A Controlled Natural Language for Addressing the Problem of Vagueness in Commercial Contracts

MSc Computer Science – Dissertation

Candidate: Katharine Bryony Sparks

Supervisor: Prof. Christopher D. Clack

2023

Department of Computer Science
University College London

Abstract

This project addresses the issue of the relatively slow take up of automation within the legal domain. Specifically, it attempts to meet some of the challenges facing the creation of computable commercial contracts, through the design and implementation of a Controlled Natural Language (CNL). CNLs in this domain already exist and have permitted the creation of sound computable contracts but have thus far neglected to address the issue of vague and ambiguous language. The central aim of the project was the creation of a CNL enabling the expression of vague language structures, and the development of a dedicated parser to analyse and preserve these structures in an object of the CNL when translating it to another format. Methods for quantifying the incidence of vague language in a contract were also developed.

Table of Contents

1. Introduction

A distinguishing feature of civilized society is its organization around a system of codified laws, derived either from consensus or from some higher authority. Most post-industrial nations have extended the legal system beyond that of public and criminal law, into the spheres of industry and finance. In these contexts, interactions between private individuals and/or businesses are governed by contract law. While traditionally such contracts are bespoke products of lengthy negotiation and drafting by teams of professional lawyers, technological progress over the past several decades has created opportunities for the automation of the negotiation and drafting process. This also provides an opportunity for advanced analysis of contracts, in turn enabling the automation of contract performance.

In this chapter, I provide context to the project, a discussion of controlled natural languages for law and their limitations, and a summary of the aims and objectives of the project. I argue that the lack of support for modelling vague language in law-specific controlled natural languages is an obstacle to the popular adoption of computable contracting.

1.1. Context

The story of computing in the mid-late 20th century is one that has no natural comparands in the history of technology. The immense rate of progress in terms of efficiency and applicability is unprecedented. Computer science as a discipline has seen almost as many paradigmatic shifts in its young life as have its peers whose intellectual roots reach back to the Hellenic era.

Arguably, however, what distinguishes computing from other technological revolutions is its accessibility, widespread adoption, and permeation into almost every aspect of human society. While the advent of technologies like the printing press, the camera, aqueducts, etc., prompted significant reorganizations of culture and production, their concentration within institutions and among wealthy individuals limited their impact to one largely filtered from the top-down. In contrast, beyond the era of early computers, computation has had a far more immediate impact both on industry and on daily life.

This narrative has continued into the 21st century. The development of new internet technologies (such as cloud storage and the blockchain) have made automation accessible and affordable to a wider audience than ever before. In the legal sphere, *computable contracts* have been posited as one possible application for these technologies (Surden, 2012). Computable

contracts are distinct from traditional contracts in that their construction requires an additional layer of computer interpretability. Computable contracts take many forms and have a range of functions, ranging from the automation of relevant aspects of the contract to the analysis of its semantics.

Despite this, the adoption of automation within the legal profession has been relatively slow, particularly when compared to that of other industries and spheres of public life. A primary reason for this is the intricacy of legal language and level of expertise required to wield it effectively. In a commercial setting, well-drafted legal contracts are key not only to the unproblematic performance of a business agreement, but also to the prevention of potentially expensive future litigation. Acquiring the skills to practice law effectively generally necessitates several years of study via professional programs that often (though not always) lack training in programming languages. The intricacy of legal language is compounded by the inherent complexity of natural language itself, the syntactic and semantic quirks of which it inevitably inherits.

As such, a major block to the widespread adoption of computable contracts is the confluence of the following factors:

1. Natural language's supervenience on legal language (entailing within the latter all the complexities and inconsistencies of the former),
2. The norms and conventions of contract drafting/analysis as highly specialised knowledge,
3. The knowledge, language and culture gaps between lawyers and programmers.

The development of controlled natural languages specific to contract drafting may serve to resolve these issues by providing a bridge between computing and law.

1.2. Controlled Natural Languages

Controlled natural languages (CNLs) might best be defined in terms of how they differ from traditional programming languages. While the latter are purely formal languages - consciously designed around a predetermined set of syntactic and semantic rules/systems - CNLs directly derive from a natural language. This derivation entails the adoption of some elements of the natural language (lexical features, syntax, semantics, etc.), and the exclusion of others. The choice of what properties of the natural language will be preserved in the CNL is generally

determined by the CNL's purpose. Upon compilation, CNL and non-CNL languages coincide – i.e., a file generated with a CNL may then be analysed for its underlying logic and potentially converted to machine language code, just as a Python or Java file may be.

To illustrate this idea, consider a CNL derived from French for the purpose of taxonomizing different species of animal. To be useful to a French-speaking user, the CNL must preserve central features of the French language (such as gender and grammatical moods, etc.). To meet the domain-specific requirements of the CNL, however, some properties and systems will either be omitted or streamlined. In our example, any CNL that models taxonomizing must have functionality to allow the user to categorize ("*x* is-a *y*"). It must also preserve the structure of these categorization statements for analysis purposes (i.e., as identifiable components in an abstract syntax tree). As such, while extensive forms of conjugation exist in the natural French language, it might be unnecessary to model subjunctive and imperative conjugation. Rather, capturing the logics of conditional and indicative forms of conjugation would take priority, given the narrow purposes of the CNL.

1.3. Law-specific CNLs, Deontology, and Limitations

Several CNLs have been proposed for the purpose of drafting legal contracts (Prisacariu & Schneider, 2012), (Kowalski & Datoo, 2021). Generally, currently existing CNLs are derived from English and are primarily concerned with modelling the deontic aspects of real contracts. The deontic aspects in question concern the prohibitions, obligations and permissions that are central to legal documents. As a subcategory of modal logic, deontic logic also facilitates the formalisation of temporal expressions. In the context of legal contracting these relate to periods in which various obligations or permissions may hold. Auxiliary components include the parties to which a contract pertains, the actions those parties may take to fulfil their contractual obligations, events or states of affairs that might come to pass during a contract period, and the objects or assets involved in the contract.

The most significant challenge facing legal language CNLs has been the problem of deontic paradoxes, of which there are many (the details of which are largely outside the scope of this project). Various solutions to the problem have been posited, such as:

1. Restricting available syntax as to preclude the formulation of paradoxical statements (Pace & Schneider, 2009)

2. Allowing for paradoxical statements to be formed but providing some mechanism for the detection of inconsistencies, either directly or through reduction to the form of contradiction in classical logic (Karadotchev, 2019)

Each approach generates its own challenges, with implications for the verbosity of the CNL and its success in emulating natural language, and the complexity of its implementation code. These challenges as they relate to this project are expanded upon in chapters 3 and 4.

Avoidance of the paradoxes of deontic logic enables the drafting of sound computable contracts and goes some way towards incentivising the popular adoption of computable contracts in the legal sphere. However, a focus on deontology ignores certain vital features of real legal contracts, and as such fails to provide a comprehensive alternative for traditional contract drafting.

In chapter 2 ('Vagueness, Ambiguity and the Law'), I provide a detailed account of why vague and ambiguous language is both a significant problem for legal interpretation, and a powerful tool commonly used in contracting. I examine real cases in which the presence of vague and ambiguous language in commercial contracts led to litigation. Because legal contracts are so heavily concerned with contingencies, conditional states of affairs feature significantly in legal language. And while pre-existing CNLs account for contingency (via the classical conditional), contracts often combine the traditional language of contingency with appeals to ambiguous referents and vague states of affairs. It can and should be argued that these appeals are non-trivial to the meaning and interpretative value of legal contracts, and as such cannot be dismissed as linguistic embellishments. As such, the absence of vagueness and ambiguity in the semantics of pre-existing CNLs leaves a major blind spot unresolved. This omission limits the viability of computable contracts as an alternative to traditionally drafted ones.

1.4. Project Objectives

On the basis of the problems outlined above, the aims of this research project are as follows:

1. A controlled natural language for law, enabling the expression of vague language structures.

2. Mechanisms to analyse an object written in the CNL, to detect those vague structures, and to preserve these structures upon translation of the CNL object into another format.
3. Mechanisms for quantifying vague language.
4. An analysis of vagueness in terms of legal scholarship, logic, and linguistics, for the purpose of informing the design of the controlled natural language.

In chapter 2, I provide justification for my project objectives through an assessment of legal literature on the problem of vagueness. I highlight the reasons behind the frequency of vague language in contracts. I conclude that while vague and ambiguous language can act as a source of legal contention, its usage is necessary to the proper functioning of contract law.

In chapter 3, I provide an analysis of vagueness as a concept and as a linguistic category. I examine how vagueness manifests in natural language. I provide a taxonomy of vague types based on this analysis. I explore the difficulties inherent to modelling vagueness programmatically, and why these issues are compounded in a legal context.

In chapter 4, I outline the methodology used in designing and implementing my controlled natural language, informed by the findings of the previous sections. The grammar is provided in Backus-Naur Form. The implementation of a lexer/parser for the language is outlined, with commentary on technologies used, process, and results. Quantification mechanisms are similarly expounded here.

I conclude by summarizing the project as a whole and outlining how each of the objectives (1) – (4) were met.

2. Vagueness, Ambiguity, and the Law

This chapter provides an overview of vagueness and ambiguity through the lens of legal theory. The purpose of this section is to deliver evidence for the premise put forward in the previous section – that is, that vague language is central to contract law, and as such should insofar as is possible be modelled in relevant CNLs. Legal literature on the problem of vagueness is examined here, and a discussion is provided as to the positive role of unspecific language in contract drafting and interpretation.

2.1. Definitions and Preliminaries

Before the role of unspecific language in contract law can be explored, it is pertinent to provide some preliminary definitions. It should be noted that the terms *vagueness* and *ambiguity* are unpacked in detail in chapter 3, with discussion of their formal and conceptual properties, of their differentiation from one another, and of their specific roles in language. As such, the definitions provided here are intentionally limited for the purpose of establishing their relevance in the legal context. *Unspecific language* can be taken as an umbrella term encompassing a range of linguistic features. Here, we are focused on two such features: vagueness and ambiguity. While often used interchangeably, the two refer to different linguistic phenomena.

Vagueness can be defined as language that is so imprecise regarding its referents or truth claims as to generate potential borderline cases. Vague propositions are vague due to the lack of possible avenues of inquiry that would verify their truth value (Hu, 2017). Vague statements are often ones that concern description or categorization. There is often some degree of subjectivity attached to a vague statement.

Borderline cases are the products of statements that have multiple valid interpretations as to what they refer, many of which may conflict with one another. A borderline case is the result of a sentence that makes a claim about some feature or phenomenon (be it literal or conceptual), using language that obscures the object or truth conditions of that sentence. That is, if an interlocuter makes a statement about the world that could be reasonably interpreted as having multiple potential meanings, or reasonably be evaluated as true or false in equal measures, they have made a vague statement that generates borderline cases. In common-sense terms, borderline cases are the result of 'failing to draw the line'. An example might be "He's

a handsome man". One could reasonably judge this true or false, and it is unclear as to how handsomeness can be objectively measured to settle disagreement.

Ambiguity describes statements whose constituent parts contain words that might have multiple meanings. While this also generates potential borderline cases, ambiguous statements differ from vague statements in that upon disambiguation, the referents of the former can become fixed. This is not the case in vague statements, who maintain controversy as to their referents, meanings, or truth-values even after disambiguation is applied. Ambiguity may also be the result of misused punctuation and may be resolved through altering sentence structure.

Some examples of vague statements include "Rembrandt was short for a Dutchman", "Personhood begins in the womb", "The garden is overgrown". The properties and categories invoked do not have sharp, objective boundaries, making truth conditional verification difficult and/or impossible. Ambiguous statements include "Alice walked her dog by the bank" (i.e., river or financial bank?), "Connor waved, and the flag did too" (i.e., social act or kinetic event?), "Let's eat Grandma" (i.e., cannibalism or family dinner?). For further discussion and illustrations, see the next chapter.

2.2. Law and the Sorites Paradox

The impact of vague language can be felt across most sectors of the legal world. While our focus is on commercial contracts, it is not difficult to see how imprecision might prove problematic elsewhere. In the criminal justice system, for instance, the consequences of confusion over the definition of a crime could be catastrophic for all parties involved. Confidentiality and termination clauses must also be highly specific as to avoid the potential for future litigation. Similarly, the courts that have jurisdiction over a contract may not be left undefined. The potential for borderline cases seems, superficially at least, to be something we should strive to avoid, lest we run into controversy over interpretation.

2.2.1. The Sorites Paradox

Precision and clarity are desirable traits in normal legal practice. Furthermore, the avoidance of vague language is often treated as a virtue. The problem of vagueness has generated significant discussion in legal literature (Páll Jónsson, 2009). Much of this discussion reduces to what is termed the Sorites Paradox. Sorites concerns the problem of borderline cases. Simply

put, the paradox states that it is possible to extend a basic categorization statement indefinitely as to reach a final inference that, while inductively sound, defies common-sense or invites controversy. We can illustrate this with *modus ponens*:

1. **Base step:** A man with no hair is bald.
2. **Induction step:** A man with *n+1* hairs is no more or less bald than a man with *n* hairs (where *n* is zero hairs and *n+1* is 1 hair, and so on).
3. **Inference:** Therefore, a man with 500,000 hairs is bald.

Formally, where Φ is a predicate 'baldness', *n* is a natural number, and α_n is a value in a series for Φ:

(Induction Sorites)

$$\frac{\Phi\alpha_0 \qquad \forall n(\Phi\alpha_n \rightarrow \Phi\alpha_{n+1})}{\forall n(\Phi\alpha_n)}$$

This can also be formalized in terms of the conditionals it entails:

(Conditional Sorites)

$$\frac{\Phi\alpha_0 \qquad \Phi\alpha_1 \rightarrow \Phi\alpha_2 \qquad \Phi\alpha_2 \rightarrow \Phi\alpha_3 \qquad \ldots \qquad \Phi\alpha_{n-1} \rightarrow \Phi\alpha_n}{\Phi\alpha_n}$$

where *n* can be arbitrarily large.

The final inference seems plainly false, defying the common-sense definition of baldness. Naturally, we might attempt to resolve this by rejecting the induction step. The

problem with this is that in negating the induction step, we affirm its contrapositive – that is, we find ourselves accepting that a man with a single hair on his head cannot be described as bald. Not only is this controversial, but it forces us into drawing a sharp distinction between bald and non-bald referents. Because the induction step is a generalization that does not permit exceptions (Boolos, 1991), in rejecting the induction step, we imply the existence of a threshold *n* number of hairs that must be reached for baldness to cease to hold as a property. Further, we must accept that any proposition invoking the concept of baldness can be checked against this threshold to verify its truth value. Such a specific threshold cannot be defined without some degree of controversy or arbitrariness, resulting in the Sorites puzzle. Attempts to refine the Soritical predicate through introduction of new predicates (e.g., 'baldish') fall foul of the same issues, and lend themselves to the problem of higher-order vagueness and infinite regression.

2.2.2. Legal Scholarship on Sorites

It is possible to define vagueness itself in terms of the paradox, describing it as "[…] an indeterminacy in the applications of an idea, as to how many grains are required to make a heap, and the like." (Peirce, 1892). Sorites has acted as the starting point for many accounts of vagueness (both in conceptual-logical terms and in terms of how it manifests in natural language). These accounts are explored in chapter 3. However, here our focus must stay on legal theory, where the paradox has taken central stage in discussions on vagueness. The so-called 'Bad Samaritan Law' thought experiment can be used to illustrate the application of Sorites to legislation (Hunt, 2016).

Imagine some law is passed that automatically charges with manslaughter any individual observed not intervening to save the life of another person(s) who is clearly in peril. All individuals must provide "reasonable assistance" to the potential victim. If the individual in question was not able to come to the aid of the victim without putting themselves in peril, then demanding they assist is unreasonable. Now, consider that a child has drowned in a lake. A woman was observed walking some ways into the lake to save the child but turned around before reaching them. She is charged under the Bad Samaritan Law. In her own defence, the woman argues that while walking some steps into the lake was reasonable assistance, going as far into the lake as the child was when drowning should be considered unreasonable. She argues that if one step into the lake is not unreasonable, then two steps are not – the difference between one and two is morally trivial. Continuing this line of argument indefinitely implies moral triviality between ten and eleven steps, etc. But, the woman argues, this incremental triviality

can be chained to imply that taking five hundred steps into the lake to save the child's life is an entirely reasonable request. She argues that this is absurd. The prosecutor might argue that all she has proven is that this line of thought leads to absurdity, it does not prove that she is not liable. However, the judge is not so sure, and advises the jury that it cannot be proven at what point the woman's duty shifted from reasonable to unreasonable.

In Hunt's example, the Soritical predicate is "reasonable assistance". But the predicate could theoretically be anything, and as such could apply equally in other spheres of the legal world – most importantly for our purposes, to contract law. One response to the Sorites problem is Feinberg's argument for 'borderline zones' (Feinberg, 1984). Feinberg argues that the kind of borderline cases generated by Soritical predicates should be split into zones. These zones are demarcated based on some consensus as to the degree to which the predicate applies. Zones where the predicate certainly does or does not apply bookend middle zone(s), encompassing all cases in which consensus cannot be reached. As to the question of legal responsibility, Feinberg argues that cases falling in borderline zones should not be held liable. Here then, the burden is on the prosecutor or suing party to prove that the prosecuted party is *clearly* liable. Note that this solution comes down to the question of consensus, spotlighting the central problem of Sorites to the legal world: one of interpretation. Dworkin's proposed solution to vagueness similarly comes down to the norms of interpretation (Dworkin, 1986). Dworkin argues that vague predicates (such as "cruel" or "fair") that often feature in legal controversies are 'semantic defects' – i.e., the product of miscommunication among speakers as to the correct interpretation of these predicates. Moore similarly dismisses the problem of vagueness as the product of the limits of human language to accurately describe real-world objects and phenomena (Moore, 1992).

