Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

First Steps Towards Logical English

Author:
Vesko Karadotchev

Supervisor:
Fariba Sadri

Submitted in partial fulfillment of the requirements for the MSc degree in MSc Computing Science of Imperial College London

September 2019

Abstract

A stumbling block to automating legal reasoning is the fact that laws need to be understood by everyone; however, few people understand the executable code necessary for automation. Controlled Natural Languages (CNLs) are attempts to provide an intermediate layer between natural and computer languages. This paper presents the grammar and semantics of a controlled language called Logical English (LE). The aim is for Logical English to both be intuitively understandable by English speakers and automatically translatable to executable code.

Logical English is inspired by logic programming and is designed to be clear and unambiguous. The grammar and semantics presented in the paper make the language expressive enough for real-world applications. I demonstrate this by translating into LE key parts of a standard financial contract.

Acknowledgments

I want to especially thank Bob Kowalski for the many hours he dedicated to discussing Logical English with me, and for his patience and advice while I was trying to find my way through this project. I would also like to thank Fariba Sadri and Miguel Calejo for always being available to chat and answer my questions about Prolog and logic.

Chapter 1

Introduction

1.1 Task

In this project I set out to develop initial grammar rules and semantics for a controlled language called Logical English (LE). Three overarching objectives guided development:

- English speakers without computer science knowledge should be able to understand programs written in LE.
- There should be an automated way to reliably translate LE into executable code.
- Logical English should be expressive enough to handle the logical rules and relationships found in legal contracts.

The first two objectives are in opposition and presented the key challenge for the project. Natural language is ambiguous and imprecise. Words and phrases have changeable and incomplete meanings, which more often than not require an interpretation based on context. Natural language is easy for humans to understand (and misunderstand!) but difficult to reliably translate into unambiguous instructions for a computer. Computer languages remove ambiguity by introducing strict, often unintuitive, grammar rules which take time and effort for people to learn.

Logical English is an intermediary between these extremes – easy to comprehend by both people and machines. Translations from spoken English to Logical English have to be done by hand; however, translations from LE to executable code are fully automatable. For this report, the lower-level, executable code is written in Prolog. See figure 1.1 for a schematic representation.

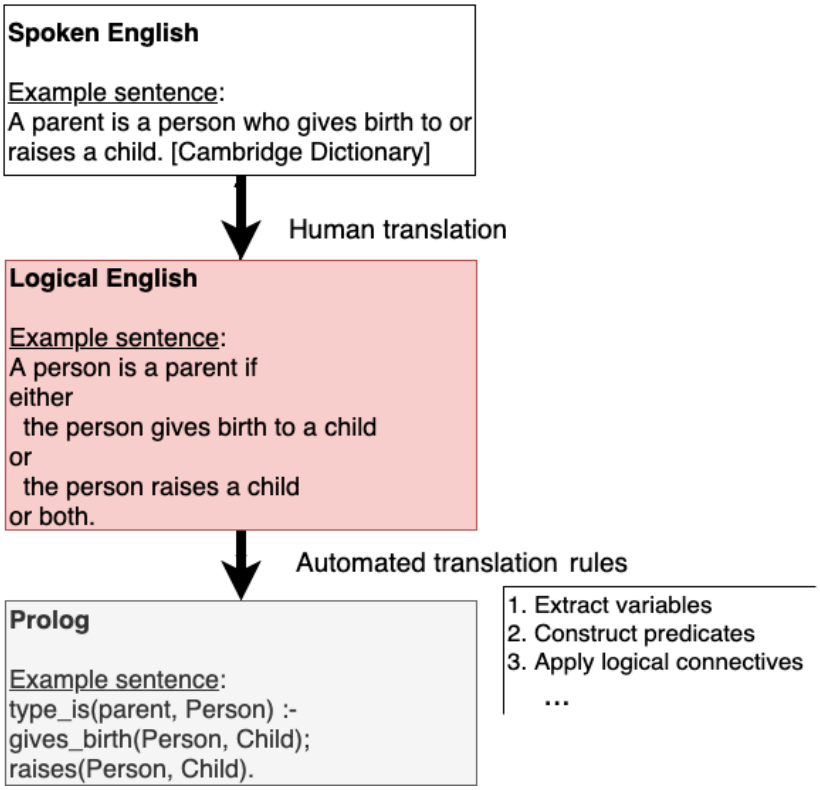


Figure 1.1: How LE relates to spoken English and Prolog

1.2 Main application of Logical English

The most natural application for Logical English is in the area of legal automation, especially the automation of contracts. Kowalski notes that legal texts can be viewed as programs which regulate humans affairs (1). Well-written legislation is precise and unambiguous, which means it lends itself to being translated into computer programs. The advantages of automation in law should be clear – vast improvements in the efficiency with which legal documents are produced, searched, analysed and interpreted. Both industry and academia have long been interested in this area (2) (3) (4).

