First-time visitors have 10 compilations free. A purchase of tokens is offered automatically after the 10th 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.*

Sponsoring

Tokens can be sponsored to other accounts, which can use but not transfer the tokens.

Cap

The supply is capped at 100 million tokens. The sale can be paused, effecting a temporary soft cap.

Price

The price for Lexon Æternity Tokens increases with the amount of tokens issued.[38] This serves as load protection for the online compiler.

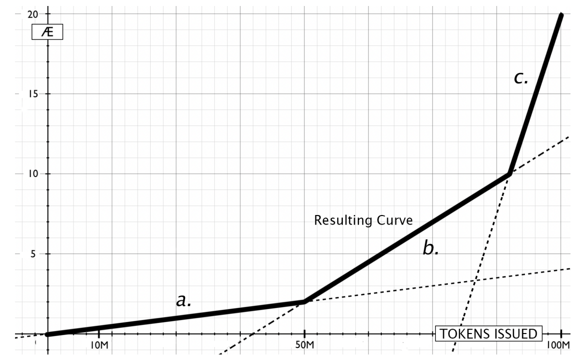

Figure 2 – Token sale price based on tokens issued[38]

Current Price

The current price, in Æ, can be learned at *http://lexon.org/laex.* The page lists the price for the *next* token sold and allows the querying of the total price for a planned purchase, e.g., how many tokens one would receive for 100 Æ.

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 Æ spent, because the same amount of Æ buys progressively fewer tokens, which drives the price progressively to a lesser degree. Dampening the effect, the initial price increase rate (*Figure 2, a.*) grows steeper after 50M tokens have been issued (*b.*) and again after 90M (*c.*). For the respective partial curves, *a.*, *b.*, *c.*, the formulae are:

PRICE POINT FORMULA

ISSUED[38]	PRICE	CURVE
< 50M	$\frac{\text{issued}}{25M}$	*a.*
≥ 50M	$\frac{\text{issued} - 40M}{5M}$	*b.*
≥ 90M	$\frac{\text{issued} - 80M}{1M}$	*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 0.4 Æ:

[35] See *Sophia*, pg. 5.
[36] AEX-9 is Æternity's fungible token standard.
[37] AirGap wallet – *https://airgap.it/*
[38] Drops and locked-in sales can be excluded.

© 2023 Henning Diedrich

Select Price Points		
issued[38]	Price	
1M	0.04	Æ
10M	0.40	Æ
100M	20.00	Æ

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 can therefore at any point be more economic to buy in one transaction, instead of spreading a purchase across multiple transactions.

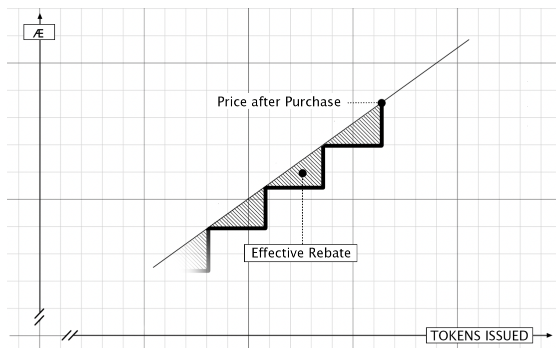

Figure 3 – Effective rebate (schematic)

CONCLUSION

Lexon's real-world impact is broad and sustained. It unites developments in computational law, cryptography, computer sciences, AI[39] 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 complement and safeguard generative AI to 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."*[40]

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 deweaponize law and level the playing field in business, protecting creativity and merit against the deep pockets of incumbents. Because Lexon is up to a million times cheaper, and a billion times faster,[41] 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. 22] has been presented to the reform committee appointed by the American Law Institute. Eventually, Lexon will be the language that the real Robotic Laws[42] 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.*[43] 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.

---

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

[40] 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 *http://lexon.org/intro.*

[41] See the *Lexon* book, ibid.

[42] See appx. *Robotic Laws*, pg. 19.

[43] See L. Lessig, 2000, *Code is Law* – *https://www.harvardmagazine.com/2000/01/code-is-law-html.*

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.[44] Its transparency and ease will unleash enormous power for good, pulling law back to a semblance of equal justice – a notion as urgently 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,[45] 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.[46] Lexon brings together deep tech that emerged from these passionate efforts, and makes it accessible.

