CoLa: A Controlled Natural Language for Computable Contracting

Simon Fattal

May 2021

Supervisor:
Prof. Christopher D. Clack

Submitted as part requirement for the degree of
Masters of Computer Science at UCL

Abstract

Automating legal contracts is desirable for several reasons. However, one of the biggest obstacles to wider adoption is that the code used to execute such contracts can be hard to understand by those without a background in Computer Science. Various Controlled Natural Languages (CNLs) have been proposed to bridge the gap between natural and computer languages. This paper evaluates existing CNLs against a set of requirements, and presents CoLa, a new CNL which is designed to meet all of these requirements. A complete grammar and syntax is provided for this language, as well as a parser which can produce a hierarchical representation of legal contracts in an intuitive manner. Example applications to real world financial contracts have also been provided. Substantial testing was performed on all aspects of code implemented, resulting in a robust application which can be used for a variety of different contracts.

Acknowledgements

I would like to thank Prof. Christopher D. Clack for introducing me to the world of computable contracting, and for the time spent discussing the design of CoLa and how it should be implemented. I am grateful for all of his patience and advice while working on this project, and extremely appreciative of him educating me on the strength and simplicity of functional programming languages.

Contents

1 Introduction

The history of contract law dates back to Ancient civilisations. Despite this, the representation of contracts has largely remained the same (i.e., a written document). Since the late 1990s, technological breakthroughs have sparked an unprecedented level of opportunity across many industries. In fact, smart contracts (Section 2.1.3) was named by the World Economic Forum as a *central element to the fourth industrial evolution* (Schwab [2016]). While many industries have experienced first-hand the benefits of such technological adoption, this has not yet extended to contract law.

1.1 Natural Language Contracts

A contract is a legally binding document that that recognises and governs the rights and duties of the parties to an agreement (Ryan [2006]). For the purposes of this work, we draw from the work of Cummins and Clack [2020], for an overview of the main components of a contract:

Objects Contract objects and their definitions clarify what the key terms of a contract mean.

Parties The individuals or groups of individuals between whom a contract exists.

Obligations, Permissions, and Prohibitions The deontic aspects (Section 1.2) of a contract govern the rights and expectations of parties to fulfil the contract.

Actions These cover the details of how a party can fulfil its obligations.

Temporal Aspects These cover details related to timing for fulfilling obligations.

Events These are the situations that may or should arise during the contract period.

Traditional natural language contracts can be difficult and often tedious to form. Moreover, once agreed upon, contracts can be challenging to analyse and require manual work to implement. Consequentially, this has led to high costs for businesses and longer times required to draft contracts. While many industries have experienced first-hand the benefits of recent technological adoption, this has not yet extended to contract law.

State of the art computable contracting (Section 2.1.2) aims to leverage computational power, resulting in significant improvements to innovation in the legal domain. Computable contracting can lower the transaction costs of traditional contracting, allow contracts to be self-executing, enable an increased scope for analysis of contract obligations and risk, and even understand the semantics of contracts to determine compliance with contract terms. As such, it is imperative that we move away from natural language contracting, characterised by being cumbersome to draft, difficult to read, and having a lack of modularity and connectedness, and towards computable contracting, characterised by being easy

to generate, universally understood by both humans and machines, and connected to the systems to which they refer.

1.2 The Role of Deontic Logic

As described by Von Wright [1951], the obligatory (that which we ought to do), the permitted (that which we are allowed to do) and the forbidden (that which we must not do) underpin the modal concept of deontic logic. Computable contracts should not only be able to express the sequence and causality of events, as natural language contracts do, but they must also account for these deontic terms, relating to the entities involved. Having an underlying logic which accounts for such deontic terms would enable us to effectively reason about contracts, which is an important step in facilitating the use of computable contracts.

However, the expressiveness of deontic logic does not come without its challenges. Formalising deontic logic has been a concentrated topic of research for several decades. The main challenge is that deontic logic is filled with paradoxes (Meyer and Dignum [1994]).[1] These paradoxes range from counterintuitive interpretations of seemingly trivial statements, to further paradoxes surrounding the concept of time (Pace and Schneider [2009]). If we are to incorporate deontic logic into the formulation of computable contracts, we need some way of remedying this. One approach, taken by Prisacariu and Schneider [2007a] is to restrict the syntax of deontic logic in a way that maintains the general intuitive properties of deontic notions in contracts but avoids many of these classical paradoxes. As will be seen in Section 4.3, this project adopts the same approach as Prisacariu and Schneider [2007a] due to its effectiveness.

1.3 Project Objectives

This objectives of this project are: (i) to analyse the requirements for a computable contracts language by evaluating existing formalisms and approaches, (ii) to design a new controlled natural language to address these requirements, (iii) to develop, validate, and test a parser which accepts controlled language contracts as input and can map this to a formal underlying logic, from which code can then be generated for a number of different languages.

[1] For a comprehensive list of classical paradoxes, see Prisacariu and Schneider [2007b].

2 Background

This section is composed of two parts. First, we define and briefly discuss some of the key terms that are relevant for a complete understanding of this work. We then move on to identify some of the primary research in the field of computable contracting, including state of the art solutions.

2.1 Technical Preliminaries

There are three terms which we explain in depth: controlled natural languages, computable contracts, and smart contracts. This aims to give the reader a concrete understanding of the fundamentals, in order to fully grasp the remainder of this paper. A limited analysis of these terms has also been provided.

2.1.1 Controlled Natural Languages

To represent knowledge in a computer, formal languages are typically used. Such languages adhere to a set of rules, have a well-defined syntax, and have clear semantics. Importantly, formal languages support reasoning, which is a crucial component to increasing innovation in the context of contract law. However, many experts in the domain of law, business and government are unfamiliar with formal languages, which has hindered their adoption for practical applications. There is a clear tension between the use of natural languages and the use of formal languages to empower analysis and automation by computers. Controlled natural languages have been proposed to address this conflict.

A controlled natural language (CNL) does not have a single generally accepted definition. For the purposes of this work, we will use the definition provided by Kuhn [2014]:

> *A controlled natural language is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax, and/or semantics, while preserving most of its natural properties.*

If we consider a CNL which can also describe complex descriptions of events and actions, then we have a highly suitable candidate for enabling the direct translation of formal contracts into an underlying logic. More specifically, this work seeks to identify a way of reasoning about contracts, by developing a CNL which is simple enough for people without a background in computer science to use, rich enough to express the essential concepts of a contract and precise enough to capture detailed semantics. By drafting a contract in this CNL, the contract can automatically be converted into an underlying logic for analysis, and can automatically be converted into low-level code.

2.1.2 Computable Contracting

A 'data-oriented' contract, which for our purposes is synonymous to a computable contract, is defined by Surden [2012] as:

> *A contract in which the parties have expressed one or more terms or conditions of their agreement in a manner designed to be processable by a computer system.*

The only difference between data-oriented contracts and computable contracts is that that data-oriented contracts are a form of computable contracts that interface with data inputs or outputs. A computable contract is understandable by both humans and computers, and supports: (i) the analysis of contract semantics, (ii) the simulation of contract performance, and (iii) the automation of some aspects of performance. This means that part, or all, of a written contract may be written in a CNL in order for it to be computer-processable. Note that in the case where we wish to automate a small part of a written contract, rather than the contract in its entirety, it is important that the code responsible for the automation is aware of other parts of the contract, which we may not wish to automate. This is to ensure that the automated code does not contradict the semantics of other aspects of the written contract, and forms an integral part of code validation.

As noted by Clack [2021], this implies that such parts of an agreement are simultaneously contract and code, understandable by humans and computers, and are able to serve both the contractual obligations of parties and the automated implementation of such obligations. By constraining the language and the complexity of the grammar of the agreement, one can obtain a way of making statements about the underlying domain, without diverging too far off from the natural language description. This ensures that it can still be understood by native language speakers while also making it easy to translate into an underlying logic.

Smart Contract Templates (SCTs), described by Clack et al. [2016], are a first step towards automation. In this approach, the contract is first written a natural language. Then, we identify the parts of the contracts that we know we will want to automate, using a markup language. These parts are annotated with a very precise expression, which could be math, code, or even an underlying logic, in order for it to be easy to code. Finally, code is written, separately, to automate the parts which are annotated. Note that SCTs do not provide computable contracts since a human without prior knowledge of programming would not understand the code.

2.1.3 Smart Contracts

While this work does not directly concern smart contracts, it is nonetheless a related concept which should be explained. The term 'smart contract', coined by Szabo [1997], is defined by Clack et al. [2016] to mean:

> *An automatable and enforceable agreement. Automatable by computer, although some parts may require human input and control. Enforceable either by legal enforcement of rights and obligations or via tamper-proof execution of computer code.*

It is worth highlighting the use of the term "automatable", rather than "automated". This is because it is possible that some aspects of a smart contract may require manual, human control, or input. To be a smart contract, it is only required that *some* aspect of the agreement *can* be automated. The rise in popularity of the cryptocurrency community has prompted new meanings to the term 'smart contract', namely involving software agents which are typically run on a shared ledger. This is different to the other widely-used meaning of term, relating to expressing and implementing legal contracts in software, a distinction observed by Stark [2016]. Clack et al. [2016] provide the above definition in order to remove any ambiguities surrounding the term, especially those that arise from the interaction of different disciplines such as Computer Science, Law, and Business. This definition is broad enough to capture the various meanings associated with the term.

There are plenty of advantages that smart contracts offer, which we do not cover in depth.[2] Through the use of encryption and other cryptographic primitives, smart contracts can be stored securely. Furthermore, since the possibility of fraud is drastically reduced, the need for third-party bookkeepers and other administrative roles is almost entirely eliminated. Smart contracts also allow transactions between anonymous parties, establishing new levels of trust and market integrity, which is essential for any market infrastructure.

One point of contention is the role that humans play in the lifecycle of smart contracts. While trivial examples of smart contract code do not require human intervention, high-value, complex, or long-lived smart contract code would require human intervention. This may be due to the need to change the code if the law changes. As such, despite smart contracts being fully self-executable, they generally do not remove the need for human intervention.

Smart contracts and computable contracts are similar in nature. At their core, they are both a type of agreement. This paper takes the view that computable contracts change the way that legal contracts are drafted, while smart contracts are more involved with the general automation of agreements (including agreements that may not be enforceable in law). Computable contracts can be viewed as a methodology within the field of smart contracts, to facilitate the automation of legally enforceable agreements. Furthermore, with an appropriate CNL, a computable contract could be the exclusive element of an entire legal agreement, representing the contract, code and semantic specification in a single structure, which is ultimately what a computable contract is meant to be (Clack [2021]).

[2] For a more complete understanding of the advantages of smart contracts, the reader may wish to consult Temte [2019].

2.2 Literature Review

In Section 2.1.2 we explained what computable contracts are. Now, we provide a broad review of surrounding literature in the area of computable contracting. In Section 3, we take a more detailed look at specific existing CNLs for computable contracting, which are more closely related to this particular work.

Since 2012, there have been numerous efforts to expand on Surden's work. As noted by Cummins and Clack [2020], many attempts to improve the efficiency of contracting have been done in such a way that any coding required to support automation is completely separate from the creation of the contract itself. If we wish to see the advancement of computable contracting, it is imperative that modern approaches attempt to bridge the gap between contract creation and the required coding for its automation. Several such approaches are mentioned below.

Based at Stanford University's CodeX Centre for Legal Informatics, the Stanford Computable Contracts Initiative (SCCI)[3] is working to develop a general protocol for expressing computable contracts, along with a series of open-source software tools to create them. The SCCI will design a standard protocol for expressing computable contracts. This protocol also aims to be flexible (to cover the vast variety of contracting scenarios), domain tailorable (providing templates for specific domains), technologically independent (not tied to any particular implementation) and interdisciplinary (integrating best-practices across a variety of disciplines). The SCCI are also working on developing a universal Contract Definition Language, that will "allow terms and contracts to be represented in a machine-understandable way" with its foundations based on declarative rules-based logic programming.

Another interesting work is that of Pace and Rosner [2009], who define a paired logic and CNL which can be used for drafting and analysing contracts. They present an abstract-syntax definition of the CNL along with operators describing the paired deontic logic. This approach embeds part of the contract logic as an embedded language in Haskell, in order to enable easier automated manipulation of contracts while keeping the desired syntactic structure intact. A related paper by Pace and Schneider [2009] helps to provide a better understanding of Pace and Rosner [2009], and both works will be explored in greater detail in Section 3.2. The purpose of Pace and Rosner [2009] is for contract simulation, but future work intends to perform contract analysis, including the detection of conflicts and superfluous clauses.

As mentioned in Section 1.2, Prisacariu and Schneider [2007a] define a contract language which is able to avoid most of the paradoxes of deontic logic, and also provide a translation of the contract language into their logic. Paradoxes such as Ross's paradox, the Free Choice Permission paradox, Satre's dilemma, and more (explained in Section 3.1 of

[3] `https://law.stanford.edu/projects/stanford-computable-contracts-initiative/`

Prisacariu and Schneider [2007b]) are all avoided. This is because they are either not expressible in the language, or they are excluded by the translation into the underlying logic. This worked formed the basis for $\mathcal{CL}$ (Prisacariu and Schneider [2009]), a contract specification language whose purpose is largely for contract simulation, which combines deontic logic with propositional dynamic logic. It features synchronous actions, conflict relation, and an action negation expression (i.e. an action which takes us outside of the trace of some other action). This approach of applying the deontic notions of obligation, permission and prohibition to actions is often used in the formalisation of contracts, such as in Martínez et al. [2010]. In terms of the contract clause formalisation, both Prisacariu and Schneider [2009] and Martínez et al. [2010] are similar to the work of Wöhrer and Zdun [2020], though the domain-specific language proposed in Wöhrer and Zdun [2020] is highly programmatic (i.e. it looks and feels like a programming language).

Fuchs et al. [2008] propose a controlled natural language, Attempto Controlled English (ACE)[4]. While this language is constrained by the bounds of first-order logic, ACE is still able to express relations, rules, commands and queries in a way that can be understood by machines. However, the applications of this language is better suited for the domain of the semantic web, rather than contract law, due to its lack of deontic terms.

Developed for the business domain, RuleCNL (Njonko et al. [2014]) is a CNL for defining business rules. Such business rules are often used in contracts. The RuleCNL tool aids end-users in the drafting process and provides automatic mappings into the Semantics of Business Vocabulary and Business Rules standard (SBVR [2008]). SBVR is grounded in first-order logic with its own semantic formulation description. While RuleCNL does an excellent job at bridging the gap between business domain experts and formalised business rules that can be understood by computers, it is difficult to apply this CNL to other domains. RuleSpeak[5] is defined in the SBVR specification as a CNL to express business rules, though this is not an explicit language, but rather a set of guidelines for expressing business rules in a concise fashion.

Kowalski's development of Logical English (LE) (Kowalski [2020]) is another attempt to devise a contract drafting language which can be understood by both humans and machines. With prototypes written in the logical programming language Prolog, LE aims to be understandable by a reader without any training in computing, logic or mathematics. LE admits three kinds of sentences: simple sentences, which express facts about individuals; general rules, which express properties of groups of individuals, and complex sentences, which express properties of both individuals and other properties. Furthermore, it has been shown that LE can represent sophisticated and financial contracts, such as various sections of the widely-used ISDA Master Agreement (Karadotchev [2019]). Born from the observation that legislation and logic programming share many linguistic similarities, LE actively

[4] `http://attempto.ifi.uzh.ch`

[5] `http://www.rulespeak.com/en/`

supports the goals of legal agreements and is also suitable for general computation.

Camilleri et al. [2014] present a CNL for verbalising *C-O Diagrams*, for the purpose of simulating and analysing electronic contracts. *C-O Diagrams* provide a visual representation for contracts, and give the possibility to express deontic terms. The language for verbalising such diagrams include contract clauses (which consist of an agent, modality, and action), constraints (each clause has its own timer, which can be referred to in any timing restriction), conjunctions (to combine multiple contracts), and names (to allow referencing of other clauses). Furthermore, both a web-based tool allowing editing in this CNL, and another tool for visualising and manipulating the diagrams interactively have been provided.

Finally, Lexon (Diedrich [2020]) is a computer language with an emphasis on readability. No prior knowledge in programming is required to understand or write Lexon. It can be used to write normal contracts that work as blockchain smart contracts; the same text is both the program and legal agreement. Each Lexon document starts with a title, the Lexon version it is built for, and a short preamble explaining the purpose of the contract. Following this, terms are defined, such as the parties involved in the contract, their roles, as well as other objects, such as pre-defined amounts. Then the main contract starts, represented through a list of clauses, each with their own clause name. The Lexon language is highly promising, as seen through the many complex example contracts it is capable of expressing. According to Idelberger [2020], it is the most advanced digital contract stack available.

3 Requirements and Analysis

After reviewing the existing literature surrounding the space of CNLs and computable contracting, we now present the requirements that a CNL must have in order to enable computable contracting, followed by an analysis of existing literature evaluated against these requirements.

3.1 Requirements of a CNL for Computable Contracting

This section presents a novel set of requirements that a CNL must have in order to adequately enable computable contracting. While Hvitved [2011] proposes a set of 16 properties that "ideal" contract *formalisms* should support, we identify the key features that a practical CNL should possess in order to facilitate computable contracting more generally (e.g., including translation into lower-level languages). Naturally, some aspects of the underlying formalism will be reflected in the CNL. The 11 requirements are presented below.

(R_1) *Uses a natural language (such as English), with a restricted syntax and grammar.*

This is the mere definition of a CNL, so any attempt to enable computable contracting using a CNL must at the bare minimum meet this requirement. There are no explicit criteria for how the syntax or grammar should be restricted.

(R_2) *Expresses the key components of a contract, as identified in Section 1.1.*

If a CNL is unable to satisfy this requirement it cannot be regarded as a sufficient language for contracts to be drafted in. There is no use in a CNL which cannot capture the key components of a contract and hence this should not be considered valid for the purposes of enabling computable contracting.

(R_3) *Can be both generated and understood by humans without a formal background in Logic or Computer Science.*

One must not need to be able to program, know what a data type is, or know how an operating system works in order to use such a language. This is essential if we want to ensure the mass adoption of a CNL by lawyers, who are the primary party responsible for drafting contracts.

(R_4) *Captures deontic, temporal, and operational semantics.*

The CNL must be able to account for obligations, permissions, and prohibitions. These deontic terms are essential when deriving the meaning of clauses. The CNL must also be able to account for time. This can be done through attaching dates to events or actions, or comparing the times of events. Furthermore, the CNL must be able to account for operational semantics, by being able to describe actions.

(R_5) *Avoids some, or all, of the classic paradoxes of deontic logic.*

This is highly related to the underlying formal logic of the CNL. It is critical that the classical paradoxes are avoided in order for the language to be unambiguous. To satisfy this requirement, at least one of the following paradoxes must be avoided: Ross's paradox, the Free Choice Permission paradox, the Good Samaritan paradox, the Gentle Murderer paradox, Sartre's dilemma, or Chisholm's dilemma.[6] These paradoxes are the most important ones in deontic logic, according to Prisacariu and Schneider [2012]. These can be avoided in two ways, either by (i) not allowing paradoxical statements to be expressible in the language, or by (ii) allowing paradoxical statements in the language but excluding them in the translation to the underlying logic.

(R_6) *Can be easily translated into lower-level languages.*

The CNL should be based on an underlying formalism that facilitates both formal analysis of the contract and translation of parts of the contract into code. To achieve such analysis, we need to be able to convert the CNL into lower-level languages, in a way such that the drafted contract can be passed through the language stack, described by Clack [2021].

(R_7) *Supports the detection of inconsistencies.*

The CNL must enable the drafting lawyer to detect any inconsistent clauses. This of course depends on the interface given to the lawyer. For example, if the lawyer is given a syntax directed editor, this editor may be able to flag, or even refuse to permit, possible inconsistencies when they are detected. Alternatively, if no editor is provided, when the CNL attempts to parse the contract, inconsistencies can be detected and reported as a '`TypeError`' or another error similar in nature.

(R_8) *Supports building new contracts from existing contracts.*

This has two distinct applications. First, the CNL must give drafting lawyers the ability to construct new contracts which are based on existing ones. For example, if a lawyer wishes to draft a contract which is very similar to a contract that has previously been drafted, the CNL should anticipate this and aid in the drafting of such a contract. Second, it is often the case that a contract implements aspects of an inherited template contract, with some alterations (for example, the ISDA Schedule[7] is part of the ISDA Master Agreement, where parties choose whether and how certain provisions will apply). This method of contract drafting must also be supported by the CNL.

(R_9) *Supports overlapping clauses where one takes precedence over another.*

Relating to R_8, if there is a scenario in which a contract takes precedence over another contract from which it has inherited clauses, the CNL must be able to give precedence to

[6] These paradoxes are described in depth in Prisacariu and Schneider [2007a].

[7] `https://www.isda.org/book/schedule-to-the-2002-isda-master-agreement/`

the clauses in the inherited contract, or vice versa (the lawyer should be able to choose this). To make this point clearer, consider a Master Agreement and a Schedule Agreement, with overlapping clauses. The CNL should allow either the clauses in the Master Agreement or the clauses in the Schedule Agreement to take precedence, depending on what the drafting lawyer wishes.

(R_{10}) *Can be extended.*

It is important that a CNL is extensible in order for it to be well-equipped to handle vagueness. Where vagueness in part of a contract exists, be it intentionally or not, the CNL should offer extensibility as a way around this vagueness. This can either be done through extending the CNL, by allowing it to use a wider variety of words, or by offering the drafting lawyer the ability to write this part of the contract using a natural language, known as an 'escape mechanism'.[8]

(R_{11}) *Can reference other clauses.*

It is not essential that a CNL meet this requirement in order to suitably model contracts. However, it is a known fact that clauses often refer to other clauses in contracts. This can either happen when a definition stated earlier in a contract is being used in a new context, or when some clause is being overwritten or extended by a new clause (this can reference the original clause implicitly or explicitly). This is a desirable feature for a CNL to support.