One outstanding issue in contract automation is the difficulty of verifying that a piece of code faithfully represents and preserves the logical structure of the corresponding legal document (5). Lawyers usually cannot read computer code and programmers are not lawyers, so there is a disconnect between the legal document and its representation as a computer program. Logical English contributes to solving this problem because it is designed to be understandable by English speakers with no programming knowledge. A document written in LE could be verified by a lawyer simply by comparing the source text to the translation and using judgement to determine how closely they correspond.

An additional benefit of Logical English is that its sentence structure makes logical relationship explicit (see the next section), removing some of the ambiguities inher-

ent in natural language. This can be beneficial both for writing law, as the writer is forced to think carefully about all relationships they want to express, and for understanding law, as the implications of legal statements are made more obvious.

1.3 Writing Logical English

The main contribution of my project consists in defining the grammar and semantics for Logical English, and translating into LE key parts of the ISDA Master Agreement – a standard financial contract used for trading derivatives.

Logical English can be described by its grammar and semantics. The grammar is the set of rules which distinguish random strings of words from correct LE constructions. Semantics deals with how we interpret grammatical sentences when deciding whether they are true or false. This section outlines how LE sentences can be constructed and how they should be interpreted.

LE admits three kinds of sentences – simple sentences, which express facts about individuals; general rules, which express properties of groups of individuals, and complex sentences[1], which talk about properties of both individuals and other properties.

Simple sentences

The simplest meaningful Logical English construction is called a simple sentence. There are three types of simple sentences.

The first type is constructed from proper names, verbs, determiners (“a”, “the”, etc.), and prepositions (“in”, “to”, “from”, etc.).[2] Proper names are the subjects and objects of verbs. Prepositions help identify the semantic role names in sentences and make Logical English easier to read. For example:

```
1  Jane has a Hat.
2  Jack goes Home.
3  The UK leaves the EU in October.
```

are all simple sentences. Note that capitalised words are interpreted as proper names. For example “Hat” in the first sentence refers to a specific individual named “Hat” and not hats in general. If we want to explicitly specify that “Hat” names an entity of type “hat”, in LE we can say:

```
1  Jane has a Hat. Hat is a hat.
```

[1] These correspond to English complex sentences with adverb clauses. An example is the sentence “Mary wishes [that she is home]”, where the subordinated adverb clause modifies the verb “wishes” from the main clause.

[2] As shown later in this section, the word “not” is also allowed directly after verbs.

Importantly, the sequence of words:

```
Tom has a hat.
```

is not a sentence, as the common noun "hat" is not the name of a specific entity, but rather the name for a type of entity.

The second kind of simple sentences in LE consist of a proper name, the linking verb "to be", and an adjective:

```
1  Jane is young.
2  Jack is old.
```

These sentences have the usual English interpretation and are used to assign an attributes individuals.

Finally, and in a similar vein, Logical English admits the "is a" construction. The sentences:

```
1  Joey is a kangaroo.
2  Iago is a parrot.
```

are both simple sentences and are used when we wish to assign a type to an individual. For a detailed discussion of types see Section 3.12.

New nouns and verbs can be constructed by concatenating English adverbs to verbs and adjectives to nouns using the camelCase notation. For example:

```
1  Jane runsQuickly to the Finish.
2  The EiffelTower isLocated in Paris.
```

Simple sentences in Logical English have the same structure as "ground facts" in logic programming – i.e. they do not contain any variables. Note that when we are dealing with constants, the preceding determiners ("the" in "the Finish") do not have a logical meaning and are ignored at lower levels of translation. Determiners become important in LE rules.

Rules

Simple LE sentences can make statements about specific individuals – Tom, Jane, and the like. However, this is not enough to represent the kinds of things usually talked about in legal texts. More often than not, we need to make general claims about types of individuals. This is where rules come in.