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, fashionable cyborg dreams aside: *language.*

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

Payment for its use is made with the Lexon Æternity Tokens. The tokens can be purchased at *http://lexon.org/laex.*

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://www.gnu.org/licenses/gpl-3.0.txt*

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.[47]

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. See the License text at *https://lexon.org/license.*

The described tokens are not for investment; they may not work as a store of value. There may be no secondary market for the tokens. 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

There is no claim to the products of the Lexon compiler. Any text you write in Lexon and anything you create using the Lexon compiler is yours or determined by arrangements you made.

This document, including appendices, is licensed under Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0);[48] sources and grammar under AGPL3.[49] Basically, you can quote, share or modify this document but must give credit and allow the same.

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

[45] 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*

[46] See the *Lexon* book, ibid.

[47] Cf. appx. *The Principles of Newspeak* in G. Orwell, 1949, *Nineteen-Eighty-Four.* Orwell essentially argues that words must be ambiguous to be meaningful.

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

[49] *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 7 – Lexon code example

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

```
@compiler >=6

main contract Escrow =

  record state = {
           payer      : address,
           payee      : address,
           arbiter    : address,
           amount     : int,
           fee        : int
        }

  entrypoint init(payee : address, arbiter : address, fee : int) = {
            payer = Call.caller,
            payee = payee,
            arbiter = arbiter,
            amount = Call.value,
            fee = fee
        }

  stateful function transfer(to : address, amount : int) =
      Chain.spend(to, amount)

  function permit(authorized : address) =
    require(Call.caller == authorized,
      "no access")

  stateful entrypoint pay_out() =
    permit(state.arbiter)
    transfer(state.arbiter, state.fee)
    transfer(state.payee, Contract.balance)

  stateful entrypoint pay_back() =
     permit(state.arbiter)
     transfer(state.arbiter, state.fee)
     transfer(state.payer, Contract.balance)
```

Source 8 – Sophia result (barebones)

Using the *all auxiliaries* option, the Lexon compiler translates the Lexon code from the previous page into the following Sophia program. Its core functionality is identical to the *barebones* version, but it has additional features and comments.

```
@compiler >=6

include "Option.aes"

/* Lexon-generated Sophia code

   code:             Escrow
   file:             escrow.lex
   compiler:         lexon 0.3 alpha 85
   grammar:          0.2.20 / subset 0.3.8 alpha 79 - English / Reyes
   backend:          sophia 0.3.1/85
   target:           sophia 7+
   parameters:       --sophia  --all-auxiliaries
*/

/** LEX Escrow.
 *
 *  "Payer" is a person.
 *  "Payee" is a person.
 *  "Arbiter" is a person.
 *  "Amount" is an amount.
 *  "Fee" is an amount.
 *
 *  The Payer pays an Amount into escrow, appoints the Payee,
 *  appoints the Arbiter, and fixes the Fee.
**/

main contract Escrow =

    record state = {
                payer        : address,
                payee        : address,
                arbiter      : address,
                amount       : int,
                fee          : int }

    entrypoint init(payee : address, arbiter : address, fee : int) =
                payer = Call.caller,
                payee = payee,
                arbiter = arbiter,
                amount = Call.value,
                fee = fee }

    /* token transfer */
    stateful function transfer(to : address, amount : int) =
        Chain.spend(to, amount)

    /* built-in require function */
    function permit(authorized : address) =
        require(Call.caller == authorized, "no access")

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

    stateful entrypoint pay_out() =
        permit(state.arbiter)
        transfer(state.arbiter, state.fee)
        transfer(state.payee, Contract.balance)

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

    stateful entrypoint pay_back() =
        permit(state.arbiter)
        transfer(state.arbiter, state.fee)
        transfer(state.payer, Contract.balance)
```

Source 9 – Lexon compilation example (all auxiliaries)

LEXON FOR LAW

Lexon allows for law to be executed as a program. Asst. prof. Carla L. Reyes of SMU pioneers the use of Lexon to write statute – shown below – in her seminal 2021 paper *Creating Cryptolaw for the Uniform Commercial Code.*[50] She created the following Lexon code as a proposal to the commission that is tasked with the reform of the U.S. trade law, which she advises on blockchain topics. This code could become model law, be adapted by states to be executed on the computers of their local agencies and protect billions of dollars of collateral.