Clearly then, the problem of vagueness and the paradoxes it entails are central issues in legal theory. It should be noted however that much of the literature we have touched upon raise questions that are far beyond the scope of this project and will not be explored here. Feinberg, Dworkin, and Moore's proposals have generated extensive discussion in legal theory around the applicability of classical logic to law (due to the central role of bivalence in Sorites), epistemicism, and other topics in ethics and philosophy that do not pertain to the aims of our project. The purpose of this section is to illustrate that vague language is a non-trivial feature of legal language, and to identify aspects of legal literature that relate to the design of our controlled natural language. As such, we limit our exploration of the topic with this in mind.

2.3. Necessary and Desirable Vagueness

Returning to our project, we might conclude that vagueness is completely undesirable for a contracting CNL. We might assume then that in designing a CNL for use by legal professionals, we should aim to eliminate the possibility of vague language use, insofar as that is possible. However, this would be undesirable. Vague language plays an important role in the correct functioning of contract law.

Consider *Force majeure*, a common clause in commercial contracts. Force majeure clauses allow lawyers to include contingency plans for unforeseen circumstances, such as natural disasters. This is particularly important for the avoidance of further litigation in business agreements that rely heavily on smooth logistical operation. It is in the interests of all parties that a disruption in the chain of supply or production does not lead to the contract becoming void, with terms vague enough to encompass a variety of possible contingencies. Similarly, dispute resolution or arbitration clauses are often included in commercial contracts. These may be unspecific as to the triggers for arbitration, its length, or opt-out conditions.

Academic literature on vagueness is not limited to negative readings of the subject. Many legal approaches to vagueness highlight its productive capacities. Endicott, for instance, argues that legal controversies generated by cases like the one outlined in the Bad Samaritan thought experiment can be read as opportunities for clarification (Endicott, 2011). To Endicott, vague language prompt courts to specify the details of a law. He also argues that the existence of vague laws can incentivise lawyers to draft highly specific contracts, to avoid risk. Similarly, Soames sees controversy over vague language as a conflict over interpretation that may prompt the drafting of new laws or contracts (Soames, 2011). Soames argues that this serves to perfect the legal system through the process of 'precisification'.

In contrast to Endicott and Soames, who stress the beneficial aspects of vagueness in legislation and interpretation, Bernheim and Whinston focus on the utility of vagueness to contract law (Bernheim & Whinston, 1998). On this position, when some contingencies, objects or other features of a contract are unverifiable, some deontic aspects of that contract may be left vague to the benefit of all parties. The presence of any form of vagueness renders the contract incomplete. However, in a departure from the traditional perspective, incompleteness is presented as being an equally sound property as completeness. The argument is that when optimal completeness cannot be achieved, an intentional form of incompleteness should be pursued. This position stresses the formal aspects of contracts. Framed in game

theoretical terms, well-designed contracts are posited to be those that respond to the unverifiable or vague aspects of a contract by intentionally obfuscating the conditions of verification for other aspects, even when the latter are in fact measurable or open to clarification. This is posited as a desirable remedy if the unverifiable aspects initially tilt the balance of interpretation in favour of one party or another. This reading of vagueness is motivated by the premise that vagueness often cannot be avoided, either in terms of language or in terms of contingencies, and that therefore some mechanism towards securing fair interpretation is of paramount importance.

Vagueness then is not so two-dimensional as to be an entirely negative aspect of language. Nor is there a clear way in which it can be avoided completely – both as a feature of legal language and as a reflection of the complexities and contingencies of the material realities contracts pertain to. At worst, the attitude of legal scholars to vagueness might be summarised as one of grudging acceptance. At best, we see a tentative optimism as to its role in the perfection of norms and interpretation.

2.4. Ambiguity and Reference

The focus thus far has been on *vague* language and its implications for contract law. While *ambiguity* is a fundamental feature of legal language, it is side-lined in our survey of the literature. This is because its specific properties – those that differentiate it from vagueness - have no unique role to play in law. By this, we mean that the problems of reference and verification that ambiguity generates for legal language reduce to the same problems in the context of natural language. And, crucially and in contrast to vague propositions, fuzzy referents and borderline cases generated by ambiguous propositions can be solved universally through the application of disambiguation. This is not available in the resolution of vagueness (see chapter 3 for more on this). The role of vagueness in law is an extension of its role in natural language in general, but one that generates problems specific to that domain (which, in turn, generate particular norms of interpretation and contracting).

This is not to say that ambiguity cannot be highly problematic in a legal context. Plainly, the correct fulfilment of an obligation becomes impossible if the parties involved have no consensus as to what that obligation entails. An example might be the obligation for a distributor to deliver toothpicks to a restaurant. The restaurant is expecting wooden table toothpicks typical to the culinary context. But the distributor might deliver plastic toothpicks

of the kind sold in a chemist. The distributor might claim when challenged that the restaurant was not specific enough when placing their order and that technically the contract was fulfilled. Potentially expensive litigation might be triggered in such an example.

Much like vagueness, ambiguity also plays a necessary role in the drafting of legal contracts. Natural language could not function without the use of non-rigid designators (Kripke, 1981). As an extension of natural language, legal language is no different. To accurately reflect real world states-of-affairs, lawyers must be allowed to utilise unfixed referents in deontic statements. A simple example of this might be a clause that obligates one party to pay the other interest on a loaned asset for a set period, where the rate of interest paid is pinned to UK mortgage rates year-by-year. Because these rates are varying, "interest" as a term in such a contract is a non-rigid designator. While technically ambiguous, this is unproblematic here.

While closely related to vagueness, ambiguity is a distinct linguistic phenomenon and must be treated as such. Modelling ambiguity in a CNL also poses unique challenges, as will be explored in the next two chapters.

2.5. Application to Commercial Contracts

Much of the discussion on imprecise language in law outlined in previous subsections has centred on legislation and lawfulness, although we have introduced work that specifically relates to commercial law. Because our project is directly concerned with the business sector and contracts pertaining to it, it is useful to point to examples of where vague and ambiguous language has been cited as significant to commercial contract litigation. Our findings here directly informed the design of our CNL grammar (see subsection 4.1.5).

In the case *RTS v Müller* (RTS Flexible Systems Ltd V Molkerei Alois Müller Gmbh & Co KG, 2010), a dispute arose between the parties listed over the timeliness of services provided by the former to the latter. Briefly, RTS brought a claim against Müller for unpaid fees. Müller countered by arguing that RTS had not yet fulfilled their contractual obligations, and therefore had no right to request payment. The parties had initially agreed to enter into a detailed contract outlining the terms upon which the work was to be carried out. However, the exact terms of the contract were not finalized before work began. As such, an interim contract was entered into, with loosely agreed-upon terms and conditions. The contract outlined obligations for payment and delivery, but wrapped them in vague language regarding timeframe for fulfilment ('[…] *within* 30 to 90 days of takeover […]', '[...] payment of 30%

after delivery'). Contention therefore arose due to ambiguity over what complete delivery consisted of, and disagreement over the timeframes within which delivery and payment should coincide with one another. The case went to court and was appealed twice. Presiding judges in the initial case and in its appeal ruled differently from one another. The third and final judge identified the source of disagreement between the judges as a result of '[both parties] submitting that at *some stage* a contract came into existence which governed their relationship', and, given the vagueness of the language used, both parties being reasonably able to argue in favour different interpretations of that contract. While this case no doubt adds credence to the notion that vague language can enable miscommunication and lead to expensive and lengthy litigation, it also illustrates why vague language is useful and necessary in commercial law. As the final presiding judge puts it '[the case] demonstrates the perils of beginning work without agreeing the precise basis upon which it is to be done.' The counter to this argument is that it is assumes an ideal context in which real world constraints do not exist. As it stands, the parties initially intended to enter into a complete contract but were forced by time constraints to enter into an incomplete one – the use of vague language enabled them to do so. While in this case, incompleteness led to litigation, this should not be universalized. As such, vague language has a place and purpose in commercial contracts.

The case *Chartbrook v Persimmon* (Chartbrook Ltd v Persimmon Homes Ltd and Others, 2009) further illustrates the problem of imprecise language. In this case, a dispute arose between the parties over semantics. A clause in the contested contract stipulated that Persimmon's fee for using Chartbrook's land to construct properties would increase 'if the properties sold for more than expected'. Chartbrook requested an increase in payment on the basis that the contingency to which they agreed had been realized. Persimmon refused on the basis that the language used to define the contingency and its connotations was too loosely defined. The presiding judge in this case remarked that '[…] the meaning of words varies according to the circumstances with respect to which they are used. […] the contract does not use algebraic symbols. It uses labels […] they are usually chosen as a distillation of the meaning or purpose of a concept intended to be more precisely stated in the definition […] [the definition] may help elucidate ambiguities.' The appeal to definition in the resolution of contested contingency measures illustrates where ambiguity and vagueness differ (as expressions of the latter kind cannot be disambiguated), but also where they may pose a common problem in commercial contract interpretation.

2.6. Conclusions for Project

We now have a sense as to how unspecific language is treated in legal theory and scholarship. It is often a source of frustration for the creation and upholding of interpretative norms. Further, attempts to avoid it or to resolve the paradoxes it leads to have been unsuccessful (for reasons largely beyond the scope of this paper). Nevertheless, there is also significant literature that frames it in a far more favourable light.

Ultimately then, we might summarise vagueness and ambiguity as 'necessary evils' in this context. We must now measure the relevance of this to our aim: the design and implementation of a controlled natural language for the legal domain. Vagueness as it manifests in natural language generates borderline cases – this seems inevitable. Use of the Soritical predicate is a simple way of formalizing this problem. However, as we have seen, the formal properties of natural language that substantiate vagueness and ambiguity are wide-ranging and cannot be reduced to a single linguistic mechanism. Similarly, the impact of imprecision in a deontic context – the context of contract law – is felt differently depending on its use case (e.g., force majeure vs. an ambiguous referent). The conclusion we can draw from this is that while our CNL *could* be designed around a singular predicate for vagueness, this would obscure the specific roles its different forms play in the structure of a contract. As such, our CNL should be informed by an analysis of vagueness and ambiguity as features of language, and a taxonomy based on this analysis.

We should not negate the problematic aspects of vague language in the process of accounting for it. Precision and conciseness are desirable in most domains, and particularly so in the legal context where so much (material or otherwise) may be at stake for the parties involved. Given the scope of the project, a reasonable remedy is to provide a mechanism for quantifying the degree of vagueness present in a contract drafted in our CNL. This should provide information on types of vagueness and their frequencies. Enforcing limitations on vague language is not desirable in this context. The range of attitudes encountered in this section suggests that, although we have a definition of vagueness and the problems it generates, its role in the legal context is contested and to some degree subjective. The end-user should be free to edit or preserve their contract based on information provided by the quantification mechanism.

3. Analysis, Taxonomy, and Models of Vagueness

Having established the relevance of unspecific language to law, we must now consider how best to model it in a legal CNL. While the previous chapter provided a general overview of vagueness and ambiguity, a more comprehensive analysis is necessary. This analysis is two-fold.

First, we consider how vagueness and ambiguity manifest as properties of natural language. Various methodologies have been proposed for demarcating these properties into distinct categories. We draw from these to formulate a working taxonomy of vague and ambiguous types.

Second, we consider relevant literature on vague propositions and their formal character. This will inform the design of the grammar underpinning our CNL. This analysis leads into a discussion of the theoretical challenges that arise in the application of antecedent theories of vagueness to the context of legal language. I argue that the outcomes of pre-existing analyses of vague propositions cannot be unproblematically applied to the unique requirements of legal CNLs. Many prominent accounts of vagueness as a linguistic feature focus on philosophical questions around a speaker's agency and belief. In the context of legal language, such questions are largely secondary. I consider the implications of this to the design of our CNL.

3.1. Analysis

In their wide-ranging survey of vagueness as a special topic in formal semantics, Kamp and Sassoon introduce the question as "[…] an ultimate challenge. An enormous diversity of literature on the topic has accumulated over the years, with no hint of a consensus emerging" (Kamp & Sassoon, 2016). This lack of consensus is reflected in the multitude of vague taxonomies that have previously been proposed. The lesson from this is that we should embrace the fact that, while vagueness is an observable phenomenon in natural language, its interpretation, measurement, and typification is to some degree subjective.

There is some agreement on the generic indicators of vague expressions. An exploration of these indicators will help us formulate our working taxonomy. Flags for vague expressions include:

1. Susceptibility to the Sorites Paradox *(Section 3.1.1)*
2. Dependence on a comparison class for meaning *(Section 3.1.2)*
3. Subjectivity in truth value assessment (*Section 3.1.3)*
4. Sensitivity to context in truth value judgement *(Section 3.1.3)*
5. Presence of tolerance-governed predicates *(Section 3.1.4)*
6. The generation of borderline cases and fuzzy boundaries *(Section 3.1.4)*
7. Modifiers that select for certain types of vague predicates *(Section 3.1.5)*
8. Behaviour under ellipsis; resistance to disambiguation *(Section 3.1.6)*

Note that in natural language, the terms and structures that entail these indicators are often interrelated and co-dependent. Our aim in this subsection is a finely grained grammatical analysis of vagueness, but one that remains relevant to the design of our controlled natural language. Because we explored the Sorites Paradox in depth in the previous section we will only mention it briefly here. Some indicators of vagueness are evident on the atomic and propositional levels, while others only manifest in context of larger inference structures. The purpose of this subsection is to expound these indicators and their relationships with one another, and from this to move towards a taxonomy of the categories that draw them together.

3.1.1. Soritical Predicates

While we are dispensing with further discussion of the Sorites Paradox, the idea of a Soritical predicate as the primary mechanism behind vagueness is largely mirrored in the broader literature on the topic. Put simply, discourse on vague language often centres on the vagueness of predicates. Predicates are often measured in terms of their application domain. A *positive extension* of a predicate is the domain to which that predicate certainly applies, while the *negative extension* is the inverse of this. Vagueness persists in the *truth value gap*, i.e., the set of all borderline cases the predicate may apply to (Zadeh, 1965).

3.1.2. Comparison Class Dependency

It is in this context that the notion of comparison class dependency appears (Klein, 1980). A statement that utilizes adjectives is one that relies on a comparison class for meaning. A particular object will instantiate several conceptual and material properties. But where the

central meaning of a proposition is one that relies upon knowledge/reference to a particular property or cluster of properties, that proposition relies upon a comparison class to secure the referential validity of the proposition (Frege, 1948) (Russell, 1905). A comparison class is the set of entities determined to *definitely* instantiate a property, used as a criterion for judging the truth value of a proposition that assigns that property to an object.

Note that comparison class dependency forms the basis for other indicators of vagueness that involve the use of adjectives. Two such indicators are the possibility of subjectivity as to the exact membership of an invoked comparison class, and sensitivity to context in the assessment of descriptive and categorizing propositions (subsection 3.1.3). *Tolerance-governed predicates* (subsection 3.1.4) describe a subset of adjectival predicates and so also indicate comparison class dependency. As such, comparison class dependency will be referenced in discussions of those indicators.

Adjectives are the most common form of predicate that reference a comparison class, but not all adjectives are vague. For instance, so-called *sharp* adjectives such as 'prime' or 'even' to describe quantity or numeric value do not generate borderline cases, and so do not function as vague predicates. This contrasts with the vast majority of adjectives – *relative* adjectives – which do. Examples of relative adjectives include 'big', 'smart', 'tall', 'bald', etc.

3.1.3. Subjectivity and Sensitivity in Truth Value Determination

This also introduces two further indicators of vagueness: that of subjectivity in truth value assessment, and of sensitivity to context in truth value judgement (Graff, 2000) (Kennedy, 2007) (Barker, 2002) (Crespo & Veltman, 2019). Propositions of the former entail *comparison class-determining for-phrases* (Cruse, 2010) (Gehrke & Castroviejo, 2015), while propositions of the latter feature *judgement-determining phrases* (Rips & Turnbull, 1980).

An example of subjective indication is the statement "this meal is small for its price". The speaker is invoking an adjective 'small', inviting the listener to judge the validity of their statement through comparison to various potential classes – e.g., the set of all meals, the set of all small meals, the set of all expensive meals, relative portion sizes, etc. Subjectivity here is further compounded by the object itself. A meal can be comprised of various food and drink types, each of which with different comparison classes for judgment of smallness, expensiveness, etc. A meal made entirely of caviar, for instance, would have a significantly different set of criteria for unreasonable size-to-expense ratios to be judged against than a meal

of rice and peas. There may also be a degree of fuzziness as to the boundaries of the food types themselves – e.g., the point at which pasta al dente becomes standard pasta, which may provide further context for truth value assessment based on some comparison class or another. This implements the concept of nested comparison class statements, with the potential of transforming the overall subjectivity (and so possible indicators of vagueness) of the larger proposition.

An example of a judgement-determining phrase might be the statement "In my opinion, this street is clean". The comparison class is implied, but the difference is that it is framed in terms of the speaker's judgement. This judgment adds additional context to truth value assessment (e.g., the speaker's beliefs, expectations, known variations in context that might weigh for or against the truth value of the proposition, etc.) for the listener to reason with. The proposition may variably be judged as true or false dependent on context. Regardless of sentential differences, however, the matter of comparison class dependency persists. And, along with it, persists the problem of borderline cases. For even when the comparison class and context of a predicate is fixed, there may be indeterminacy as to the correct tripartition of its extensions.