Logical English can express rules of the form *HEAD if BODY*, where *HEAD* has similar structure to the simple sentences but admits common nouns (i.e. lower-case nouns like "hat", "person", "animal", etc.) in addition to proper names. *HEAD* alone is not

strictly a simple sentence as it can contain common nouns and it can be substituted for a complex sentence (see next section). The constructions which can legally appear in heads of rules are called LE formulas. *BODY* is conjunction of disjunctions of LE formulas or other conjunctions.

For example, the following is a rule about animals:

```
1  A thing is an animal if
2  the thing is a bird.
```

Common nouns stand in for any individual of the type named by the common noun – e.g., "an animal" can refer to any individual animal. In logic terms, common nouns preceded by indefinite determiners ("a", "an", "some") introduce new variables in a rule. Common nouns preceded by definite determiners are interpreted as referring to a variable introduced earlier int he rule. Thus, "the thing" in line 2 of the pink box always refers to the same individual as "a thing" from line 1.

Sometimes we want to introduce multiple variables of the same type, and then refer back to them later in the rule. To disambiguate between the occurrence of two variables of the same type, Logical English makes use of ordinals:

```
1  A person likes a second person if
2  the second person is good.
```

In the example above, "the second person" is matched to the second new variable of type person (introduced with the phrase "a second person").

LE formulas in the body of a rule can be joined together with the words "and", "ether... or..." and "either... or... or both" to form conjunctions and disjunctions inside the bodies of rules. For example:

```
1  An animal can fly if
2  either
3    the animal is a bird
4  or
5    the animal is a bat.
```

is a valid rule. Logically, "either... or..." is to be interpreted as an exclusive disjunction – i.e., an animal cannot be both a bird and a bat at the same time. See Section 3.5 for more on disjunctions.

Because Logical English is ultimately translated into the programming language Prolog, it takes advantage of some predefined predicates in Prolog. Reserved phrases which correspond to specific logic predicates can be used in the heads or bodies of rules. For example, LE supports simple arithmetic operations and date and time comparisons:

```
1  A person is late for a meeting if
2  the meeting is for a time , and
3  the person arrives at a second time , and
4  the second time is after the first time.
5
6  A first number is a product of
7    a second number and a third number if
8  the first number equals the second number times
9    the third number.
```

Here the word "after" is interpreted as the arithmetic operation "$>$", the word "equals" means "$=$", and the word "times" indicates multiplication "$*$".

Another example of reserved syntax is the phrase "is contained in" which corresponds to the membership check predicate (e.g., 2 is contained in the list [1, 2, 3]). Other examples of reserved syntax are given throughout this report.

Importantly, all variables introduced in the head of a valid LE rule have to be referenced at least once in the body of the rule. So the sequence of words:

```
A pig flies if
the PrimeMinister is sane.
```

is not an LE rule, as "a pig" is not referenced anywhere in the body. In logic programming, this is called range restriction. At the LE level, it removes a possible source of confusion – when we say "a pig" in the sentence above, it is unclear if we refer to a specific individual or to pigs in general.

Complex sentences with an adverb clause

The final kind of LE sentence corresponds to complex English sentences with a subordinated adverb clause. Consider the English sentence:

```
Mary notifies her landlord that she's terminating her tenancy.
```

In Logical English the sentence becomes:

```
1  Mary notifies MarysLandlord that
2   Mary terminates MarysTenancyAgreement.
```

The sentence has an intuitive interpretation. The word "notifies" says something about an action that Mary is performing (i.e. terminating the agreement). In logic terms, *notifies/3* is a meta-predicate and *termiantes/2* is both a predicate and a term in a meta-predicate. For more on meta-predicates, see Section 3.11.

Negation

Logical English supports both strong and weak negation. Weak negation corresponds to the phrase "it cannot be shown that" and applies to simple LE formulas excluding ground facts. In Logical English, it is implemented with reserved syntax:

```
1  A plaintiff is innocent if
2  it cannot be shown that
3  the plaintiff is guilty.
```

The above rule states that every plaintiff is innocent unless we explicitly prove that she is guilty.

Strong negation corresponds to "it is not the case that" (as in, it is not the case that $2 + 2 = 5$). In LE it is represented by placing the constant "not" immediately after a verb:

```
1  John is not tall.
2  A thing canFly not if
3  the thing is a penguin.
```

For more on the two kinds of negation and why they are needed, see Section 3.8.