The salient point is that the law itself, without further changes *is* the program that the respective office runs to implement the law. The productivity gains of Lexon could not be illustrated better.

The motivation for this proposal is a concrete shortfall of the existing statute. Asst. prof. Reyes writes (emphasis added):

> *"Under certain conditions, security interests not only bind the creditor and debtor, but also third-party creditors seeking to lend against the same collateral. To receive this extraordinary benefit, creditors must put the world on notice, usually by filing a financing statement with the state in which the debtor is located. Unfortunately, the Uniform Commercial Code (U.C.C.) Article 9 filing system fails to provide actual notice to interested parties and introduces risk of heavy financial losses. To solve this problem, this Article introduces a smart-contract-based U.C.C.-1 form built using Lexon, an innovative new programming language that enables the development of smart contracts in English. The proposed "Lexon U.C.C. Financing Statement" does much more than merely replicate the financing statement in digital form; it also performs several U.C.C. rules so that, for the first time, the filing system works as intended. In* ***demonstrating that such a system remains compatible with existing law****, the Lexon U.C.C. Financing Statement also reveals important lessons about the interaction of technology and commercial law."* [ibid. 50]

```
LEX UCC Financing Statement.

LEXON: 0.2.12

"Financing Statement" is this contract.
"File Number" is data.
"Initial Statement Date" is a time.
"Filer" is a person.
"Debtor" is a person.
"Secured Party" is a person.
"Filing Office" is a person.
"Collateral" is data.
"Digital Asset Collateral" is an amount.
"Reminder Fee" is an amount.
"Continuation Window Start" is a time.
"Continuation Statement Date" is a time.
"Continuation Statement Filing Number" is data.
"Lapse Date" is a time.
"Default" is a binary.
"Continuation Statement" is a binary.
"Termination Statement" is a binary.
"Termination Statement Time" is a time.
"Notification Statement" is a text.

The Filer fixes the Filing Office, fixes the Debtor, fixes the Secured Party, and fixes the Collateral.
```

[50] *Washington and Lee Law Review – https://papers. ssrn.com/sol3/papers.cfm?abstract_ id=3809901*

```
Clause: Certify.
The Filing Office may certify the File Number.

Clause: Set File Date.
The Filing Office may fix the Initial Statement Date as the current time.

Clause: Set Lapse.
The Filing Office may fix the Lapse Date.

Clause: Set Continuation Start.
The Filing Office may fix the Continuation Window Start.

Clause: Pay Fee.
The Secured Party may pay a Reminder Fee into escrow.

Clause: Notice.
The Filing Office may fix the Notification Statement.

Clause: Notify.
The Filing Office may, if the Continuation Window Start has passed, send the Notification Statement to
the Secured Party.

Clause: Pay Escrow In.
The Debtor may pay the Digital Asset Collateral into escrow.

Clause: Fail to Pay.
The Secured Party may declare Default.

Clause: Take Possession.
The Filing Office may, if Default is declared, pay the Digital Asset Collateral to the Secured Party.

Clause: File Continuation.
The Secured Party may file the Continuation Statement.

Clause: Set Continuation Lapse.
The Filing Office may, if the Continuation Statement is filed, fix the Continuation Statement Date.

Clause: File Termination.
The Secured Party may file a Termination Statement, and certify the Termination Statement Time as the
then current time.

Clause: Release Escrow.
The Filing Office may, if the Termination Statement is filed, return the Digital Asset Collateral to the
Debtor.

Clause: Release Reminder Fee.
The Filing Office may, if the Termination Statement is filed, return the Reminder Fee to the Secured
Party.

Clause: Termination Period.
"Termination Period" is defined as 365 days after the Termination Statement Time.

Clause: Terminate and Clear.
The Filing Office may, if the Termination Period has passed, terminate this contract.
```