3.2 Analysis of Existing Work

As noted by Cummins and Clack [2020], many of the existing approaches to improving the computability of smart contracts separate the creation of the contract from the coding required to support the automation of the contract. This section identifies the relevant CNLs that attempt to bridge this gap, and analyses how well these works fit the requirements listed in Section 3.1.

For the purposes of this analysis, five CNLs were chosen. Of these, three were constructed with (or at least inspired by) the aim of representing legal contracts. These three are the most promising CNLs for enabling computable contracting. The other two are more general purpose CNLs, but have been included to give the reader a benchmark for how a regular CNL would perform. For each of the contract-specific CNLs considered, a worked example of using the CNL has been provided, followed by an evaluation against the proposed requirements. A checkmark indicates that the requirement is met. A question mark denotes that it is possible for this requirement to be met. A cross denotes that the requirement is

[8] We have intentionally omitted the requirement to prevent vagueness in a CNL, as many contracts would like vagueness to exist, in order for parties to agree on such contracts in the first place.

not met. For a more complete explanation of the provided worked examples, the reader may wish to refer to the original papers which propose the CNLs themselves.

3.2.1 Pace and Rosner [2009]

Overall, Pace and Rosner [2009] make an excellent attempt at defining a paired logic and CNL which can be used for drafting contracts. Their justification for the use of their underlying logic is good, since by restricting the syntax of their proposed deontic logic, in a similar fashion to Prisacariu and Schneider [2007a], many of the classical paradoxes of deontic logic are dealt with.

That said, there are some critiques that can be made. A 'basic action' is never explicitly defined. They could have made the interaction clearer between the two fundamentally different levels: an action-expression and a contract. It appears as though the action-expression is a way of defining a clause, while a contract defines how these action-expressions are combined. No explanation of the interaction between these two terms has been provided. Fortunately, the separate work of Pace and Schneider [2009] provides a clearer explanation of the concept of choice through more explicit definitions of each of the logical operators used. Finally, there is no concrete explanation of what a 'job' is. This makes things semantically unclear: is a 'job' the *result* of some actions, or is it the *doing* of some actions? One may argue that the actual CNL itself may be too verbose, but this is deemed necessary in order to capture the full semantics relating to a clause. The CNL does not entertain the possibility of an escape mechanism.

Drawing from the work of Pace and Rosner [2009], we can inspect an example of this CNL to justify whether it meets the requirements described above. Additional examples with greater explanation can be found in Pace and Rosner [2009]. This example provides a short excerpt of natural language, followed by the equivalent representation in the CNL.

Natural Language:

> Upon accepting a job, the system guarantees that results will be available within an hour unless cancelled in the meantime.

CNL:

> If SYSTEM accepts Job, then during one hour it is obligatory that SYSTEM make available results of Job unless SOMEONE cancels Job.

With the help of this example, we can now begin to analyse this language against the provided requirements. Note that the language has been analysed holistically, rather than just on the basis of how it performs in the provided example.

(R_1) ✓ Clearly, the controlled version of the original statement uses a natural language (English), and the syntax and grammar is restricted.

(R_2) ✓ This example incorporates objects, agents (i.e. parties), and events (in the form of agent-action-object). It also uses a deontic logic, showing it is able to support the key components of a contract.

(R_3) ✓ The controlled statement can quite easily be read and understood by humans. It can also be easily generated by humans, since the controlled statement closely mirrors the original.

(R_4) ✓ As obligations have been accounted for in this sentence, the CNL clearly captures deontic terms. Permissions and prohibitions are also accounted for in the definition of action-expressions. Temporal aspects are covered too; the example provided mentions 'one hour'.

(R_5) ✓ Pace and Rosner [2009] decide to use a more general logic, which can then be restricted, syntactically or semantically, to isolate potential problems. By restricting the syntax of the CNL, many classical deontic paradoxes are avoided.

(R_6) ✓ Accompanying this CNL is an underlying logic, which uses operators which closely match the CNL. This would enable translation from CNL into lower level languages with some further work.

(R_7) x The CNL does not provide explicit features to detect inconsistencies. It is possible that this could be extended depending on how lawyers interface with the language, but in its current form such a feature does not exist.

(R_8) ✓ By examining the abstract syntax used, we indeed observe that contracts can be created from existing ones. This can be achieved using the `+` (choice) or `&` (conjunction) operator in the provided logic.

(R_9) x While the abstract syntax does include operators which cover sequential composition, no explicit attempt has been made to support overlapping clauses or the concept of precedence.

(R_{10}) ✓ The CNL of Pace and Rosner [2009] can be easily extended by simply increasing the span of words that can be used. They do mention the conflict between the higher and lower level reasoning of a contract, and how if any conflicts are discovered, these can be expressed in a natural language. Despite this not being quite the same as an escape mechanism, the language can clearly be extended.

(R_{11}) x There is no unique identifier associated with clauses, so referencing earlier clauses is not possible.

3.2.2 Kowalski [2020]

Logical English (LE) is a CNL which can be understood by a reader without any background in Computer Science. Modelled on the language of law, which, according to

Kowalski [1995], can itself be viewed as a programming language that is executed by humans rather than computers, LE is intended to be suitable for general computation (for example, tasks such as programming, databases and knowledge representation).

It is worth noting that there have been several variants of LE, focussed primarily on legal applications. The most promising aspects of LE are that it is inspired the language of law, and it is intended to be developed as a series of extensions. These two factors combined make LE highly promising for future legal applications. Furthermore, since LE can be directly translated to Prolog, this facilitates an effortless conversion into lower layers of the language stack, which could enable it to generate smart contract code.

Kowalski [2020] presents an informal introduction to LE, along with some examples of LE in action. The example below is taken from Karadotchev [2019], who applied LE to various well known financial contracts. We first provide the natural language contract, followed by its representation in LE. This example was derived from Section 2 of the ISDA Master Agreement.

Natural Language:

Each party will make each payment or delivery specified in each Confirmation to be made by it, subject to the other provisions of this Agreement.

CNL:

An action is an obligation for a party if
a transaction specifies the action , and
the transaction is a transaction between the party and
    a counterparty , and
the action is directed from the party to
    the counterparty , and
either
    the action is a payment or
    the action is a delivery ;
and
it cannot be shown that
the action is not an obligation for the party .

Using this example, we can analyse LE against the provided requirements, as done previously. We find that LE does not fail to meet a single requirement.

(R_1) ✓ The above example demonstrates that LE uses a natural language (English) with a restricted syntax and grammar.

(R_2) ✓ This example incorporates agents (parties and counterparties), events (actions) and objects (the definitions provided). Deontic terms have also been expressed.

(R_3) ? LE can easily be understood by a human without a formal background in Computer Science, but it may be difficult for such an individual to generate LE. From the example provided, LE appears to be quite verbose, and the entire structure of the natural language is changed. Without clear and extensive documentation, writing LE may prove to be challenging.

(R_4) ✓ Deontic terms are used throughout LE. In the provided example, we see that an obligation has been defined. Permission and prohibition can be defined in a similar fashion. LE is always written in present tense, and is capable of expressing temporal aspects, such as the time or date of a given event. Operational aspects are captured through LE's descriptive nature of permissible actions.

(R_5) ✓ While LE is able to capture deontic terms, it is based on a more general logic for abductive logical programming (ALP) (Kakas et al. [1992]), with goals in first-order logic. ALP is distinct to deontic logic, and as such, this requirement is satisfied.

(R_6) ✓ Translations from LE to executable Prolog code are possible and can even be fully automated. Karadotchev [2019] provides an interpreter which can automatically translate Simplified Logical English (LE but with fewer features) into Prolog code. No indication is provided as to whether code can be generated for other low-level programming languages.

(R_7) ? It is possible for the detection of inconsistencies to be performed, through the use of strong negation (Karadotchev [2019], Section 3.8). This could be done by adding a new rule for each property, which is the negation of the original property. If a program is able to prove that both the property and its negation are true, then an inconsistency exists.

(R_8) ? While LE does not explicitly address the idea of building contracts based on other contracts, since it can be translated into Prolog it would not be difficult to meet this requirement. This is because the concept of inheritance is native to Prolog.[9]

(R_9) ✓ As seen in the application of LE to the ISDA Master Agreements in Section 4.5.2, LE is capable of supporting overlapping clauses.

(R_{10}) ✓ LE prides itself on being extensible; it is intended to be developed as a series of extensions. It is also written in a way that closely resembles natural language. To date, there have already been three implementations of variants of LE, focussed primarily on legal applications (Kowalski [2020]), suggesting that it can be extended.

[9]One explanation of inheritance in Prolog can be found in `https://sicstus.sics.se/sicstus/docs/3.12.11/html/sicstus/Inheritance.html`

(R_{11}) ? While LE does not advertise the use of unique identifiers for clauses, it has been used in several examples from Karadotchev [2019] when implemented in Prolog. This suggests that there may be a way to reference other clauses.

3.2.3 Diedrich [2020]

Lexon enables us to write normal contracts that work as blockchain smart contracts; the same text is both a program and a legal agreement. While this is quite an ambitious objective, Lexon indeed does perform both of these roles whilst remaining highly readable for people across all domains. Lexon can be very easily understood by individuals without any background in Logic, Computer Science, or Law. It can also, without requiring any modification, be run as a program (e.g., as a smart contract on the blockchain).

We now present an example of Lexon. At its core, Lexon was built so that it could represent both contracts and code, simultaneously. Therefore, in this case, we only provide the Lexon version of an example contract, since it was designed to be completely human readable and acts as the natural language version too.

CNL:

LEX Escrow Contract.

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

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

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

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

(R_1) ✓ Lexon uses a (highly readable) form of English to express contracts, with a restricted base vocabulary of roughly 130 words.

(R_2) ✓ All Lexon contracts follow the same format. The basic structure of a contract in Lexon is: Head; Definitions; Recitals; Clauses. Objects and parties are captured in the 'Definitions' section, initial actions and temporal aspects are captured in

the 'Recitals' section, and further deontic and operational aspects are captured in the 'Clauses' section.

(R_3) ✓ Anyone can read Lexon and understand what it means. It can be drafted very easy using the simple format mentioned as well as the extensive supporting documentation.

(R_4) ✓ In the 'Clauses' section of a Lexon agreement, deontic and operational aspects are captured. Temporal aspects also exist; clauses can make use of time references by referencing specific dates.

(R_5) ✓ Lexon does not use deontic logic, beyond the atomic keyword 'may'. Instead, Boolean 'true' and 'false' values are used to articulate contracts. Therefore, the paradoxes of deontic logic are avoided.

(R_6) ✓ Currently, the Lexon compiler can create both Ethereum solidity smart contracts and Aeternity Sophia smart contracts from Lexon code. More target platforms will be added in the future, suggesting that converting Lexon into lower-level languages is certainly possible.

(R_7) ? While Lexon does not support the detection of inconsistencies, it does prohibit inconsistencies from being written. If a Lexon contract has an inconsistency and tries to get executed, it would not compile. Furthermore, using the organised format the a Lexon contract is written in, detecting inconsistencies by reading the contract would be a lot easier since all definitions are grouped together.

(R_8) ✓ In a similar style to how an uncontrolled natural language contract can be duplicated in order to form a new contract based on an existing one, Lexon works in much the same way. Lexon contracts can be viewed as templates, which can be duplicated and modified.

(R_9) ✓ Lexon code can be embedded, e.g., as a Schedule of a larger Master Agreement. The `LEX` keyword in the header provides a clear separation between the different parts of such a larger contract. Therefore, it is possible to distinguish overlapping clauses in different parts of a contract.

(R_{10}) ✓ There are several ways in which Lexon can be extended. First, any nouns are permitted, so this already gives the language some degree of extensibility. Second, clauses can have any name. In fact, it is possible (and common) to name clauses using a descriptive phrase. Finally, the Lexon Extension Form can be used to define a new verb, and give that verb an equivalent piece of code which the Lexon compiler can generate when the contract is executed.

(R_{11}) ? While each clause in a Lexon contract has a unique name, which may make it easier to reference other clauses, there is not description of how these clause names

would be used to reference other clauses. It is theoretically possible to reference other clauses in a contract, but not an explicit feature of Lexon.

3.2.4 Fuchs et al. [2008]

Attempto Controlled English (ACE) is a general purpose CNL, which is human and machine understandable. Originally intended for use in software specification, ACE has shifted towards a focus on knowledge representation, specifically in the domain of the semantic web. The vocabulary of ACE consists of three key elements: predefined function words (e.g., determiners, functions and prepositions), pre-defined phrases (e.g., "there is", "it is false that"), and content words (nouns, verbs, adjectives, and adverbs). Since ACE is not a CNL designed for the purposes of representing or analysing legal contracts, we refrain from a full analysis against the provided requirements. It is, however, still included in the requirements comparison matrix seen in Section 3.3.

3.2.5 SBVR [2008]

The Semantics of Business Vocabulary and Business Rules (SBVR) specification defines an English vocabulary for which rules can be stated, known as Structured English.[10] Structured English is a CNL which was first developed for business rules. For rule expression, Structured English uses two main styles: a prefixed rule keyword style and an embedded rule keyword style. The vocabulary is extensible and consists of four key elements: terms, names, verbs and keywords. Each of these sentence constituents has its own style and colour, which can help the reader to better understand the composition of rules and sentences. The language strictly defines the allowed sentence constituents, but is more relaxed when it comes the ordering of such constituents. Similarly to ACE, SBVR too was not designed for the purpose of computable contracting. Therefore, we do not provide a complete requirements analysis, but still include it in the requirements comparison matrix of Section 3.3 in order to serve as a benchmark for comparison.

[10] `https://www.omg.org/spec/SBVR/1.0/PDF`

3.3 Summary and Gap Analysis

In Table 1, the five selected CNLs are evaluated against the proposed requirements based on the preceding analysis. The three most promising CNLs for enabling computable contracting perform well, and all achieve a high score. The two general purpose CNLs are, understandably, unable to perform as well against these requirements. As can be seen in this table, there are certain requirements which appear to be more challenging to meet than others. These requirements include (R_7) "Supports the detection of inconsistencies", (R_9) "Supports overlapping clauses where one takes precedence over another", and (R_{11}) "Ability to reference other clauses". These requirements share a common theme: they require that the individual components of a contract can, in certain ways, interact with other components.

Table 1: Requirements comparison matrix.
Pac is Pace and Rosner [2009], LE is Kowalski [2020], Lex is Diedrich [2020], ACE is Fuchs et al. [2008], SBVR is SBVR [2008]. A checkmark (score = 1) denotes that the requirement is met. A question mark (score = 0.5) denotes that it is possible. A cross (score = 0) denotes that the requirement is not met.
Constructed using guidance from Kuhn [2014].

	Pac	LE	Lex	ACE	SBVR
R_1	✓	✓	✓	✓	✓
R_2	✓	✓	✓	x	x
R_3	✓	?	✓	✓	✓
R_4	✓	✓	✓	?	✓
R_5	✓	✓	✓	✓	x
R_6	✓	✓	✓	✓	x
R_7	x	?	?	x	x
R_8	✓	?	✓	x	x
R_9	x	✓	✓	x	x
R_{10}	✓	✓	✓	✓	✓
R_{11}	x	?	?	?	?
Total Score	8	9	10	6	4.5

Based on this analysis, there is a clear need for a CNL which places an emphasis on the connectedness of the clauses of a contract. This would significantly help in meeting these more challenging requirements. Section 4 introduces a new language, devised with this goal in mind.

4 The Controlled Components Contract Language (CoLa)

As seen in Section 3.3, there does not exist a CNL which can satisfy all of the requirements defined in Section 3.1. The best attempt in existing literature meets nine out of the eleven requirements provided. This is a clear identification of the need for a new CNL which can meet *all* of the requirements given. The Controlled Components Contract Language, or CoLa for short, is a novel CNL which attempts to meet all of the requirements given, while remaining simple enough for lawyers without a background in Logic or Computer Science to use when drafting contracts. This section explains the design and implementation of CoLa, provides the complete grammar for CoLa, and demonstrates a full end-to-end example of how it can be used. This is followed by a comparative example of CoLa against other CNLs.

4.1 Design

Before providing a complete formal grammar for CoLa in Section 4.3, we present an overview of its design. At its core, CoLa works by modelling contracts as lists of components. Each component is one of the following:

Definition. A definition is used to give a value to a name. For example, we may wish to say that `PartyA` is `The Employer` and that `PartyB` is `An Employee`. Alternatively, we may wish to say that the `HourlyPay` is `10 pounds`. In all cases, a definition is composed of two parts. Note that these are not type expressions, but rather they associate values to names. Definitions can be only be joined in conjunction. A discussion of contradictory definitions is deferred until Section 4.1.1.

Conditional Definition. Conditional definitions are definitions which only apply when a certain Boolean expression is true. They are composed of two parts: a definition, as explained above, and an expression. These parts are combined using an `IF` token, and an optional `THEN` token. For example, an expression might be something such as `PartyA paid more than PartyB`, and the conditional definition can either be `IF PartyA paid more than PartyB THEN PartyB is the Debtor`, or `PartyB is the Debtor IF PartyA paid more than PartyB`. Expressions always include a comparison operator (less than, equal to, or more than), and are never found alone; they must be coupled with a definition to form a conditional definition. Expressions can be joined both in conjunction and disjunction. For example, we can form a disjunction of expressions in a conditional definition by saying `IF PartyA paid more than PartyB OR PartyC paid more than PartyB THEN PartyB is the Debtor`. We can form a conjunction of expressions by saying `IF PartyA paid more than PartyB AND PartyC paid more than PartyB THEN PartyB is the Debtor`.

Statement. A statement is a central component in all contracts. It is a declaration of an obligation, permission, or prohibition belonging to a given subject. An exam-

ple statement may be something such as `Alice must pay 10 pounds on the 1st January 2021`. Statements are made up of a subject, deontic term, verb, object, and date. Statements can be joined both in conjunction and disjunction.

Conditional Statement. Conditional statements are statements which only apply when a certain condition is true. They are composed of two parts, a statement and a condition. In the same manner as conditional definitions, these parts are combined using an `IF` token, and an optional `THEN` token. For example, a condition may be something such as `Alice paid 4 pounds on the 31st December 2020`, and the conditional statement may be `IF Alice paid 4 pounds on the 31st December 2020 THEN Alice must pay 6 pounds on the 1st January 2021`, or `Alice must pay 6 pounds on the 1st January 2021 IF Alice paid 4 pounds on the 31st December 2020`. Furthermore, CoLa allows two types of conditions: (i) a condition about something that has happened (e.g., `Alice paid 4 pounds on the 31st December 2020`), and (ii) a condition that is about the existence of an obligation, permission, or prohibition (e.g., `Alice must pay 4 pounds on the 31st December 2020`). Conditions can be joined both in conjunction and disjunction.

One of the main reasons for structuring contracts in this way is that the components can be joined together to build larger, more complex contracts, capable of expressing a variety of different clauses. Each component described above is composed of smaller elements, which can be seen in the formal grammar, provided in Section 4.3. This modular structure is particularly advantageous as it allows for a wide degree of flexibility when expressing contracts. Furthermore, the components introduced above are mainstream in the legal domain; there is no new vocabulary of terms for lawyers to learn, reinforcing the fact that no prior knowledge of programming is required.

4.1.1 Challenging Requirements

When designing this CNL, it was necessary to ensure that it met all of the requirements provided in Section 3.1, including the more challenging requirements identified in Section 3.3. A complete requirements analysis of CoLa can be found in Section 5.2. For now, we briefly mention how CoLa addresses some of the more challenging requirements that other CNLs were not able to meet.

Section 3.3 recognised that the common hurdles for many of the CNLs were (R_7) "Supports the detection of inconsistencies", (R_9) "Supports overlapping clauses where one takes precedence over another", and (R_{11}) "Ability to reference other clauses". With this in mind, CoLa was designed in such a way to meet these requirements, without compromising the other, more easily achieved, requirements.

The way we tackle these specific requirements is as follows. Every component in a contract must have an associated `ID`. This `ID` is listed at the beginning of every component,

and is a numeric term inside square brackets. We assert that the component with the greater `ID` takes precedence in the event of an inconsistency. For example, assume that the Master Agreement of some contract, when written in CoLa, contains the following definitions:

[11] (Subject TheSeller) `IS` (Subject Alice)

[12] (Subject TheBuyer) `IS` (Subject Bob)

It is possible that the corresponding Schedule, which is used to identify additions to the Master Agreement, contains the following definitions:

[296] (Subject TheSeller) `IS` (Subject Charlie)

[297] (Subject TheBuyer) `IS` (Subject David)

These latter definitions may be written in the Schedule to be used in the event that Alice and Bob are no longer present, for some given reason. At the surface, this is clearly a scenario where we have overlapping clauses, and an inconsistency. The CNL has, in two separate instances, identified conflicting definitions for `TheSeller` and `TheBuyer`. The resolution that CoLa proposes is simply for the components with the greater `ID` to take precedence, in this case [296] and [297]. Therefore, in this scenario, `TheSeller` would be `Charlie` and `TheBuyer` would be `David`.

In addition to this, we allow explicit precedences to be stated. This is done by allowing `ID`s to take the form `[X(Y)]`, which means that component with `ID [X]` takes precedence over component with `ID [Y]`. In the above example, this would be expressed as:

[296(11)] (Subject TheSeller) `IS` (Subject Charlie)

[297(12)] (Subject TheBuyer) `IS` (Subject David)

This demonstrates that component [296] takes precedence over component [11] and component [297] takes precedence over component [12]. This example shows how we address inconsistencies, overlapping clauses where one takes precedence over another, and the ability to reference other clauses.

4.2 Implementation

Following a brief description of the design of CoLa, we now turn to its implementation. CoLa is implemented entirely in Miranda (Turner [1985], Clack et al. [1995]), a polymorphic, higher order functional programming language which strongly influenced the design of Haskell. This decision was made since Miranda is lightweight programming language, which was designed to have a very small memory footprint, making it very easy to get a quick program up and running, allowing for rapid prototyping when building the CNL.