1.4 Report structure

The remainder of the report is organised as follows. Chapter 2 reviews relevant literature and languages similar to LE. Chapter 3 discusses the grammar rules in more detail, providing justification for why they were selected. Chapter 4 describes a Prolog interpreter for a simplified version of LE, demonstrating how a string of text can be automatically converted to a logic program (second arrow in Figure 1.1). Chapter 5 demonstrates a practical application of Logical English – translating parts of a frequently used financial contract template (first arrow in Figure 1.1). Chapter 6 presents some user-testing results. Chapter 7 concludes and discusses future work.

Throughout the report I give examples of sentences in English, Logical English, Prolog, and formal logic. For the ease of the reader, grammatical LE sentences are always written in pink boxes, while Prolog sentences (usually direct translations from LE) are written in grey boxes. All arithmetic operations in the Prolog code are done using the *clpfd* library and notation (6).

Chapter 2

Literature review

2.1 Logic programming and Prolog

I mentioned that LE sentences take the form of Logic Program clauses which are eventually translated down to Prolog clauses. These concepts have specific meanings in logic that require explaining.

Modern logic has two main components – a formal language and a semantics (7). Formal languages have syntax rules and inference rules. Syntax rules tell us which strings of symbols are part of a given language and which are not. Inference rules tells us how we can derive true sentences from other true sentences.

Semantics is the idea that the valid strings of symbols in a formal language can have an interpretation (or meaning). Interpretations are usually mathematical models which serve as pictures of the world. An interpretation assigns the values "True" and "False" to sentences of the formal language.

Thus, logic can be used to construct models of the world in carefully defined formal languages. The key attraction of logic for programming is that, in principle, machines can automatically apply inference rules to strings of symbols, giving them the ability to reason about the underlying models. Indeed, the driving force behind logic programming research is the principle that "logic subsumes computation" (8). That is, any computational problem can be represented as a logic program.

A logic program is a set of rules of the form:

$H \leftarrow A_1,\ \ A_2, ...,\ \ A_n,\ \ not\ \ B_1,\ \ not\ \ B_2, ...\ \ not\ \ B_m$ for $n, m \geq 0$

where H is called the head of the rule and consists of a single predicate (8). If H is "False", the clause is called a constraint. For the purpose of this report, I do not consider constraints part of logic programs. A_i and $not\ \ B_i$ are predicates/negated predicates, respectively, and their conjunction is called the body of the rule. All variables which appear in both the head and the body are universally quantified, and variables which appear only in the body are existentially quantified.

For example, "All persons are mortal" can be represented by the rule $mortal(X) \leftarrow person(X)$, and the simple fact "John has red hair", can be represented by a rule without a body – $red\_hair(john)$.

Reasoning with logic programs can be automated efficiently by a process called SLD resolution. Resolution is a refutation procedure which proves that a theorem G is implied by a logic program P by deriving a contradiction from $\neg G \cup P$ and the repeated application of a pattern-matching rule (8).

Prolog is a powerful computer language based on logic programming. Programs in Prolog are sets of clauses of the form:

```
HEAD.
HEAD :- BODY.
```

Computation in Prolog usually proceeds by resolution. Theorems in Prolog are called "queries". Given a database of rules, users can pose queries to check if the query is consistent with the preexisting rules. As will be seen later, Prolog has some predefined predicates which include computation beyond pure resolution. An example is the *findall/3* predicate which is used in Chapter 5.

2.2 Controlled Natural Languages

Background

Logical English is an example of a Controlled Natural Language (CNL). I follow Kuhn in defining a CNL as having the following key characteristics (9):

i A CNL is based on a natural language like English;

ii A CNL is more restricted than its base language, having a smaller lexicon and simpler syntax and semantics;

iii A CNL should be similar enough to the base language as to be intuitively understandable by speakers of the base language;

iv A CNL is artificially constructed and its rules are all explicitly defined (it is not a product of an implicit natural process).

According to the purpose for which they were developed, CNLs can be split in two categories – human-oriented and machine-oriented (10). The purpose of human-oriented languages is to make (usually technical) text easier to understand. For example, Standard Technical English (STE) was developed in the 1980s for writing and maintaining technical documents in the commercial aviation industry (11). STE is comprised of a set of writing rules and a dictionary. The dictionary contains easily recognisable English words and follows the rule of "one word per meaning, one meaning per word". For example, the dictionary contains the word "start", but not the words "commence", "begin", "initiate" (11).

Machine-oriented CNLs, on the other hand, are developed to be easily processable by computers, usually for the purpose of creating knowledge bases and performing automated logical reasoning and analysis (10). An example here is Attempto Controlled English (ACE), which was the first CNL wholly translatable in first order logic (12).