3.1.4. Tolerance-governed Predicates; Borderline Cases and Fuzzy Boundaries

Vague expressions rarely exist in isolation. The ability to make inferences is at the centre of natural language. It is here that the notion of *tolerance* comes into view. As an indicator of vagueness, tolerance is the notion that small differences have no consequences for membership (Dummett, 1995) (Wright, 2021). Adjectival predicates have varying tolerance ranges. For instance, 'tall', as a Soritical predicate, is a relative adjective and so resists the imposition of rigid definition. In contrast 'prime' is a sharp adjective – it describes a category that has no tolerance range, precluding fuzzy boundaries and borderline cases. Tolerance is often used to measure the degree of vagueness an adjectival predicate introduces to a proposition. Tolerance-governed adjectives may be split into two subsets: absolute and non-absolute (Burnett, 2014). Where both the positive and negated form of a predicate have a tolerance range, the predicate is non-absolute, and allows for symmetric forms of inference. Where only one predicate form has a tolerance range, the predicate is absolute (maximal-criterion for the positive form, minimal-criterion for its negative form). Asymmetric inference patterns are permitted here. 'Tall' and 'short' are examples of non-absolutes, while 'straight' and 'bent' are maximum-

criterion absolute and minimum-criterion absolute respectively. The difference comes down to the falsifiability of propositions about membership. Non-absolute predicates' positive and negation forms may both be used in Soritical arguments, whereas absolute predicates have one extension (negative or positive) that precludes Sorites arguments from being constructed for it. Neither are sharp, but the latter are taken to be 'less vague' than the former.

3.1.5. Vague Modifiers

In natural language, vagueness generally manifests in combinatorial relations, rather than in the simple predicate form our analysis has centred on so far. This is to say that an effective analysis of vagueness must be mereological. Modified vague expressions take the generic form <*modifier*> + <*expression*>, with variation in ordering and structural properties determined by the expression type. An expression can take many forms, atomic or compound, including but not limited to predicate expressions, descriptive expressions, deontic expressions, conditional expressions, temporal expressions, etc. Vague modifiers can generally be classified into three categories: adjectival modifiers, quantifying expressions, and frequency adverbs (Lakoff, 1973) (Lewis, 1975).

Examples are provided to illustrate how modifiers of each category act on expressions. In each example below, four pairs of sets are defined. In each pair of sets, the first set contains modifiers, while the second contains expressions they may modify. Example sentences are given. These contain modified expressions that may be generated by combining elements of the modifier set with elements of the expression set.

Note that in English sentences, modifiers may not be grammatically adjacent to all expressions they modify. This is particularly true for quantifying expressions, where the quantifier is often adjacent to the variable(s) it quantifies, rather than adjacent to the expression whose domain the quantifier is restricting (e.g., "*Some x* are *large"*, where the quantifier *some* and adjectival expression *large* are separated by the variable *x* and the linking verb *are*).

Adjectival modifiers (or adverb modifiers) often act as indicators of the presence or absence of vagueness in a sentence or set of sentences, and may include straightforward negation/negational predicates:

a) {slightly, perfectly} {clean, tidy} → This street is perfectly clean.
b) {a little bit, very} {dirty, messy} → Her room is a little bit messy.

c) {extremely, not very} {tall, short} → Jack is extremely tall.
d) {unusually, worryingly} {obese, thin} → That cat is worryingly thin.

Taken as indicators alone, choice of modifier can be divorced from the semantic content of its applicant. On a functional level, they either strengthen or lessen the vagueness of the adjectival predicate they apply to. However, from our example set above it is clear that in actual language, norms of sense and meaning can be lost in the selection of modifiers (e.g., in describing a room as 'perfectly messy' we go outside the norms of use for that modifier/adjective combination). This goes beyond the question of vagueness but does illustrate the notion of modifiers selecting for certain instantiations of vagueness. What is relevant is that strengthening adverb modifiers may not negate vagueness completely; 'Jack is extremely tall' generates the same borderline cases as does any proposition that depends upon a comparison class. Similarly, modifiers such as 'unusually' and 'worryingly' both strengthen their adjectival predicate and invite analysis of the proposition's comparison class. The latter point is particularly relevant. There is significant controversy over whether generality is a form of vagueness, or a closely related but distinct type in the vein of ambiguity (Carlson & Pelletier, 1995) (Zhang, 1998) (Lakoff, 1970). In the context of adjectival modifiers, however, sentences of kind (d) feature adjectival modifiers that strengthen, weaken, or otherwise highlight their objects' genericity or lack thereof. This modification is framed in terms of the proposition's comparison class dependencies, and so can be reasonably described as a form of general adjectival modification.

Quantifying expressions are linguistic structures that measure and describe sets (including empty sets). Quantifying expressions typically involve *domain restrictions* (von Fintel, 1994). Quantifiers delimit the application domain of an expression; vague quantifiers only loosely delimit:

a) {almost all, more than half, some} {unemployed, employed} → More than half of his friends are unemployed.
b) {every, no} {married, single} → No bachelor is married. *(not vague)*
c) {around, nearly} {ten, a million} → He lost nearly a million pounds.
d) {between, before} {2024, January/July} → The election will be held between January and July.

Modifiers of the kind found in (a) indicate quantificational vagueness, with the closure modifiers in (b). While the latter sentence is trivially specific, statements of the former kind are

vague in virtue of the looseness of their modifiers, and despite the sharpness of their adjectives ('single' and 'married' as categories of marital status have complete definitions and fixed criteria for membership) (Lasersohn, 1999). (a) and (c) instantiate vague quantification in a different way to that in (d). The latter instantiates temporal vagueness in a way that leaves quantity (in this case, a quantity of days and weeks) open.

Frequency adverbs function similarly, but as indicators of temporal or quantificational norms. They are used to specify (or loosen) the scope of a state of affairs, and/or the contexts in which a property holds:

a) {usually, sometimes} {even, odd} → Our chicken usually lays an even number of eggs on Mondays.
b) {always, never} {even, odd} → Multiples of five are always odd. *(false, but not vague)*
c) {generally, typically} {weekends, weekdays} → I typically work on weekends.
d) {mostly, rarely} {vegan, pescetarian} → Fast-food is rarely vegan.

In (a), while certain elements of the sentence are fixed (including a temporal aspect, i.e., *Mondays*), and *even* is sharply defined, a broader temporal vagueness is generated in using *usually* as a frequency modifier. Contrast this with a sentence of kind (b). Here, the modifier *always* closes the sentence to frequency vagueness. The sentence communicates false information, but its truth conditions are fixed, well-defined, and verifiable.

3.1.6. Behaviour under Ellipsis; Resistance to Disambiguation

Discussions of comparison classes and borderline cases may trigger further questions as to the relationship between descriptive and categorical statements. In the context of vagueness, we might wonder why adjectival predicates are so Soritical, yet the same is not said for nouns in general – do they not themselves rely on comparison classes for meaning? The answer is that the problem of vague nouns is indeed a significant one. In addressing this, we have an opportunity to further differentiate vagueness and ambiguity.

Typical nouns are multidimensional and generally sharp (Kamp, 2011) (Kamp & Partee, 1995). This means that they are defined by similarity – that is, the similarity of the properties shared by the objects that instantiate them. 'Chair' or 'man' are examples of nouns

that have multiple instantiations defined by shared essential properties. And while the adjectival expressions used to describe these properties might be contentious or vague in some cases, the issue is not with the categories themselves (Hampton, 1997). While the existence of necessary properties is a contentious topic, that discussion is beyond the scope of the project. Consensus on the essential properties specific to certain categories rarely changes, and as such the nouns that refer to those categories tend to be fixed.

An exception of this is the case of vague nouns that give rise to borderline cases. Examples of these include 'heap' and 'mountain' – in both cases, the line that distinguishes positive and negative extensions of these nouns cannot be drawn, once again falling foul of Sorites. As covered in our legal analysis, ambiguous nouns can either be unspecific by virtue of an incomplete or obscured propositional scope, or unspecific due to their role as non-rigid designators. Either way, disambiguation may be applied to strengthen their referential integrity. An indicator of vague nouns, in contrast, is a resistance to disambiguation, particularly under the *ellipsis test* (Zwicky & Sadock, 1975). Consider two sentences:

a) Alex went to the bank, and John did too.
b) Thorpe Fell Top is a hill, Whernside is a mountain.

The noun 'bank' in the first sentence has only two possible readings – either Alex and John both went to the riverbank, or they both went to a financial bank. The disambiguation of the first conjunct entails the disambiguation of the second elliptical conjunct. In contrast, not only is disambiguation not applicable in the second sentence, but any line drawn between 'hill' and 'mountain' when clarifying the first conjunct will not necessarily hold for the second – the contested nouns generate borderline cases.

3.2. Taxonomy

We have now explored each indicator of vagueness in the list outlined at the beginning of the previous subsection. In summary, we have identified the following types of interrelated indicators for vagueness:

1. **Comparison class dependency** (adjectival predicates, borderline cases and fuzzy boundaries, comparison class-determining for-phrases, judgement-determining phrases)

2. **Tolerance-governed predicates** (absolute and non-absolute relative adjectives)
3. **Vague modifiers** (adjectival modifiers, frequency adverbs, quantifying expressions)
4. **Vague nouns** (resistant to disambiguation, Soritical nouns)

Taking stock of what we have learned, we can begin to formulate our taxonomy. While we might be tempted to use the indicator typology above as a direct basis for this, recall that the definition of a controlled natural language is one that only preserves relevant facets of the natural language from which it derives. Our aim is to implement a CNL with mechanisms that detect and translate to an underlying logic the features and structures of vague language. Our analysis of indicators was intentionally finely grained to reflect the complexity of natural language. However, this granulation may not be desirable from a design and implementation perspective. As such, a minimalistic taxonomy that unifies the central themes of the typology above will be provided here, to be used as the basis for our CNL's grammar and its implementation.

It is clear from our analysis that adjectives, either in predicate or modifier form, are the primary source of vagueness in natural language. However, we should also note that in the specific context of legal language, descriptive sentences do not play a central role. Legal language is concerned with deontology and contingencies. Descriptive language is used, but in service of the primary purpose of the contract. As such, our taxonomy should be designed around the role that adjectival predicates and modifiers play in legal contracts specifically, rather than around the nuances of its place in natural language.

Deontic logic is favoured for law specific CNLs partially because of its capacity for modal expressions. That is, it allows for the language of possibility, contingency, and time. For our CNL to meet its requirements, it too must have capacity for this. Further, it should reflect how vague language functions within those modal aspects. Vagueness as it relates to temporal expressions is particularly relevant in the context of law. The time periods in which obligations, rights and inhibitions hold are often central to a contract, and potential sources of contention. Controversy over the details of financial and logistical obligations are similarly common. The common root is often the presence of vagueness in a quantifying expression, like the kind explored in our analysis subsection.

Finally, vagueness in legal language may manifest as inexactness about the nature or degree of a relation between two subjects, entities, or timeframes. We saw this in our first case study in chapter 2, where repeated use of loose quantifiers about time ('between *x* and *y*', '*within x* days' etc.) led to controversy around obligations and fulfilment (section 2.5). This

form of vagueness has no natural analogue in our indicator typology. It might be defined as a hybrid between quantification and deontic, where an expression of the latter is loosened by the vague modifiers of the former kind.

Considering these observations, the following taxonomy will be used:

1. **Degree Vagueness** (adjectival predicates/modifiers, comparison class dependencies, tolerance-governed predicates, vague nouns)
2. **Quantification Vagueness** (generalizing determiners, loose quantifiers, imprecise prepositions and adverbs, frequency modifiers)
3. **Relational Vagueness** (relational prepositions and adverbs, relational quantifiers)

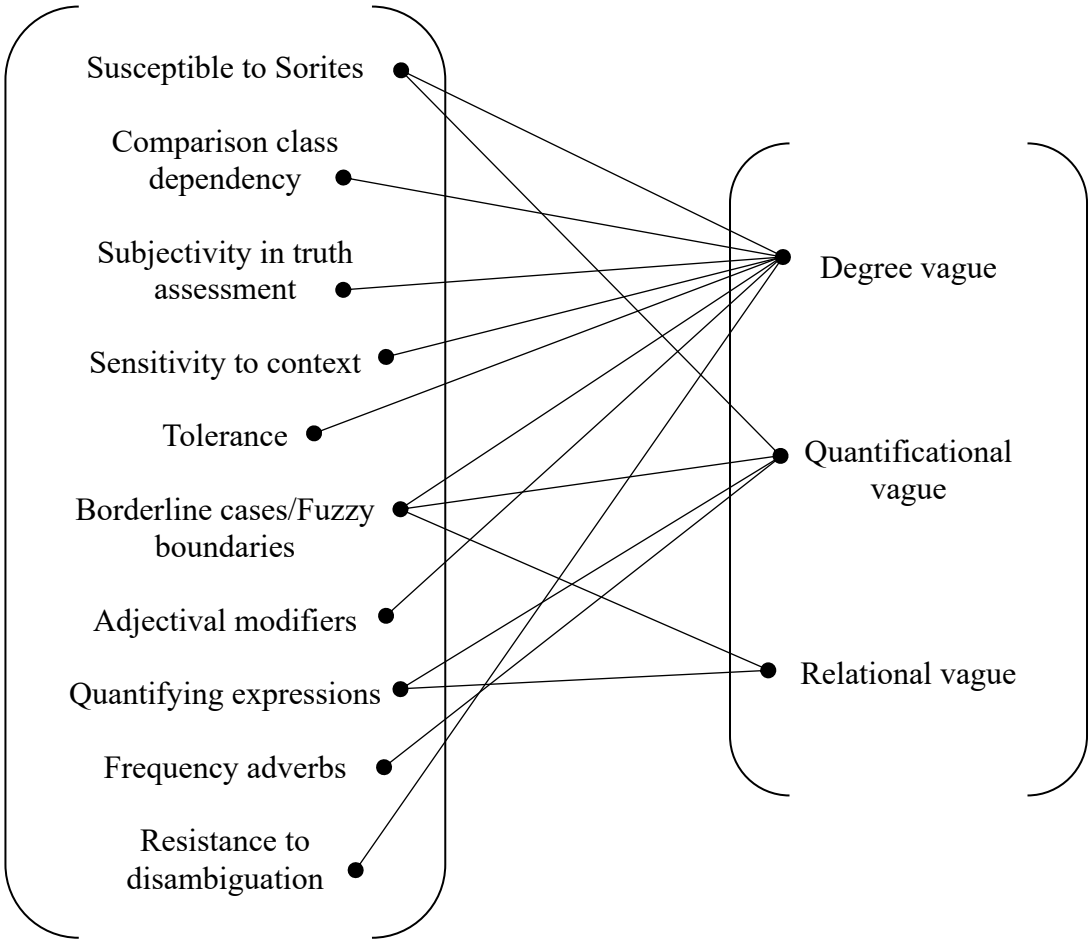
Degree vague statements are those that rely upon a comparison class for meaning and referential value. They utilise adjectives and adjectival modifiers (particularly adverbs and qualifiers). Examples include "Steve is a moderately greedy man", "you're extremely late". Soritical predicates instantiate degree vagueness.

Quantification vague statements are those that whose composite expressions have been modified or qualified to obscure their referent(s), or to loosely restrict the domain application of one or more of its composite predicates. This can apply to both numeric and temporal expressions. Examples include "I'll deliver your parcel around 7 o'clock", "I like some flavours of ice cream". Quantification vague statements are often those that express approximation ('almost'), that use generalizing determiners ('some'), etc. Generally, the truth conditions of a quantificational vague proposition are precise but are obscured by use of loosening modifiers and determiners. Such expressions are often open to *precisification* (e.g., 'by some flavours of ice cream, I meant chocolate and vanilla').

Relational vague statements are statements that define a relation between two expressions or terms, but through use of inexact modifiers leave either the nature or degree of that relation open to varying interpretation. Relational vagueness is highly related to quantificational vagueness but expresses an imprecise relationship between variables, rather than direct quantification on a variable or domain restriction of a predicate. A relational vague expression is one that draws two objects, concepts, or entities together into some relation (e.g., 'owes money'), providing upper and lower bounds for the relation to hold and/or for its conditions to be met, but lacking details on its interim points. Examples include 'You owe me between $200 and $400', 'At best, I see you as an acquaintance. At worst, you're my enemy', 'My dog is something between a Labrador and a Weimaraner', 'I'll send you a minimum of

200 chickens in 2014, and a maximum of 500 chickens between now and 2016' (drawing together the vague quantifiers *minimum/maximum/in* and the loose relational modifier *between*). The truth conditions of such propositions are generally fixed, but are often hard to assess, requiring further specification or context.

Figure 1: Indicators → Taxonomy
(mapping diagram)



In natural language these broad types necessarily interact with one another. A relational statement might loosely bind together two sub-statements whose referents have been made vague through quantificational or adjectival modifiers. As such, a highly modular design will be used in the design and implementation of our CNL grammar. Before we turn to methodology, however, a final aspect of our analysis remains.

3.3. Modelling and Applicative Challenges

We might reasonably assume that the abundance of literature explored in this section and in the previous one would provide a clear direction for approaching the implementation of our controlled natural language, and for implementing mechanisms to measure vague content. However, this is not the case. Various obstacles exist for the application of formal models of vagueness to a law-specific controlled natural language. Because they have impacted both the design and implementation process of the project, it is worth expounding the central challenges encountered here. However, this discussion will be limited – this is because many of the issues here go beyond our scope. As such, we will only highlight relevant issues, and will limit technical discussion of the models touched upon for the sake of brevity.