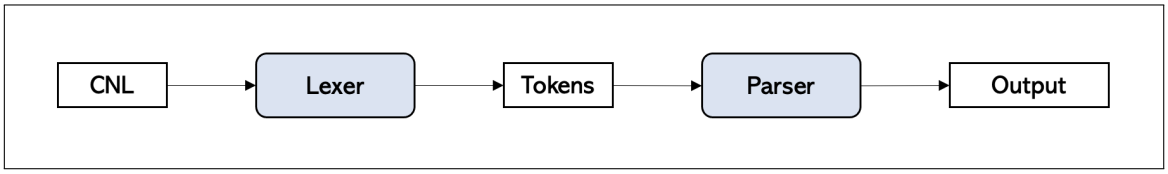

Figure 1: Converting a controlled natural language into a parse tree. A lawyer will draft the contract, in a controlled natural language. The lexer will then tokenize this input. Finally, the parser takes the list of tokens and converts this into the CoLa representation, consisting of a parse tree and the components in the original CNL.

CoLa's implementation was built in two separate parts, the lexer and the parser. The lexer accepts the CNL as input, and outputs a list of tokens. These tokens are then provided as input to the parser, which outputs the CoLa representation of the original input. Figure 1 gives a high level view of this process.

The final representation is in the form of a parse tree, and a list of components. The parse tree gives the user of the CNL an intuitive feel for the different hierarchies and dependencies present in the contract, and the list of components allows the user to easily identify which clauses are relevant at certain times. The output could be used for a variety of reasons, including (i) static analysis of the contract, (ii) dynamic analysis (or simulation) of contract performance, and (iii) code generation for automation of parts of the contract.

4.2.1 Lexer

The lexer is responsible for converting contracts, written in the CNL, into a list of tokens. Since it accepts a CNL as input, the lexer first converts the input into a list of words. It does this using a custom `split` function, seen in Listing 1. This split function operates on fullstops and whitespace to separate the text into sentences and the sentences and words respectively, resulting in a list of words. The lexer can ignore the presence of unnecessary whitespace. While the lexer does allow other punctuation such as commas and round brackets to be part of the input, they do not influence the resulting tokens it produces. Square brackets are reserved for component `ID`s. Note that the lexer treats all inputs, including component `ID`s, as strings.

Listing 1: The split helper function, used by the lexer.

```
split :: char -> [char] -> [[char]]
split c input = xsplit c input []
                where
                xsplit c []       []  = []
                xsplit c []       acc = [acc]
                xsplit c (c:rest) acc = [acc] ++ (xsplit c rest [])
                xsplit c (x:rest) acc = xsplit c rest (acc ++ [x])
```

Once the lexer has processed the input into a list of words, the lexer uses pattern matching to build a list of corresponding tokens. The full set of tokens can be found in Appendix A. This list of tokens is subsequently fed into the parser for further processing. The lexer can also perform trivial error handling. For example, if it detects that a sentence ends with the word "`the`", then it will show the user an error message "`Sentence ends with definite article`".

Consider what happens when the lexer sees the phrase "`It is the case that`". It pattern matches this to form a `ComponentAffirmation` token. Similarly, in the opposite scenario, "`It is not the case that`" is transformed into a `ComponentNegation` token. The parser would interpret these tokens as the component holding or not holding respectively. Another example worth noting is how the lexer converts dates into date tokens. The natural language input of "`on the 15 March 2021`" is tokenized into the tokens `[ComponentON, ComponentTHE, Number 15, Number 3, Number 2021]`. While it is easy to see how the day and year can be tokenized, the month token is formed by converting the month provided to an index in a list of integers from 1 to 12. The full code for the lexer can be found in Appendix B.1.

4.2.2 Parser

The role of the parser is to convert the list of tokens given to it as input into a parse tree, along with a separate list of components that are present. When combined with the lexer, the system is able to convert extensive portions of text, which can be hard to reason about, into a clear structured representation, identifying the definitions, conditional definitions, statements, and conditional statements present.

The parser is implemented in a highly modular fashion. Each parse function is of the same type; it will accept a list of tokens and return a list of 2-tuples, each of which contains the remainder of the input not yet parsed (as a list) and the parsed output. The code for this definition can be seen in Listing 2. Building on the work of Fokker [1995], the parser will take as input a `[*]` and return as output a `[([*],**)]`. The polymorphic types `*` and `**` have been used to keep this as generic as possible.

Listing 2: Definition of the type t_parser.

```
t_parser * ** == [*] -> [([*],**)]
```

To parse a component, the parser first parses the various subcomponents present, before deciding how it should combine the subcomponents in order to form the output representation of the component. For example, the code that parses a statement, `parse_simplestatement`, calls (a minimum of) six other functions, each dedicated to parsing different elements of the statement. Since the code for this function is long, an abbreviated version which demonstrates this point can be seen in Listing 3. We see that `parse_simplestatement` first

parses the `compID`, then parses the `holds` (i.e. the affirmation or negation component). after this, it parses the `date`, `subject`, `verb`, and `object`, in an order which depends on how the statement was provided. Each component of the statement is parsed separately, before the collective statement is converted into a component. The full code for the parser, which includes the code for `parse_simplestatement`, can be found in Appendix B.2.

Listing 3: Abbreviated code for parsing a statement.

```
parse_simplestatement :: t_parser token (maybe t_component)
parse_simplestatement = maybestatement2maybecomponent . (then a
    parse_compID_statement (then f parse_holds_statement (alt (s_parse "dsvo
    ") (alt (s_parse "dovs") alt (s_parse "sdvo") (alt (s_parse "svod") (alt
     (s_parse "ovsd") (s_parse "ovds"))))))))
                          where
                          ...
                          s_parse "dsvo" = then (merge "D") parse_d (then (
    merge "vo") parse_s (then (merge "o") parse_v parse_o))
```

As seen in Listing 3, the parser was also built in such a way to allow multiple orderings of the same sentence. For example, the phrases: "`Alice shall deliver a bicycle on the 15 March 2021`", "`On the 15 March 2021 Alice shall deliver a bicycle`", and "`Alice on the 15 March 2021 shall deliver a bicycle`" are all valid. This demonstrates that the parser is robust to the natural variants present in the ways that users may decide to draft their contracts.

Combining the intermediate outputs of the parser is done using two parser combinators: a sequencer operation and an alternator operation (Fokker [1995]). These can be seen in Listing 4. The sequencer operation allows us to parse multiple elements sequentially, and even provides additional generality which allows us to determine how we would like to combine the elements parsed. For example, `parse_definition` uses the sequencer operation to first parse a `Name` token, followed by a `ComponentIS`, followed by another `Name` token (e.g., `PartyA is Alice`). The alternator operation allows us to parse elements in alternative ways. For example, `parse_simplestatement` uses the alternator operation to accommodate for the potential multiple orderings of the same statement. Both of these helper functions are used regularly throughout the code.

Listing 4: Sequencer and alternator helper functions.

```
then :: (**->**->**) -> (t_parser * **) -> (t_parser * **) -> (t_parser *
   **)
then f p1 p2 = (concat.((map g).p1))
                 where
                 g (rem,res)  = [(rem2, f res res2) | (rem2, res2) <- (p2
   rem)]

alt :: t_parser * ** -> t_parser * ** -> t_parser * **
alt p1 p2 = g
             where
             g ip = (p1 ip)++(p2 ip)
```

When parsing components, it is often the case that multiple correct outputs exist. A 'list of successes' method (Wadler [1985]) is used to collect all of the possible outputs. This leverages one of the core features of functional programming: lazy evaluation. The list of successes method works by replacing terms that may raise an exception with a term that produces a list of values. The list of values can include the empty list, indicating a failed parsing, or one or more of the possible correct parse outputs, indicating a success. One consequence of this that was observed is that larger contracts take much longer to parse. In fact, naive combinatory parsing requires exponential time and space.[11] A discussion of the scalability of CoLa is deferred to Section 5.2.3.

Another interesting aspect of the parser is the way that it can combine multiple contracts, using a recursive call. Listing 5 shows an abbreviated version of the function `parse_contract`. To parse a contract, the parser can either parse a single component, or it can parse a single component followed by another contract in conjunction. The full code for `parse_contract`, along with other examples of this recursive style of programming being used when parsing each of the components, can be found in Appendix B.2.

Listing 5: Code for parsing a contract.

```
parse_contract :: t_parser token (maybe t_contract)
parse_contract = alt parse_component (then f parse_component (then g
    parse_contractAND parse_contract))
                 where
                 ...
                 f (Just (Contract x))    (Just (Contracts_and y)) = Just (
    Contracts_and (x:y))
                 f (Just (Contract x))    (Just (Contract y))      = Just (
    Contracts_and [x, y])
                 ...
```

[11] https://en.wikipedia.org/wiki/Parser_combinator

4.3 The CNL Syntax

After understanding the design and implementation of the CNL, we now provide the abstract syntax definition for the CNL. We then provide a concrete syntax for the CNL, followed by examples to demonstrate how CoLa is used. In order of appearance, a brief explanation of the symbols given in the syntax is provided: 'ϕ' refers to the empty set (a contract can be empty); '?' following an element indicates that the element is optional. Note the distinction between the use of `C-AND` to express the conjunction between components and `AND` to express the conjunction within components.

CoLa abstract syntax (BNF):

⟨*contract*⟩ ::= ϕ
| ⟨*component*⟩
| ⟨*component*⟩ `C-AND` ⟨*contract*⟩

⟨*component*⟩ ::= ⟨*definition*⟩
| ⟨*conditional-definition*⟩
| ⟨*statement*⟩
| ⟨*conditional-statement*⟩

⟨*definition*⟩ ::= ⟨*simple-definition*⟩
| ⟨*simple-definition*⟩ `AND` ⟨*definition*⟩

⟨*simple-definition*⟩ ::= ⟨*ID*⟩ ⟨*subject*⟩ `IS` ⟨*subject*⟩
| ⟨*ID*⟩ ⟨*subject*⟩ `EQUALS` ⟨*numerical-expression*⟩

⟨*numerical-expression*⟩ ::= ⟨*num*⟩
| ⟨*numerical-object*⟩[12]
| ⟨*numerical-expression*⟩ ⟨*operator*⟩ ⟨*numerical-expression*⟩

⟨*operator*⟩ ::= `PLUS`
| `MINUS`
| `TIMES`
| `DIVIDE`

⟨*conditional-definition*⟩ ::= ⟨*definition*⟩ `IF` ⟨*condition*⟩
| `IF` ⟨*condition*⟩ `THEN` ⟨*definition*⟩

[12]The inclusion of numerical objects in the grammar was a last-minute improvement that has not yet been fully implemented in the parser.

⟨*statement*⟩ ::= ⟨*simple-statement*⟩
| ⟨*simple-statement*⟩ `OR` ⟨*statement*⟩
| ⟨*simple-statement*⟩ `AND` ⟨*statement*⟩

⟨*conditional-statement*⟩ ::= ⟨*statement*⟩ `IF` ⟨*condition*⟩
| `IF` ⟨*condition*⟩ `THEN` ⟨*statement*⟩

⟨*simple-statement*⟩ ::= ⟨*ID*⟩ ⟨*holds*⟩? ⟨*subject*⟩ ⟨*modal-verb*⟩ ⟨*verb*⟩ ⟨*object*⟩ ⟨*date*⟩
| ⟨*ID*⟩ ⟨*holds*⟩? ⟨*subject*⟩ ⟨*date*⟩ ⟨*modal-verb*⟩ ⟨*verb*⟩ ⟨*object*⟩
| ⟨*ID*⟩ ⟨*holds*⟩? ⟨*date*⟩ ⟨*subject*⟩ ⟨*modal-verb*⟩ ⟨*verb*⟩ ⟨*object*⟩

⟨*condition*⟩ ::= ⟨*simple-condition*⟩
| ⟨*simple-condition*⟩ `OR` ⟨*condition*⟩
| ⟨*simple-condition*⟩ `AND` ⟨*condition*⟩

⟨*simple-condition*⟩ ::= ⟨*ID*⟩ ⟨*holds*⟩? ⟨*subject*⟩ ⟨*verb-status*⟩ ⟨*object*⟩ ⟨*date*⟩
| ⟨*ID*⟩ ⟨*holds*⟩? ⟨*subject*⟩ ⟨*date*⟩ ⟨*verb-status*⟩ ⟨*object*⟩
| ⟨*ID*⟩ ⟨*holds*⟩? ⟨*date*⟩ ⟨*subject*⟩ ⟨*verb-status*⟩ ⟨*object*⟩
| ⟨*ID*⟩ ⟨*holds*⟩? ⟨*subject*⟩ ⟨*modal-verb*⟩ ⟨*verb*⟩ ⟨*object*⟩ ⟨*date*⟩
| ⟨*ID*⟩ ⟨*holds*⟩? ⟨*boolean-expression*⟩

⟨*boolean-expression*⟩ ::= ⟨*subject*⟩ ⟨*verb-status*⟩ ⟨*comparison*⟩ ⟨*subject*⟩

⟨*ID*⟩ ::= '`[`' ⟨*num*⟩ '`]`'
| '`[`' ⟨*num*⟩ '`(`' ⟨*num*⟩ '`)`' '`]`'

⟨*holds*⟩ ::= ``` `it is the case that` ''
| ``` `it is not the case that` ''

⟨*subject*⟩ ::= ⟨*string*⟩

⟨*verb*⟩ ::= `DELIVER`
| `PAY`
| `CHARGE`

⟨*verb-status*⟩ ::= `DELIVERED`
| `PAID`
| `CHARGED`

⟨*comparison*⟩ ::= `LESS THAN`
| `EQUAL TO`
| `MORE THAN`

⟨*modal-verb*⟩ ::= `OBLIGATION`
| `PERMISSION`
| `PROHIBITION`

⟨*date*⟩ ::= ‘‘`on the`’’ ⟨*num*⟩ ⟨*month*⟩ ⟨*num*⟩
| ‘‘`on`’’ `ANYDATE`
| ‘‘`on`’’ `ADATE`
| ‘‘`on`’’ `THEDATE`

⟨*month*⟩ ::= ‘‘`January`’’
| ...
| ‘‘`December`’’

⟨*object*⟩ ::= ⟨*numerical-object*⟩
| ⟨*nonnumerical-object*⟩

⟨*numerical-object*⟩ ::= `POUNDS` ⟨*num*⟩
| `DOLLARS` ⟨*num*⟩
| `EUROS` ⟨*num*⟩
| `AMOUNT` ⟨*subject*⟩

⟨*nonnumerical-object*⟩ ::= `SOMECURRENCY` ⟨*string*⟩
| `REPORT` ⟨*string*⟩
| `NAMEDOBJECT` ⟨*string*⟩
| `OTHEROBJECT` ⟨*string*⟩

⟨*string*⟩ ::= ⟨*char*⟩
| ⟨*char*⟩ ⟨*string*⟩

⟨*char*⟩ ::= ‘ ’
| ‘`A`’ | ‘`B`’ | ... | ‘`Z`’
| ‘`a`’ | ‘`b`’ | ... | ‘`z`’

⟨*num*⟩ ::= ⟨*digit*⟩
| ⟨*digit*⟩ ⟨*num*⟩

⟨*digit*⟩ ::= 0 | 1 | ... | 9

The concrete syntax for CoLa defines the mapping between the terminals provided by the abstract syntax and the words recognised by the lexer. For each terminal provided in the abstract syntax definition, we provide the corresponding lexeme. In almost all cases, the terminal's corresponding lexeme is precisely the (uppercase) terminal itself. For example, the lexeme of the terminal `C-AND` is "C-AND", the lexeme of the terminal `PLUS` is "PLUS", the lexeme of the terminal `ANTDATE` is "ANYDATE", and so on. For brevity, the concrete syntax only lists terminal's whose lexeme are not precisely the terminal string. The left side of the $\mapsto$ operator is the BNF terminal, and the right hand side is the lexeme.

CoLa concrete syntax:

`DELIVER` $\mapsto$ deliver

`PAY` $\mapsto$ pay

`CHARGE` $\mapsto$ charge

`DELIVERED` $\mapsto$ delivered

`PAID` $\mapsto$ paid

`CHARGED` $\mapsto$ charged

`LESS THAN` $\mapsto$ less than

`EQUAL TO` $\mapsto$ equals | equal to

`MORE THAN` $\mapsto$ more than | greater than

`OBLIGATION` $\mapsto$ shall | must

`PERMISSION` $\mapsto$ may

`PROHIBITION` $\mapsto$ is forbidden to

`POUNDS` $\mapsto$ GBP | pounds | quid

`DOLLARS` $\mapsto$ USD | dollars | bucks

`EUROS` $\mapsto$ EUR | euros

4.4 A Contract Example

We now provide an example which demonstrates how the CNL proposed in this paper can be used. The example shows an informal explanation and three stages of an example contract: the CNL version, the tokens and the output. After the CNL is drafted and provided as input to the lexer, the tokens are produced. Then the parser, upon reading the tokens as input, will produce the output. The output is given as a list of components, and a corresponding parse tree including these components has been provided.

Explanation: Alice would like to purchase a bicycle. If she pays 100 pounds on the 1 April 2021, or she pays 120 dollars on the same date, then Bob must deliver a bicycle to Alice on the 5 April 2021. Bob may also include a receipt if he wishes, and under no circumstances is he allowed to charge a delivery fee.

CNL:

IF [1] it is the case that Alice paid POUNDS 100 on the 1 April 2021
OR
[2] it is the case that Alice paid DOLLARS 120 on the 1 April 2021

THEN [3] it is the case that Bob must deliver OTHEROBJECT “bicycle” on the 5 April 2021

C-AND
[4] it is the case that Bob may deliver REPORT “receipt” on ANYDATE
AND
[5] it is the case that Bob is forbidden to charge AMOUNT “delivery fee” on ANYDATE.

Tokens:

```
[ComponentIF, CompID "1", ComponentAffirmation, Subject "Alice", Verb_status Paid, Object (Pounds 100), ComponentON, ComponentTHE, Number 1, Number 4, Number 2021, ComponentOR, CompID "2", ComponentAffirmation, Subject "Alice", Verb_status Paid, Object (Dollars 120), ComponentON, ComponentTHE, Number 1,Number 4, Number 2021,ComponentTHEN,CompID "3", ComponentAffirmation, Subject "Bob", Modal_verb Obligation, Verb Deliver, ComponentA, Object (OtherObject "bicycle"), ComponentON, ComponentTHE, Number 5, Number 4, Number 2021, ContractAND, CompID "4", ComponentAffirmation, Subject "Bob", Modal_verb Permission, Verb Deliver, ComponentA, Object (Report "receipt"), ComponentON, Date ANYDATE, ComponentAND, CompID "5", ComponentAffirmation, Subject "Bob", Modal_verb Prohibition, Verb Charge, ComponentA, Object (OtherObject "delivery fee"), ComponentON, Date ANYDATE]
```

Output:

[1] it is the case that Alice paid POUNDS 100 on the 1 April 2021
[2] it is the case that Alice paid DOLLARS 120 on the 1 April 2021
[3] it is the case that Bob shall deliver OTHEROBJECT “bicycle” on the 5 April 2021
[4] it is the case that Bob may deliver REPORT “receipt” on ANYDATE
[5] it is the case that Bob is forbidden to charge AMOUNT “delivery fee” on ANYDATE.

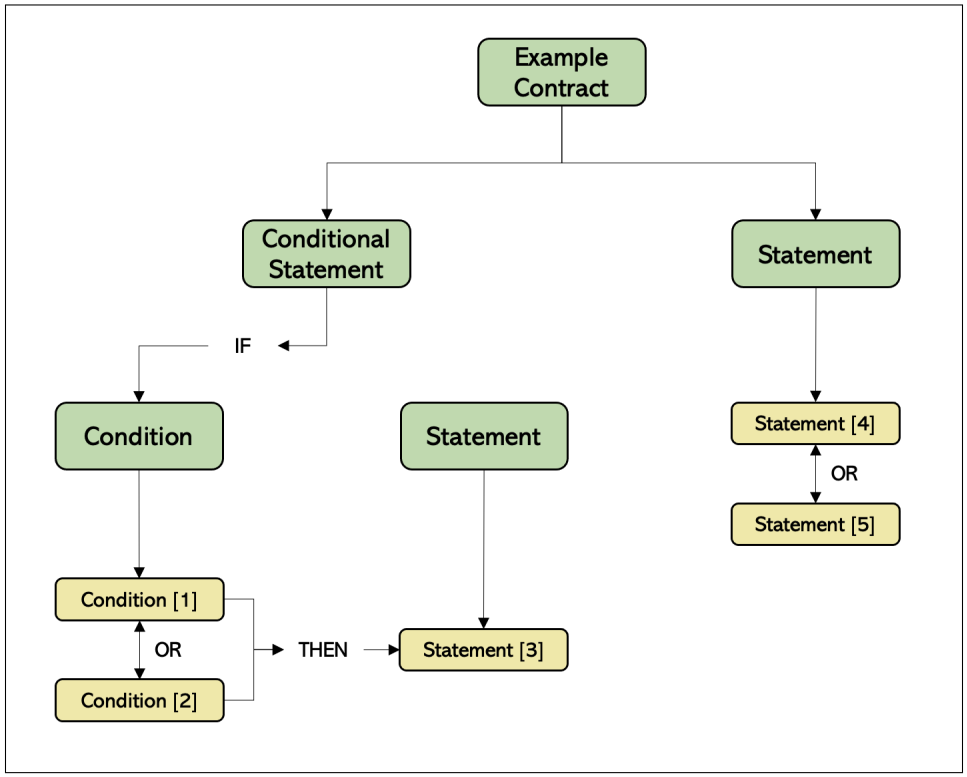

Figure 2: CoLa Parse Tree for the example contract.