Examples

Whatever the purpose, the design of existing CNLs usually follows four general principles, as outlined by Kuhn (13):

i Clearness – there should be as little ambiguity as possible, or at least clear methods to resolve ambiguity;

ii Naturalness – the language should be easy to understand;

iii Simplicity – it should be easy to detect, both for humans and machines, whether a sentence is part of a given CNL or not;

iv Expressivity – the language should be able to describe as many situations and problems as possible.

Three prominent examples of general-purpose CNLs (i.e. not developed with a specific domain in mind) are Attempto Controlled English (ACE), Processable English (PENG), and Computer Processable Language (CPL). The first two are machine-oriented, while the third is human-oriented. I briefly examine each and compare them to Logical English.

Attempto Controlled English (ACE)

As mentioned, ACE is the first controlled language that can be translated unambiguously into first-order logic. A set of construction rules define how ACE sentences can be built, and a set of interpretation rules serve to disambiguate common English constructions during translation. ACE has a vocabulary consisting of function words (like determiners and conjunctions), predefined phrases (like 'there is', 'it is false that'), and content words (which can be nouns, proper names, adjectives, and adverbs). An example of text written in ACE is:

```
1. Every company that buys at least three machines gets a discount.
2. Six Swiss companies each buy one machine.
3. A German company buys four machines.
4. Who gets a discount?
```

ACE has a mathematical knowledge base implemented in Prolog, so it is able to answer the above question correctly. In addition, the ACE parsing engine recognises about 100,000 different content words, with users being able to define additional ones (10).

As demonstrated by the example, ACE is easy to read, as its grammar is very similar to spoken English. Interestingly, ACE's "naturalness" sometimes means that ACE

sentences inherit some of the ambiguity of natural language. Consider sentence 1 again. If a company buys one machine every ten years, will it still get a discount after thirty years (when three machines have been bought), or does the discount apply only when the machines are bought in bulk?

Logical English is at an advantage here. Because it is close to Prolog and expresses relationships in logic program clauses, it is much less likely that LE sentences will suffer from such ambiguities. For example, sentence 1 can be paraphrased in Logical English as follows:

```
1  A company getsDiscount for a purchase on a date if
2  the company makes the purchase on the date , and
3  the purchase consists of a numberOfMachines , and
4  the numberOfMachines is greater than 3.
```

This is more verbose than the original formulation; however, it is less ambiguous, as it makes it clear that the machines need to be purchased on the same date for discounts to apply.

ACE may look easy to read, but it is difficult to write. Gao reports that it takes the untrained person many tries to form a grammatically correct sentence (10). Furthermore, the interpretation rules may disambiguate sentences in unexpected ways. For example, “He” in “Brad is an actor. He is handsome.” will actually be resolved to refer to the word “actor”, rather than the word “Brad” (10). A major stumbling block is ACE’s extensive and difficult to learn grammar. Logical English is again at an advantage here, as it makes use of a very small vocabulary and a very simple set of syntax rules.

Processable English (PENG)

PENG is a more light-weight controlled language compared to ACE. Both ACE and PENG’s processors are based on grammars written in definite clause grammar (DCG) notation. This requires both languages to have a predefined lexicon of content words. However, PENG has a smaller lexicon (about 3,000 words) and simpler grammar rules (10).

In comparison, the rules of Logical English do not require a large dictionary of content words. As shown, content words in LE are distinguished either symbolically (e.g., whether they begin with a capital letter) or with the aid of a small number of predefined preceding words (like determiners) and phrases. Logical English’s lexicon, even if extended, will contain well below 3,000 phrases, effectively making the language ontology free.

A notable feature of PENG is its powerful editor. Because PENG’s parser is very efficient, the editor can analyse strings in real time, while they are being typed by a user, and suggest possible grammatical continuations for the current string.

My project is concerned with the core of Logical English – the grammar and semantics. However, the next step will naturally be the development of supporting software such as a helper editor. PENG can serve as a model here.

Computer Processable Language (CPL)

CPL is a human-oriented controlled language developed by Boeing. It uses a large predefined ontology for resolving the meaning of content words. One of the defining features of the language is that it makes extensive use of heuristics in order to resolve ambiguities and produce "natural" interpretations which a person would expect (14). For example, when interpreting the sentence "A man drives a car for one hour.", CPL will correctly interpret "for" to mean the predefined predicate *duration()* and will discard the incorrect meaning of "for" as designating a beneficiary (as in "This present is for you.") (14). With its large ontology, CPL has been successfully used to pose AP-exam style questions to AURA, a system containing knowledge about chemistry, maths, and physics (15).