An overarching issue with scholarship on vagueness is its orientation towards metaphysical and epistemological questions. This is the case both in legal scholarship and in linguistics. Legal literature on the topic of vagueness often treats it as a problem to be solved. In identifying the problem, legal scholars often lean on philosophical literature on vagueness. It is here that for the most part, law-specific discourse is replaced by discussions on metaethics, ontology and epistemology, and ceases to be directly relevant for our purposes. A similar problem arises in linguistics. The literature on vagueness in formal semantics is often at pains to emphasize its narrow scope. As Kamp and Sassoon put it, "Efforts to solve the Sorites paradox have uncovered a range of important connections between vagueness and other aspects of language and thought. Linguists traditionally leave it to the philosophers to deal with the Sorites and put their own efforts into dealing with other correlates of vagueness in natural language and their consequence for grammar." (Kamp & Sassoon, 2016). However, throughout the literature, this boundary is constantly pushed up against by linguists. For instance, attempts at modelling the tripartite division of a vague predicate's application domain have been plagued with old philosophical questions as to the relationship between sense and reference. Discussion on the fuzziness of boundaries within application domains quickly turn to the questions of second, third, and higher-order forms of vagueness, and of infinite regression (Williamson, 1994) (Sorenson, 2001) (Wright, 2010). Again, to quote Kamp and Sassoon, "Since our primary focus is on the semantics of vagueness as it manifests itself in the ways we speak, and since there is no room to discuss all aspects of vagueness, we have decided not to include higher order vagueness." (Kamp and Sassoon, 2016). The result is that much of the literature on formal

modelling of vague language relies upon reference to other disciplines (and therefore potentially different models and systems of logic) to fill in its gaps.

This poses issues for our project. We are not only interested in understanding vague language as a linguistic phenomenon. We are designing a controlled natural language, intended to model vague language microcosmically, for a very specific domain. Lack of consensus as to how vague language and its paradoxes should be formalized is problematic for our purposes. Philosophical literature on the topic goes some way towards clarifying this issue, but also highlights a deeper conflict at the centre of the project: between deontic language as a means of describing *objective* states-of-affairs, and of vague language as rooted in *subjective* agency and semantic judgement. The conflict between rules-based models of language and models of language derived from natural language processing and semantic webs is also relevant, but pertains more to practical challenges of implementation. As such, detailed discussion of this is delayed to chapter 4, where it is examined in the contexts of both the CNL and of the auxiliary code that implements our vagueness quantification mechanisms.

An example of a prominent and complete approach to modelling vague language, and one that retains the principle of bivalence, is that of *supervaluationism* (Lewis, 1999) (Fine, 1975) (Keefe, 2008). Supervaluationists hold that if a statement instantiates a valid schema of classical logic, that statement is valid regardless of the verifiability of its components' truth conditions. On a supervaluationist model, the *supertruth* of a proposition $P(a) \vee \neg P(a)$ is secured regardless of whether *P* is a vague predicate, and whether *a* is a member of the truth-value gap extension of *P*'s domain. Indeed, on a supervaluationist model, the question of *P(a)*'s truth value is non-existent – alone it is neither true nor false, but rather inherits its truth value from the schema in which it is a component. Various mechanisms might be proposed for expressing relations between vague components – e.g., through the introduction of a new operators between predicates that denote the intersection of their truth-gap extensions, or, alternately, that order vague propositions based on the degree of contextual information that support or negate them.

From the point of view of designing a CNL, supervaluationism would seem useful as a basis for avoiding paradox. We could argue that, on adopting a supervaluationist approach, we would have justification for only preserving the law of classical logic into our underlying logic. We could argue that, given this, there is no further question as to whether we should restrict the Sorites Paradox at the level of syntax or through its omission in our underlying logic. And while the technical and time restraints of the project do impel us to adopt this approach (or some auxiliary form of it), there is a strong argument to be made that reduction to the rules of classical

logic obscures the unique semantic content and mechanisms of vague language, and whether this is desirable in the context of a controlled natural language.

The alternative is to adopt a form of many-valued logic that sacrifices the principle of bivalence in favour of preserving vagueness as a semantic property. An example of such a logic is *paraconsistent* logic (Priest, 1979). Put briefly, on a paraconsistent reading of $P(a) \vee \neg P(a)$, if *a* is a borderline case of *P*, then *P(a)* is both true and false. The set of possible truth values in paraconsistent logic is {1, (1, 0), 0}, where 1 denotes truth, 0 denotes falsehood, and (1, 0) denotes joint truth and falsity. Where *a* is a borderline case of *P*, *P(a)* returns a truth value of (1, 0). This of course necessitates new systems of inference and contradiction validation, the translation of which into a controlled natural language should not be limited to the legal domain - and so are beyond our scope.

All this is to illustrate the contention that plagues models of vagueness, and the problems that this generates in our project. Because contract language is by nature deontological, and deontic logic is a form of philosophical logic that preserves the classical laws of inference, in designing our CNL we might naturally favour a model of vagueness that does not throw out bivalence. The trade-off is that we are sacrificing unique properties of vague language. To some degree, this is desirable – the Soritical paradox and related problems of vagueness have no role to play here and so should not be modelled. However, restrictions on vague structures have other consequences for our CNL. For instance, asymmetric and symmetric forms of inference generated by tolerance-governed predicates cannot be modelled in a binary truth value system. In the reduction of vague language to the laws of inference, connectives, and operators of classical logic (e.g., through the translation of our CNL to Prolog), the subjective underpinnings of comparison class dependency and tolerance-governed predicates would be lost – classical logic has 'truth' as its only semantic content.

In computational models of argumentation and reasoning (similarly rooted in subjective language), rules of inference are implemented through the introduction of new operators that denote the relationship between speakers, audiences, their frames of belief, and a set of pre-established knowledge/facts (Hunter, 2022). We might be tempted to pursue a similar approach to resolve the apparent conflict between classical logic and vague speech. But ultimately, this would be misplaced in the context of a legal CNL (at least, one of this scope). Legal language is concerned with the language of prescription, lacking speakers and listeners in the traditional sense. It is interpreted, and in that sense might be said to have 'listeners', but its content is not determined by the subjectivity and empirical judgements of the lawyer(s) who drafted it.

Further, a contract exists to construct a restricted universe consisting of various states-of-affairs, obligations, parties, and contingencies. Outside of data driven contracts, which may automatically check propositions about some stored set of information (stock prices, GPS locations, bank balances, etc.), contract language is not oriented towards the notion of truth conditions and their validation or falsification – such concerns are *a posteriori.* A contract's fundamental logical requirement is that it should not contain internal contradictions. It should not express any information about the subjectivity of its author. Contracts lack frames of belief, and instead are framed by the norms of the legal sector they pertain to (i.e., commercial law). These norms determine how contracts are interpreted, and so are in this sense subjective, but do not involve individual speaker agency in the traditional sense (i.e., intentionality, mental causality, etc). Subjectivity therefore is limited to the judgement of its interpreters; and the interpretation of the content of a contract is to a significant degree pre-determined by the norms of language usage, e.g., in the business sector (which aims towards some consensus on the extensions of commonly used adjectives such as 'early', 'late', 'best efforts'). Where those norms fail to settle controversy over interpretation is where vague language becomes significant (either through its intentional use in the drafting process, or as a source of conflict over meaning between parties after the fact).

The properties specific to contract language distinguish it from those of traditional speech, and how speech is typically analysed and formalised. On a semantic level, we have shown how vague language centres around the agent who uses it and the listener who judges its contents. It is oriented towards an external world that its propositions reference and upon whom its truth conditions rely for verification. This is the central conflict that must be reconciled when modelling both vague and deontic semantics.

The following chapter ('Methodology and Implementation') outlines the design and implementation of our CNL in light of this conflict. It outlines what mechanisms were used to strike a balance in fulfilling both the deontic and vague language requirements of the project. It also explores the advantages and drawbacks of our approach. It also offers some suggestions as to how the project might be expanded.

4. Methodology and Implementation

This chapter outlines the design and execution of our controlled natural language. Functionality for quantifying and 'scoring' the vague content of contracts drafted in the language is also expounded here.

We begin by outlining the Backus-Naur form of our CNL's grammar, and how the taxonomy introduced in the previous section informed its design. We identify the core aspects of the grammar and how they relate to the specific requirements of contract law. The design of the quantification mechanism is discussed in the context of similar algorithms. We provide an example of a simple contract written in our CNL.

A technology review is provided. We discuss Haskell and its Parsec library, our reasoning for using it, and the properties it locked into the project code. We discuss the benefits and drawbacks of the Parsec library as a tool for modelling vague and ambiguous language. The benefits, challenges, and restraints of our choice of implementation language are identified.

Key features of our code are then summarized, with commentary on how they implement the project requirements. The contract example from our exposition of the grammar is sent as input to our parser, and the parse tree it generates is outlined and explained.

4.1. Grammar

The design of our controlled natural language was informed by its primary requirements:

1. a capacity to capture common syntactic structures present in legal language,
2. a clear mechanism for expressing deontic statements,
3. a degree of flexibility to allow for non-specific and vague language to be used where necessary.

The first step of the design process was identifying the relationship between the different points of the above criteria. In real legal contracts, typically all three criterium are present as features of legal language and work in concert with one another to generate meaning. As such, any grammar that models legal language must be modular – that is, it must allow for the chaining of different statement types together into compound statements, without discarding the specific character of the composite parts. Where that character is vague, our controlled natural language

must reflect the different forms it might take – taken directly from the taxonomy outlined in the previous chapter.

Before we move onto an examination of the grammar itself, the question of nouns should be addressed. As previously established, ambiguity and vagueness are distinct linguistic phenomena, though often play similar roles in natural language. Due to technological restrictions (outlined in the next section), it is out of the scope of this project to resolve the problem of reference for ambiguous and vague nouns. Regardless, the issue has not been ignored. Indeed, the decision to preserve degree vagueness into the underlying logic, and the weightings that tokens of this type were given for quantification (section 4.2.) was taken as a response to the potential problems that imprecise nouns may generate in legal contracts (as discussed in section 2.4.). This is to say that while ambiguous nouns are not explicitly present in the grammar, they have directly informed the design process.

4.1.1. Introduction to the Grammar

First and foremost, we were concerned with providing reasonable accommodations for modal, deontic, and vague language (and, most importantly, expressions that combine all elements of all three). Note that although deontic language is a subset of modal language, we make a distinction here for the purpose of emphasizing differences in application. That is, by modal we mean tensed expressions (future verbs, etc) and expressions about contingency. Deontic expressions here specifically relate to obligations, permissions, and inhibitions.

The broader categories of the taxonomy are instantiated by particular statements, expressions, and predicates. Degree vagueness manifests in the form of descriptive expressions (‘The chicken is frozen’) and within broader deontic statements (‘Alex will deliver to John a gold bar’). Relational vagueness is represented through certain modifiers and determiners (‘John will pay Alex maximum $100 minimum $500’, ‘Between 2022 and 2024 Alex will deliver to John 500 geese’). The latter example instantiates form of temporal statement (the other form being that which manifests quantificational vagueness). Quantificational vagueness manifests in the form of approximation expressions (‘John refunded Alex around $70’) and minimal-maximum generality (‘Alex will deliver to John at least 10 televisions’).

We were also concerned with balancing expressivity with a sufficient level of uniformity so as not to over-complicate sentences of the CNL (undesirable on a usability level). Some restriction of expressive freedom is also obviously a necessity for parser implementation. As such, while a degree of freedom is given to the end-user when drafting contracts, structure

is enforced through pattern and keyword requirements. The following subsections outline the key design concepts in the grammar, while the subsection containing the abstract syntax (4.1.5) gives them explicit form and provides an example of a contract drafted in the grammar.

4.1.2. Statements

The fundamental building blocks of our grammar are statements, expressions, and predicates. This subsection will outline statements; subsections 4.1.3. and 4.1.4. will expound expressions and predicates. Statements are constructed either from other statements, from chained expressions (most of which are composites of other expressions), or from some combination of both. This format was chosen to better emulate the modularity of natural language.

Sentences of our controlled natural language fall under one of nine statement types, based on their structure. Statements are demarcated and connected by the period/full stop character. Note that, due to specific properties of our implementation language (see section 4.3/4), the full stop is a conjunction operator for statements, and so *must not* be used to *end* a set of statements. Here we will provide brief definitions of each statement type, and an example. The defining aspects of the statements are outlined in italic; see the abstract syntax (4.1.5) for all keywords and/or patterns that define each statement type.

1. **Deontic** statements are either an obligation, an inhibition, or a permission. E.g., "Alex *will pay* John $200".
2. **Conditional** statements instantiate the 'if-then' form. E.g., "*If* Alex paid John $200 *then* John delivered to Alex 100 eggs".
3. **Time** statements are prefixed with a specific date or with an expression pertaining to time. E.g., "*On* 1 January 2000 Alex paid John $200".
4. **Degree vague** statements are descriptive. E.g., "Alex *is very* litigious".
5. **Disjunction** statements express exclusive choice. E.g., "*Either* John delivers to Alex 100 eggs *or* John will refund Alex $200".
6. **Negation** statements outline a non-existent situation, event, or property. E.g., "*It is not the case that* John delivered to Alex 100 eggs".
7. **Conjunction** statements connect two affirmative statements. E.g., "*It is the case* that John refunded Alex $200 *and* it is not the case that Alex has permission to sue".

8. **Contingency** statements express a potential future event relevant to the contract's performance. This goes some way towards implementing force majeure when used in concert with conditional statements. E.g., "The warranty *will* expire".
9. A **Component** is a pair of statements connected by a period. E.g., "Alex will pay John $200. Either John delivers to Alex 100 eggs or John will refund Alex $200".

A *contract* is comprised of one or more statements. Where more than one statement is present, these will be connected as components. An object in our CNL must be comprised of language structures that map to statements – that is, incomplete statements or lone expressions outside of a statement structure are illegal under our grammar.

When designing our statements, where possible the aim was to keep their forms as naturalistic as possible. Given that controlled natural languages are programming languages with parsing requirements, this was not always possible – conjunction and negation, for instance, are fairly unwieldly ('it is the case', 'it is not the case'). Expanding the grammar for naturalism could form the basis for future work.

4.1.3. Expressions

Expressions can be atomic (such as a *noun* string) or comprised of other expressions (like *date*, combining *day/month/year* expressions). They can also contain *predicates*. An example of the latter is the expression type *vague quantifier*, which is comprised of a modifying predicate (e.g., 'around') and a quantifying expression representing the quantifier and the variable being quantified (e.g., '100 bicycles'). See the next subsection 4.1.4 for more on predicates.

Expressions represent a range of grammatical elements, types, and reference terms, all of which correspond to constituents in parse trees for objects of our CNL. Expression types range in complexity and are instantiated by different syntactic patterns and keywords. The decision to include elements that are not traditional 'expressions' (e.g., currency types, quantifiers) was made for implementation purposes (see section 4.4), and for the sake of modularity. As such, expressions under our grammar should be taken as (a) mechanisms for modelling certain semantic elements of natural language relevant to our project goals, (b) as a 'glue' that binds the constituent elements of a statement, and (c) as a means of systematically assigning token type to the constituents of a parse tree.

When designing the grammar, the decision was made to focus expressive functionality in deontic statements as it is there that potential miscommunication could arise. As such, while

the structure of *obligation* expressions is fixed as a binary relation between two parties, characterized by the deontic aspect and tense of that relation (i.e., "<noun> will deliver to/will refund/will pay <noun>"), the details of the object or quantity involved in the obligation are more open, through the availability of adjective insertion and quantification. An expression of obligation might therefore take various forms: e.g., "John delivered to Alex a broken clock", John will refund Alex at least $200", "John will deliver to Alex around 100 clocks", etc. Similarly, deontic statements can be nested in larger statements that add context to their deontic properties, e.g., through the use of predicates in a time statement "Between January and March John will deliver to Alex fruit", or through the use of composite conditional and degree vague statements: "If Alex is dissatisfied then Alex has permission to exchange".

Note that on our grammar, deontic verbs have a special status – that is, they are not tagged under the expression type *verb*, but under the particular deontic aspect they are expressing, and the tense that deontic aspect is expressed in (e.g., *future obligation*). See the complete syntax (4.1.5) for further information.

4.1.4. Predicates

During design, a decision had to be made as to whether to include non-vague modifiers, quantifiers, relational operators, etc., in the grammar. Ultimately, it was decided that strengthening or loosening adjectival modifiers ("very", "slightly", etc.) would be accounted for by the *adverb* expression type and implemented using pattern matching on statements. Non-vague quantifiers ('all', 'exactly', etc.) would be excluded. It was judged that the absence of a vague quantifier in a quantifying expression (e.g., "John delivered to Alex 100 clocks") entails specificity without the need for dedicated predicates.

The decision was made to limit predicates to those that instantiate vague language. Informed by our taxonomy, predicates are either quantificational or relational vague. It was also determined that the set of predicate words recognizable to the parser/lexer should be limited for scope and to limit excessive verbosity. Our choice of words was informed by the example legal text (outlined in chapter 2). The most frequently used terms were 'between' and 'within' – the former is present in our grammar as under *vague relation*; the latter is under *vague quantifier*. Other words ('in', 'after', 'before', 'around', 'at least', etc.) are also included in the grammar. Vague predicates are further differentiated by whether they pertain to temporal elements of a statement or to objects.

It should be noted that in the implementation, vague quantifiers fall under the `Pred` type, whereas vague relations fall under `Op`. Terms recognizable to the lexer/parser fall under `reserved` and `reservedOp`. This was required for reasons related to implementation strategy (see subsection 4.4 on infix and prefix operations) but does not impact the CNL's use and/or alter its syntax.

4.1.5. Syntax

The following syntax outlines how objects of our CNL may be constructed. Syntactical elements are contained within chevrons (< >). A valid statement "Between 2022 and 2023 John will pay Alex around $200" instantiates the syntactical form <temporal-range> <noun> <future-payment> <object-quantifier> <usd>. The presence of a question mark (?) immediately following an element indicate that the element is optional. Square brackets ([ ]) denote a list. Wherever an aspect of the syntax implements vague language, it is bolded.