Reading the parse tree in Figure 2 should be done in a depth-first approach, scanning each branch (or component) as far as you can before moving onto the next. Note that since components can only be combined in conjunction, we omit the `C-AND` between them in the parse tree. In this simple example, we have two components: one conditional statement (composed of `IDs [1], [2], [3]`) and one statement (composed of `IDs [4], [5]`). The conditional statement conveys they central objective of the example contract, that Alice can pay some amount in exchange for Bob delivering a bicycle. The statement conveys the additional information, that Bob can provide a receipt but is not allowed to charge a delivery fee.

4.5 Comparative Examples

In this section, we provide in depth examples of the same contract expressed in four different CNLs. Specifically, we consider the representation of Section 2(c) of the ISDA Master Agreement, concerning the 'Netting of Payments', seen in Figure 3.

(c) ***Netting of Payments.*** If on any date amounts would otherwise be payable:—

(i) in the same currency; and

(ii) in respect of the same Transaction,

by each party to the other, then, on such date, each party's obligation to make payment of any such amount will be automatically satisfied and discharged and, if the aggregate amount that would otherwise have been payable by one party exceeds the aggregate amount that would otherwise have been payable by the other party, replaced by an obligation upon the party by which the larger aggregate amount would have been payable to pay to the other party the excess of the larger aggregate amount over the smaller aggregate amount.

Figure 3: Section 2(c) of the 2002 ISDA Master Agreement.

The ISDA Master Agreement is used to provide legal and credit protection for parties who enter into over-the-counter (i.e., without the supervision of an exchange) derivatives transactions. To the untrained reader, understanding the *Netting of Payments* excerpt can be quite challenging when reading it for the first time. The rule states that, under certain conditions, multiple payments involving the same parties will be replaced by a single payment worth the net amount. As will be shown, when this contract is drafted in a CNL, it can be far easier to understand.

In what follows, we represent this section of the ISDA Master Agreement in four CNLs:

(i) Pace and Rosner [2009],

(ii) Diedrich [2020],

(iii) Kowalski [2020],

(iv) the Controlled Components Contract Language.

After showing what this contract looks like in each CNL, a brief discussion on the strengths and weaknesses of each representation is given. We then provide a summary describing the differences between the four CNLs' representations. Section 5.2 provides an updated requirements comparison matrix which compares the four CNLs against the proposed requirements.

4.5.1 Pace and Rosner [2009]

The first CNL we consider is that of Pace and Rosner [2009]. A central idea to this CNL is that a sentence can describe a *simple event* in the form of an *agent-action-object* triple. Using this CNL, we model Section 2(c) of the ISDA Master agreement below:[13]

[13] Applying the CNL to this example was done with approval from the authors.

If it is obligatory that PARTY A pay PARTY B some amount on a date and it is obligatory that PARTY B pay PARTY A some amount on the same date, in respect of the same currency and the same transaction, then, on such date, each party's obligation to make payment is automatically satisfied and discharged, and, replaced by an obligation on PARTY A to pay PARTY B the excess of the larger aggregate amount over the smaller aggregate amount on the given date, if PARTY A's aggregate amount payable exceeds PARTY B's aggregate amount payable, or an obligation on PARTY B to pay PARTY A the excess of the larger aggregate amount over the smaller aggregate amount on the given date, if PARTY B's aggregate amount payable exceeds PARTY A's aggregate amount payable.

This representation of Section 2(c) of the ISDA Master Agreement closely follows the original text. It is not clear whether the original or the controlled version would be easier for a drafting lawyer to read. One benefit of this representation is that all events are clearly underlined, and there is a clear identification of two distinct parties. The semantics of the original contract are also accurately conveyed. However, this controlled form is highly verbose, and while it is consistent with the original style through using only a single sentence, this does not make it any easier to interpret. There is no attempt at reorganising the structure of the contract, so the difficulty of understanding the original contract remains.

In order to make this CNL more readable, several improvements could be made. First, increasing the use of whitespace, through new lines or paragraphs, would greatly help. As an initial suggestion, each action could be separated into its own line. This is related to the second improvement that could be made, which is to break the contract up into smaller, more comprehensible, sections. This would enable the reader to establish breakpoints when reading the CNL, and more easily isolate particular elements of the contract. Finally, there is an abundance of repeated phrases throughout. This could be tackled by instantiating variables which could act as placeholders for certain phrases of text.

4.5.2 Kowalski [2020]

We now consider Logical English (LE) (Kowalski [2020]). Karadotchev [2019] provides a case study of applying LE to various aspects of the ISDA Master Agreement, including Section 2(c). The original example was incomplete, and has been extended accordingly:

It is not the case that
        it is an obligation that a party pays to a counterparty
        an amount in a currency for a transaction on a date
if   it is an obligation that the party pays to the counterparty
        a net amount in the currency for the transaction on the date.

It is not the case that
        it is an obligation that a party pays to a counterparty
        an amount in a currency for a transaction on a date
if   it is an obligation that the counterparty pays to the party
        a net amount in the currency for the transaction on the date.

It is an obligation that a party pays to a counterparty
        a net amount in a currency for a transaction on a date
if   the net amount is a larger aggregate amount minus a smaller aggregate
        amount
and   the larger aggregate amount is the sum of each amount of each payment by
        the party to the counterparty in the currency for the transaction on the date
and   the smaller aggregate amount is the sum of each amount of each payment by
        the counterparty to the party in the currency for the transaction on the date.

We observe that the approach taken here is to reverse the obligation. Rather than following the format of Section 2(c) of the ISDA Master Agreement, Karadotchev [2019] starts by stating "It is not the case that ...". This could potentially confuse the lawyers who intend to use Logical English. Section 2(c) uses a single sentence to explain the clause. Separating this into paragraphs does certainly improve the readability. However, it is unclear how the various paragraphs are connected, if it all. Is one paragraph contingent on another? We simply do not know. While this excerpt of LE is fairly verbose, it does do a good job at isolating each of the relevant components line by line.

While in this current representation there is no way of referencing different clauses, when this contract is implemented in Prolog, payments become normalised and are referred to by their `IDs` . This presents a considerable strength for Logical English, indicating that it can refer to other clauses in a contract. Furthermore, LE aims to make all logical relationships explicit. Often, inconsistencies arise because of vague language, or things left implicit. By writing contracts directly in LE, contradictions would be easier to spot.

4.5.3 Diedrich [2020]

Lexon is a highly readable CNL which can be used to write normal contracts that function as blockchain smart contracts. Drawing on the work of Clack [2021], we now provide

Lexon's version of Section 2(c) of the ISDA Master Agreement:

```
LEX Netted Payment #1
Comment: To be used as one contract per day and transaction.

"Party One" is a person.
"Party Two" is a person.
"Total Payable of Party One" is an amount.
"Total Payable of Party Two" is an amount.

CLAUSE: Register Payable By Party One.
The Total Payable of Party One is increased by a given Amount.

CLAUSE: Register Payable By Party Two.
The Total Payable of Party Two is increased by a given Amount.

CLAUSE: Daily Netting.
If the Total Payable of Party One is greater than the Total Payable of Party Two,
then Party One pays the Net Amount to Party Two.
If the Total Payable of Party Two is greater than the Total Payable of Party One,
then Party Two pays the Net Amount to Party One.
Afterwards, terminate the contract.

CLAUSE: Net Amount.
"Net Amount" is defined as the difference between the Total Payable of Party One
and the Total Payable of Party Two.
```

There are clear actions provided to the reader, unlike in other CNLs where deontic aspects are used. This is expected, since Lexon does not use deontic terms (other than the word 'may') was purposefully designed to generate smart contracts. Furthermore, Lexon's representation is pleasantly simple; the spirit of the contract can be quickly understood from reading the provided title, terms are clearly defined at the start, and sentences are straightforward and concise throughout.

It is challenging to find fault with Lexon. One could argue that, for consistency and completeness, "Net Amount" should be defined at the start of the contract, as it too is an amount. However, this exclusion does not hinder the understanding of this contract in any way. Although each clause has its own description, there is no explicit identifier attached to each clause, which could make it difficult to refer to other clauses in the contract. In a similar vein to Kowalski [2020], there is no guidance surrounding the ordering of the clauses. While it is natural to assume that the clauses should be read chronologically, there is no explicit mention as to whether the positioning of the clauses is significant.

4.5.4 CoLa

Finally, we can use CoLa to represent Section 2(c) of the ISDA Master Agreement:

IF [1] it is the case that PartyA shall pay AMOUNT 'A' on ADATE
AND
[2] it is the case that PartyB shall pay AMOUNT 'B' on THEDATE
THEN [3] it is not the case that PartyA shall pay AMOUNT 'A'
on THEDATE
AND
[4] it is not the case that PartyB shall pay AMOUNT 'B'
on THEDATE
C-AND
[5] it is the case that ExcessParty shall pay AMOUNT "ExcessAmount"
on THEDATE
C-AND
IF [6] it is the case that PartyA paid more than PartyB
THEN [7] ExcessParty IS PartyA
AND
[8] ExcessAmount EQUALS AMOUNT 'A' MINUS AMOUNT 'B'
C-AND
IF [9] it is the case that PartyB paid more than PartyA
THEN [10] ExcessParty IS PartyB
AND
[11] ExcessAmount EQUALS AMOUNT 'B' MINUS AMOUNT 'A'.

Using CoLa, the sophisticated nature of ISDA 2c is expressed using a conditional statement (`IDs [1] - [4]`), a statement (`ID [5]`), and two conditional definitions (`IDs [6] - [8]` and `[9] - [11]`). The conditional statement is used to convey the initial obligations that exist, the statement is used to express the replacement of the initial obligations with the new obligation, and the conditional definition provides clarity on the relevant entities in this new obligation. For completeness, the output parse tree can be found in Appendix C.

While this CNL does appear more regimented than the previous examples, it is effective in several ways. The use of `IDs` removes any doubts surrounding the order in which to read the contract. Using capitalised keywords in the margin provides the reader with a clear overview of the structure of the contract. Furthermore, by also capitalising parties, objects,

and unspecified dates, the key elements of a component stand out, which can be beneficial when quickly scanning the contract. Unlike other CNLs, where to get an approximation of the size of the contract you would have to manually count the paragraphs or clauses, in CoLa you can simply look for the largest `ID` in the CNL, which will always be at the very end.

Nevertheless, CoLa does have its drawbacks. In the case of statements and conditions, it can at times be difficult to distinguish one from another. Consider the structure of the components with `ID [1]` and `ID [3]` respectively. On the face of it, it is not obvious which is a condition and which is a statement; they are almost identical. It is only with familiarity of the BNF does one find that the component with `ID [1]` is a condition and the component with `ID [3]` is a statement, due to the syntax of the `IF` and `THEN` keywords. In addition, unlike the previous CNLs, the format of CoLa does not look like a natural written contract that lawyers would be familiar with. While this may not be a weakness, it does indicate that more could be done to improve its readability. Further evaluation of the CoLa CNL is given in Section 5.3.1.

4.5.5 Summary

We have compared Section 2(c) of the ISDA Master Agreement in four different CNLs. While each CNL represents this contract in a unique way, their representations share many similarities. All four languages use a restricted syntax to express the contract. Pace and Rosner [2009] provide a reasonable first step in representing Section 2(c) of the ISDA Master Agreement, but do not make any significant progress in improving the contract's readability. Kowalski [2020] does an excellent job at making the contract understandable to both lawyers and to computers, and is a very promising attempt at achieving the goals of computable contracting. Diedrich [2020] stands out for its readability and presentation. Lexon looks and feels like a traditional contract, despite being a CNL capable of generating smart contracts. Finally, CoLa is distinctive for its precise, itemised, and orderly style. It is sufficiently readable for lawyers to understand, and acts as a computable contract using the provided parser. A critical evaluation of the CoLa implementation and parse tree is given in Sections 5.3.2 and 5.3.3 respectively.

5 Verification and Validation

Following the introduction of the CNL presented in this work, we now discuss how CoLa was verified and validated. We begin by discussing the verification, addressing how the CoLa lexer and parser were tested as well as debugged. Following this, we validate this CNL against the requirements proposed in Section 3.1, discuss its scalability, and contrast its performance against the other CNLs discussed in Section 3.2. Finally, a critical evaluation of CoLa is given.

5.1 Verification

The approach taken for testing both the lexer and parser was as follows. First, unit tests were implemented, to ensure that the various smaller fragments of the overall system worked as expected. These were implemented for the entire set of defined tokens for the lexer, as well as for each individual parse function. Once it was shown that these smaller functions worked correctly, larger end-to-end tests were implemented. These end-to-end tests were designed to evaluate both the lexer and the parser, as well as their relationship. In total, 140 unit tests were written, and 11 end-to-end tests were written. All tests successfully passed, and complete code coverage was achieved.

5.1.1 Unit Tests

To verify the CoLa implementation, unit tests were implemented for all aspects of the lexer, as well as every parser function. For the lexer, this was done by comparing the expected output of the lexer on a given string with the actual output of the lexer on the same string. For the parser, this was done by comparing the expected output of the evaluation of the parser on a certain list of tokens with the actual output of the same evaluation on the same list of tokens. When unit testing the parser, due to how large it was, the approach taken was to test each individual parse function, starting from the smaller ones, such as the parsing of a date, up until the larger ones, such as the parsing of a full contract consisting of several components. An example of both a lexer and parser unit test can be seen in Listings 6 and 7. For a test to pass, the actual output of the evaluation (of the lexer or parser) must match the expected printed output character for character. This is done through a manual inspection. A wider selection of unit tests can be seen in Appendix D.1.

Listing 6: An example lexer unit test.

```
lexer "[1] The Student must deliver a report 'Dissertation' on the 31
   April 2021",
show "[CompID 1, Subject Student, Modal_verb Obligation, Verb Deliver,
   ComponentA, Object (Report Dissertation), ComponentON, ComponentTHE,
   Number 31, Number 4, Number 2021]"
```

Listing 7: An example parser unit test.

```
parse_statement [CompID 1, Subject Student, Modal_verb Obligation, Verb
    Deliver, ComponentA, Object (Report Dissertation), ComponentON,
    ComponentTHE, Number 31, Number 4, Number 2021],
 show "[([],Just (CompStatement (Statement 1 Holds Obligation (31,4,2020)
    Student Deliver (Report Dissertation))))]"
```

5.1.2 End-to-end Testing

The main purpose of end-to-end testing was to ensure that the lexer and the parser worked seamlessly together. These were written by comparing a controlled natural language string of text against the output of this text when fed into the lexer as well as the output of the parser on this text. Unlike the unit tests, which separately verify that a CNL can be converted into tokens (for the case of the lexer) and that those tokens can then be converted into the final output (for the case of the parser), these end-to-end tests were designed to verify that all stages of the lifecycle of using CoLa worked as intended, from the CNL to the tokens to the final output. An example end-to-end test can be seen in Listing 8. A wider selection of end-to-end tests can be found in Appendix D.2.

Listing 8: An example end to end test.

```
show "[1] It is the case that the Student must deliver a report ‘
    Dissertation’ on the 31 April 2021."
[CompID 1, ComponentAffirmation, Subject Student, Modal_verb Obligation,
    Verb Deliver, ComponentA, Object (OtherObject dissertation), ComponentON
    , ComponentTHE, Number 31, Number 4, Number 2021]
Just (Contract (CompStatement (Statement 1 Holds Obligation (31,4,2021)
    Student Deliver (OtherObject dissertation))))
```

Along with this, a full separate test suite has been written for both the contract example provided in Section 4.4, as well as Section 2(c) of the ISDA Master Agreement, seen in Section 4.5.4. These can be found in Appendix D.3 and Appendix D.4 respectively.

Debugging the implementation of the parser proved to be more challenging. Miranda does not provide rich error messages, but it does indicate the approximate line of code at which the compilation failed, as well as telling the programmer what type of error occurred (e.g., `TypeError`, `ProgramError`, `Undefined Name`). The most effective technique for debugging the parser was often to compare the data type that a function expected as input with the true data type that was provided. Furthermore, the total Miranda script is over 1500 lines of code. As such, we did encounter a segmentation fault on numerous occasions. This was due to the lack of heap space to compile the program. This was remedied using the Miranda `-heap N` command, which allowed us to increase the size of the heap to `N` bytes.

5.2 Validation

We now validate the proposed CNL by evaluating its performance against the requirements provided in Section 3.1. This is followed by a discussion of the scalability of CoLa.

5.2.1 Requirements

(R_1) ✓ *Uses a natural language (such as English), with a restricted syntax and grammar.*

The proposed CNL uses a controlled version of English, which is indeed a natural language. It also restricts the syntax of the language in a certain way, described in Section 4.3.

(R_2) ✓ *Expresses the key components of a contract, as identified in Section 1.1.*

The proposed CNL accounts for four key components, which form the basis for all contracts. These components capture the main components of a contract described in Section 1.1.

(R_3) ✓ *Can be both generated and understood by humans without a formal background in Logic or Computer Science.*

Generating the CNL only requires an understanding of English and the CNL's syntax rules. The syntax uses well-known English words and does not leverage domain-specific terminology.

(R_4) ✓ *Captures deontic, temporal, and operational semantics.*

Deontic terms are clearly captured in this CNL, through the ⟨*modal-verb*⟩ element in the BNF. Temporal terms are captured too, through the use of specific and unspecified dates. Operational semantics are accounted for by the statement component.

(R_5) ✓ *Avoids some, or all, of the classic paradoxes of deontic logic.*

As the proposed CNL expresses all statements as distinct components, which contain their own ⟨*modal-verb*⟩, this prevents paradoxical statements from being expressed in the language. Many of the classical paradoxes of deontic logic are avoided. For example, both Ross's Paradox and the Free Choice Permission Paradox[14] are remedied by using this structure.

(R_6) ✓ *Can be easily translated into lower-level languages.*

In Section 5.2.2, we show that CoLa can be mapped fairly well to an underlying logic. As such, this makes it more likely that CoLa can be translated into a lower-level language.

[14] These paradoxes are explained in Section 3.1 of Prisacariu and Schneider [2007b].

(R_7) ? *Supports the detection of inconsistencies.*

The detection of inconsistencies does not exist in the current form of CoLa. However, as described in Section 4.1.1, the avoidance of inconsistencies does exist. This is done by the semantics that the clause with a greater `ID` takes priority. To detect inconsistencies, the provided editor could be configured to alert the user of components which reference the same arguments (for example, two definitions which reference the same subject).

(R_8) ✓ *Supports building new contracts from existing contracts.*

The syntax of this CNL is written in such a way that a contract is just the conjunction of numerous components. Therefore, if a user wishes to draft a contract in this CNL based on an existing contract, it would be possible to do so by joining the existing contract together with an empty contract, and then changing the relevant components. Additionally, alterations to inherited template contracts can easily be made through the use of the `[X(Y)]` component `ID` form, described in 4.1.1.

(R_9) ✓ *Supports overlapping clauses where one takes precedence over another.*

Once again, using the `[X(Y)]` component `ID` form, described in 4.1.1, allows the drafter of a contract to identify where one clause takes precedence over another. The partial example of the ISDA Master Agreement given in Section 4.5.4 further supports this.

(R_{10}) ✓ *Can be extended.*

The contract parser was built with possible extensions in mind; it is written in a highly modular way. To extend the lexer, this could simply be done using further pattern matching. Extending the grammar would be non-trivial, but the same argument could be made against all CNLs. It is possible to add an escape mechanism through the form of a new, uniquely identifiable component, which could capture uncontrolled natural language.

(R_{11}) ? *Can reference other clauses.*

CoLa's syntax does not currently support expressions that reference other `ID`s. `ID`s are set, but never used. Referencing earlier clauses can made possible through extending the syntax. To reference an earlier clause, one could simply refer to the `ID` of the clause, which could be added as an optional parameter to the existing components. It is worth noting that the `[X(Y)]` component `ID` style does allow implicit referencing of other clauses.

For ease of reference, we provide the requirements summary matrix again in Table 2, with the CoLa appended to the last column. We find that CoLa meets the proposed requirements exceptionally well, matching the highest score of the CNLs presented. It fully satisfies nine out of the eleven requirements. Interestingly, it performs identically to Lexon when evaluated against these requirements.

Table 2: Requirements comparison matrix, including CoLa.

	Pac	LE	Lex	ACE	SBVR	CoLa
R_1	✓	✓	✓	✓	✓	✓
R_2	✓	✓	✓	x	x	✓
R_3	✓	?	✓	✓	✓	✓
R_4	✓	✓	✓	?	✓	✓
R_5	✓	✓	✓	✓	x	✓
R_6	✓	✓	✓	✓	x	✓
R_7	x	?	?	x	x	?
R_8	✓	?	✓	x	x	✓
R_9	x	✓	✓	x	x	✓
R_{10}	✓	✓	✓	✓	✓	✓
R_{11}	x	?	?	?	?	?
Total Score	8	9	10	6	4.5	10

5.2.2 Logical Underpinning

An important aspect of validation is ensuring that CoLa's output parse tree is compatible with an underlying logic. One logic that appears to be a suitable fit is provided by Pace and Rosner [2009]. The deontic logic that they use for their CNL has operators which closely match those found in CoLa's parse tree. The operators that fit CoLa's parse tree well include: (i) action negation, which, in CoLa, is done through the use of the ⟨*holds*⟩ property, (ii) obligation, permission, and prohibition, which correspond to CoLa's use of ⟨*modal-verb*⟩, and (iii) disjunction and conjunction, which is equivalent to CoLa's use of `OR` and `AND`. Other attributes of the logic that fit in well with the CoLa parse tree include the ability to combine smaller contracts together to form larger contracts, and the ability to reference specific dates.

On the other hand, some operators used by Pace and Rosner [2009] are related to CoLa's parse tree, but do not map perfectly onto it, such as the use of conditional statements. The logic of Pace and Rosner [2009] defines conditional statements in the form of: $a_1 \triangleleft \alpha \triangleright a_2$ (i.e., if condition α is true, perform action a_1, otherwise perform action a_2). However, CoLa's parse tree permits conditionals in the following two forms: `IF` *condition* `THEN` *statement*, and *statement* `IF` *condition*. The difference here is that, while Pace and Rosner [2009] use

a single expression to define the action that occurs in the event of the the condition being either true or false, CoLa only defines the resulting action of a conditional statement in the case that the condition is true. As such, CoLa requires two conditional statements, one where the condition holds and one where the condition does not hold, to express the equivalent single conditional statement in Pace and Rosner [2009]. Despite this, the deontic logic of Pace and Rosner [2009] is a sufficiently close fit for CoLa's parse tree.