In contrast to CPL, Logical English does not admit ambiguities, and requires that sentences have a clear logical form. This can be viewed as both a drawback and a feature. It is a drawback because unambiguous sentences may be harder to write and clumsier to read. It is a benefit because, once written, an LE sentence is unlikely to be interpreted in unintended ways. CPL sentences, on the other hand, can be misinterpreted. If there are two competing interpretations, the CPL engine may chose one, and a human reader the other.

CPL-lite is a version of CPL which is much closer to Logical English – it is not designed to deal with ambiguities and uses a much shorter, deterministic grammar (14). LE's handling of the "is a" construction is inspired by CPL-lite.

Importantly, CPL interprets a sentence following a 10-step pipeline. One step generates a parse tree for the sentence, another step resolves references, yet another assigns labels to semantic rules (as in, deciding whether "for" means duration or refers to a beneficiary), and so on (14). This is in contrast to ACE, where the translation from a CNL sentence to the corresponding first-order logic formula is much more direct.

The simple interpreter for (a simplified) Logical English takes a similar layered approach to translation. As demonstrated in Chapter 4, translation proceeds by iteratively analysing a list of words, with every iteration analysing particular word-patterns, and every subsequent iteration using the analysis from previous ones. The full interpreter, which is an important goal for future work, should maintain this "layered" approach to string analysis.

An example of a bi-directional grammar

The languages discussed above are mostly focused on providing the mechanisms for translating restricted English sentences into a lower-level executable code. This is an example of uni-directional translation. Bi-directional translation, in contrast, allows

the low-level language to be modified (either manually or automatically), and then translated back into the higher-level CNL. In a recent paper, Schwitter outlines a bi-directional grammar for PENG^{ASP}, an answer set programming extension for the PENG language (16).

Logical English does not attempt bi-directional translation, as the practical benefits of such an approach are unclear. If the high-level language is expressive and understandable enough, there should be no need to manipulate the lower-level Prolog. Furthermore, Prolog is less "well behaved" than PENG^{ASP} in the sense that arbitrary Prolog programs are not necessarily written in a style which allows for automatic translation to LE.

2.3 The law as a use case for Logical English

Use of examples from legal texts

As is shown in Chapter 3, each feature of LE is motivated by specific examples of English text. Legal texts are particularly well suited for the purpose. A forthcoming white paper notes that most legal texts are built up of simple statements combined using basic logical connectives such as *if-then*, and *if and only if* (17). This structure lends itself well to logical analysis. Sergot et al. showed that well-written legislation, like the British Nationality Act, can be translated into a logic program which can, in turn, be executed in Prolog (18).

More recently, Flood et al. showed that simple financial contracts, like loan agreements, can be formalised as Discrete Finite Automata (DFA) (19). A DFA is a five-tuple of elements $< Q, S, d, q, F >$, where:

i Q is the set of finite possible states of the world – eg, the loan being extended;

ii S is a finite set of events (both external events and actions) that can occur and potentially change one state of the world into another;

iii d is a mapping $d : Q \times S \mapsto Q$ which shows which actions can convert which states into different states;

iv $q \in Q$ is the starting state of the world – eg, when the contract is signed;

v $F \subseteq Q$, contains the final states of the world – eg, those that terminate the contract.

A financial contract then is simply a description of possible sequences of the form $state \Rightarrow event \Rightarrow state$. Flood et al. note that only a small portion of the state-event sequences explored in a contract describe the expected or "happy" path of events. The majority of clauses in the contract concern themselves with describing possible deviations from the happy path and the consequences and remedies for such deviations (19).

Drafting principles

Good legislation is written with the explicit purpose of being unambiguous and easily understood. In his analysis of what "the rule of law" means, Lord Bingham states that the law should be accessible, intelligible, clear, and predictable (20). In his PhD thesis, Cormacain proposes 18 drafting rules which, when followed, should produce legislation in line with Bingham's four principles (21).

For example, Cormacain's rule of "easification" suggests that long lines of text with multiple conditions should be written in a way which brings out their logical form (21):

```
IF
    X or Y applies
THEN a person must
    Do A or B.
```

This shows how a good code editor with indentation support can structure sentences written in Logical English. Throughout the report, I use indentation in LE sentences to clearly demarcating the beginning and end of predicates and to disambiguate complex disjunctions.