Source 10 – Lexon code example: U.C.C. Filing Statement

The above example is compiled to the following Sophia code, using the --harden option to make the code safe against certain attacks. The produced code makes use specifically of Sophia's highly precise handling of undefined values, employing the *option type*, a hybrid of a normal atomic type and a value meaning that no value is given, *None*. The Lexon text is again used verbatim for comments, exploiting that Lexon code is per se self-documenting.

```
/* Lexon-generated Sophia code

   code:        UCC Financing Statement

   file:        statement.lex

   code tagged: 0.2.12

   compiler:    lexon 0.3 alpha 85

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

   backend:     sophia 0.3.1/85

   target:      sophia 6+

   options:     --sophia  --harden
*/

@compiler >=6

include "Option.aes"
using Option


/** LEX UCC Financing Statement.
 *
 *  LEXON: 0.2.12
 *
 *  "Financing Statement" is this contract.
 *  "File Number" is data.
 *  "Initial Statement Date" is a time.
 *  "Filer" is a person.
 *  "Debtor" is a person.
 *  "Secured Party" is a person.
 *  "Filing Office" is a person.
 *  "Collateral" is data.
 *  "Digital Asset Collateral" is an amount.
 *  "Reminder Fee" is an amount.
 *  "Continuation Window Start" is a time.
 *  "Continuation Statement Date" is a time.
 *  "Continuation Statement Filing Number" is data.
 *  "Lapse Date" is a time.
 *  "Default" is a binary.
 *  "Continuation Statement" is a binary.
 *  "Termination Statement" is a binary.
 *  "Termination Statement Time" is a time.
 *  "Notification Statement" is a text.
 *
 *  The Filer fixes the Filing Office, fixes the Debtor, fixes the Secured Party, and fixes the
Collateral.
**/

main contract UCCFinancingStatement =

    record state = {
            file_number : option(string),
            initial_statement_date : option(int),
            filer : option(address),
            debtor : option(address),
            secured_party : option(address),
            filing_office : option(address),
            collateral : option(string),
            digital_asset_collateral : option(int),
            reminder_fee : option(int),
            continuation_window_start : option(int),
            continuation_statement_date : option(int),
            continuation_statement_filing_number : option(string),
            lapse_date : option(int),
            _default : option(bool),
            continuation_statement : option(bool),
            termination_statement : option(bool),
            termination_statement_time : option(int),
            notification_statement : option(string),
```

```
            terminated : bool
        }

    datatype event = Message(indexed address, indexed address, string)

    entrypoint init(filing_office : address, debtor : address, secured_party : address,
collateral : string) = {
            file_number = None,
            initial_statement_date = None,
            filer = Some(Call.caller),
            debtor = Some(debtor),
            secured_party = Some(secured_party),
            filing_office = Some(filing_office),
            collateral = Some(collateral),
            digital_asset_collateral = None,
            reminder_fee = None,
            continuation_window_start = None,
            continuation_statement_date = None,
            continuation_statement_filing_number = None,
            lapse_date = None,
            _default = None,
            continuation_statement = None,
            termination_statement = None,
            termination_statement_time = None,
            notification_statement = None,
            terminated = false
        }


    stateful function termination() =
        put(state{terminated = true})

    function check_termination() =
        require(!state.terminated, "contract system terminated before")

    stateful function transfer(to : address, amount : int) =
        Chain.spend(to, amount)

    function send(to : address, message : string) =
        Chain.event(Message(Call.caller, to, message))

    function permit(authorized : option(address)) =
        require(Call.caller == force(authorized), "access")

    /*
     *  Clause: Certify.
     *  The Filing Office may certify the File Number.
     */

    stateful entrypoint certify(file_number : string) =
        check_termination()
        permit(state.filing_office)
        put(state{file_number = Some(file_number)})

    /*
     *  Clause: Set File Date.
     *  The Filing Office may fix the Initial Statement Date as the current time.
     */

    stateful entrypoint set_file_date() =
        check_termination()
        permit(state.filing_office)
        put(state{initial_statement_date = Some(Chain.timestamp)})

    /*
     *  Clause: Set Lapse.
     *  The Filing Office may fix the Lapse Date.
     */

    stateful entrypoint set_lapse(lapse_date : int) =
        check_termination()
        permit(state.filing_office)
        put(state{lapse_date = Some(lapse_date)})

    /*
     *  Clause: Set Continuation Start.
     *  The Filing Office may fix the Continuation Window Start.