5.2.3 Scalability

Typically, contracts begin with several definitions, followed by numerous other clauses, such as statements or conditional statements. As such, when building a parser for legal contracts, it is imperative that the parser can handle contracts with many clauses. Figure 4 shows the time it takes to parse a contract as the number of clauses in the contract increases, using data reported by the Miranda compiler. We observe that the time taken increases exponentially with the number of clauses in the contract itself. As mentioned in Section 4.2.2, CoLa was designed using the 'list of successes' approach (Wadler [1985]). This creates lists of all the possible correct parses of the input (multiple correct parses exist due to ambiguities in the grammar). This causes the parser to produce exponentially more outputs as the number of clauses in the contract increases, due to the recursive-like nature of this approach. While there is no obvious way to determine what is causing the exponential time complexity observed, the list of successes approach is a likely candidate. It is worth noting that any such parser built using this approach would face this problem, suggesting that there may be a larger issue surrounding this technique when applied to parsing.

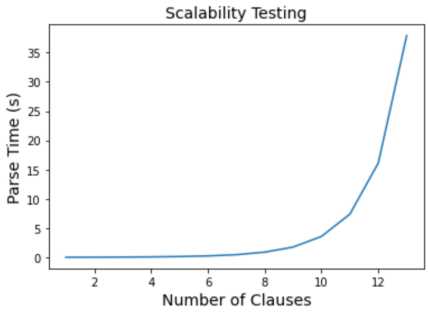

Figure 4: Measuring parse time against number of clauses.

5.3 Critical Evaluation

While a brief evaluation of the CoLa CNL was provided in Section 4.5.4, we now provide a further critical evaluation of it, as well as an evaluation of the implementation of CoLa, and CoLa's output parse tree.

5.3.1 CNL

The CNL is able to capture a wide range of different expressions. However, one of the drawbacks of having such an expressive language is that it is able to express clauses that it should not be able to. For example, despite being a common phrase, it is not natural to have statements such as "one is not permitted to do something" appear in the CNL.

This should be rewritten as "one is forbidden to do something". CoLa, in its current implementation, has no way of converting the negation of a permission into a prohibition, which gives rise to an ambiguity in the language. While this is not a severe problem, nor is it unique to CoLa, it is something that could be improved upon.

Another ambiguity, identified in Section 4.5.4, concerns conditions. The CNL captures two type of conditions: (i) those which concern something that has already happened (e.g., "the employee delivered a report"), and (ii) those which concern the existence of an obligation, permission, or prohibition (e.g., "the employee shall deliver a report"). This second type of condition arises commonly in contracts where the lawyer wishes to set out a rule that applies regardless of the details of other rules. Both conditions are needed to give the CNL its desired expressiveness. However, the second type of condition is syntactically indistinguishable from a statement, which can lead to potential confusion when reading the CNL.

Moreover, CoLa currently does not support any type of escape mechanism (i.e., the use of uncontrolled text in cases where a clause cannot be accurately expressed by the CNL). This is a limitation of the CNL, since it is possible that more complex clauses exist, which CoLa may not be able to be capture. To address this, inspiration can be taken from Lexon. The use of a title or optional comments within a contract would enable supplementary information, in the form of uncontrolled natural language, to be provided. While this is not a complete escape mechanism, this could make CoLa more attractive for lawyers to use.

5.3.2 Implementation

While CoLa is extremely quick to execute smaller contracts, it does take longer for larger contracts, comprised of more clauses, to be executed. If an alternative approach was used when parsing, such as one that did not collect all possible correct outputs but rather just the first correct output, this problem would be quickly remedied. Moreover, if implemented in a different functional programming language with a compiler that produced faster code, such as Haskell, this problem would be less severe.

CoLa requires a high level of precision when tokenizing contracts via the lexer, due to the use lexer's use of pattern matching. If there is a typographical error in the input, then this would not be registered by the lexer, resulting in a failed execution. This makes the lexer fragile to any inputs that do not exist in the token vocabulary. On the other hand, this does ensure that those drafting contracts are more precise. One could claim that using natural language processing (NLP) techniques[15] could replace the need for the lexer, and would allow lawyers to be less precise. However, NLP techniques are never 100% correct. They require a lot of computational resources, as well as a long time to train, yet do not

[15] `https://en.wikipedia.org/wiki/Natural_language_processing`

guarantee perfect results, as the lexer does. Just as uncontrolled natural language contracts in English require lawyers to use the correct words and spelling, so too does a CNL. This issue is not unique to CoLa, but any CNL.

5.3.3 Parse Tree

The visual aid provided, for which examples can be found in Section 4.4 Figure 2, and Appendix C, is effective for several reasons. First, the parse tree allows the reader to see a quick high-level overview of the contract that is being expressed, without even having to read any of the clauses explicitly. This allows to reader to easily determine the structure of the contract, and gives an indication of what they key components are. Second, the use of directed arrows in the parse tree gives the contract a clear sense of order. When reading the parse tree in a depth-first manner, conjunctive terms such as `and`, `or`, `if`, and `then` are used to connect smaller elements of the contract, enabling the reader to see how each component contributes to the overall contract. Third, since each component is expressed with its own unique ID, it is easy to get an approximation of how large the contract is by looking for the largest `ID` in the tree, which will typically be at the bottom right.

On the other hand, producing the parse tree is a manual process which can be cumbersome for longer contracts. While this is a significant barrier to wider use, automating the production of CoLa parse trees is possible and would address this. It is also worth noting that, without a written list of all of the components in the contract, the parse tree can only provide an indication of the structure, dependencies, and hierarchy of the contract, but there is no way of understanding the actual meaning of the components from the parse tree alone. The list of contract components must be provided alongside the parse tree, in order for it to effectively convey contract specific information.

6 Conclusion

This project concludes with an evaluation of the achievements against the stated objectives, as set out in Section 1.3, followed by a discussion of possible future work.

6.1 Project Evaluation

This core objectives of this project were composed of three main goals: to identify the requirements for a computable contracts language by reviewing existing formalisms; to design a new CNL which could address these requirements; and to implement a parser capable of representing real-world contracts. All of these goals, and more, were met.

In Section 3, a set of requirements which a CNL must meet in order to practically enable computable contracting was provided. This was crafted from an extensive literature review, seen in Section 2.2. Existing CNLs were evaluated against these requirements to determine any common trends and gauge the overall existing performance.

Section 4 introduced The Controlled Components Contract Language (CoLa), a novel CNL which leverages existing approaches in literature, and was designed with specific features in mind in order to meet the proposed requirements. A full definition of the CNL's syntax has been provided, along with a description of its design and implementation. Furthermore, a complete code generator has been implemented, capable of transforming contracts, in an end-to-end fashion, from a controlled natural language into their logical representations. Section 4.4 provided a full example demonstrating how CoLa can be used, and Section 4.5.4 translated a key section of a popular financial contract into a CoLa representation, demonstrating that CoLa is expressive enough to be applied in real-world legal contexts.

Finally, Section 5 illustrated the robustness of CoLa, through the implementation of a rigorous suite of unit tests, and CoLa was subsequently evaluated against the initial requirements. We observe that CoLa performed as well as the best performing existing CNL when evaluated against the proposed requirements. While being a fairly new language, with provisional implementation written in Miranda, CoLa forms the foundation for future research in the area of computable contracting.

6.2 Future Work

Future work should focus on making the CoLa infrastructure more adaptive to different terms used in natural language, improving the speed of the CoLa parser when handling larger contracts, and the general understanding of meta-clauses.

First, a possible solution to making CoLa more adaptive to the different terms used in natural language is to develop a comprehensive library for future tokenization. This library would be familiar with the various synonyms that exist in the English language, and could

be formed by importing an existing business ontology into the lexer. Depending on the user interface provided to the users of CoLa, such a solution could possibly even detect common typographical errors and correct them.

Second, the speed of the parser can be improved. Current tests suggest that the parser could do a better job at reducing the time it takes to convert tokens into the output representation. As seen in Section 5.2.3, the current implementation performs at an exponentially increasing runtime, which is rather disconcerting. Furthermore, this may suggest that the 'list of successes' method proposed by Wadler [1985] may not be well suited for the application of efficient parsing.

Third, more thought should be given to 'meta-clauses'. These are clauses which concern other clauses. They can drastically change the way that the performance of a contract is understood, yet there is virtually no existing literature which addresses them. One scenario where meta-clauses are used is the replacement of existing obligations with new obligations. For example, as indicated by Section 2(c) of the ISDA Master Agreement, we would be instructed to replace 4 payments of PartyA to PartyB and 3 payments of PartyB to PartyA with a single payment from PartyA to PartyB. Other scenarios where meta-clauses can be used are in the insertion of new ways of satisfying obligations, the replacing of times of events with new times, and the insertion of logic regarding combinations of obligations and events.

References

J. J. Camilleri, G. Paganelli, and G. Schneider. A cnl for contract-oriented diagrams. In *International Workshop on Controlled Natural Language*, pages 135–146. Springer, 2014.

C. D. Clack. Languages for smart and computable contracts, 2021.

C. D. Clack, C. Myers, and E. Poon. *Programming with Miranda*. Prentice Hall, 1995.

C. D. Clack, V. A. Bakshi, and L. Braine. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771*, 2016.

J. Cummins and C. Clack. Transforming commercial contracts through computable contracting. *arXiv preprint arXiv:2003.10400*, 2020.

H. Diedrich. Lexon bible: Hitchhiker's guide to digital contracts. *Independently published*, 2020.

J. Fokker. Functional parsers. In *International School on Advanced Functional Programming*, pages 1–23. Springer, 1995.

N. E. Fuchs, K. Kaljurand, and T. Kuhn. Attempto controlled english for knowledge representation. In *Reasoning Web*, pages 104–124. Springer, 2008.

T. Hvitved. *Contract formalisation and modular implementation of domain-specific languages*. PhD thesis, Citeseer, 2011.

F. Idelberger. Merging traditional contracts (or law) and (smart) e-contracts–a novel approach. 2020.

A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of logic and computation*, 2(6):719–770, 1992.

V. Karadotchev. First steps towards logical english. Master's thesis, Imperial College London, September 2019.

R. Kowalski. Logical english. *Logic and Practice of Programming (LPOP)*, https://www.doc.ic.ac.uk/ rak/papers/LPOP.pdf, 2020.

R. A. Kowalski. Legislation as logic programs. In *Informatics and the Foundations of Legal Reasoning*, pages 325–356. Springer, 1995.

T. Kuhn. A survey and classification of controlled natural languages. *Computational linguistics*, 40(1):121–170, 2014.

E. Martínez, G. Díaz, M. E. Cambronero, and G. Schneider. A model for visual specification of e-contracts. In *2010 IEEE International Conference on Services Computing*, pages 1–8. IEEE, 2010.

J. Meyer and F. P. M. Dignum. *The paradoxes of deontic logic revisited: A computer science perspective (or: Should computer scientists be bothered by the concerns of philosophers?)*, volume 1994. Unknown Publisher, 1994.

P. B. F. Njonko, S. Cardey, P. Greenfield, and W. El Abed. Rulecnl: A controlled natural language for business rule specifications. In *International Workshop on Controlled Natural Language*, pages 66–77. Springer, 2014.

G. J. Pace and M. Rosner. A controlled language for the specification of contracts. In *International Workshop on Controlled Natural Language*, pages 226–245. Springer, 2009.

G. J. Pace and G. Schneider. Challenges in the specification of full contracts. In *International Conference on Integrated Formal Methods*, pages 292–306. Springer, 2009.

C. Prisacariu and G. Schneider. A formal language for electronic contracts. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 174–189. Springer, 2007a.

C. Prisacariu and G. Schneider. Towards a formal definition of electronic contracts. *Research report http://urn. nb. no/URN: NBN: no-35645*, 2007b.

C. Prisacariu and G. Schneider. $\mathcal{CL}$: An action-based logic for reasoning about contracts. In *International Workshop on Logic, Language, Information, and Computation*, pages 335–349. Springer, 2009.

C. Prisacariu and G. Schneider. A dynamic deontic logic for complex contracts. *The Journal of Logic and Algebraic Programming*, 81(4):458–490, 2012.

F. Ryan. Round hall nutshells contract law. *Thomson Round Hall*, page 1, 2006.

O. SBVR. Semantics of business vocabulary and business rules (sbvr), version 1.0, 2008.

K. Schwab. The fourth industrial revolution (geneva: World economic forum). *Ekonomica Preduzeca*, 2016.

J. Stark. Making sense of blockchain smart contracts. *Coindesk. com*, 2016.

H. Surden. Computable contracts. *UCDL Rev.*, 46:629, 2012.

N. Szabo. The idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials*, 6, 1997.

M. N. Temte. Blockchain challenges traditional contract law: Just how smart are smart contracts. *Wyo. L. Rev.*, 19:87, 2019.

D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer, 1985.

G. H. Von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.

P. Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

M. Wöhrer and U. Zdun. Domain specific language for smart contract development. 2020.

Appendices

A Complete Set of Tokens

```
token ::= ContractAND | ComponentOR | ComponentAND | ComponentIF |
    ComponentTHEN | ComponentDEFIS | ComponentDEFEQ | Word [char] | Number
    num | ComponentON | ComponentTHE | ComponentA | ComponentAN | Modal_verb
     t_statement_type | Verb t_verb | Object t_object | Subject [char] |
    Verb_status t_verb_status | ComponentAffirmation | ComponentNegation |
    CompID [char] | Comparison t_comparison | NumericalExpr t_numericalexpr
    | Operator t_operator | Date t_other_date
```

(Return to Section 4.2.1).

B Code

B.1 Lexer

```
lexer :: [char] -> [token]
lexer input =
    concat (map (xlexer . split_into_words) (split_into_sentences input))
    where
    xlexer []                                   = []
    xlexer ([]: rest)                           = xlexer rest
    xlexer ("IF"  :rest)                        = (ComponentIF   : xlexer
    rest)
    xlexer ("THEN":rest)                        = (ComponentTHEN : xlexer
    rest)
    xlexer ("more":("than":rest))               = ((Comparison MoreThan) :
    xlexer rest)
    xlexer ("equal":("to":rest))                = ((Comparison EqualTo) :
    xlexer rest)
    xlexer ("less":("than":rest))               = ((Comparison LessThan) :
    xlexer rest)
    xlexer ("on":("ANYDATE":rest))              = ComponentON : Date ANYDATE
    : xlexer rest
    xlexer ("on":("ADATE":rest))                = ComponentON : Date ADATE :
    xlexer rest
    xlexer ("on":("THEDATE":rest))              = ComponentON : Date THEDATE
    : xlexer rest
    xlexer ("on":("the":(dd:(mm:(yy:rest))))) = ComponentON : ComponentTHE
    : (Number (numval dd)) : (Number (conv mm)) : (Number (numval yy)) :
    xlexer rest
    xlexer ("It":("is":("the":("case":("that":rest)))))         =
    ComponentAffirmation: xlexer rest
    xlexer ("it":("is":("the":("case":("that":rest)))))         =
    ComponentAffirmation: xlexer rest
    xlexer ("It":("is":("not":("the":("case":("that":rest))))))  =
    ComponentNegation: xlexer rest
    xlexer ("it":("is":("not":("the":("case":("that":rest))))))  =
    ComponentNegation: xlexer rest
    xlexer ("is":("forbidden":("to":rest)))   = (Modal_verb Prohibition):(
    xlexer rest)
    xlexer ("IS"      :rest)                    = ((ComponentDEFIS) : (xlexer
     rest))
    xlexer ("EQUALS"  :rest)                    = ((ComponentDEFEQ) :
    NumericalExpr (NumericalExprNum (numval (hd rest))) : xlexer (tl rest)),
     if ((isdigit (hd (hd rest))) & (tl rest)=[])
    xlexer ("EQUALS"  :rest)                    = ((ComponentDEFEQ) :
    NumericalExpr (NumericalExprObject (OtherObject (hd rest))) : xlexer (tl
     rest)), if ((isupper (hd rest)) & (tl rest)=[])
    xlexer ("EQUALS"  :rest)                    = ((ComponentDEFEQ) :
    NumericalExpr (NumericalExprExpr (NumericalExprObject (OtherObject (hd
    rest))) (operatorconstructors!(pos operators (hd (tl rest)))) (
```

```
    NumericalExprNum (numval (hd (tl (tl rest)))))) : xlexer (tl (tl (tl
    rest)))), if ((member operators (hd (tl rest))) & (isdigit (hd (hd (tl (
    tl rest))))))
    xlexer ("EQUALS"  :rest)                    = ((ComponentDEFEQ) :
    NumericalExpr (NumericalExprExpr (NumericalExprObject (Amount (hd (tl
    rest)))) (operatorconstructors!(pos operators (hd (tl (tl rest))))) (
    NumericalExprObject (Amount (hd (tl (tl (tl (tl rest)))))))) : xlexer (
    tl (tl (tl (tl (tl rest)))))), if (member operators (hd (tl (tl rest))))
    xlexer ("EQUALS"  :rest)                    = ((ComponentDEFEQ) : (xlexer
     rest))
    xlexer (word : rest)                        = (CompID (f word)   : (
    xlexer rest)),
                                                  if ((hd word)='[') & ((last
     word)=']') & (isdigit (hd (f word)))
                                                = error "Square brackets
    detected outside of ComponentID",
                                                  if ((hd word)='[') & ((last
     word)=']')
                                                = (ContractAND   : (xlexer
    rest)), if (word="C-AND")
                                                = (ComponentOR   : (xlexer
    rest)), if (word="OR")
                                                = (ComponentAND  : (xlexer
    rest)), if (word="AND")
                                                = (ComponentA) : (Object (
    Report (hd (tl rest)))): (xlexer (tl (tl rest))),
                                                  if (word="A" \/ word="a") &
     ((hd rest)="report") & (rest ~= [])
                                                = (ComponentAN) : (Object (
    Report (hd (tl rest)))): (xlexer (tl (tl rest))),
                                                  if (word="An" \/ word="an")
     & ((hd rest)="report") & (rest ~= [])
                                                = (ComponentA) : (Object (
    Amount (hd (rest))): (xlexer (tl (rest))),
                                                  if (word="AMOUNT") & (rest
    ~= [])
                                                = (ComponentAN) : (Object (
    Amount (hd (tl rest)))): (xlexer (tl (tl rest))),
                                                  if (word="An" \/ word="an")
     & ((hd rest)="AMOUNT") & (rest ~= [])
                                                = (ComponentA) : (Object (
    OtherObject (hd rest))): (xlexer (tl rest)),
                                                  if (word="A" \/ word="a") &
     (rest ~= [])
                                                = (ComponentAN) : (Object (
    OtherObject (hd rest))): (xlexer (tl rest)),
                                                  if (word="An" \/ word="an")
     & (rest ~= [])
                                                = error "Sentence ends with
    indefinite article",
```

```
                                                  if (member indefarticles
    word) & (rest=[])
                                                = (Modal_verb (v2t (
    modalconstructors!(pos modalverbs word)))):(xlexer rest),
                                                  if (member modalverbs word)
                                                = (Verb (verbconstructors!(
    pos verbs word))):(xlexer rest),
                                                  if (member verbs word)
                                                = (Verb_status (
    verbstatusconstructors!(pos verbstatuses word))):(xlexer rest),
                                                  if (member verbstatuses
    word)
                                                = (ComponentTHE) : (Subject (
    hd rest)): (xlexer (tl rest)),
                                                  if ((member defarticles
    word) & (rest ~= []) & ~(isdigit (hd (hd rest))) & (isupper (hd rest)))
                                                = (ComponentTHE) : (Subject (
    word)): (xlexer rest),
                                                  if (isupper word)
                                                = error "Expected The (or the
    ) to be followed by a number.",
                                                  if (member defarticles word
    ) & (rest=[])
                                                = error "Sentence ends with
    definite article",
                                                  if (member defarticles word
    ) & (rest=[])
                                                = (Object (Pounds (numval
    word))): (xlexer (tl rest)),
                                                  if (rest ~=[]) & (isdigit (
    hd word)) & (member pounds (hd rest))
                                                = (Object (Dollars (numval
    word))): (xlexer (tl rest)),
                                                  if (rest ~=[]) & (isdigit (
    hd word)) & (member dollars (hd rest))
                                                = (Object (Euros (numval word
    ))): (xlexer (tl rest)),
                                                  if (rest ~=[]) & (isdigit (
    hd word)) & (member euros (hd rest))
                                                = (Number (numval word)) : (
    xlexer rest),
                                                  if (isdigit (hd word))
                                                = (Word (word)) : (xlexer
    rest), otherwise
                                                  where
                                                  f word = xf (tl word)
                                                            where
                                                            xf [] = []
                                                            xf [']'] = []
```

```
                                                        xf (x:xs) = x:(xf
    xs)
              conv mm = [1,2,3,4,5,6,7,8,9,10,11,12]!(pos ["January", "
    February", "March", "April", "May", "June", "July", "August", "September
    ", "October", "November", "December"] mm)
```

(Return to Section 4.2.1).