Chapter 3

Technical considerations

This chapter discusses the general issues which motivated the grammar rules presented in Section 1.3. It also gives some "best practices" advice on how Logical English should be written.

3.1 Readability and writeablity

Logical English should be automatically translatable to Prolog. To achieve this, the LE version I present follows the formalist approach[1] and has "a small set of grammar rules that do not conflict with each other or offer multiple alternatives" (14). Grammatical sentences in LE are modelled after Prolog clauses, which in turn take of the form *HEAD if BODY* for rules or *HEAD* for facts. This simplifies automatic translation as the logical structure of LE sentences is explicit and close to the executable code.

At the same time, Logical English should be easy to read, so that legal professionals with no programming background can use it in their work. To that end, LE should be viewed as a subset of everyday English, but with carefully restricted grammar.

Paraphrasing English sentences as logic program clauses is not always simple. Often times multiple translations are available, and the best choice differs from one problem to the next. For example the sentence:

```
John needs to go home before 10pm.
```

can be translation in either of the following ways:

```
1  John needsToGoHome at a time if
2  the time is before 2200.
3
```

[1] Formalist here means that no ambiguity is allowed at the Logical English level. Some languages like CPL have facilities to handle ambiguous statements by evaluating competing logical formulations and selecting the "most natural" one.

```
4  John needs that John goesHomeBefore 2200.
```

Additionally, LE writing involves more than just paraphrasing. For example, we might ask why does John need to go home before 10pm. Suppose that going home before 10pm is only a sub-goal, and what John really cares about is not being grounded by his parents. If this is the intended meaning of the sentence, we can say:

```
1  John is grounded if
2  John getsHome at a time , and
3  the time is after 2000.
4
5  John wants not that John is grounded.
```

In the above example, we did not just paraphrase the sentence, but changed its content to more accurately capture the intended meaning. Writing LE sentences (and writing in general) involves thinking about both form and content. This makes Logical English non-trivial to write – first-time users will not be able to produce sentences which are both grammatically correct and useful for a given application. However, writing well is difficult in any language, and especially in similar CNLs like ACE or PENG. As previously stated, the goal of my project is readability and not writeability. Still, the small set of grammar rules, and a library of useful Logical English texts and translations makes learning to write LE comparatively easy.

3.2 Ambiguity and vagueness

Natural language is frequently both ambiguous and vague. Ambiguity arises when a natural language sentence or phrase does not have a unique logical interpretation. Vagueness arises when certain words admit of relative interpretations. As an example of vagueness, consider the word “tall”. We all agree on its general meaning but we can argue over whether a person with height 1.8m is tall or not.

Ambiguity is difficult to avoid and usually occurs as a mistake, without the explicit intention of the writer. When it does occur, ambiguity can cause problems, especially in law. Consider the following clause in an imaginary contract (22):

```
The artist shall paint the model nude.
```

It is unclear whether the artist should be nude or the model. A sentence like this is not permitted in Logical English. In more real-world cases, ambiguity often arises when we disagree on the meaning of certain words. Consider this clause from a real contract (22):

> These [out-of-pocket] expenses **include** court reporting services, expert witness fees, reasonable travel expenses, if any, fees paid to trial witnesses and the cost to create demonstrative trial exhibits.

In this case, there was a conflict about the meaning of the word “include”. One side argued that the list of included expenses was only partial, and so other expenses, not explicitly mentioned in the clause, should also be covered. The other side maintained that the clause enumerates all the expenses which are covered, and so no other expenses should be considered.

Legal ambiguity frequently leads to litigation and courts have developed rules to deal with it. For example, in buyer-seller relationships, there is a saying rule that an “ambiguous contract is to be interpreted against the seller” (23).

Unlike ambiguity, vagueness is a welcome feature in legal writing. Certain concepts can be left vague on purpose, to allow discretion on a case-by-case basis. Consider the following statement form the British Nationality Act (18):

> An application for registration of an adult or young person as a British citizen [...] must not be granted unless the Secretary of State is satisfied that the adult or young person is **of good character**.

The meaning of the statement is not ambiguous – being of good character is a necessary precondition for receiving British citizenship. Arguably, the phrase “of good character” is not ambiguous, as most English speakers will agree on its general meaning. However, the precise definition of the phrase is vague; it is purposely left out of the clause, giving the Secretary of State some discretion when deciding individual cases.