Abstract syntax:

<contract>	::=	[<statement>]
	|	<statement>
<statement>	::=	<component>
	|	<conditional>
	|	<time>
	|	**<degree-vague>**
	|	<disjunction>
	|	<negation>
	|	<conjunction>
	|	<contingency>
	|	<deontic>
<conditional>	::=	'if' <statement> 'then' <statement>
<time>	::=	**<temporal-range>** <statement>
	|	**<temporal-quantifier>** <month> <statement>
	|	**<temporal-quantifier>** <year> <statement>
	|	'on' <date> <statement>
<degree-vague>	::=	**<description>**
<disjunction>	::=	'either' <statement> 'or' <statement>
<contingency>	::=	<future-event>

<disjunction> ::= ‘either’ <statement> ‘or’ <statement>

<deontic> ::= <obligation>
| <inhibition>
| <permission>

<expression> ::= <noun>
| <verb>
| **<adjective>**
| **<adverb>**
| <num>
| <usd>
| <gbp>
| <future>
| <past>
| **<vague-relation>**
| **<vague-quantifier>**
| <day>
| <month>
| <year>
| <date>
| **<description>**
| <inhibition>
| <permission>
| <obligation>
| **<quantity>**

<vague-quantifier> := **<temporal-quantifier>**
| **<object-quantifier>**

<temporal-quantifier> ::= ‘within’
| ‘after’
| ‘before’
| ‘in’

<object-quantifier> ::= ‘approximately’
| ‘around’
| ‘at least’
| ‘at most’

<vague-relation> ::= **<temporal-range>**
| **<object-range>**

<temporal-range> ::= ‘between’ <month> ‘and’ <month>
| ‘between’ <year> ‘and’ <year>

<object-range> ::= ‘between’ <gbp> ‘and’ <gbp>
| ‘between’ <usd> ‘and’ <usd>
| ‘between’ <num> ‘and’ <num>

<inhibition> ::= <noun> ‘may not’ <verb>

<permission> ::= <noun> ‘has permission to’ <verb>

<obligation> ::= <future-payment> **<object-quantifier>?** <usd>
| <future-payment> **<object-quantifier>?** <gbp>
| <future-payment> **<object-quantifier>?** <num>
| <future-delivery> <noun>
| <future-delivery> **<quantity>**
| <future-delivery> ‘a’ **<adjective>?** <noun>
| <future-delivery> ‘the’ **<adjective>?** <noun>
| <future-refund> **<object-quantifier>?** <usd>
| <future-refund> **<object-quantifier>?** <gbp>
| <future-refund> **<object-quantifier>?** <num>
| <past-payment> **<object-quantifier>?** <usd>
| <past-payment> **<object-quantifier>?** <num>
| <past-delivery> **<object-quantifier>?** <num>? <noun>
| <past-delivery> ‘a’ **<adjective>?** <noun>
| <past-delivery> ‘the’ **<adjective>?** <noun>
| <past-refund> **<object-quantifier>?** <usd>
| <past-refund> **<object-quantifier>?** <gbp>
| <past-refund> **<object-quantifier>?** <num>

<future> ::= <future-payment>
| <future-delivery>
| <future-refund>
| <future-event>

<future-event> ::= ‘the’ <noun> ‘will’ <verb>
| ‘the’ <noun> <verb>
| <noun> <verb>

<future-payment> ::= <noun> ‘will pay’ <noun>

<future-delivery> ::= <noun> ‘will deliver to’ <noun>
| <noun> ‘delivers’ <noun>

<quantity> ::= **<object-quantifier>** <num> <noun>

<future-refund> ::= <noun> ‘will refund’ <noun>
| <noun> ‘must refund’ <noun>

<past> ::= <past-payment>
| <past-delivery>
| <past-refund>

<past-payment> ::= <noun> ‘paid’ <noun>

<past-delivery> ::= <noun> ‘delivered to’ <noun>

<past-refund>	::=	<noun> ‘refunded’ <noun>
<description>	:=	‘the’ <noun> ‘is’ **<adverb>?** <adjective>
	|	<noun> ‘is’ <adverb>? <adjective>
<date>	::=	<day> <month> <year>
<day>	::=	<integer>
<num>	::=	<integer>
<month>	::=	<string>
<year>	::=	<integer>
<string>	::=	<char>
	|	<char> <string>
<char>	::=	‘A’ | … | ‘Z’
	|	‘a’ | … | ‘z’
<identifier>	:=	<string>
<integer>	::=	0 | … | 9
<noun>	::=	<string>
<usd>	::=	‘$’ <integer>
<gbp>	::=	‘£’ <integer>
<verb>	::=	<string>
<adjective>	::=	<string>
<adverb>	::=	<string>

An example of a simple contract comprised of valid statements in the CNL grammar is provided (figure 2). In section 4.4., this example is given as input to our parser, and a parse tree is generated from it.

Figure 2: An example contract in our CNL

> “On 23 June 2023 Alex paid John $3000. Within 2024 John will deliver to Alex a computer. If delivery is late then John will pay Alex around $100. Between 2024 and 2026 the warranty is valid. If the computer breaks then before 2027 Alex has permission to complain. If Alex will complain then either John delivers Alex a new computer or John will refund Alex minimum $2500. In 2027 the warranty will expire. If the warranty is expired then Alex may not complain”

The grammar outlines the syntax of the controlled natural language, and an abstract look at the rules the parsing functions follow in our implementation. In the implementation, token types follow those of the grammar. The nested/branching structure of the grammar was chosen for the sake of generating more detailed AST constituents (and so representing more semantic content), and for technical reasons - to take advantage of Parsec's reserved names and operator functionality, the latter of which was used to implement predicates and relational operations (e.g., deontic obligations between two parties). Upon quantification, we return to the overarching taxonomy to measure overall instances of vagueness.

4.2. Quantification

4.2.1. Challenges

A primary requirement is the development of an auxiliary mechanism for measuring vague language (in our case, via a post-hoc analysis of a parse tree). The implementation of this posed a dilemma. According to the OED, the English language contains at least 450,000 words (Oxford English Dictionary, n.d.), a significant proportion of which might be categorized as degree vague (descriptive) under our taxonomy. Such words are highly sensitive to the more social aspects of language (context, definitional consensus, subjective judgement, etc.), posing additional problems for parsing. Our grammar is designed not around the detection of finely grained grammatical properties, but around types of vague expressions, modifiers, and syntactic indicators. Writing functions to correctly parse and sort all 170,000 words into categories would be both an insurmountable challenge and completely nonsensical. As such, we faced a choice. We could restrict parsing so tightly as to only allow use of a limited set of words and expressions commonly used in contract law (e.g., 'late', 'on time', 'on or before', 'in the range of', 'reasonably priced', etc.) that had been pre-tagged as instances of vagueness types. This would dramatically reduce the expressive capacity of our CNL, but it would avoid the problem of having to find a way of parsing and categorizing unexpected language input. The other choice would be to design some functionality to draw from a semantic net database (e.g., WordNet), match each term encountered during parsing with its analogue in the database, flag certain properties or usages as vague, then sort on this basis.

4.2.2. Solution

Ultimately the decision was made that the use of databases like WordNet would exceed the scope of the project's timeframe and resources. As such, the decision was made that a looser form of parsing restriction would be implemented, where pattern matching based on some keywords would be used to identify and categorize structures of vague language. This choice impacted the design of our measurement mechanism. Instead of focusing on quantifying the 'raw' natural language input, our function would instead measure the output of the parsing function. One of the few pre-existing algorithms for measuring vague language uses the database approach. It is worth touching on here as it directly influenced the design of our scoring mechanism. VAGO (Mondeca, Icard, & Claveau, 2023) is an algorithm that draws from a lexical database. VAGO's database is also partitioned by a taxonomy of vagueness, but one that significantly differs from our own, having been directly lifted from Alston's typology of vague expressions (Alston, 1964). In VAGO, a score for vagueness and subjectivity is defined as the ratio of vague words to number of sentences in a given text (Airbus France, Guélorget, & Icard, 2021).

This form of measurement is adapted in our mechanism, but with two fundamental changes (beyond the difference in taxonomies used). First, we do not measure words to sentences, but rather vague tokens to total number of components. Second, while VAGO returns a total score based on its encounter rate for vague words, our algorithm returns a vagueness score, based upon the weighted sums of each token type, in range [0,1]. It does also return a ratio, but not of words-to-sentences but of vague tokens to total tokens. The weightings (0.25, 0.5, and 0.75) were chosen for the sake of consistent balance/distribution, where more 'significant' vague language lies higher on a scale out of 1 and so adds more to an overall vagueness score. The decision not to weigh each type equally was made to reflect their differing levels of impact in the context of law. This of course entails a degree of subjectivity. However, our weightings were not chosen arbitrarily. In our legal analysis, discussions of vagueness often manifested as controversy as to at what point in a contract or law an obligation kicks in. Similarly, the use of loose language (e.g., in the use of approximation, generalized determiners, etc.) was demonstrated in our judgement ruling example (section 2.5). On our taxonomy, such language falls under the category of quantificational vagueness. As such, tokens of this type are given a weight of 0.5.

The same logic determines the weighting of relational vague tokens. These are given a weight of 0.75. This is because, functionally, they generate similar problems for interpretation

as quantificational vagueness, but to an even greater extent. While quantifying expressions such as 'around $200' are loose, the tolerance range of $200 has reasonable limits, which can be invoked in the process of legal interpretation. In contrast, relational vague statements such as 'between $1000 and $2000' have a much wider tolerance range attached to them. In the context of deontic language, imprecision of this kind can be dangerous. As such, relational tokens are given a heavier weighting to reflect the fact that upon the norms of legal language, this form of vagueness should be avoided wherever possible.

Finally, degree vague tokens are given a weighting of 0.25. It may initially seem surprising that the category entailing adjectival predicates and modifiers is given the lowest weighting, given the central role they occupy in vagueness analyses. Indeed, discussion on the Sorites Paradox generally concerns descriptive language. The reasoning for this weighting concerns ambiguity and non-rigid designators. While the legal theory on vagueness often does centre on Soritical predicates, this is generally discussed within the context of judicial interpretation (e.g., in our Bad Samaritan thought experiment). Our case studies in chapter 2 suggest that commercial litigation may be more likely to come as the result of referential ambiguity than from the use of tolerance-governed predicates. As we have established, the decision not to use a semantic database restricts the breadth of our parser code. It means that we cannot *a priori* identify the grammatical character of a string – the flagging of nouns, verbs, etc., must be done through contextual pattern matching. This means that we have no reference point for distinguishing different kinds of nouns. As such, the ability to differentiate rigid and non-rigid designators is beyond the scope of our project code, as is the ability to differentiate between a vague noun ('heap') and an ambiguous one ('chicken'). In the context of commercial law, ambiguity over whether a contract calls for frozen chicken or live poultry is more pressing an issue than the question of a heap's fuzzy boundaries. Paradoxically, degree vague language allows for precisification when it comes to ambiguous nouns (e.g., in prefacing 'chicken' with 'live' or 'frozen'). The decision was made then to use adjectives and adverbs as a means to lessen the problem of ambiguity, or at the very least to offer the end user the option to do so. The non-zero weighting reflects the fact that degree vague language is not innocuous – it still contributes to the overall vagueness score, compounding with frequent use.

4.2.3. Formulae

The implementation code for our scoring system is provided in section 4.5. The following formulae formalise the process.

Before the final score is calculated, the three categories of vagueness are quantified via an analysis of the parse tree for a contract denoted by ϕ. For every token that instantiates a particular form of vagueness, a variable x is added to the set *X*. Each *x* is either 0.25, 0.5, or 0.75, depending on the category of vagueness being measured for. *X* sets contain the total number *n* of *x* values for each category. The contents of *X* are then summed to get a total *score* for each vagueness category. This is used to calculate the total vagueness score for the contract. The cardinalities/*card* of each *X* set are used to calculate the total ratio of vague tokens to total tokens in a contract, giving information on the frequency of vague language relative to the overall contract.

(Degree vague)

$$x = 0.25$$
$$X = \{x_1, \dots, x_n\}$$
$$D_{score_\phi} = \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$$
$$D_{card_\phi} = |X| = n$$

(Relational vague)

$$x = 0.75$$
$$X = \{x_1, \dots, x_n\}$$
$$R_{score_\phi} = \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$$
$$R_{card_\phi} = |X| = n$$

(Quantificational vague)

$$x = 0.5$$
$$X = \{x_1, \dots, x_n\}$$
$$Q_{score_\phi} = \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n$$
$$Q_{card_\phi} = |X| = n$$

When all categories have been quantified, their scores are summed and then divided by 10 to obtain a total vagueness score. A higher score indicates a broader degree of vagueness in the contract being analysed. Division by 10 was chosen to keep scores small and readable/meaningful to the user. Scores may be demarcated into zones. A score below 1 indicates non-extensive use of vague language. As vague language increases beyond that point, each new integer passed indicates entry into a new zone.

The scoring function is best used in concert with the frequency (or ratio) function where a contract is very short. This is because in shorter contracts, a lower vagueness score may disguise a situation in which vague language is used in limited quantity yet still takes up the lion's share of the contract's overall contents.

(Total score)

$$R_{score}(\phi) = \frac{D_{score_\phi} + R_{score_\phi} + Q_{score_\phi}}{10}$$

(Ratio)

$$R_{ratio}(\phi) = \frac{D_{card_\phi} + D_{card_\phi} + D_{card_\phi}}{N_\phi}$$

where ϕ is a contract and N_ϕ is the total number of contract components.

4.3. Technologies, Challenges and Considerations

4.1.3. Haskell and Parsec

Haskell was chosen for the implementation language underlying our CNL. It was chosen for two primary reasons: for its status as a functional language, and for access to its Parsec lexical analysis and parsing library.

The English language is highly modular. Its sentences are dense sets of interrelated lexical and syntactic units. Being derived from English, our controlled natural language is similarly compositional, albeit far less dense. The emphasis on vague language units and their

impact on the sentences in which they appear reinforces the 'participatory' nature of our CNL's cascading component-statement-expression-predicate structure. The unique syntactical types introduced in our grammar should be preserved through parser implementation. Because of this, the ability to define new algebraic data types (ADTs) for use in parsing was a top requirement. Although unique ADTs can be defined in some object-oriented languages, they are a fundamental feature of functional languages. Strong functionality for pattern-matching was also considered desirable in aligning design with implementation. Because Haskell is a functional language, the use of currying and partial application is a central feature, making it intuitively well-suited for our purposes.

While parsing libraries exist for many languages, Haskell gives access to the Parsec library (Leijen, Martini, & Latter, 2006). Parsec provides functionality for writing higher-order parser combinators (i.e., parsers that take smaller and more specific parsers as input). This ideal for parsing a controlled natural language in which different sets of grammatical rules interact to govern how a larger input string is interpreted. This is achieved through use of monadic structures. Parsec provides functionality for both lexical analysis and parsing, precluding the need to write separate functions for each. It provides solid functionality for returning error information upon parse failure, which is useful during design and debugging. While Parsec has been adapted to other languages beyond Haskell, there was no incentive to adopt a reimplementation. The use of Haskell and Parsec necessitates some additional considerations (particularly around vanilla Haskell, Parsec, and user-defined algebraic type compatibility) that other languages do not. However, this is necessary for access to their unique features – e.g., monads, functors and `fmap` methods, the `do` syntactic sugar, and combinators - all of which played a central role in the development process.

4.1.4. Lexing in Parsec

The decision was made to use Parsec's pre-built lexers in concert with custom parsing functions and lexer constructors, using the former in a limited capacity for the purposes of identifying whitespace, strings, numbers, and currency symbols. This means that token types identified through lexing are intermediate data, attached to specific ADTs (nouns, adjectives, etc.) by way of broader parsing and pattern-matching functions. As such, pre-built tokens are not represented in the abstract syntax tree output. While the use of Haskell and its Parsec library provides distinct benefits and functionality, it also introduces and 'locks in' features. While

most of these features were either innocuous or beneficial, some created challenges in aligning the project code with the controlled natural language's intended design and usage.

Parsec's built-in lexing functions identify tokens based on predetermined criteria, checking strings for whitespace, symbols, etc. They also do the work of extracting identifiers from 'noise' (parentheses, quotation marks, etc). These functions are accessible as part of the `Text.Parsec.Token` language building module. Utilizing this module brings with it the advantage of not having to write separate parsers for parsing standard lexical features (whitespace, common punctuation marks, etc.), freeing up development time for parsers that enforce rules specific to the structures of the controlled natural language. The module allows the programmer to define a language according to a pre-determined set of token constructors, then instantiate token parsers based on this definition. Token constructors include `commentStart` (escape characters, e.g., `#` in Python), `caseSensitive` (a Boolean value), `parens` (used to parse the value within parentheses), `identifier` (returns identifiers based on the rule(s) outlined in the language definition), etc. Note that the module allows the user to import pre-defined language definitions as templates for their own definition (e.g., `haskellDef` for the Haskell language, `javaStyle` for Java-like languages, etc.). In our case, `emptyDef` was imported. It contains the minimal pre-determined token definition and was therefore most suitable for tokenizing input based on our context-specific controlled natural language. There is strong functionality for customization (e.g., the ability to define the Parsec type of the first character in identifiers).