B.2 Parser

```
|| 1. Contract level

t_contract ::= TrueContract | Contract t_component | Contracts_and [
    t_component] || list of components implies ANDing of components

parse_contract :: t_parser token (maybe t_contract)
parse_contract = alt parse_component (then f parse_component (then g
    parse_contractAND parse_contract))
                 where
                 f Nothing                 any                       =
    Nothing
                 f any                     Nothing                   =
    Nothing
                 f (Just (Contract x))     (Just (Contracts_and y)) = Just (
    Contracts_and (x:y))
                 f (Just (Contract x))     (Just (Contract y))      = Just (
    Contracts_and [x, y])
                 g Nothing                 any                       =
    Nothing
                 g any                     Nothing                   =
    Nothing
                 g x                       y                         = y

parse_contractAND :: t_parser token (maybe t_contract)
parse_contractAND (ContractAND:rest) = [(rest,Just (TrueContract))]
parse_contractAND any                = [(any ,Nothing)]

|| 2. Component level

t_component  ::= CompDefinition t_definition | CompStatement t_statement |
    CompCondition t_condition |
                 CompCondStatement t_condition t_statement |
    CompStatStatement t_statement t_statement | TrueComponent | ||
    TrueComponent simplifies the code
                 CompExprDefinition t_expression t_definition |
    CompExpression t_expression

parse_component :: t_parser token (maybe t_contract)
parse_component = maybecomponent2maybecontract . (alt parse_definition (alt
     parse_statement (alt parse_condition (alt parse_conditional_statement
```

```
    parse_conditional_definition))))
                  where
                  maybecomponent2maybecontract ops = filter ((~=Nothing).
    snd) (map g ops)
                                                     where
                                                     g (rem, Nothing) = (
    rem, Nothing)
                                                     g (rem, Just s)  = (
    rem, Just (Contract s))

|| 3. Definitions

t_definition ::= Def_IS t_ID t_subject t_subject | Def_EQ t_ID t_subject
    t_numericalexpr | Definitions_and [t_definition]

t_numericalexpr ::= NumericalExprNum t_num | NumericalExprObject t_object |
     NumericalExprExpr t_numericalexpr t_operator t_numericalexpr |
    NoNumericalExpr || NoNumericalExpr needed for intermediate parsing
t_operator      ::= PLUS | MINUS | TIMES | DIVIDE | NoOperator ||
    NoOperator needed for intermediate parsing
empty_operator = NoOperator

parse_definition :: t_parser token (maybe t_component)
parse_definition = alt parse_simpledefinition (then f
    parse_simpledefinition (then g parse_componentAND parse_definition))
                  where
                  f Nothing                    any
                    = Nothing
                  f any                        Nothing
                    = Nothing
                  f (Just (CompDefinition x)) (Just (CompDefinition (
    Definitions_and y)))  = Just (CompDefinition (Definitions_and (x:y)))
                  f (Just (CompDefinition x)) (Just (CompDefinition y))
                    = Just (CompDefinition (Definitions_and [x, y]))
                  g Nothing                    any
                    = Nothing
                  g any                        Nothing
                    = Nothing
                  g x                          y
                    = y

parse_simpledefinition :: t_parser token (maybe t_component)
parse_simpledefinition = alt parse_simpledefinitionIS
    parse_simpledefinitionEQ

parse_simpledefinitionIS :: t_parser token (maybe t_component)
parse_simpledefinitionIS =
    maybedefinition2maybecomponent . (then a parse_compID_defIS (d_parse "
    s1ISs2"))
    where
```

```
    a x       Nothing = Nothing
    a Nothing y       = Nothing
    a (Just (Def_IS id1 s11 s12)) (Just (Def_IS id2 s21 s22)) = Just (
    Def_IS id1 s21 s22) || Def_IS2 but CompID1 overrides
    d_parse "s1ISs2" = then (merge "s1") parse_s12d (then (merge "s2")
    parse_IS parse_s22d)
    d_parse any = g
    g ip = [([], Nothing)]
    merge c x Nothing = Nothing
    merge c Nothing y = Nothing
    merge "s2" (Just (Def_IS id1 s11 s12)) (Just (Def_IS id2 s21 s22)) =
    Just (Def_IS id1 s11 s22) || first definition with second s2
    merge "s1" (Just (Def_IS id1 s11 s12)) (Just (Def_IS id2 s21 s22)) =
    Just (Def_IS id2 s11 s22) || second definition with first s1
    maybedefinition2maybecomponent ops = filter ((~=Nothing).snd) (map g
    ops)
                                       where
                                       g (rem, Nothing) = (rem, Nothing)
                                       g (rem, Just d)  = (rem, Just (
    CompDefinition d))

|| Parsing the first subject into a defintion
parse_s12d :: t_parser token (maybe t_definition)
parse_s12d = maybesubject2maybedefinition . (then f parse_THE_subject
    parse_subject)
             where
             f x       Nothing = Nothing
             f Nothing y       = y
             f x       y       = y
             maybesubject2maybedefinition ops = filter ((~=Nothing).snd) (
    map g ops)
                                              where
                                              g (rem, Nothing) = (rem,
    Nothing)
                                              g (rem, Just s1) = (rem,
    Just (Def_IS empty_ID s1 empty_subject))
|| Parsing the second subject into a defintion
parse_s22d :: t_parser token (maybe t_definition)
parse_s22d = maybesubject2maybedefinition . (then f parse_THE_subject
    parse_subject)
             where
             f x       Nothing = Nothing
             f Nothing y       = y
             f x       y       = y
             maybesubject2maybedefinition ops = filter ((~=Nothing).snd) (
    map g ops)
                                              where
                                              g (rem, Nothing) = (rem,
    Nothing)
```

```
                                                g (rem, Just s2) = (rem,
    Just (Def_IS empty_ID empty_subject s2))

parse_compID_defIS :: t_parser token (maybe t_definition)
parse_compID_defIS ((CompID x): rest) = [(rest, Just (Def_IS x
    empty_subject empty_subject))]
parse_compID_defIS any                = [(any, Nothing)]

parse_IS :: t_parser token (maybe t_definition)
parse_IS (ComponentDEFIS:rest) = [(rest,Just (Def_IS empty_ID empty_subject
     empty_subject))]
parse_IS any                   = [(any, Nothing)]

parse_simpledefinitionEQ :: t_parser token (maybe t_component)
parse_simpledefinitionEQ =
    maybedefinition2maybecomponent . (then a parse_compID_defEQ (d_parse "
    sEQne"))
    where
    a x Nothing = Nothing
    a Nothing y = Nothing
    a (Just (Def_EQ id1 s1 n1)) (Just (Def_EQ id2 s2 n2)) = Just (Def_EQ
    id1 s2 n2) || Def_EQ2 but CompID1 overrides
    d_parse "sEQne"  = then (merge "s") parse_s2d (then (merge "ne")
    parse_EQ parse_ne2d)
    d_parse any = g
    g ip = [([], Nothing)]
    merge c x Nothing = Nothing
    merge c Nothing y = Nothing
    merge "ne"  (Just (Def_EQ id1 s1 n1)) (Just (Def_EQ id2 s2 n2)) = Just
    (Def_EQ id1 s1 n2)
    merge "s"   (Just (Def_EQ id1 s1 n1)) (Just (Def_EQ id2 s2 n2)) = Just
    (Def_EQ id2 s1 n2)
    maybedefinition2maybecomponent ops = filter ((~=Nothing).snd) (map g
    ops)
                                        where
                                        g (rem, Nothing) = (rem, Nothing)
                                        g (rem, Just d)  = (rem, Just (
    CompDefinition d))

|| Parsing the subject into a defintion
parse_s2d :: t_parser token (maybe t_definition)
parse_s2d = maybesubject2maybedefinition . (then f parse_THE_subject
    parse_subject)
            where
            f x       Nothing = Nothing
            f Nothing y       = y
            f x       y       = y
            maybesubject2maybedefinition ops = filter ((~=Nothing).snd) (
    map g ops)
                                               where
```

```
                                                g (rem, Nothing) = (rem,
    Nothing)
                                                g (rem, Just s)  = (rem,
    Just (Def_EQ empty_ID s NoNumericalExpr))

|| Parsing the numerical expression into a defintion
parse_ne2d :: t_parser token (maybe t_definition)
parse_ne2d = maybenumericalexpr2maybedefinition . parse_numercialexpr
             where
             maybenumericalexpr2maybedefinition ops = filter ((~=Nothing).
    snd) (map g ops)
                                                      where
                                                      g (rem, Nothing) = (
    rem, Nothing)
                                                      g (rem, Just ne) = (
    rem, Just (Def_EQ empty_ID empty_subject ne))

parse_numercialexpr :: t_parser token (maybe t_numericalexpr)
parse_numercialexpr ((NumericalExpr ne): rest) = [(rest, Just (ne))]
parse_numercialexpr any                        = [(any, Nothing)]

parse_compID_defEQ :: t_parser token (maybe t_definition)
parse_compID_defEQ ((CompID x): rest) = [(rest, Just (Def_EQ x
    empty_subject NoNumericalExpr))]
parse_compID_defEQ any                = [(any, Nothing)]

parse_EQ :: t_parser token (maybe t_definition)
parse_EQ (ComponentDEFEQ:rest) = [(rest,Just (Def_EQ empty_ID empty_subject
     NoNumericalExpr))]
parse_EQ any                   = [(any, Nothing)]

|| 4. Conditional Definitions

t_expression ::= Expression t_ID t_holds t_subject t_verb_status
    t_comparison t_subject | Expressions_or [t_expression] | Expressions_and
     [t_expression]
t_comparison ::= LessThan | EqualTo | MoreThan | NoComparison
empty_comparison = NoComparison

parse_conditional_definition :: t_parser token (maybe t_component)
parse_conditional_definition
  = alt (then f parse_definition (then h parse_IF parse_expression)) (then
    g (then h parse_IF parse_expression) (then h parse_THEN parse_definition
    ))
    where
    f Nothing                   y                          = Nothing
    f x                         Nothing                    = Nothing
    f (Just (CompDefinition x)) (Just (CompExpression y)) = Just (
    CompExprDefinition y x)
    g Nothing                   y                          = Nothing
```

```
    g x                          Nothing                    = Nothing
    g (Just (CompExpression y)) (Just (CompDefinition x)) = Just (
    CompExprDefinition y x)
    h Nothing                    y                          = Nothing
    h x                          Nothing                    = Nothing
    h x                          y                          = y

parse_expression :: t_parser token (maybe t_component)
parse_expression = alt parse_simpleexpression (alt (then f
    parse_simpleexpression (then g parse_componentOR parse_expression)) (
    then h parse_simpleexpression (then g parse_componentAND
    parse_expression)))
                   where
                   f Nothing any = Nothing
                   f any Nothing = Nothing
                   f (Just (CompExpression x)) (Just (CompExpression (
    Expressions_or y))) = Just (CompExpression (Expressions_or (x:y)))
                   f (Just (CompExpression x)) (Just (CompExpression y)) =
    Just (CompExpression (Expressions_or [x, y]))
                   g Nothing any = Nothing
                   g any Nothing = Nothing
                   g x y = y
                   h Nothing any = Nothing
                   h any Nothing = Nothing
                   h (Just (CompExpression x)) (Just (CompExpression (
    Expressions_and y))) = Just (CompExpression (Expressions_and (x:y)))
                   h (Just (CompExpression x)) (Just (CompExpression y)) =
    Just (CompExpression (Expressions_and [x, y]))

parse_componentOR :: t_parser token (maybe t_component)
parse_componentOR ((ComponentOR):rest) = [(rest, Just (TrueComponent))]
parse_componentOR any                  = [(any, Nothing)]

parse_componentAND :: t_parser token (maybe t_component)
parse_componentAND ((ComponentAND):rest) = [(rest, Just (TrueComponent))]
parse_componentAND any                   = [(any, Nothing)]

parse_simpleexpression :: t_parser token (maybe t_component)
parse_simpleexpression = maybeexpression2maybecomponent . (then a
    parse_compID_expression (then b parse_holds_expression (e_parse "svcs"))
    )
                         where
                         a Nothing any = Nothing
                         a any Nothing = Nothing
                         a (Just (Expression id1 h1 s11 vs1 c1 s12)) (Just
    (Expression id2 h2 s21 vs2 c2 s22)) = Just (Expression id1 h2 s21 vs2 c2
     s22)
                         b Nothing any = any || holds is optional
                         b any Nothing = Nothing
```

```
                        b (Just (Expression id1 h1 s11 vs1 c1 s12)) (Just
    (Expression id2 h2 s21 vs2 c2 s22)) = Just (Expression id2 h1 s21 vs2 c2
     s22)
                        e_parse "svcs" = then (merge "s1") parse_s12e (
    then (merge "cs2") parse_v2e (then (merge "s2") parse_c2e parse_s22e))
                        e_parse any    = g
                        g ip = [([], Nothing)]
                        merge c x Nothing = Nothing
                        merge c Nothing y = Nothing
                        merge "s2"  (Just (Expression id1 h1 s11 vs1 c1
    s12)) (Just (Expression id2 h2 s21 vs2 c2 s22)) = Just (Expression id1
    h1 s11 vs1 c1 s22)
                        merge "cs2" (Just (Expression id1 h1 s11 vs1 c1
    s12)) (Just (Expression id2 h2 s21 vs2 c2 s22)) = Just (Expression id1
    h1 s11 vs1 c2 s22)
                        merge "s1"  (Just (Expression id1 h1 s11 vs1 c1
    s12)) (Just (Expression id2 h2 s21 vs2 c2 s22)) = Just (Expression id2
    h2 s11 vs2 c2 s22)
                        maybeexpression2maybecomponent ops = filter ((~=
    Nothing).snd) (map g ops)
                                                             where
                                                             g (rem,
    Nothing) = (rem, Nothing)
                                                             g (rem, Just
    e)  = (rem, Just (CompExpression e))

parse_holds_expression :: t_parser token (maybe t_expression)
parse_holds_expression (ComponentAffirmation:rest) = [(rest,Just (
    Expression empty_ID Holds empty_subject empty_verb_status
    empty_comparison empty_subject))]
parse_holds_expression (ComponentNegation:rest)    = [(rest,Just (
    Expression empty_ID NotHolds empty_subject empty_verb_status
    empty_comparison empty_subject))]
parse_holds_expression any                         = [(any,Nothing)]

parse_compID_expression :: t_parser token (maybe t_expression)
parse_compID_expression ((CompID x): rest) = [(rest, Just (Expression x
    empty_holds empty_subject empty_verb_status empty_comparison
    empty_subject))]
parse_compID_expression any                = [(any, Nothing)]

|| Parsing the first subject into an expression
parse_s12e :: t_parser token (maybe t_expression)
parse_s12e = maybesubject2maybeexpression . (then f parse_THE_subject
    parse_subject)
             where
             f x       Nothing = Nothing
             f Nothing y       = y
             f x       y       = y
```

```
            maybesubject2maybeexpression ops = filter ((~=Nothing).snd) (
    map g ops)
                                                where
                                                g (rem, Nothing) = (rem,
    Nothing)
                                                g (rem, Just s1) = (rem,
    Just (Expression empty_ID empty_holds s1 empty_verb_status
    empty_comparison empty_subject))

|| Parsing a verb status into an expression
parse_v2e :: t_parser token (maybe t_expression)
parse_v2e = maybeverb_status2maybeexpression . parse_verb_status
            where
            maybeverb_status2maybeexpression ops = filter ((~=Nothing).snd)
     (map g ops)
                                                  where
                                                  g (rem, Nothing) = (rem,
     Nothing)
                                                  g (rem, Just vs) = (rem,
     Just (Expression empty_ID empty_holds empty_subject vs empty_comparison
     empty_subject))

|| Parsing a comparison into an expression
parse_c2e :: t_parser token (maybe t_expression)
parse_c2e = maybecomparison2maybeexpression . parse_comparison
            where
            maybecomparison2maybeexpression ops = filter ((~=Nothing).snd)
    (map g ops)
                                                where
                                                g (rem, Nothing) = (rem,
    Nothing)
                                                g (rem, Just c)  = (rem,
    Just (Expression empty_ID empty_holds empty_subject empty_verb_status c
    empty_subject))

parse_comparison :: t_parser token (maybe t_comparison)
parse_comparison ((Comparison x):rest) = [(rest, Just (x))]
parse_comparison any                   = [(any, Nothing)]

|| Parsing the second subject into an expression
parse_s22e :: t_parser token (maybe t_expression)
parse_s22e = maybesubject2maybeexpression . (then f parse_THE_subject
    parse_subject)
             where
             f x       Nothing = Nothing
             f Nothing y       = y
             f x       y       = y
             maybesubject2maybeexpression ops = filter ((~=Nothing).snd) (
    map g ops)
                                                where
```

```
                                                    g (rem, Nothing) = (rem,
    Nothing)
                                                    g (rem, Just s2) = (rem,
    Just (Expression empty_ID empty_holds empty_subject empty_verb_status
    empty_comparison s2))

|| 5. Statements

t_statement ::= Statement t_ID t_holds t_statement_type t_date t_subject
    t_verb t_object | Statements_or [t_statement] | Statements_and [
    t_statement]
t_ID == [char]
t_holds ::= Holds | NotHolds | UnknownHolds
t_statement_type ::= Obligation | Permission | Prohibition |
    NoStatementType || We need NoType during intermediate parsing
t_date == (num,num,num) || day, month, year
t_other_date ::= ANYDATE | ADATE | THEDATE | NoOtherDate || ANYDATE =
    (00,00,01), ADATE = (00,00,02), THEDATE = (00,00,03)
t_num == num
t_subject == [char]
t_object ::= Pounds num | Dollars num | Euros num | Amount [char] |
    SomeCurrency [char] | Report [char] | NamedObject [char] | OtherObject [
    char] | NoObject || We need NoObject during intermediate parsing
t_verb_phrase == (t_statement_type, t_verb)
t_verb ::= Deliver | Pay | Charge | NoVerb || We need NoVerb during
    intermediate parsing

empty_statement :: t_statement
empty_statement = Statement empty_ID empty_holds empty_statement_type
    empty_date empty_subject empty_verb empty_object
empty_ID = ""
empty_holds :: t_holds
empty_holds = UnknownHolds
empty_statement_type :: t_statement_type
empty_statement_type = NoStatementType
empty_date :: t_date
empty_date = (0,0,0)
empty_other_date :: t_other_date
empty_other_date = NoOtherDate
empty_subject :: t_subject
empty_subject = ""
empty_verb :: t_verb
empty_verb = NoVerb
empty_object :: t_object
empty_object = NoObject

parse_statement :: t_parser token (maybe t_component)
parse_statement = alt parse_simplestatement (alt (then f
    parse_simplestatement (then g parse_componentOR parse_statement)) (then
    h parse_simplestatement (then g parse_componentAND parse_statement)))
```

```
                    where
                    f Nothing                    any
                      = Nothing
                    f any                        Nothing
                      = Nothing
                    f (Just (CompStatement x))  (Just (CompStatement (
    Statements_or y)))     = Just (CompStatement (Statements_or (x:y)))
                    f (Just (CompStatement x))  (Just (CompStatement y))
                      = Just (CompStatement (Statements_or [x, y]))
                    g Nothing                    any
                      = Nothing
                    g any                        Nothing
                      = Nothing
                    g x                          y
                      = y
                    h Nothing                    any
                      = Nothing
                    h any                        Nothing
                      = Nothing
                    h (Just (CompStatement x))  (Just (CompStatement (
    Statements_and y)))    = Just (CompStatement (Statements_and (x:y)))
                    h (Just (CompStatement x))  (Just (CompStatement y))
                      = Just (CompStatement (Statements_and [x, y]))

parse_simplestatement :: t_parser token (maybe t_component)
parse_simplestatement = maybestatement2maybecomponent . (then a
    parse_compID_statement (then f parse_holds_statement (alt (s_parse "dsvo
    ") (alt (s_parse "dovs") (alt (s_parse "sdvo") (alt (s_parse "svod") (
    alt (s_parse "ovsd") (s_parse "ovds"))))))))
                        where
                        a Nothing any = Nothing
                        a any Nothing = Nothing
                        a (Just (Statement id1 h1 ty1 da1 su1 ve1 ob1)) (
    Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement id1 h2
    ty2 da2 su2 ve2 ob2) || s2 but s1 CompID overrides
                        f Nothing any = Nothing
                        f any Nothing = Nothing
                        f (Just (Statement id1 h1 ty1 da1 su1 ve1 ob1)) (
    Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement id2 h1
    ty2 da2 su2 ve2 ob2) || s2 but s1 hold overrides
                        s_parse "dsvo" = then (merge "D") parse_d (then (
    merge "vo") parse_s (then (merge "o") parse_v parse_o))
                        s_parse "dovs" = then (merge "D") parse_d (then (
    merge "vs") parse_o (then (merge "s") parse_v parse_s))
                        s_parse "sdvo" = then (merge "S") parse_s (then (
    merge "vo") parse_d (then (merge "o") parse_v parse_o))
                        s_parse "svod" = then (merge "S") parse_s (then (
    merge "od") parse_v (then (merge "d") parse_o parse_d))
                        s_parse "ovsd" = then (merge "O") parse_o (then (
    merge "sd") parse_v (then (merge "d") parse_s parse_d))
```

```
                        s_parse "ovds" = then (merge "O") parse_o (then (
    merge "sd") parse_v (then (merge "s") parse_d parse_s))
                        s_parse any    = g
                        g ip = [([], Nothing)]
                        merge c    x
    Nothing                                 = Nothing
                        merge c    Nothing                                  y
                                            = Nothing
                        merge "o"  (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id1 h1 ty1 da1 su1 ve1 ob2) || keep h1
                        merge "s"  (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id1 h1 ty1 da1 su2 ve1 ob1) || keep h1
                        merge "d"  (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id1 h1 ty1 da2 su1 ve1 ob1) || keep h1
                        merge "vo" (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id1 h1 ty2 da1 su1 ve2 ob2) || keep h1
                        merge "vs" (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id1 h1 ty2 da1 su2 ve2 ob1) || keep h1
                        merge "od" (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id1 h1 ty1 da2 su1 ve1 ob2) || keep h1
                        merge "sd" (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id1 h1 ty1 da2 su2 ve1 ob1) || keep h1
                        merge "D"  (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id2 h2 ty2 da1 su2 ve2 ob2) || keep h2
                        merge "S"  (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id2 h2 ty2 da2 su1 ve2 ob2) || keep h2
                        merge "O"  (Just (Statement id1 h1 ty1 da1 su1 ve1
    ob1)) (Just (Statement id2 h2 ty2 da2 su2 ve2 ob2)) = Just (Statement
    id2 h2 ty2 da2 su2 ve2 ob1) || keep h2
                        maybestatement2maybecomponent ops = filter ((~=
    Nothing).snd) (map g ops)
                                                              where
                                                              g (rem, Nothing
    ) = (rem, Nothing)
                                                              g (rem, Just s)
      = (rem, Just (CompStatement s))

parse_holds_statement :: t_parser token (maybe t_statement)
parse_holds_statement (ComponentAffirmation:rest) = [(rest,Just (Statement
    empty_ID Holds empty_statement_type empty_date empty_subject empty_verb
    empty_object))]
```

```
parse_holds_statement (ComponentNegation:rest)    = [(rest,Just (Statement
    empty_ID NotHolds empty_statement_type empty_date empty_subject
    empty_verb empty_object))]
parse_holds_statement any                         = [(any,Nothing)]

parse_compID_statement :: t_parser token (maybe t_statement)
parse_compID_statement ((CompID x): rest) = [(rest, Just (Statement x
    empty_holds empty_statement_type empty_date empty_subject empty_verb
    empty_object))]
parse_compID_statement any                = [(any, Nothing)]

parse_d :: t_parser token (maybe t_statement) || date_phrase
parse_d = alt (parse_other_d) (maybedate2maybestatement . (then f
    parse_ON_date (then g parse_THE_date (maybenum2maybedate . (then h
    parse_day (then h parse_month parse_year))))))
      where
      f x       Nothing  = Nothing
      f Nothing y        = Nothing
      f x       y        = y
      g x       Nothing  = Nothing
      g Nothing y        = Nothing
      g x       y        = y
      h Nothing y        = Nothing
      h x       Nothing  = Nothing
      h (Just x)(Just y) = Just (x + y*1000)
      maybenum2maybedate ops = filter ((~=Nothing).snd) (map g ops)
                               where
                               g (rem, Nothing) = (rem, Nothing)
                               g (rem, Just y)  = (rem, Just (y mod
    1000, (y div 1000) mod 1000, (y div 1000000)))
      maybedate2maybestatement ops = filter ((~=Nothing).snd) (map g
    ops)
                                     where
                                     g (rem, Nothing) = (rem, Nothing)
                                     g (rem, Just d)  = (rem, Just (
    Statement empty_ID empty_holds empty_statement_type d empty_subject
    empty_verb empty_object))

parse_other_d :: t_parser token (maybe t_statement)
parse_other_d = maybedate2maybestatement . (then f parse_ON_date (
    maybeotherdate2maybedate . parse_other_date))
            where
            f x       Nothing = Nothing
            f Nothing y       = Nothing
            f x       y       = y
            maybeotherdate2maybedate ops = filter ((~=Nothing).snd) (
    map g ops)
                                           where
                                           g (rem, Nothing) = (rem,
    Nothing)
```

```
                                                    g (rem, Just (ANYDATE)) = (
    rem, Just (00,00,01))
                                                    g (rem, Just (ADATE))   = (
    rem, Just (00,00,02))
                                                    g (rem, Just (THEDATE)) = (
    rem, Just (00,00,03))
                maybedate2maybestatement ops = filter ((~=Nothing).snd) (
    map g ops)
                                                         where
                                                         g (rem, Nothing) = (rem
    , Nothing)
                                                         g (rem, Just d)  = (rem
    , Just (Statement empty_ID empty_holds empty_statement_type d
    empty_subject empty_verb empty_object))

parse_other_date :: t_parser token (maybe t_other_date)
parse_other_date ((Date x): rest) = [(rest, Just (x))]
parse_other_date any              = [(any, Nothing)]

parse_ON_date :: t_parser token (maybe t_date)
parse_ON_date (ComponentON:rest) = [(rest,Just empty_date)] || the value is
     not used
parse_ON_date any                = [(any, Nothing)]

parse_THE_date :: t_parser token (maybe t_date)
parse_THE_date (ComponentTHE:rest) = [(rest,Just empty_date)] || the value
    is not used
parse_THE_date any                 = [(any, Nothing)]

parse_day :: t_parser token (maybe t_num)
parse_day = parse_number

|| For now we only permit a month to be entered as a number
parse_month :: t_parser token (maybe t_num)
parse_month = parse_number

parse_year :: t_parser token (maybe t_num)
parse_year = parse_number

parse_number :: t_parser token (maybe t_num)
parse_number ((Number x): rest) = [(rest, Just x)]
parse_number any                = [(any, Nothing)]

|| The following code deals with the parsing of objects to statements
parse_o :: t_parser token (maybe t_statement)
parse_o = maybeobject2maybestatement . (then f (alt (parse_A) (parse_AN))
    parse_object)
          where
          f x       Nothing = Nothing
          f Nothing y       = y
```

```
          f x       y       = y
          maybeobject2maybestatement ops = filter ((~=Nothing).snd) (map g
    ops)
                                           where
                                           g (rem, Nothing) = (rem, Nothing
    )
                                           g (rem, Just o)  = (rem, Just (
    Statement empty_ID empty_holds empty_statement_type empty_date
    empty_subject empty_verb o))

parse_A :: t_parser token (maybe t_object)
parse_A (ComponentA:rest) = [(rest, Just empty_object)] || for intermediate
     parsing
parse_A any               = [(any, Nothing)]

parse_AN :: t_parser token (maybe t_object)
parse_AN (ComponentAN:rest) = [(rest, Just empty_object)] || for
    intermediate parsing
parse_AN any                = [(any, Nothing)]

parse_object :: t_parser token (maybe t_object)
parse_object ((Object x):rest) = [(rest, Just (x))]
parse_object any               = [(any, Nothing)]

|| The following code deals with the parsing of subjects to statements
parse_s :: t_parser token (maybe t_statement)
parse_s = maybesubject2maybestatement . (then f parse_THE_subject
    parse_subject)
          where
          f x       Nothing = Nothing
          f Nothing y       = y
          f x       y       = y
          maybesubject2maybestatement ops = filter ((~=Nothing).snd) (map g
     ops)
                                           where
                                           g (rem, Nothing) = (rem,
    Nothing)
                                           g (rem, Just s)  = (rem, Just (
    Statement empty_ID empty_holds empty_statement_type empty_date s
    empty_verb empty_object))

parse_subject :: t_parser token (maybe t_subject)
parse_subject ((Subject x):rest) = [(rest, Just (x))]
parse_subject any                = [(any, Nothing)]

parse_THE_subject :: t_parser token (maybe t_subject)
parse_THE_subject (ComponentTHE:rest) = [(rest,Just empty_subject)] || the
    value is not used
parse_THE_subject any                 = [(any, Nothing)]