     */

    stateful entrypoint set_continuation_start(continuation_window_start : int) =
        check_termination()
        permit(state.filing_office)
        put(state{continuation_window_start = Some(continuation_window_start)})
```

```
    /*
     *  Clause: Pay Fee.
     *  The Secured Party may pay a Reminder Fee into escrow.
     */

    stateful payable entrypoint pay_fee() =
        check_termination()
        permit(state.secured_party)
        switch(state.reminder_fee)
            None => put(state{reminder_fee = Some(Call.value)})
            Some(_) => put(state{reminder_fee = Some(force(state.reminder_fee) + Call.value)})

    /*
     *  Clause: Notice.
     *  The Filing Office may fix the Notification Statement.
     */

    stateful entrypoint notice(notification_statement : string) =
        check_termination()
        permit(state.filing_office)
        put(state{notification_statement = Some(notification_statement)})

    /*
     *  Clause: Notify.
     *  The Filing Office may, if the Continuation Window Start has passed, send the
Notification Statement to the Secured Party.
     */

    entrypoint notify() =
        check_termination()
        permit(state.filing_office)
        if(state.continuation_window_start =< Some(Chain.timestamp))
            send(force(state.secured_party), state.notification_statementx)

    /*
     *  Clause: Pay Escrow In.
     *  The Debtor may pay the Digital Asset Collateral into escrow.
     */

    stateful payable entrypoint pay_escrow_in() =
        check_termination()
        permit(state.debtor)
        switch(state.digital_asset_collateral)
            None => put(state{digital_asset_collateral = Some(Call.value)})
            Some(_) => put(state{digital_asset_collateral =
Some(force(state.digital_asset_collateral) + Call.value)})

    /*
     *  Clause: Fail to Pay.
     *  The Secured Party may declare Default.
     */

    stateful entrypoint fail_to_pay() =
        check_termination()
        permit(state.secured_party)
        put(state{_default = true})

    /*
     *  Clause: Take Possession.
     *  The Filing Office may, if Default is declared, pay the Digital Asset Collateral to the
Secured Party.
     */

    stateful entrypoint take_possession() =
        check_termination()
        permit(state.filing_office)
        if(state._default != None)
            transfer(force(state.secured_party), state.digital_asset_collateral)

    /*
     *  Clause: File Continuation.
     *  The Secured Party may file the Continuation Statement.
     */

    stateful entrypoint file_continuation(continuation_statement : bool) =
        check_termination()
        permit(state.secured_party)
        put(state{continuation_statement = Some(continuation_statement)})

    /*
     *  Clause: Set Continuation Lapse.
     *  The Filing Office may, if the Continuation Statement is filed, fix the Continuation
Statement Date.
     */
```

```
    stateful entrypoint set_continuation_lapse(continuation_statement_date : int) =
        check_termination()
        permit(state.filing_office)
        if(state.continuation_statement != None)
            put(state{continuation_statement_date = Some(continuation_statement_date)})

    /*
     *  Clause: File Termination.
     *  The Secured Party may file a Termination Statement, and certify the Termination
Statement Time as the then current time.
     */

    stateful entrypoint file_termination(termination_statement : bool) =
        check_termination()
        permit(state.secured_party)
        put(state{termination_statement = Some(termination_statement)})
        put(state{termination_statement_time = Some(Chain.timestamp)})

    /*
     *  Clause: Release Escrow.
     *  The Filing Office may, if the Termination Statement is filed, return the Digital Asset
Collateral to the Debtor.
     */

    stateful entrypoint release_escrow() =
        check_termination()
        permit(state.filing_office)
        if(state.termination_statement != None)
            transfer(force(state.debtor), state.digital_asset_collateral)

    /*
     *  Clause: Release Reminder Fee.
     *  The Filing Office may, if the Termination Statement is filed, return the Reminder Fee to
the Secured Party.
     */

    stateful entrypoint release_reminder_fee() =
        check_termination()
        permit(state.filing_office)
        if(state.termination_statement != None)
            transfer(force(state.secured_party), state.reminder_fee)

    /*
     *  Clause: Termination Period.
     *  "Termination Period" is defined as 365 days after the Termination Statement Time.
     */

    entrypoint termination_period() =
        Some(state.termination_statement_time + (365 * 86400))

    /*
     *  Clause: Terminate and Clear.
     *  The Filing Office may, if the Termination Period has passed, terminate this contract.
     */

    stateful entrypoint terminate_and_clear() =
        check_termination()
        permit(state.filing_office)
        if(termination_period() =< Some(Chain.timestamp))
            termination()
```