A drawback to using this method for lexing is that we are constrained to working within the parameters of the pre-determined token constructors. As is clear from a cursory look at the importable language definitions and the token constructors themselves, the module is oriented towards parsing traditional programming languages. During development, this was mainly felt when using the `reservedOp` (operations) and `reserved` (names) constructors. In a traditional programming language, the demarcation between the two constructors would be straightforward, with the former holding the set of all operators used in the language ('&&', '=', etc.), and the latter used to differentiate identifier strings from keywords ('do', 'in', 'for', etc). However, given that the generic end-user of our CNL is a legal professional with little to no programming experience, more verbose reserved string patterns were necessary. While it would be possible to ignore the `reservedOp` constructor and focus entirely on populating `reserved`'s definition with key words and phrases, the use of Parsec's infix and prefix operator parse rules was desirable for the representation of binary and unary relations (particularly

deontic obligations and quantificational vague modifiers). As such, key words and phrases were divided between `reserved` and `reservedOp` based on semantic properties/roles in the CNL.

4.1.5. Formal Verification

A broader question was that of formal verification according to the laws of classical logic. Because a database was not used, lexical types are identified using pattern matching on string input. When a contract is sent as string input to the contract parser (`parseContract`), lower parsers and pattern matching helper functions identify legal values. The identification of legal values in turn determines which type constructors should be called to generate a parse tree that reflects the contract's structure and content. This is based on what keywords are encountered when parsing the contract's *statements* (i.e., the contract string's sub-strings, demarcated by the period symbol). Lexical component types (such as 'noun', 'currency' and 'verb') have as their legal value free variable strings that are 'carried over' from the contract input. The end-user has freedom to enter any string or character for use as this legal value, and it is preserved in the parse tree (a necessity for specifying the details of a contract).

This causes some issues for implementing correctness checking. Because Parsec's error catching mechanisms are designed to catch syntax errors based upon a set of rules determined by the programmer, it is highly useful when restricting legal input to ensure the correctness of parse tree output. It means that keywords can be used to identify how statements and sub-statements fit together (i.e., conditionals, conjunctions, etc.). And while logical validation can be implemented at the point of parsing, this is most efficient when:

1. the string being parsed is subject to highly restricted syntax rules,
2. the input string is parsed holistically (not split into separate and self-contained sub-strings governed by different parse rules determined by their content, as is the case with statements in our language)
3. the length of the string is limited, meaning less use of `lookahead` (or, alternately, less 'rollback' of consumed input when an error is encountered further on when using `try`)

Consider a basic language with a small set of keywords to denote basic operators $\{\wedge,\vee,\neg,\rightarrow,\leftrightarrow\}$ and with a limited alphabet $\{a,b,c,d\}$ available for naming variables. A limited set of well-formed sentences in that language are possible, and so a limited set of parser rules are necessary to catch contradiction, tautology, etc. Even on such a language, as propositions

become nested and input grows in length, input may be consumed successfully earlier in parsing that becomes unsound in the broader structure, e.g., $((a \wedge b) \leftrightarrow (\neg a))$. Catching contradictions on longer and more complex input requires more parse rules to be specified, and frequent rollbacks pose the risk of increased computational cost. These issues would only be compounded in parsing our CNL, with its multitude of statement structures and keywords (i.e., the strings that constitute our reserved names and operators). For these reasons, we did not attempt to use Parsec to validate the logical structure of the contracts it processes. Rather, it was used to enforce the rules of the controlled natural language and to generate parse trees from objects written in it.

It must be noted that Parsec *can* be used to parse the results of tokenizers. The library has its own implementation of `prim/primops`: the `Text.Parsec.Prim` module, which can be used to implement custom `satisfy` methods. These methods can in turn be supplied with custom functions that test input for some user-determined criterion and output a Bool. This is a possible method for testing a parse tree for logical validity. However, given the timeframe and primary requirements of the project, the generation of multiple parse trees (and associated Boolean functions) was beyond our current scope. Ultimately, the requirements of the project were (a) the design of a controlled natural language for the purpose of expressing vague structures in a deontic context, (b) program code for parsing objects written in this language, for identification of their vague structures, and for generating parse trees that are correct by the laws of the CNL's grammar/syntax, and (c) code that gives a vagueness 'rating'.

The scope of the project was centred on the expression and identification of vague and deontic statements, not on proofs of their conjectures, nor on the assessment of their logical soundness and/or validity. The aim was a controlled natural language for expressing particular structures of language, not a modelling language for representing forms of knowledge. Further, as we have shown in previous sections, the paradoxes specific to vague and deontic structures are highly tied up with semantic content and, in the case of the former, are not necessarily resolvable within the laws of classical deduction. As such, the focus when developing our project code was not on mechanisms for the *a priori* identification of contradictions and tautologies, nor on the resolution of paradoxes, but rather on ensuring and preserving expressive capabilities. If this functionality is not implemented through the restriction of syntax at the level of parsing – which it was not here, for the reasons outlined above - it must be realised through other dedicated mechanisms. This could form the basis for future work.

4.4. Implementation

We begin with a brief overview of algebraic types before moving on to an example of how combinatorial parsers are used to generate correct parse trees for our controlled natural language. We then illustrate how quantification analysis is applied to parse tree output to return score and ratio values. In the interests of brevity limited code snippets will be used, and only then to illustrate key ideas and features. A full copy of the code is included in the appendix. An example of controlled natural language input and parser output is given at the end of the chapter.

4.4.1. A Brief Note on Haskell and Notation

Before we begin, the following is a brief explanation of common Haskell notation that appears in our code. In Haskell, the `>>` function is used when we want to ignore the result of a first operation and move to a second (e.g., `a >> b`). This is frequently used in parsing as it allows us to consume (and skip over) characters that we do not want to use in further operations. The `>>=` function is used when we want to use the output of one operation as an argument for another (e.g., `a >>= b`). We also frequently utilize the `<|>` and `$` operators. Put very simply, the former indicates choice ('or'). The latter is used in function composition. It allows functions to be chained together without needing to use parentheses to control evaluation order.

We frequently make use of `do` notation in the project code. This used as syntactic sugar when using the `>>` and `>>=` operators. `do` notation emulates some aspects of imperative programming, allowing us to do *something akin to* (but *not actually identical to*, as Haskell uses functional language compilation and memory management) storing local variables within a process to hold interim results using `<-`:

```
do result1 <- a
result2 <- b result1
   c result2
```

which is equivalent to:

```
a >>= (b >>= c)
```

We also use the `try` notation throughout the project code. In simple terms, `try` is a parser combinator that allows us to 'roll back' on previously consumed input when an error is encountered. This is important when checking an input string against a set of closely related but distinct parse rules. For instance, when parsing a string "howdy!" using two parsers checking for "hello" and "howdy" respectively, we don't want the failure of the "hello" parser at the second character of "howdy" to keep the "h" is consumed, as this precludes the second parser from correctly identifying the target string. Using `try` lets us undo the consumption of input and avoid this problem.

Finally, it should be noted that Haskell's status as a functional language means I/O is handled differently from languages like Python, Java, etc. (WikiHaskell, n.d.). For this reason, much of our testing was carried out in `GHCi`, an interpreter that comes with the Haskell platform (Haskell.org, n.d.). Our project code was then compiled and runs as a program that outputs `.o/.hi` object files. Users may run the program in `GHCi` to access individual functions, and to see the parse tree output visualized in the command line.

4.4.2. Types

As outlined in our grammar, the highest type in our controlled natural language is the contract. This carries over into our implementation code. Contracts are comprised of statements, which in turn contain varying combinations of other types. Statements can be (a) recursively constructed from other statements, (b) comprised of non-statement types, or (c) some combination of both. This modularity is reflected in our implementation code.

All non-statement types are either expressions, predicates, or operators. Expressions accept integers, identifiers (defined here as non-reserved strings), operators, predicates, and other expressions as legal values. Note that in parse tree output, constructors name constituents. The set of expression constructors is extensive, and often recursively construct other expressions. This was necessary for all possible statements made of the CNL to be parsed.

In introducing predicates in our grammar (4.1.4), we noted that *vague quantifier* and *vague relation* are implemented as predicates (`Pred`) and operators (`Op`) in our code. These are constructed through use of `Infix` and `Prefix` operations (see subsection 4.4.2). Predicates identify modifiers and determiners that alter the expression in which they are used. Operators are constituents that identify a relation between two or more parties, objects, etc. Figures 3-5 below provide code snippets to illustrate this.

Figure 3: Contract and Statement type definitions

```
newtype Contract = Contract Stmt
        deriving (Eq, Show)

data Stmt = Deontic Expr
          | Component Stmt Stmt
          | Conditional Stmt Stmt
          | DegreeVague Expr
          | Disjunction Stmt Stmt
          | Negation Stmt
          | Conjunction Stmt Stmt
          | Contingency Expr
          | Time Expr Stmt
          deriving (Eq, Show)
```

Figure 4: Expression type definition

```
data Expr = Noun Id
          | Num Integer
          | USD Integer
          | GBP Integer
          | Past Op Expr Expr
          | Future Op Expr Expr
          | VagueRelation Op Expr Expr
          | VagueQuantifier Pred Expr
          | Predicate Pred Expr
          | Verb Id
          | Adjective Id
          | Adverb Id
          | Month Id
          | Year Integer
          | Day Integer
          | Date Expr Expr Expr
          | Description' Expr Expr Expr
          | Description Expr Expr
          | Inhibition Expr Expr
          | Permission Expr Expr
          | Obligation Expr Expr
          | Quantity Expr Expr
          deriving (Eq, Show)
```

Figure 5: Operator and Predicate type definitions

```
data Op = TemporalRange
        | ObjectRange
        | Payment
        | Delivery
        | Refund
        | Event
        deriving (Eq, Show)

data Pred = Within
        | After
        | Before
        | Minimum
        | Maximum
        | Approximation
        deriving (Eq, Show)
```

4.4.3. Parsing

Contracts are constructed using parser combinators. When an object written in the CNL is input to the `parseContract` method, a parse tree is generated. This is achieved via incremental calls to higher-order parser combinators, sub-parsers, and auxiliary functions. The highest functions are `getcontract`, `statement`, and `statement'`. `getcontract` sends input to `statement`, which demarcates its content into substrings by our designated separation character (the period symbol). As they are separated, substrings are sent as input to consecutive calls to another combinator, `statement'`, which identifies statement type according to criteria defined in its sub-parsers. At each step of contract parsing, sections of the original input are consumed, and constituents of the final parse tree are constructed. The particulars of those constituents are dependent on which parsing functions succeeded or failed.

The constituent returned by `statement'` is highly variable and dependant on the output of the combinator's sub-parsers (and, in turn, the output(s) of their own sub-parsers). There are eight statement parsers, each of which either (a) test their input for the presence of `reserved/reservedOp` keywords, (b) pass all or parts of their input to other statement parsers to check for the presence of sub-statements, (c) pass all or parts of their input to expression parsers, the success or failure of which determine the success of the statement parser, or (d) do all of the above. Figures 6-8 below illustrate this.

Figure 6: Contract-building functions (1)

```
parseContract :: String -> Contract
parseContract str =
    case parse parser' "" str of
        Left e -> error $ show e
        Right r -> r

parser' :: Parser Contract
parser' = whiteSpace >> getcontract

getcontract :: Parser Contract
getcontract = do
    contract <- statement
    return $ Contract contract
```

Figure 7: Contract-building functions (2)

```
statement :: Parser Stmt
statement = do
    components <- sepBy1 statement' (reserved ".")
    if length components == 1
        then return $ head components
    else return $ foldr1 Component components
```

Figure 8: Statement parser

```
statement' :: Parser Stmt
statement' = try deonticStmt
            <|> ifStmt
            <|> disjunctionStmt
            <|> conjunctionStmt
            <|> negationStmt
            <|> (try descriptionStmt
            <|> contingentStmt)
            <|> timeStmt
```

If, for instance, a string fails upon all statement parsers before finally succeeding on `timeStmt`, the following process has occurred. First, part of its input has been successfully consumed by one of its lower expression parsers. Second, the remaining input has been recursively sent back to `statement'`, and in isolation succeeds on some test in the `deonticStmt` parser.

Figure 9: Temporal vague statement parser

```
timeStmt :: Parser Stmt
timeStmt = do
    e <- (reserved "On" >> getdate) <|> try betweencheck <|> timePrefix <|>
timePrefix' <|> timePrefix'' <|> tempredexpr
    Time e <$> statement
```

Figure 10: Deontic statement parser

```
deonticStmt :: Parser Stmt
deonticStmt = do
    e <- try clauseExp <|> try getObligation
    return $ Deontic e
```

Both `timeStmt` and `deonticStmt` depend on the output of other expression parsers for the `Expr` values attached to their `Stmt` type outputs. `getObligation`, for instance, returns an `Obligation` constituent. Depending on which of its sub-parsers succeed, it also outputs specific combinations of other constituents. Nuances in input are caught by bespoke parser functions at this level. This ensures that the relative expressive freedom the CNL user has when specifying deontic structures (e.g., through use of vague modifiers, adjectives, etc.) is correctly reflected in the constituents of the parse tree.

Figure 11: Obligation expression parser

```
getObligation :: Parser Expr
getobligation :: Parser Expr
getobligation = do
    e1 <- try binexpression
    e2 <- try objrange <|> try vaguequantity <|> try getquantity <|> try
predexpression <|> try getobjectdescription <|> compoundnounterm <|>
binexpression
```

The `binexpression` operation and its associated helper functions utilize Parsec's `Text.ParserCombinators.Parsec.Expr` module and its built-in operator constructors (specifically, `Infix` and `Prefix`). This streamlines the process of attaching user-defined identifier

variables to their associated constituents. *Obligation* expressions are modelled as binary relations between two parties; they are constructed with use of `Infix` operations, matching obligation (an `Op` type: `Delivery`, `Refund`, `Payment`) with its predicated tense (a `Pred` type: `Future`, `Past`).

Figure 12: Parsing binary obligation expressions (1)

```
binexpression :: Parser Expr
binexpression = buildExpressionParser binoperators term

term =  fmap Noun identifier
        <|> fmap Num integer
        <|> (symbol "$" >> fmap USD integer)
        <|> (symbol "£" >> fmap GBP integer)
        <|> parens binexpression
```

Figure 13: Parsing binary obligation expressions (2)

```
binoperators :: [[Operator Char st Expr]]

binoperators = [ [
              Infix (reservedOp "will pay" >> return (Future Payment))
AssocNone,
              Infix (reservedOp "paid"  >> return (Past Payment)) AssocNone,
              Infix (reservedOp "will deliver to" >> return (Future Delivery))
AssocNone,
              Infix (reservedOp "delivers" >> return (Future Delivery))
AssocNone,
              Infix (reservedOp "delivered to"  >> return (Past Delivery))
AssocNone,
              Infix (reservedOp "will refund" >> return (Future Refund))
AssocNone,
              Infix (reservedOp "must refund" >> return (Future Refund))
AssocNone,
              Infix (reservedOp "refunded" >> return (Past Refund)) AssocNone]
]
```

The code snippets above illustrate how we implemented parsing for sentences in our CNL that contain both vague and deontic structures. A sentence that would successfully parse under rules outlined above is “Between January and February Duncan paid Jack $300”. This models relational vague structures. The keyword *between* binds the nouns ‘Duncan’ and ‘Jack’ in a deontic relation *past payment*, but one with tensed yet imprecise temporal aspects. A successful parse on this statement adds, among others, the constituent `VagueRelation` to the parse tree.

If the sentence were to be modified to include vague quantifiers (e.g., if the keyword *approximately* were inserted before dollar amount paid), the outer statement type would not change from `Deontic`, whose sub-parsers provide some flexibility when parsing different input string patterns. Rather than returning an error, an additional vague structure `VagueQuantifier` would become a constituent in the overall parse tree. This is necessary for correctness as our CNL allows for limited chaining of vagueness modifiers and expressions within a single sentence of the language.

Vague language type have the property of multiple instantiation; that is, distinct lexical units in natural language can fall under a common vague category (e.g., 'always' and 'before' both instantiate quantificational vagueness). The design of our grammar reflects this, and the translation of multiple distinct input patterns to single common constituents in the final parse tree implements that in our code.

4.4.4. Quantification

When CNL input has been successfully parsed, post-hoc analysis is carried out to determine a vagueness score. A ratio value is also given as to the proportion of vague tokens to all tokens. The decision was made to quantify the parse tree rather than the input string. Our quantification method was based on our vagueness taxonomy. As such, the development process was streamlined by measuring encounter rates for the token types that instantiate our vagueness categories, rather than by counting particular strings that the lexer checks for when generating tokens. For a similar reason, the decision was made to convert the parse tree to a string for quantification, rather than use Parsec's `getState/putState` functions. These functions would be useful for directly checking input for the number of occurrences of certain words held in state – but, given this had already been done during parsing itself, we felt it was not necessary here. Instead, the parse tree was checked for the presence of `VagueQuantifier`, `VagueRelation`, and `Adjective/Adverb` tokens. The latter pair instantiate degree vagueness directly – to avoid double counting, we did not check for the `DegreeVague` token that accompanies these tokens.

Both the scoring and ratio calculations rely heavily on function composition, the order of which will be explained here. First, we will explain the scoring function, `assessment'`. A parse tree is sent to the `combinedScores` function. This is composed of two other functions, `splitTree` and `summation`. `splitTree` takes the parse tree, converts it to a string, and splits it on white space into a list of strings. This list is then sent to `summation`.

Figure 14: Scoring functions (1)

```
assessment' :: String -> String
assessment' x = printf "Score: %.2f" $ calculation $ combinedScores $
parseContract x

combinedScores :: Contract -> Float
combinedScores x = summation $ splitTree x

splitTree :: Contract -> [String]
splitTree x = words $ show x

summation :: [String] -> Float
summation x = quantScore x + relScore x + degScore x
```

Summation sends the list to `quantScore`, `relScore` and `degScore`. Each of these in turn call `degcheck'`, `relcheck'`, and `qcheck'` on the list, replacing the strings inside the list with floats (0.25, 0.5, or 0.75 for strings that match the token type being checked for, and 0 for any other token). The members of this list of floats are then summed. 0 values are insignificant to the calculation and so not dropped at this point (but will be in our `ratio` calculation function). The sum value from each of the scoring functions is returned to `summation` and added together. The process is identical (save the specifics of the float list for each type), so for the sake of brevity we will illustrate with the quantificational vague scoring functions.