```

```
|| The following code deals with the parsing of verbs to statements
parse_v :: t_parser token (maybe t_statement)
parse_v = maybeverb_phrase2maybestatement . (then f parse_MODAL parse_verb)
          where
          f x               Nothing        = Nothing
          f Nothing         y              = Nothing
          f (Just (x1,y1)) (Just (x2,y2)) = Just (x1, y2)
          maybeverb_phrase2maybestatement ops = filter ((~=Nothing).snd) (
    map g ops)
                                                where
                                                g (rem, Nothing) = (rem,
    Nothing)
                                                g (rem, Just (st_type, verb
    )) = (rem, Just (Statement empty_ID empty_holds st_type empty_date
    empty_subject verb empty_object))

parse_MODAL :: t_parser token (maybe t_verb_phrase)
parse_MODAL ((Modal_verb x):rest) = [(rest, Just (x, NoVerb))]
parse_MODAL any                   = [(any, Nothing)]

parse_verb :: t_parser token (maybe t_verb_phrase)
parse_verb ((Verb x):rest) = [(rest, Just (NoStatementType, x))]
parse_verb any             = [(any, Nothing)]

|| 6. Conditional statements

t_condition ::= Condition t_ID t_holds t_date t_subject t_verb_status
    t_object | Condition2 t_ID t_holds t_statement_type t_date t_subject
    t_verb t_object | Conditions_or [t_condition] | Conditions_and [
    t_condition]
t_verb_status ::= Delivered | Paid | Charged | NoStatus
empty_condition :: t_condition
empty_condition = Condition empty_ID empty_holds empty_date empty_subject
    empty_verb_status empty_object
empty_verb_status :: t_verb_status
empty_verb_status = NoStatus

parse_conditional_statement :: t_parser token (maybe t_component)
parse_conditional_statement
  = alt (then f parse_statement (then h parse_IF parse_condition)) (then g
    (then h parse_IF parse_condition) (then h parse_THEN parse_statement))
    where
    f Nothing                  y                          = Nothing
    f x                        Nothing                    = Nothing
    f (Just (CompStatement x)) (Just (CompCondition y)) = Just (
    CompCondStatement y x)
    g Nothing                  y                          = Nothing
    g x                        Nothing                    = Nothing
    g (Just (CompCondition y)) (Just (CompStatement x)) = Just (
    CompCondStatement y x)
```

```
    h Nothing                   y                           = Nothing
    h x                         Nothing                     = Nothing
    h x                         y                           = y

parse_condition :: t_parser token (maybe t_component)
parse_condition = alt parse_eithersimplecondition (alt (then f
    parse_eithersimplecondition (then g parse_componentOR parse_condition))
    (then h parse_eithersimplecondition (then g parse_componentAND
    parse_condition)))
                  where
                  f Nothing                   any
                    = Nothing
                  f any                       Nothing
                    = Nothing
                  f (Just (CompCondition x))  (Just (CompCondition (
    Conditions_or y)))     = Just (CompCondition (Conditions_or (x:y)))
                  f (Just (CompCondition x))  (Just (CompCondition y))
                    = Just (CompCondition (Conditions_or [x, y]))
                  g Nothing                   any
                    = Nothing
                  g any                       Nothing
                    = Nothing
                  g x                         y
                    = y
                  h Nothing                   any
                    = Nothing
                  h any                       Nothing
                    = Nothing
                  h (Just (CompCondition x))  (Just (CompCondition (
    Conditions_and y)))    = Just (CompCondition (Conditions_and (x:y)))
                  h (Just (CompCondition x))  (Just (CompCondition y))
                    = Just (CompCondition (Conditions_and [x, y]))

parse_eithersimplecondition :: t_parser token (maybe t_component)
parse_eithersimplecondition = alt parse_simplecondition
    parse_simplecondition2

parse_simplecondition :: t_parser token (maybe t_component)
parse_simplecondition = maybecondition2maybecomponent . (then a
    parse_compID_condition (then f (parse_holds_condition) (alt (s_parse "
    dsvo") (alt (s_parse "dovs") (alt (s_parse "sdvo") (alt (s_parse "svod")
     (alt (s_parse "ovsd") (s_parse "ovds")))))))
                        where
                        a Nothing any = Nothing
                        a any Nothing = Nothing
                        a (Just (Condition id1 h1 da1 su1 vs1 ob1)) (Just (
    Condition id2 h2 da2 su2 vs2 ob2))   = Just (Condition id1 h2 da2 su2
    vs2 ob2)
                        f Nothing any = Nothing
                        f any Nothing = Nothing
```

```
                      f (Just (Condition id1 h1 da1 su1 vs1 ob1))  (Just
(Condition id2 h2 da2 su2 vs2 ob2))  = Just (Condition id1 h1 da2 su2
vs2 ob2)
                      s_parse "dsvo" = then (merge "D") parse_d2c (then (
merge "vo") parse_s2c (then (merge "o") parse_v2c parse_o2c))
                      s_parse "dovs" = then (merge "D") parse_d2c (then (
merge "vs") parse_o2c (then (merge "s") parse_v2c parse_s2c))
                      s_parse "sdvo" = then (merge "S") parse_s2c (then (
merge "vo") parse_d2c (then (merge "o") parse_v2c parse_o2c))
                      s_parse "svod" = then (merge "S") parse_s2c (then (
merge "od") parse_v2c (then (merge "d") parse_o2c parse_d2c))
                      s_parse "ovsd" = then (merge "O") parse_o2c (then (
merge "sd") parse_v2c (then (merge "d") parse_s2c parse_d2c))
                      s_parse "ovds" = then (merge "O") parse_o2c (then (
merge "ds") parse_v2c (then (merge "s") parse_d2c parse_s2c))
                      s_parse any    = g
                      g ip = [([], Nothing)]
                      merge c    x
Nothing                          = Nothing
                      merge c    Nothing                                 y
                                 = Nothing
                      merge "o"  (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id1 h1 da1
 su1 vs1 ob2)
                      merge "s"  (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id1 h1 da1
 su2 vs1 ob1)
                      merge "d"  (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id1 h1 da2
 su1 vs1 ob1)
                      merge "vo" (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id1 h1 da1
 su1 vs2 ob2)
                      merge "vs" (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id1 h1 da1
 su2 vs2 ob1)
                      merge "od" (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id1 h1 da2
 su1 vs1 ob2)
                      merge "sd" (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id1 h1 da2
 su2 vs1 ob1)
                      merge "ds" (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id1 h1 da2
 su2 vs1 ob1)
                      merge "D"  (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id2 h2 da1
 su2 vs2 ob2)
                      merge "S"  (Just (Condition id1 h1 da1 su1 vs1 ob1)
) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id2 h2 da2
```

```
    su1 vs2 ob2)
                        merge "O"  (Just (Condition id1 h1 da1 su1 vs1 ob1)
    ) (Just (Condition id2 h2 da2 su2 vs2 ob2)) = Just (Condition id2 h2 da2
    su2 vs2 ob1)
                        maybecondition2maybecomponent ops = filter ((~=
    Nothing).snd) (map g ops)
                                                            where
                                                            g (rem, Nothing
    ) = (rem, Nothing)
                                                            g (rem, Just x)
     = (rem, Just (CompCondition x))

parse_simplecondition2 :: t_parser token (maybe t_component)
parse_simplecondition2 = maybecondition2maybecomponent . (then a
    parse_compID_condition2 (then f parse_holds_condition2 (s_parse "sstvod"
    )))
                        where
                        a Nothing any = Nothing
                        a any Nothing = Nothing
                        a (Just (Condition2 id1 h1 st1 d1 s1 v1 o1)) (Just
    (Condition2 id2 h2 st2 d2 s2 v2 o2)) = Just (Condition2 id1 h2 st2 d2
    s2 v2 o2)
                        f Nothing any = Nothing
                        f any Nothing = Nothing
                        f (Just (Condition2 id1 h1 st1 d1 s1 v1 o1)) (Just
    (Condition2 id2 h2 st2 d2 s2 v2 o2)) = Just (Condition2 id1 h1 st2 d2
    s2 v2 o2)
                        s_parse "sstvod" = then (merge "S2") parse_s2c2 (
    then (merge "vod2") parse_st2c2 (then (merge "od2") parse_v2c2 (then (
    merge "d2") parse_o2c2 parse_d2c2)))
                        s_parse any    = g
                        g ip = [([], Nothing)]
                        merge c    x
    Nothing                         = Nothing
                        merge c    Nothing
    y                               = Nothing
                        merge "S2"    (Just (Condition2 id1 h1 st1 d1 s1
    v1 o1)) (Just (Condition2 id2 h2 st2 d2 s2 v2 o2)) = Just (Condition2
    id2 h2 st2 d2 s1 v2 o2)
                        merge "vod2"  (Just (Condition2 id1 h1 st1 d1 s1
    v1 o1)) (Just (Condition2 id2 h2 st2 d2 s2 v2 o2)) = Just (Condition2
    id1 h1 st1 d2 s1 v2 o2)
                        merge "od2"   (Just (Condition2 id1 h1 st1 d1 s1
    v1 o1)) (Just (Condition2 id2 h2 st2 d2 s2 v2 o2)) = Just (Condition2
    id1 h1 st1 d2 s1 v1 o2)
                        merge "d2"    (Just (Condition2 id1 h1 st1 d1 s1
    v1 o1)) (Just (Condition2 id2 h2 st2 d2 s2 v2 o2)) = Just (Condition2
    id1 h1 st1 d2 s1 v1 o1)
                        maybecondition2maybecomponent ops = filter ((~=
    Nothing).snd) (map g ops)
```

```
                                                                where
                                                                g (rem,
    Nothing) = (rem, Nothing)
                                                                g (rem, Just x
    )  = (rem, Just (CompCondition x))

595

parse_holds_condition :: t_parser token (maybe t_condition)
parse_holds_condition (ComponentAffirmation:rest) = [(rest,Just (Condition
    empty_ID Holds empty_date empty_subject empty_verb_status empty_object))
    ]
parse_holds_condition (ComponentNegation:rest)    = [(rest,Just (Condition
    empty_ID NotHolds empty_date empty_subject empty_verb_status
    empty_object))]
parse_holds_condition any                         = [(any,Nothing)]
600

parse_holds_condition2 :: t_parser token (maybe t_condition)
parse_holds_condition2 (ComponentAffirmation:rest) = [(rest,Just (
    Condition2 empty_ID Holds empty_statement_type empty_date empty_subject
    empty_verb empty_object))]
parse_holds_condition2 (ComponentNegation:rest)    = [(rest,Just (
    Condition2 empty_ID NotHolds empty_statement_type empty_date
    empty_subject empty_verb empty_object))]
parse_holds_condition2 any                          = [(any,Nothing)]
605

parse_compID_condition :: t_parser token (maybe t_condition)
parse_compID_condition ((CompID x): rest) = [(rest, Just (Condition x
    empty_holds empty_date empty_subject empty_verb_status empty_object))]
parse_compID_condition any                = [(any, Nothing)]

610 parse_compID_condition2 :: t_parser token (maybe t_condition)
parse_compID_condition2 ((CompID x): rest) = [(rest, Just (Condition2 x
    empty_holds empty_statement_type empty_date empty_subject empty_verb
    empty_object))]
parse_compID_condition2 any                = [(any, Nothing)]

|| The following code deals with the parsing of dates into conditions
615 parse_d2c :: t_parser token (maybe t_condition) || date_phrase
parse_d2c = alt (parse_other_d2c) (maybedate2maybecondition . (then f
    parse_ON_date  (then g parse_THE_date (maybenum2maybedate . (then h
    parse_day (then h parse_month parse_year))))))
            where
            f x       Nothing = Nothing
            f Nothing y       = Nothing
620         f x       y       = y
            g x       Nothing = Nothing
            g Nothing y       = Nothing
            g x       y       = y
            h x       Nothing = Nothing
625         h Nothing y       = Nothing
```

```
            h (Just x)(Just y)= Just (x + y*1000)
            maybenum2maybedate ops = filter ((~=Nothing).snd) (map g ops)
                                     where
                                     g (rem, Nothing) = (rem, Nothing)
                                     g (rem, Just y)  = (rem, Just (y mod
    1000, (y div 1000) mod 1000, (y div 1000000)))
            maybedate2maybecondition ops = filter ((~=Nothing).snd) (map g
    ops)
                                           where
                                           g (rem, Nothing) = (rem, Nothing
    )
                                           g (rem, Just d)  = (rem, Just (
    Condition empty_ID empty_holds d empty_subject empty_verb_status
    empty_object))

parse_other_d2c :: t_parser token (maybe t_condition)
parse_other_d2c = maybedate2maybecondition . (then f parse_ON_date (
    maybeotherdate2maybedate . parse_other_date))
                  where
                  f x       Nothing = Nothing
                  f Nothing y       = Nothing
                  f x       y       = y
                  maybeotherdate2maybedate ops = filter ((~=Nothing).snd) (
    map g ops)
                                                 where
                                                 g (rem, Nothing) = (rem,
    Nothing)
                                                 g (rem, Just (ANYDATE)) =
    (rem, Just (00,00,01))
                                                 g (rem, Just (ADATE))   =
    (rem, Just (00,00,02))
                                                 g (rem, Just (THEDATE)) =
    (rem, Just (00,00,03))
                  maybedate2maybecondition ops = filter ((~=Nothing).snd) (
    map g ops)
                                                 where
                                                 g (rem, Nothing) = (rem,
    Nothing)
                                                 g (rem, Just od) = (rem,
    Just (Condition empty_ID empty_holds od empty_subject empty_verb_status
    empty_object))

|| The following code deals with the parsing of objects to conditions
parse_o2c :: t_parser token (maybe t_condition)
parse_o2c = maybeobject2maybecondition . (then f (alt (parse_A) (parse_AN))
     parse_object)
            where
            f x       Nothing = Nothing
            f Nothing y       = y
            f x       y       = y
```

```
            maybeobject2maybecondition ops = filter ((~=Nothing).snd) (map
    g ops)
                                            where
                                            g (rem, Nothing) = (rem,
    Nothing)
                                            g (rem, Just o)  = (rem, Just
    (Condition empty_ID empty_holds empty_date empty_subject
    empty_verb_status o))

|| The following code deals with the parsing of subjects to conditions
parse_s2c :: t_parser token (maybe t_condition)
parse_s2c = maybesubject2maybecondition . (then f parse_THE_subject
    parse_subject)
            where
            f x       Nothing = Nothing
            f Nothing y       = y
            f x       y       = y
            maybesubject2maybecondition ops = filter ((~=Nothing).snd) (map
     g ops)
                                             where
                                             g (rem, Nothing) = (rem,
    Nothing)
                                             g (rem, Just s)  = (rem, Just
     (Condition empty_ID empty_holds empty_date s empty_verb_status
    empty_object))

|| The following code deals with the parsing of a verb status to a
    condition
parse_v2c :: t_parser token (maybe t_condition)
parse_v2c = maybeverb_status2maybecondition . parse_verb_status
            where
            maybeverb_status2maybecondition ops = filter ((~=Nothing).snd)
    (map g ops)
                                                  where
                                                  g (rem, Nothing) = (rem,
    Nothing)
                                                  g (rem, Just vs) = (rem,
    Just (Condition empty_ID empty_holds empty_date empty_subject vs
    empty_object))

parse_verb_status :: t_parser token (maybe t_verb_status)
parse_verb_status ((Verb_status x):rest) = [(rest, Just (x))]
parse_verb_status any                    = [(any, Nothing)]

|| The following code deals with the parsing of subjects to conditions (
    type 2)
parse_s2c2 :: t_parser token (maybe t_condition)
parse_s2c2 = maybesubject2maybecondition . (then f parse_THE_subject
    parse_subject)
             where
```

```
            f x       Nothing = Nothing
            f Nothing y       = y
            f x       y       = y
            maybesubject2maybecondition ops = filter ((~=Nothing).snd) (map g ops)
                                                where
                                                g (rem, Nothing) = (rem, Nothing)
                                                g (rem, Just s)  = (rem, Just (Condition2 empty_ID empty_holds empty_statement_type empty_date s empty_verb empty_object))

parse_st2c2 :: t_parser token (maybe t_condition)
parse_st2c2 = maybemodalverb2maybecondition . parse_modal_verb
             where
             maybemodalverb2maybecondition ops = filter ((~=Nothing).snd) (map g ops)
                                                   where
                                                   g (rem, Nothing) = (rem, Nothing)
                                                   g (rem, Just st) = (rem, Just (Condition2 empty_ID empty_holds st empty_date empty_subject empty_verb empty_object))

parse_modal_verb :: t_parser token (maybe t_statement_type)
parse_modal_verb ((Modal_verb x):rest) = [(rest, Just (x))]
parse_modal_verb any                   = [(any, Nothing)]

|| The following code deals with the parsing of verbs to a condition (type 2)
parse_v2c2 :: t_parser token (maybe t_condition)
parse_v2c2 = maybeverb2maybecondition . parse_verb2
            where
            maybeverb2maybecondition ops = filter ((~=Nothing).snd) (map g ops)
                                                  where
                                                  g (rem, Nothing) = (rem, Nothing)
                                                  g (rem, Just v) = (rem, Just (Condition2 empty_ID empty_holds empty_statement_type empty_date empty_subject v empty_object))

parse_verb2 :: t_parser token (maybe t_verb)
parse_verb2 ((Verb x):rest) = [(rest, Just (x))]
parse_verb2 any             = [(any, Nothing)]

|| The following code deals with the parsing of objects to a condition (type 2)
parse_o2c2 :: t_parser token (maybe t_condition)
```

```
parse_o2c2 = maybeobject2maybecondition . (then f (alt (parse_A) (parse_AN)
    ) parse_object)
            where
            f x       Nothing = Nothing
            f Nothing y       = y
            f x       y       = y
            maybeobject2maybecondition ops = filter ((~=Nothing).snd) (map
     g ops)
                                             where
                                             g (rem, Nothing) = (rem,
    Nothing)
                                             g (rem, Just o)  = (rem, Just
     (Condition2 empty_ID empty_holds empty_statement_type empty_date
    empty_subject empty_verb o))

|| The following code deals with the parsing of dates into conditions (type
     2)
parse_d2c2 :: t_parser token (maybe t_condition) || date_phrase
parse_d2c2 = alt (parse_other_d2c2) (maybedate2maybecondition . (then f
    parse_ON_date  (then g parse_THE_date (maybenum2maybedate . (then h
    parse_day (then h parse_month parse_year))))))
            where
            f x       Nothing = Nothing
            f Nothing y       = Nothing
            f x       y       = y
            g x       Nothing = Nothing
            g Nothing y       = Nothing
            g x       y       = y
            h x       Nothing = Nothing
            h Nothing y       = Nothing
            h (Just x)(Just y)= Just (x + y*1000)
            maybenum2maybedate ops = filter ((~=Nothing).snd) (map g ops)
                                     where
                                     g (rem, Nothing) = (rem, Nothing)
                                     g (rem, Just y)  = (rem, Just (y mod
    1000, (y div 1000) mod 1000, (y div 1000000)))
            maybedate2maybecondition ops = filter ((~=Nothing).snd) (map g
     ops)
                                           where
                                           g (rem, Nothing) = (rem,
    Nothing)
                                           g (rem, Just d)  = (rem, Just (
    Condition2 empty_ID empty_holds empty_statement_type d empty_subject
    empty_verb empty_object))

parse_other_d2c2 :: t_parser token (maybe t_condition)
parse_other_d2c2 = maybedate2maybecondition . (then f parse_ON_date (
    maybeotherdate2maybedate . parse_other_date))
                   where
                   f x       Nothing = Nothing
```

```
                  f Nothing y       = Nothing
                  f x       y       = y
                  maybeotherdate2maybedate ops = filter ((~=Nothing).snd)
    (map g ops)
                                                 where
                                                 g (rem, Nothing) = (rem,
    Nothing)
                                                 g (rem, Just (ANYDATE)) =
     (rem, Just (00,00,01))
                                                 g (rem, Just (ADATE))   =
     (rem, Just (00,00,02))
                                                 g (rem, Just (THEDATE)) =
     (rem, Just (00,00,03))
                  maybedate2maybecondition ops = filter ((~=Nothing).snd)
    (map g ops)
                                                 where
                                                 g (rem, Nothing) = (rem,
    Nothing)
                                                 g (rem, Just od) = (rem,
    Just (Condition2 empty_ID empty_holds empty_statement_type od
    empty_subject empty_verb empty_object))

|| The following code deals with the parsing of IF into components
parse_IF :: t_parser token (maybe t_component)
parse_IF = maybecondition2maybecomponent . p_IF
           where
           p_IF (ComponentIF:rest) = [(rest, Just empty_condition)]
           p_IF any                = [(any, Nothing)]
           maybecondition2maybecomponent ops
             = filter ((~=Nothing).snd) (map g ops)
               where
               g (rem, Nothing) = (rem, Nothing)
               g (rem, Just x)  = (rem, Just (CompCondition x))

|| The following code deals with the parsing of THEN into components
parse_THEN :: t_parser token (maybe t_component)
parse_THEN = maybestatement2maybecomponent . p_THEN
             where
             p_THEN (ComponentTHEN:rest) = [(rest, Just empty_statement)]
             p_THEN any                  = [(any, Nothing)]
             maybestatement2maybecomponent ops
               = filter ((~=Nothing).snd) (map g ops)
                 where
                 g (rem, Nothing) = (rem, Nothing)
                 g (rem, Just x)  = (rem, Just (CompStatement x))
```