Source 11 – Lexon compilation example (hardened): U.C.C. Filing Statement

ABSTRACT SYNTAX TREE

This is a part of the *abstract syntax tree* (AST), the internal model the compiler creates when processing the grammar and text discussed in chapter *Grammar*, pg. 3. It reflects natural language grammar rather than programming logic. Such a tree can be created from any Lexon text using the *flat tree* options.

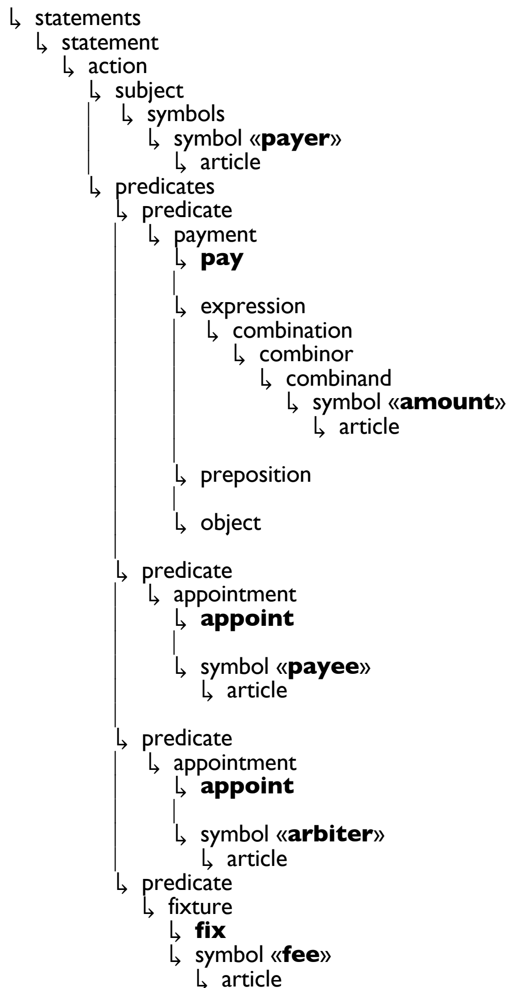

Figure 4 – Example of a Lexon abstract syntax tree

To create such a tree for your own Lexon text, at *https://lexon.org/sophia* 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.***

ROBOTIC LAWS

The science fiction author Isaac Asimov coined the term *robotic laws*[51] in the 1940s for the science fiction universe over-arching his short stories and novels. He evolved them over time and explored 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 when writing statute to rein in autonomous machines.

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.

◊

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

INDICES

INDEX OF FIGURES

Figure 1 – Compiler screen at lexon.org/sophia 5
Figure 2 – Token sale price based on tokens issued 7
Figure 3 – Effective rebate (schematic) 8
Figure 4 – Example of a Lexon abstract syntax tree 18

INDEX OF TABLES

Table 1 – Token price formula 7
Table 2 – Token price points 8

INDEX OF SOURCES

Source 1 – Lexon digital contract example 2
Source 2 – Lexon Grammar Form (LGF) example 4
Source 3 – Lexon sentence grammar (detail) 4
Source 4 – Lexon code example sentence 4
Source 5 – Lexon document structure 4
Source 6 – Compilation example (barebones) 5
Source 7 – Lexon code example 10
Source 8 – Sophia result (barebones) 10
Source 9 – Lexon compilation example (all auxiliaries) 11
Source 10 – Lexon code example: U.C.C. Filing Statement 13
Source 11 – Lexon compilation example (hardened): U.C.C. Filing Statement 17

release 1

© 2023 Henning Diedrich