Figure 15: Scoring functions (2)

```
quantScore :: [String] -> Float
quantScore x = sum $ qcheck' x

qcheck' :: [String] -> [Float]
qcheck' x = quantValues x

checkQuants :: String -> Float
checkQuants n
 | n == "(VagueQuantifier" = 0.5
 | otherwise = 0

quantValues :: [String] -> [Float]
quantValues = map checkQuants
```

Next, `summation`'s output is passed to `combinedScores`. This returns to the outermost function, `assessment'`, which applies `calculation` to it. `calculation` divides the float by 10 and returns. The value returned is our quantification score.

Figure 16: Final calculation of score

```
calculation :: Float -> Float
calculation x = x/10
```

Figure 17: Scoring a contract in GHCi

```
ghci> assessment' "between June and July Alex will deliver to John
around 200 bikes. John will pay Alex at least $200"
"Score: 0.18"
```

Calculation of ratio uses some common functions. `splitTree` is similarly called on the parse tree. When it returns its list of strings, the list is given as an argument to the `freqCalc` function. This in turn sends the list to the `vagueCards` and `tokenCardinality` functions.

Figure 18: Ratio calculation (1)

```
ratio :: String -> String
ratio x = printf "Ratio: %.2f" $ freqCalc $ splitTree $ parseContract x

freqCalc :: [String] -> Float
freqCalc x =  (fromIntegral $ vagueCards x) / (fromIntegral $ tokenCardinality
x)
```

`vagueCards` sends the list to three further functions (`getDegreeLength`, `getQuantLength`, and `getRelLength`). These functions send the list to `degcheck'`, `relcheck'`, and `qcheck'` to convert to a list of floats, then send the float list to dedicated functions for dropping zeroes. Lists that have been trimmed of zeroes are measured for length, returning an `Int` value for each category. Again, the process is mostly identical for each category so we will only illustrate with quantificational vague functions. `tokenCardinality` calls the `totalTokens` function on the list, returning an `Int` representing the total number of tokens. When `vagueCards` and `tokenCardinality` return to `freqCalc`, their `Int` values are converted to `Float` using `fromIntegral`. The `ratio` calculation is then performed.

Figure 19: Ratio calculation (2)

```
vagueCards :: [String] -> Int
vagueCards x = getDegreeLength x + getQuantLength x + getRelLength x

getQuantLength :: [String] -> Int
getQuantLength x = quantLength $ dropQuants x

dropQuants :: [String] -> [Float]
dropQuants x = dropZeroesQ $ qcheck' x

dropZeroesQ :: [Float] -> [Float]
dropZeroesQ x = filter (==0.5) x

quantLength :: [Float] -> Int
quantLength qua
 | qua == [] = 0
 | otherwise = length qua

tokenCardinality :: [String] -> Int
tokenCardinality x = totalTokens x

totalTokens :: [String] -> Int
totalTokens comps
 | comps == [] = 0
 | otherwise = length comps
```

Figure 20: Contract ratio in GHCi

```
ghci> ratio "between June and July Alex will deliver to John
around 200 bikes. John will pay Alex at least $200"
"Ratio: 0.08"
```

In `GHCi`, `Float` values with leading zeroes after their point are printed in scientific notation. It was judged that this might cause problems for the intended audience (i.e., legal professionals). For the sake of readability, the decision was made to use the `printf` function to format the scores to round to two decimal places. This formatting is maintained in the `GHC` compiled program (i.e., when run outside of `GHCi`).

4.4.5. Complete Example

The code snippets and discussion above illustrate the general process by which statement parsers, expression parsers, and auxiliary functions combine to process CNL input and output abstract syntax trees. Extensive expression parsers and helper functions are used, most of which were not explored in this chapter but can be viewed in the appendix code. The following is a complete example of a short contract written in our natural language, the parse tree it outputs when given as input to our parsing code, and the score it returns when assessed by our quantification functions.

Figure 21: CNL input in GHCi

```
ghci> parseContract "On 23 June 2023 Alex paid John $3000. Within 2024 John will deliver to Alex a computer. If delivery is late then John will pay Alex around $100. Between 2024 and 2026 the warranty is valid. If the computer breaks then before 2027 Alex has permission to complain. If Alex will complain then either John delivers Alex a new computer or John will refund Alex minimum $2500. In 2027 the warranty will expire. If the warranty is expired then Alex may not complain"
```

As the input string is consumed by the parsing functions, constituents are added to the parse tree. Each constituent corresponds to a syntactical type in the grammar, and, as we have shown, is implemented through use of algebraic data types in the underlying Haskell code. The parse tree preserves certain aspects of the input, determined during the parsing process. These are attached to constituents. Constituents of this form derived from our input include `(Noun "Alex")` and `(Year 2023)`. The outermost constituent in a branch of the parse tree corresponds to the statement type it represents in our grammar. So, for example, a deontic statement within our example ("John will refund Alex minimum $2500") is represented in the parse tree as a relation between two nouns, with a vaguely quantified unit of currency as its object: `(Component (Deontic (Obligation (Future Refund (Noun "John") (Noun "Alex")) (VagueQuantifier Minimum (USD 2500))))`. The full parse tree is presented below.

Figure 22: Parse tree output (visualization)

```
Contract (Time (Date (Day 23) (Month "June") (Year 2023)) (Component (Deontic
(Obligation (Past Payment (Noun "Alex") (Noun "John")) (USD 3000))) (Time
(VagueQuantifier Within (Year 2024)) (Component (Deontic (Obligation (Future
Delivery (Noun "John") (Noun "Alex")) (Noun "computer"))) (Conditional
(DegreeVague (Description (Noun "delivery") (Adjective "late"))) (Component
(Deontic (Obligation (Future Payment (Noun "John") (Noun "Alex")) (VagueQuantifier
Approximation (USD 100)))) (Time (VagueRelation TemporalRange (Year 2024) (Year
2026)) (Component (DegreeVague (Description (Noun "warranty") (Adjective
"valid"))) (Conditional (Contingency (Future Event (Noun "computer") (Verb
"breaks"))) (Time (VagueQuantifier Before (Year 2027)) (Component (Deontic
(Permission (Noun "Alex") (Verb "complain"))) (Conditional (Contingency (Future
Event (Noun "Alex") (Verb "complain"))) (Disjunction (Deontic (Obligation (Future
Delivery (Noun "John") (Noun "Alex")) (Description (Adjective "new") (Noun
"computer")))) (Component (Deontic (Obligation (Future Refund (Noun "John") (Noun
"Alex")) (VagueQuantifier Minimum (USD 2500)))) (Time (VagueQuantifier Within
(Year 2027)) (Component (Contingency (Future Event (Noun "warranty") (Verb
"expire"))) (Conditional (DegreeVague (Description (Noun "warranty") (Adjective
"expired"))) (Deontic (Inhibition (Noun "Alex") (Verb
"complain"))))))))))))))))))))
```

As mentioned in the introduction to this subsection (4.4.1), Haskell programs have particular IO requirements. We will not go into the details of why that is the case here. We will however give an illustration of the main function in the project code that allows end users to enter input that is sent to our parsing and quantification functions for processing, and that returns a parse tree as program output.

Figure 23: Main function

```haskell
main :: IO Contract
main = do
    putStrLn "Enter a contract."
    con <- getLine
    let contract = parseContract con
    let score = assessment' con
    let rate = ratio con
    putStrLn score
    putStrLn rate
    return contract
```

The following is output of our example contract, having being run in the compiled program at the command line interface. This outputs `.o` and `.hi` files in the folder where the compiled program is stored.

Figure 24: Program running

```
./Parser
Enter a contract.
On 23 June 2023 Alex paid John $3000. Within 2024 John will
deliver to Alex a computer. If delivery is late then John will pay
Alex around $100. Between 2024 and 2026 the warranty is valid. If
the computer breaks then before 2027 Alex has permission to
complain. If Alex will complain then either John delivers Alex a
new computer or John will refund Alex minimum $2500. In 2027 the
warranty will expire. If the warranty is expired then Alex may not
complain
Score: 0.42
Ratio: 0.07
```

5. Conclusion

Revisiting the project aims and objectives as outlined in chapter 1, we can point to how each was met and where. The requirements of the project were fourfold (research, design, implementation, quantification), and have been met. The research and development processes that were carried out in meeting them have been chronicled throughout the paper.

The project's primary requirement was the development of a controlled natural language for use in the commercial law sector, designed to enable the expression of vague language structures. To ensure our language was relevant to its proposed use case, a review of legal scholarship on the question of vagueness in law was carried out. Further, examples of real-world commercial litigation were analysed and used to directly inform the design process of the language. The results of this review were presented and analysed in chapter 2.

To optimize the design of our language and to inform the implementation process, an analysis of vagueness as a linguistic phenomenon was required. A review of literature on vagueness in formal semantics, philosophy and logic was carried out. From this, a taxonomy of vague types was developed, forming the basis of our controlled natural language and its implementation. This was carried out in chapter 3.

The findings resulting from the research period of the project heavily influenced the design of the controlled natural language. The grammar was introduced, explained, and systematized into Backus-Naur Form in chapter 4.

The other primary requirement of the project was the development of mechanisms to analyse an object written in our controlled natural language, to detect vague structures within it, and to preserve these structures upon translation of that object to another format. This was implemented with the development of code to parse input and generate abstract syntax trees. Project code was written in Haskell and using its Parsec library, as was also outlined in chapter 4.

Quantification mechanisms required for analysis of legal contracts drafted in the CNL were also a primary requirement of the project. These were similarly introduced, formalized, and illustrated in chapter 4.

Bibliography

Airbus France, Guélorget, P., & Icard, B. (2021). *Combining vagueness detection with deep learning to identify fake news.* Institut Jean-Nicod, Paris.

Alston, W. (1964). *Philosophy of Language.* Englewood Cliffs: Prentice Hall.

Barker, C. (2002). The Dynamics of Vagueness. *Linguistics and Philosophy, 25*(1), 1-36.

Bernheim, D., & Whinston, M. (1998). Incomplete Contracts and Strategic Ambiguity. *The American Economic Review, 88*(4), 902-932.

Boolos, G. (1991). Zooming Down the Slippery Slope. *Nous*(25), 695-706.

Burnett, H. (2014). A Delineation solution to the puzzles of absolute adjectives. *Linguistics and Philosophy, 37*(1), 1-39.

Carlson, G., & Pelletier, F. (1995). *The Generic Book.* Chicago: University of Chicago press.

Chartbrook Ltd v Persimmon Homes Ltd and Others (UK House of Lords July 1, 2009).

Crespo, I., & Veltman, F. (2019). Tasting and Testing. *Linguistics and Philosophy*(42), 617-653.

Cruse, A. (2010). *Meaning in Language: An Introduction to Semantics and Pragmatics.* Oxford: Oxford University Press.

Dummett, M. (1995). Bivalence and Vagueness. *Theoria, 61*(3), 201-216.

Dworkin, R. (1986). *Law's Empire.* Cambridge, Mass.: Harvard University Press.

Endicott, T. (2011). The Value of Vagueness. In S. Soames, & A. Marmor, *Philosophical Foundations of Language in the Law.* Oxford: Oxford University Press.

Feinberg, J. (1984). *Harm to Others.* New York City: Oxford University Press.

Fine, K. (1975). Vagueness, Truth and Logic. *Synthese, 30*(3), 265-300.

Frege, G. (1948). Sense and Reference. *The Philosophical Review, 57*(3), 209-230.

Gehrke, B., & Castroviejo, E. (2015). Manner and Degree: An Introduction. *Natural Language & Linguistic Theory, 33*(3), 745-790.

Graff, D. (2000). Shifting Sands: An Interest-Relative Theory of Vagueness. *Philosophical Topics, 28*(1), 45-81.

Hampton, J. (1997). Psychological representations of concepts. In M. Conway, *Cognitive Models of Memory.* Hove: Psychology Press.

Haskell.org. (n.d.). *Glasgow Haskell Compiler*. Retrieved from Haskell: https://downloads.haskell.org/ghc/9.0.1/docs/html/users_guide/

Hu, I. (2017). 'Vague' at Higher Orders. *Mind*(126), 1189-1216.

Hunt, L. W. (2016). What the Epistemic Account of Vagueness Means for Legal Interpretation. *Law and Philosophy, 35*(1), 29-54.

Hunter, A. (2022). Understanding Enthymemes in Deductive Argumentation Using Semantic Distance Measures. *Thirty-Sixth AAAI Conference on Artifical Intelligence.* Association for the Advancement of Artificial Intelligence.

Kamp, H. (2011). Two theories about adjectives. In E. Keenan, *Forman Semantics of Natural Language.* Cambridge: Cambridge University Press.

Kamp, H., & Partee, B. (1995). Prototype theory and compositionality. *Cognition*(57), 129-191.

Kamp, H., & Sassoon, G. (2016). Vagueness. In M. Aloni, & P. Dekker, *The Cambridge Handbook of Formal Semantics.* Cambridge: Cambridge University Press.

Karadotchev, V. (2019). First Steps Towards Logical English. *MSc Thesis*. Imperial College London.

Keefe, R. (2008). Vagueness: Supervaluationism. *Philosophy Compass, 3*(2), 315-324.

Kennedy, C. (2007). Vagueness and grammar: The semantics of relative and absolute gradable adjectives. *Linguistics and Philosophy, 30*(1), 1-45.

Klein, E. (1980). A Semantics for Positive and Comparative Adjectives. *Linguistics and Philosophy, 4*, 1-45.
Kowalski, R., & Datoo, A. (2021). Logical English meets legal English for swaps and derivatives. *Artificial Intelligence and the Law*(30), 163-197.
Kripke, S. (1981). *Naming and Necessity.* Wiley-Blackwell.
Lakoff, G. (1970). A Note on Vagueness and Ambiguity. *Inquiry, 1*(3), 357-359.
Lakoff, G. (1973). Hedges: A Study in Meaning Criteria and the Logic of Fuzzy Concepts. *Journal of Philosophical Logic, 2*(4), 458-508.
Lasersohn, P. (1999). Pragmatic Halos. *Language, 75*(3), 522-551.
Leijen, D., Martini, P., & Latter, A. (2006). *Hackage*. Retrieved from Haskell: https://hackage.haskell.org/package/parsec
Lewis, D. (1975). Adverbs of quantification. In E. Keenan, *Formal Semantics of Natural Language* (pp. 339-359). Cambridge: Cambridge University Press.
Lewis, D. (1999, May). Letter to Timothy Williamson on Vagueness.
Mondeca, Icard, B., & Claveau, V. (2023). *Measuring vagueness and subjectivity in texts: from symbolic to neural VAGO.* Institut Jean-Nicod; Mondeca, Paris.
Moore, M. (1992). Law as a Functional Kind. In R. P. George, & R. P. George (Ed.), *Natural Law Theory.* Oxford: Oxford University Press.
Oxford English Dictionary. (n.d.). *OED*. Retrieved from https://www.oed.com/information/about-the-oed/the-oed-today/
Páll Jónsson, Ó. (2009). Vagueness, interpretation, and the law. *Legal Theory, 15*(03), 193-214.
Pace, G., & Schneider, G. (2009). Challenges in the Specification of Full Contracts. *International Conference on Integrated Formal Methods.* Berlin: Springer.
Peirce, C. S. (1892, October 27). Review of Alfred Sidgwick's Distinction and Criticism of Belief. *The Nation*. New York City, NY: The Nation Company, L.P.
Priest, G. (1979). The Logic of Paradox. *Journal of Philosophical Logic, 8*(1), 219-241.
Prisacariu, C., & Schneider, G. (2012). A dynamic deontic logic for complex contracts. *The Journal of Logic and Algebraic Programming*(81), 458-490.
Rips, L., & Turnbull, W. (1980). How big is big? Relative and absolute properties in memory. *Cognition, 8*(2), 145-174.
RTS Flexible Systems Ltd V Molkerei Alois Müller Gmbh & Co KG (UK Supreme Court March 10, 2010).
Russell, B. (1905). On Denoting. *Mind, 14*(56), 479-493.
Soames, S. (2011). What Vagueness and Inconsistency Tell Us about Interpretation. In S. Soames, & A. Marmor, *Philosophical Foundations of Language in the Law.* Oxford: Oxford University Press.
Sorenson, R. (2001). *Vagueness and Contradiction.* Oxford: Oxford University Press.
Surden, H. (2012). Computable Contracts. *UC Davis Law Review, 46*.
von Fintel, K. (1994). *Restrictions on Quantifier Domains, PhD Thesis.* Amherst: University of Massacheusetts at Amherst.
WikiHaskell. (n.d.). *Wiki*. Retrieved from Haskell: https://wiki.haskell.org/IO_inside
Williamson, T. (1994). *Vagueness.* New York City: Routledge.
Wright, C. (2010). The Illusion of Higher-Order Vagueness. In R. Dietz, & S. Moruzzi, *Cuts and Clouds: Vagueness, its Nature, and its Logic* (pp. 523-549). Oxford: Oxford University Press.
Wright, C. (2021). *The Riddle of Vagueness: Selected Essays 1975-2020.* Oxford: Oxford University Press.
Zadeh, L. (1965). Fuzzy sets. *Information and Control, 8*(3), 338-353.

Zhang, Q. (1998). Fuzziness - vagueness - generality - ambiguity. *Journal of Pragmatics, 29*(1), 13-31.
Zwicky, A., & Sadock, J. (1975). Ambiguity Tests and How to Fail them. *Syntax and Semantics*(4).