(Return to Section 4.2.2).

C ISDA Section 2(c) CoLa Parse Tree

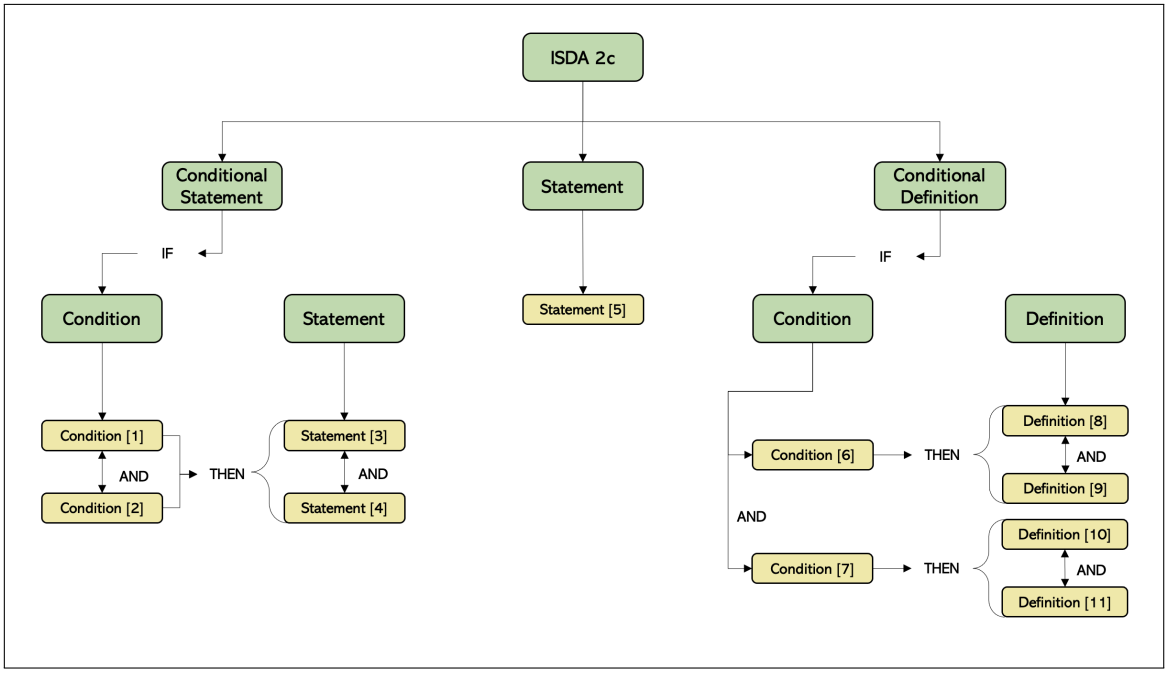

Figure 5: CoLa Parse Tree for Section 2(c) of the 2002 ISDA Master Agreement.

Output:

[1] it is the case that PartyA shall pay AMOUNT 'A' on ADATE
[2] it is the case that PartyB shall pay AMOUNT 'B' on THEDATE
[3] it is not the case that PartyA shall pay AMOUNT 'A' on THEDATE
[4] it is not the case that PartyB shall pay AMOUNT 'B' on THEDATE
[5] it is the case that ExcessParty shall pay AMOUNT "ExcessAmount" on THEDATE
[6] it is the case that PartyA paid more than PartyB
[7] ExcessParty IS PartyA
[8] ExcessAmount EQUALS AMOUNT 'A' MINUS AMOUNT 'B'
[9] it is the case that PartyB paid more than PartyA
[10] ExcessParty IS PartyB
[11] ExcessAmount EQUALS AMOUNT 'B' MINUS AMOUNT 'A'

(Return to Section 4.5.4, Section 5.3.2).

D Testing

D.1 Unit Tests

Listing 9: A wider selection of parser unit tests.

```
parse_conditional_definition || definition if expression
[CompID "2", Subject "PartyB", ComponentDEFIS, Subject "Debtor",
    ComponentIF, CompID "1", ComponentAffirmation, Subject "PartyA",
    Verb_status Paid, Comparison MoreThan, Subject "PartyB"],
show "[([],Just (CompExprDefinition (Expression 1 Holds PartyA Paid
    MoreThan PartyB) (Def_IS 2 PartyB Debtor)))]"

parse_conditional_definition || if expression then definition
[ComponentIF, CompID "1", ComponentAffirmation, Subject "PartyA",
    Verb_status Paid, Comparison MoreThan, Subject "PartyB", ComponentTHEN,
    CompID "2", Subject "PartyB", ComponentDEFIS, Subject "Debtor"],
show "[([],Just (CompExprDefinition (Expression 1 Holds PartyA Paid
    MoreThan PartyB) (Def_IS 2 PartyB Debtor)))]"

parse_statement || statement
[CompID 1, ComponentAffirmation, ComponentTHE, Subject Consultant,
    Modal_verb Permission, Verb Deliver, ComponentA, Object (Report r),
    ComponentON, ComponentTHE, Number 31, Number 3, Number 2020],
show "[([],Just (CompStatement (Statement 1 Holds Permission (31,3,2020)
    Consultant Deliver (Report r))))]"

parse_condition || negated condition
[CompID 1, ComponentNegation, ComponentTHE, Subject Consultant, Verb_status
     Delivered, ComponentA, Object (Report r), ComponentON, ComponentTHE,
    Number 31, Number 3, Number 2020],
show "[([],Just (CompCondition (Condition 1 NotHolds (31,3,2020) Consultant
     Delivered (Report r))))]"

parse_conditional_statement || statement if condition1 and condition2
[CompID 1, ComponentAffirmation, ComponentTHE, Subject ConsultantA,
    Modal_verb Obligation, Verb Deliver, ComponentA, Object (Report r),
    ComponentON, ComponentTHE, Number 1, Number 5, Number 2021, ComponentIF,
     CompID 2, ComponentAffirmation, ComponentTHE, Subject ConsultantB,
    Verb_status Paid, ComponentA, Object (Pounds 10), ComponentON,
    ComponentTHE, Number 31, Number 3, Number 2021, ComponentAND, CompID 3,
    ComponentAffirmation, ComponentTHE, Subject ConsultantB, Verb_status
    Paid, ComponentA, Object (Pounds 20), ComponentON, ComponentTHE, Number
    31, Number 4, Number 2021],
show "([],Just (CompCondStatement (Conditions_and [Condition 2 Holds
    (31,3,2021) ConsultantB Paid (Pounds 10),Condition 3 Holds (31,4,2021)
    ConsultantB Paid (Pounds 20)]) (Statement 1 Holds Obligation (1,5,2021)
    ConsultantA Deliver (Report r))))]"
```

(Return to Section 5.1.1).

D.2 End-to-end Tests

Listing 10: A wider selection of end-to-end tests.

```
show "[1] It is the case that Bob shall deliver a bicycle on the 1 March
    2021."
[CompID "1", ComponentAffirmation, Subject "Bob", Modal_verb Obligation,
    Verb Deliver, ComponentA, Object (OtherObject "bicycle"), ComponentON,
    ComponentTHE, Number 1, Number 3, Number 2021]
Just (Contract (CompStatement (Statement "1" Holds Obligation (1,3,2021) "
    Bob" Deliver (OtherObject "bicycle"))))

show "[1] It is the case that Bob shall deliver a bicycle on the 1 March
    2021 OR [2] it is the case that Bob shall deliver a bicycle on the 2
    March 2021."
[CompID "1", ComponentAffirmation, Subject "Bob", Modal_verb Obligation,
    Verb Deliver, ComponentA, Object (OtherObject "bicycle"), ComponentON,
    ComponentTHE, Number 1, Number 3, Number 2021, ComponentOR, CompID "2",
     ComponentAffirmation, Subject "Bob", Modal_verb Obligation, Verb
    Deliver, ComponentA, Object (OtherObject "bicycle"), ComponentON,
    ComponentTHE, Number 2, Number 3, Number 2021]
Just (Contract (CompStatement (Statements_or [Statement "1" Holds
    Obligation (1,3,2021) "Bob" Deliver (OtherObject "bicycle"), Statement "
    2" Holds Obligation (2,3,2021) "Bob" Deliver (OtherObject "bicycle")])))

show "[1] It is the case that Bob shall deliver a bicycle on the 1 March
    2021 C-AND [2] It is not the case that Bob shall pay a fee on the 1
    March 2021."
[CompID "1", ComponentAffirmation, Subject "Bob", Modal_verb Obligation,
    Verb Deliver, ComponentA, Object (OtherObject "bicycle"), ComponentON,
    ComponentTHE, Number 1, Number 3, Number 2021, ContractAND, CompID "2"
    , ComponentNegation, Subject "Bob", Modal_verb Obligation, Verb Pay,
    ComponentA, Object (OtherObject "fee"), ComponentON, ComponentTHE,
    Number 1, Number 3, Number 2021]
Just (Contracts_and [CompStatement (Statement "1" Holds Obligation
    (1,3,2021) "Bob" Deliver (OtherObject "bicycle")), CompStatement (
    Statement "2" NotHolds Obligation (1,3,2021) "Bob" Pay (OtherObject "fee
    "))])
```

(Return to Section 5.1.2).

D.3 Example Contract Test

Listing 11: Test for the example contract.

```
show "IF [1] it is the case that Alice paid 100 pounds on the 1 April 2021
    OR [2] it is the case that Alice paid 120 dollars on the 1 April 2021
    THEN [3] it is the case that Bob must deliver a bicycle on the 5 April
    2021 C-AND [4] it is the case that Bob may deliver a report receipt on
    ANYDATE AND [5] it is the case that Bob is forbidden to charge a
    delivery_fee on ANYDATE."
```

```
[ComponentIF, CompID "1", ComponentAffirmation, Subject "Alice",
    Verb_status Paid, Object (Pounds 100), ComponentON, ComponentTHE, Number
     1, Number 4, Number 2021, ComponentOR, CompID "2", ComponentAffirmation
    , Subject "Alice", Verb_status Paid, Object (Dollars 120), ComponentON,
    ComponentTHE, Number 1, Number 4, Number 2021, ComponentTHEN, CompID
    "3", ComponentAffirmation, Subject "Bob", Modal_verb Obligation, Verb
    Deliver, ComponentA, Object (OtherObject "bicycle"), ComponentON,
    ComponentTHE, Number 5, Number 4, Number 2021, ContractAND, CompID "4",
    ComponentAffirmation, Subject "Bob", Modal_verb Permission, Verb Deliver
    , ComponentA, Object (Report "receipt"), ComponentON, Date ANYDATE,
    ComponentAND, CompID "5", ComponentAffirmation, Subject "Bob",
    Modal_verb Prohibition, Verb Charge, ComponentA, Object (OtherObject "
    delivery_fee"), ComponentON, Date ANYDATE]

Just (Contracts_and [CompCondStatement (Conditions_or [Condition "1" Holds
    (1,4,2021) "Alice" Paid (Pounds 100), Condition "2" Holds (1,4,2021) "
    Alice" Paid (Dollars 120)]) (Statement "3" Holds Obligation (5,4,2021) "
    Bob" Deliver (OtherObject "bicycle")), CompStatement (Statements_and [
    Statement "4" Holds Permission (0,0,1) "Bob" Deliver (Report "receipt"),
     Statement "5" Holds Prohibition (0,0,1) "Bob" Charge (OtherObject "
    delivery_fee")])])
```

Quotation marks have been used in this example. in order to make it easier to match up the tokens with the output when verifying correctness. (Return to Section 5.1.2).

D.4 ISDA Master Agreement Section 2(c) Test

Listing 12: Test for the Section 2(c) of the ISDA Master Agreement.

```
show "IF [1] it is the case that PartyA shall pay AMOUNT A on ADATE AND [2]
     it is the case that PartyB shall pay AMOUNT B on THEDATE THEN [3] it is
     not the case that PartyA shall pay AMOUNT A on THEDATE AND [4] it is
    not the case that PartyB shall pay AMOUNT B on THEDATE C-AND [5] it is
    the case that ExcessParty shall pay AMOUNT ExcessAmount on THEDATE C-AND
     IF [6] it is the case that PartyA paid more than PartyB THEN [7]
    ExcessParty IS PartyA AND [8] ExcessAmount EQUALS AMOUNT A MINUS AMOUNT
    B C-AND IF [9] it is the case that PartyB paid more than PartyA THEN
    [10] ExcessParty IS PartyB AND [11] ExcessAmount EQUALS AMOUNT B MINUS
    AMOUNT A."

[ComponentIF, CompID "1", ComponentAffirmation, ComponentTHE, Subject "
    PartyA", Modal_verb Obligation, Verb Pay, ComponentA, Object (Amount "
    A"), ComponentON,Date ADATE, ComponentAND, CompID "2",
    ComponentAffirmation, ComponentTHE, Subject "PartyB", Modal_verb
    Obligation, Verb Pay, ComponentA, Object (Amount "B"), ComponentON, Date
     THEDATE, ComponentTHEN, CompID "3", ComponentNegation, ComponentTHE,
    Subject "PartyA", Modal_verb Obligation, Verb Pay, ComponentA, Object (
    Amount "A"), ComponentON, Date THEDATE, ComponentAND, CompID "4",
```

```
    ComponentNegation, ComponentTHE, Subject "PartyB", Modal_verb
    Obligation, Verb Pay, ComponentA, Object (Amount "B"), ComponentON, Date
     THEDATE, ContractAND, CompID "5", ComponentAffirmation, ComponentTHE,
    Subject "ExcessParty", Modal_verb Obligation, Verb Pay, ComponentA,
    Object (Amount "ExcessAmount"), ComponentON, Date THEDATE, ContractAND,
    ComponentIF, CompID "6", ComponentAffirmation, ComponentTHE, Subject "
    PartyA", Verb_status Paid, Comparison MoreThan, ComponentTHE, Subject "
    PartyB", ComponentTHEN, CompID "7", ComponentTHE, Subject "ExcessParty",
     ComponentDEFIS, ComponentTHE, Subject "PartyA", ComponentAND, CompID "
    8", ComponentTHE, Subject "ExcessAmount", ComponentDEFEQ, NumericalExpr
    (NumericalExprExpr (NumericalExprObject (Amount "A")) MINUS (
    NumericalExprObject (Amount "B"))), ContractAND, ComponentIF, CompID "9"
    , ComponentAffirmation, ComponentTHE, Subject "PartyB", Verb_status Paid
    , Comparison MoreThan, ComponentTHE, Subject "PartyA", ComponentTHEN,
    CompID "10", ComponentTHE, Subject "ExcessParty", ComponentDEFIS,
    ComponentTHE, Subject "PartyB", ComponentAND, CompID "11", ComponentTHE,
     Subject "ExcessAmount", ComponentDEFEQ, NumericalExpr (
    NumericalExprExpr (NumericalExprObject (Amount "B")) MINUS (
    NumericalExprObject (Amount "A")))]

Just (Contracts_and [CompCondStatement (Conditions_and [Condition2 "1"
    Holds Obligation (0,0,2) "PartyA" Pay (Amount "A"), Condition2 "2" Holds
     Obligation (0,0,3) "PartyB" Pay (Amount "B")]) (Statements_and [
    Statement "3" NotHolds Obligation (0,0,3) "PartyA" Pay (Amount "A"),
    Statement "4" NotHolds Obligation (0,0,3) "PartyB" Pay (Amount "B")]),
    CompCondition (Condition2 "5" Holds Obligation (0,0,3) "ExcessParty" Pay
     (Amount "ExcessAmount")), CompExprDefinition (Expression "6" Holds "
    PartyA" Paid MoreThan "PartyB") (Definitions_and [Def_IS "7" "
    ExcessParty" "PartyA", Def_EQ "8" "ExcessAmount" (NumericalExprExpr (
    NumericalExprObject (Amount "A")) MINUS (NumericalExprObject (Amount "B"
    ))]), CompExprDefinition (Expression "9" Holds "PartyB" Paid MoreThan "
    PartyA") (Definitions_and [Def_IS "10" "ExcessParty" "PartyB", Def_EQ "
    11" "ExcessAmount" (NumericalExprExpr (NumericalExprObject (Amount "B"))
     MINUS (NumericalExprObject (Amount "A")))])])
```

(Return to Section 5.1.2).