Appendix

```
import Control.Monad
import Text.Parsec.Language ( emptyDef )
import Text.ParserCombinators.Parsec
    ( char,
      letter,
      string,
      alphaNum,
      sepBy1,
      (<|>),
      many,
      parse,
      try,
      Parser, anyChar, oneOf )
import Text.ParserCombinators.Parsec.Expr
    ( buildExpressionParser,
      Assoc(AssocNone),
      Operator(Prefix, Infix) )
import Text.ParserCombinators.Parsec.Language ( emptyDef )
import qualified Text.ParserCombinators.Parsec.Token as Token
import Control.Monad.State as S
    ( MonadTrans(lift), modify, execState, State )
import Text.Parsec (ParsecT, runParserT, runParser, string, getState, putState,
noneOf)
import Data.Typeable
import Data.Text (splitOn)
import Text.Printf

---------- algebraic data types implementing grammar
data Op = TemporalRange
        | ObjectRange
        | Payment
        | Delivery
        | Refund
        | Event
        deriving (Eq, Show)

data Pred = Within
        | After
        | Before
        | Minimum
        | Maximum
        | Approximation
        deriving (Eq, Show)

data Expr = Noun Id
          | Num Integer
          | USD Integer
          | GBP Integer
```

```
          | Past Op Expr Expr
          | Future Op Expr Expr
          | VagueRelation Op Expr Expr
          | VagueQuantifier Pred Expr
          | Predicate Pred Expr
          | Verb Id
          | Adjective Id
          | Adverb Id
          | Month Id
          | Year Integer
          | Day Integer
          | Date Expr Expr Expr
          | Description' Expr Expr Expr
          | Description Expr Expr
          | Inhibition Expr Expr
          | Permission Expr Expr
          | Obligation Expr Expr
          | Quantity Expr Expr
         deriving (Eq, Show)

data Stmt = Deontic Expr
          | Component Stmt Stmt
          | Conditional Stmt Stmt
          | DegreeVague Expr
          | Disjunction Stmt Stmt
          | Negation Stmt
          | Conjunction Stmt Stmt
          | Contingency Expr
          | Time Expr Stmt
         deriving (Eq, Show)

newtype Contract = Contract Stmt
        deriving (Eq, Show)

type Id = String -- used as shorthand for strings when using lexing functions

--------------- language definition---------------
languageDef =
     emptyDef    { Token.commentStart    = ""
                 , Token.commentEnd      = ""
                 , Token.commentLine     = ""
                 , Token.nestedComments  = False
                 , Token.caseSensitive   = False
                 , Token.identStart      = letter
                 , Token.identLetter     = alphaNum
                 , Token.reservedNames   = [  "."
                                           , "if"
                                           , "then"
                                           , "else"
                                           , ","
                                           , "between"
```

```
                                        , "either"
                                        , "is"
                                        , "is a"
                                        , "has permission to"
                                        , "may not"
                                        , "between"
                                        , "on"
                                        , "or"
                                        , "and"
                                        ,"it is the case that"
                                        , "it is not the case that"
                                        , "the"
                                        , "will"
                                        , "within"
                                        , "after"
                                        , "before"
                                        , "in"
                                        ]
                , Token.reservedOpNames = [ "will pay"
                                        , "paid"
                                        , "will deliver to"
                                        , "delivers"
                                        , "delivered to"
                                        , "will refund"
                                        , "must refund"
                                        , "refunded"
                                        , "minimum"
                                        , "maximum"
                                        , "approximately"
                                        , "around"
                                        ]
                }

-------------lexing functions-----
lexer = Token.makeTokenParser languageDef

identifier = Token.identifier lexer
reserved   = Token.reserved   lexer
reservedOp = Token.reservedOp lexer
semi       = Token.semi       lexer
parens     = Token.parens     lexer
integer    = Token.integer    lexer
whiteSpace = Token.whiteSpace lexer
symbol = Token.symbol lexer



ifStmt :: Parser Stmt
ifStmt = do
   reserved "If"
```

```
   s1 <- statement
   reserved "then"
   Conditional s1 <$> statement

deonticStmt :: Parser Stmt
deonticStmt = do
    e <- try clauseExp <|> try getobligation
    return $ Deontic e

timeStmt :: Parser Stmt
timeStmt = do
    e <- (reserved "On" >> getdate) <|> try betweencheck <|> timePrefix <|>
timePrefix' <|> timePrefix'' <|> tempredexpr
    Time e <$> statement

disjunctionStmt :: Parser Stmt
disjunctionStmt = do
   reserved "Either"
   s1 <- statement
   reserved "or"
   Disjunction s1 <$> statement

negationStmt :: Parser Stmt
negationStmt = do
   reserved "It is not the case that"
   Negation <$> statement

descriptionStmt :: Parser Stmt
descriptionStmt = do
    e1 <- (reserved "The" >> (try moddescription <|> try simpledescription)) <|>
try moddescription <|> try simpledescription
    return $ DegreeVague e1

contingentStmt :: Parser Stmt
contingentStmt = do
    e <- (reserved "The" >> (try makeFuture <|> try makeFuture')) <|> try
makeFuture <|> try makeFuture'
    return $ Contingency e

conjunctionStmt :: Parser Stmt
conjunctionStmt = do
   reserved "It is the case that"
   s1 <- statement
   reserved "and"
   Conjunction s1 <$> statement
```

```
statement :: Parser Stmt
statement = do
    components <- sepBy1 statement' (reserved ".")
    if length components == 1
        then return $ head components
    else return $ foldr1 Component components

getcontract :: Parser Contract
getcontract = do
    contract <- statement
    return $ Contract contract

----------highest statement parser----------
statement' :: Parser Stmt
statement' = try deonticStmt
            <|> ifStmt
            <|> disjunctionStmt
            <|> conjunctionStmt
            <|> negationStmt
            <|> (try descriptionStmt
            <|> contingentStmt)
            <|> timeStmt

----- expression building functions-------
binexpression :: Parser Expr
binexpression = buildExpressionParser binoperators term

term =  fmap Noun identifier
        <|> fmap Num integer
        <|> (symbol "$" >> fmap USD integer)
        <|> (symbol "£" >> fmap GBP integer)
        <|> parens binexpression

--- infix operations for binary relationships (deontic)

binoperators :: [[Operator Char st Expr]]

binoperators = [ [
              Infix (reservedOp "will pay" >> return (Future Payment)) AssocNone,
              Infix (reservedOp "paid"  >> return (Past Payment)) AssocNone,
              Infix (reservedOp "will deliver to" >> return (Future Delivery))
AssocNone,
              Infix (reservedOp "delivers" >> return (Future Delivery)) AssocNone,
              Infix (reservedOp "delivered to"  >> return (Past Delivery))
AssocNone,
              Infix (reservedOp "will refund" >> return (Future Refund)) AssocNone,
              Infix (reservedOp "must refund" >> return (Future Refund)) AssocNone,
              Infix (reservedOp "refunded" >> return (Past Refund)) AssocNone] ]
```

```haskell
-- check for inhibition and permission patterns
clauseExp :: Parser Expr
clauseExp = do
    e <- try inhibition <|> try permission
    return e

-- non-vague quantification of objects
getquantity :: Parser Expr
getquantity = do
    e1 <- predints          -- quantity
    e2 <- term              -- object
    return $ Quantity e1 e2

-- this is called when vague quantification of objects is detected in a deontic
statement
vaguequantity :: Parser Expr
vaguequantity = do
    e1 <- predexpression  -- approximately/etc
    e2 <- term     -- integer
    return $ Quantity e1 e2

--- used when we want to ignore determiners in a description
compoundnounterm = (string "a " >> fmap Noun identifier)
        <|> (reserved "the " >> fmap Noun identifier)
        <|> fmap Noun identifier

-- description in a deontic statement
getobjectdescription :: Parser Expr
getobjectdescription = do
    (string "a ") <|> (string "the ")
    e1 <- adjterm
    e2 <- term
    return $ Description e1 e2

-- used when parsing a non-deontic statement with a verb e.g. contingency
verbterm = fmap Verb identifier

-- used for contingency expressions, future tense actions
makeFuture' :: Parser Expr
makeFuture' = do
    e1 <- term --noun
    e2 <- verbterm -- verb
    return $ Future Event e1 e2

makeFuture :: Parser Expr
makeFuture = do
    e1 <- compoundnounterm -- noun
    reserved "will"
    e2 <- verbterm -- verb
    return $ Future Event e1 e2
```

```haskell
-- used when building expressions with vague quantification
predexpression :: Parser Expr
predexpression = buildExpressionParser predicates predterm

-- currency or number
predterm = predints

predints = (symbol "$" >> fmap USD integer)
        <|> (symbol "£" >> fmap GBP integer)
        <|> fmap Num integer

------ used for constructing vague quantifying expressions
predicates :: [[Operator Char st Expr]]

predicates = [[ Prefix (reservedOp "approximately" >> return (VagueQuantifier
Approximation)),
                Prefix (reservedOp "around" >> return (VagueQuantifier
Approximation)),
                Prefix (reservedOp "minimum" >> return (VagueQuantifier Minimum)),
                Prefix (reservedOp "maximum" >> return (VagueQuantifier Maximum)),
                Prefix (reservedOp "at least" >> return (VagueQuantifier Minimum)),
                Prefix (reservedOp "at most" >> return (VagueQuantifier Maximum)),
                Prefix (reservedOp "within" >> return (VagueQuantifier Within)),
                Prefix (reservedOp "after" >> return (VagueQuantifier After)),
                Prefix (reservedOp "before" >> return (VagueQuantifier Before))
                ]]

-- used in temporalrange
predtempterm = fmap Month identifier <|> fmap Year integer

--- modified description i.e. one with an adverb
moddescription :: Parser Expr
moddescription = do
    e <- term
    reserved "is"
    e1 <- adverbterm
    e2 <- adjterm
    return $ Description' e e1 e2

--- non-adverb adjectival expression
simpledescription :: Parser Expr
simpledescription = do
    e <- term
    reserved "is"
    e2 <- adjterm
    return $ Description e e2

--- specific fmap fucntions for degree vague instantiation
adjterm :: Parser Expr
adjterm = fmap Adjective identifier
```

```
adverbterm :: Parser Expr
adverbterm = fmap Adverb identifier

-- deontic methods

----- gets specifics for an obligation statement, i.e. quantity, presence of vague
modifiers, etc
getobligation :: Parser Expr
getobligation = do
    e1 <- try binexpression    -- subjects + relation
    e2 <- try objrange <|> try vaguequantity <|> try getquantity <|> try
predexpression <|> try getobjectdescription <|> compoundnounterm <|> binexpression
-- object(s)
    return $ Obligation e1 e2

--- parses an inhibition deontic statement
inhibition :: Parser Expr
inhibition = do
   e1 <- term
   reserved "may not"
   Inhibition e1 <$> verbterm

-- parses a permission deontic statement
permission :: Parser Expr
permission = do
   e1 <- term
   reserved "has permission to"
   Permission e1 <$> verbterm

-- temporal fmap parsing functions based on context pattern
----------
dayterm :: Parser Expr
dayterm = fmap Day integer
monthterm :: Parser Expr
monthterm = fmap Month identifier
yearterm :: Parser Expr
yearterm = fmap Year integer

--- temp terms - slight differences based on usage
tempterm = monthterm <|> yearterm
tempexpr = buildExpressionParser binoperators tempterm
tempredexpr = buildExpressionParser predicates tempterm

getdate :: Parser Expr
getdate = do
   e1 <- dayterm
   e2 <- monthterm
   Date e1 e2 <$> yearterm
```

```haskell
---- check for temporal range
betweencheck :: Parser Expr
betweencheck = do
    reserved "Between"
    e1 <- tempterm
    reserved "and"
    e2 <- tempterm
    return $ VagueRelation TemporalRange e1 e2

------ checks for temporal quant-------
timePrefix :: Parser Expr
timePrefix = do
    reserved "Before"
    e <- tempterm
    return $ VagueQuantifier Before e

timePrefix' :: Parser Expr
timePrefix' = do
    reserved "Within" <|> reserved "In"
    e <- tempterm
    return $ VagueQuantifier Within e

timePrefix'' :: Parser Expr
timePrefix'' = do
    reserved "After"
    e <- tempterm
    return $ VagueQuantifier After e

-- used for 'between x and y' relations of quantity. should be used for currency
objrange :: Parser Expr
objrange = do
    reserved "Between"
    e1 <- predints
    reserved "and"
    e2 <- predints
   -- e3 <- term
    return $ VagueRelation ObjectRange e1 e2 --

---------- functions for quantifying vagueness ------
-- takes string copy of parse tree, applies each quantification test, sums up
summation :: [String] -> Float
summation x = quantScore x + relScore x + degScore x

-- creates a string copy of parse tree
splitTree :: Contract -> [String]
splitTree x = words $ show x

-- higher function for splitTree and summation
combinedScores :: Contract -> Float
combinedScores x = summation $ splitTree x
```

```haskell
-- get a vagueness score between 0,1
calculation :: Float -> Float
calculation x = x/10

-- input is a string written in rules of CNL, output is a score
assessment :: String -> Float
assessment x = calculation $ combinedScores $ parseContract x

-- functions to calculate scores
----- scores quant tokens
quantScore :: [String] -> Float
quantScore x = sum $ qcheck' x

qcheck' :: [String] -> [Float]
qcheck' x = quantValues x

--- scores rel tokens
relScore :: [String] -> Float
relScore x = sum $ relcheck' x

relcheck' :: [String] -> [Float]
relcheck' x = relationalValues x

---- Degree Vague
-- scores degree tokens
degScore :: [String] -> Float
degScore x = sum $ degcheck' x

degcheck' :: [String] -> [Float]
degcheck' x = degreeValues x

--- Vague Quantifiers
--- turns tokens into numbers
checkQuants :: String -> Float
checkQuants n
 | n == "(VagueQuantifier" = 0.5
 | otherwise = 0

quantValues :: [String] -> [Float]
quantValues = map checkQuants

---- Vague Relations
-- turns tokens to numbers
checkRelations :: String -> Float
checkRelations n
 | n == "(VagueRelation" = 0.75
 | otherwise = 0

relationalValues :: [String] -> [Float]
relationalValues = map checkRelations
```

```haskell
---- Degree Vague
-- tokens to numbers
checkDegree :: String -> Float
checkDegree n
 | n == "(Adverb" = 0.25
 | n == "(Adjective" = 0.25
 | otherwise = 0

degreeValues :: [String] -> [Float]
degreeValues = map checkDegree

--- measures ast string for length
totalTokens :: [String] -> Int
totalTokens comps
 | comps == [] = 0
 | otherwise = length comps

--- apply these to list with zeroes dropped
degreeLength :: [Float] -> Int
degreeLength deg
 | deg == [] = 0
 | otherwise = length deg

relLength :: [Float] -> Int
relLength rel
 | rel == [] = 0
 | otherwise = length rel

quantLength :: [Float] -> Int
quantLength qua
 | qua == [] = 0
 | otherwise = length qua

-- use this if you want to get only the total number of tokens when parsing
directtokenCard :: String -> Int
directtokenCard x = totalTokens $ splitTree $ parseContract x

--drop for quants. gets rid of zeroes for cardinality
dropZeroesQ :: [Float] -> [Float]
dropZeroesQ x = filter (==0.5) x

-- drop for degrees
dropZeroesD :: [Float] -> [Float]
dropZeroesD x = filter (==0.25) x

-- drop for relationals
dropZeroesR :: [Float] -> [Float]
dropZeroesR x = filter (==0.75) x
```

```haskell
-- on list of token strings, returns weightings as a list and drops zeroes from it
dropQuants :: [String] -> [Float]
dropQuants x = dropZeroesQ $ qcheck' x

dropDegree :: [String] -> [Float]
dropDegree x = dropZeroesD $ degcheck' x

dropRel :: [String] -> [Float]
dropRel x = dropZeroesR $ relcheck' x

------ cardinalities
getDegreeLength :: [String] -> Int
getDegreeLength x = degreeLength $ dropDegree x

getQuantLength :: [String] -> Int
getQuantLength x = quantLength $ dropQuants x

getRelLength :: [String] -> Int
getRelLength x = relLength $ dropRel x

-- getting cardinalities for ratio calculation
vagueCards :: [String] -> Int
vagueCards x = getDegreeLength x + getQuantLength x + getRelLength x

tokenCardinality :: [String] -> Int
tokenCardinality x = totalTokens x

----- calculates the total ratio of vague tokens to all tokens
freqCalc :: [String] -> Float
freqCalc x =  (fromIntegral $ vagueCards x) / (fromIntegral $ tokenCardinality x)

-- for total ratio - this uses printf to display a rounded up 2 decimal place
string message
ratio :: String -> String
ratio x = printf "Ratio: %.2f" $ freqCalc $ splitTree $ parseContract x

--this prints ratio in scientific notation if leading number is zero after decimal
point
ratio' :: String -> Float
ratio' x = freqCalc $ splitTree $ parseContract x

-- for total score - this uses printf to display a rounded up 2 decimal place
string message
assessment' :: String -> String
assessment' x = printf "Score: %.2f" $ calculation $ combinedScores $ parseContract
x

---- ignores white space and sends all else to getcontract for parsing
parser' :: Parser Contract
parser' = whiteSpace >> getcontract
```

```haskell
----- function on a string, generates a parse tree
parseContract :: String -> Contract
parseContract str =
    case parse parser' "" str of
        Left e -> error $ show e
        Right r -> r

main :: IO Contract
main = do
    putStrLn "Enter a contract."
    con <- getLine
    let contract = parseContract con
    let score = assessment' con
    let rate = ratio con
    putStrLn score
    putStrLn rate
    return contract
```