As Logical English is automatically translatable to Prolog code, ambiguities are much less likely to arise in LE texts. This is an advantage over natural language. On the other hand, when we need vague concepts, we can still make use of them by simply omitting an explicit definition.

3.3 Isomorphism

Isomorphism in knowledge representation is the idea that when we translate from a natural language to a controlled language, we should aim to have a one-to-one correspondence between source and translation. Bench-Capon and Coenen specify five conditions which, if met, guarantee that the controlled-language representation of a legal text is isomorphic to its source (24). Of the five, three are relevant for Logical English:

i Each legal source should be represented separately.

ii The representation preserves the structure of each legal source. This means that clauses in Logical English should roughly correspond to clauses in the source document.

iii The representation preserves traditional mutual relationships, references and connections between legal sources. That is to say, if a contract refers to “Section 2, paragraph 5”, there should be an equivalent reference incorporated in the Logical English translation.

The main benefit of isomorphism is that it will make verifying that LE contracts are faithful to the source text much easier. If a lawyer wishes to verify a section of a contract, they will only need to look at the equivalent section of the translation. Furthermore, future software tools can take advantage of isomorphic correspondence to make reading LE text even easier. For example, a visual editor can present the source text and LE translation side by side in two panes. Whenever the user highlights a section of the source text in one pane, the LE clauses which correspond to it can be highlighted in the other pane (and vice versa).

The main drawback of isomorphism is that, if the source legislation is badly written (say it is logically ambiguous), forcing a one-to-one correspondence with the translation will result in an unpredictable or impossible to run logic program. Overall, writers of LE programs should follow the principles of isomorphism as a rule of thumb, but abandon them when necessary.

3.4 Reactive rules and logic program clauses

Logic program clauses are statements of the form *HEAD if BODY* while production rules take the from *If CONDITIONS then DO ACTIONS*. At a first glance, it might seem that these are just two ways of expressing an equivalent conditional; however, Kowalski and Sadri show that one set of rules cannot be reduced to the other (25).

Computation with logic program clauses proceeds via queries. Whenever a query matches the $HEAD$ of a rule, the query is evaluated as *True* if the $BODY$ of the rule can be satisfied. Thus, logic programs can be thought of as a way to check whether a given statement is a logical consequence of a set of causal laws.

Reactive rules, on the other hand, are most naturally interpreted as goals for agents. An agent attempts to do the *ACTIONS* of a rule whenever the *CONDITIONS* are fulfilled. Computation here is analogous to trying to solve an abduction task, where new facts are introduced in an attempt to make all reactive rules true.

Prolog, through an extension language called LPS, can support both types of computation described above. However, in the current project I restrict my attention to representing logic programs only. This is because my main use-case is translating legal contracts, and contracts are naturally viewed as "static" sets of rules rather than as agents who "want" to perform actions or make states of affairs true.

Let me try to clarify what I mean by "static" rules. As stated by Flood and Goodenough, legal contracts specify one "happy sequence" of events, various unhappy ones, and the different ways in which agents can access one sequence from another (19). For example, if the borrower in a loan contract fails to make a payment, the parties to the contract leave the happy sequence of events and enter the "borrower default" sequence, with all its reprecussions.

The lender and borrower are agents who perform actions (and observe events), and their actions depend on the specific sequence of events they find themselves in.

The contract, on the other hand, has no goals and expresses no preference for one sequence over another. The contract is just a set of causal laws which describe how the world can change. Even though the states the agents are in may change, the clauses of the contract do not. In that sense, it is better to view contracts "static", and represent them with logic clauses rather than reactive rules.

3.5 Logical connectives

Logical English has a small lexicon of reserved words. The most important of these are the logical connective words which indicate the form of the underlying logic programming clause.

Table 3.1 lists the key connectives and how they are represented at the LE-level. The examples in the table give a definitions of the concept "animal".

Connective	**LE example**	**Prolog form**
if	A thing is an animal **if** the thing is a person	type_is(animal, X) :- type_is(person, X).
and	A thing is an animal if the thing moves **, and** the thing reproduces.	type_is(animal, X) :- moves(X), reproduces(X).
exclusive or	A thing is an animal if **either** the thing is a bird **or** the thing is a fish.	type_is(animal, X) :- (type_is(bird, X); type_is(fish, X)), \+(type_is(bird, X), type_is(fish, X)).
inclusive or	A thing is an animal if either the thing can fly or the thing is a mammal, **or both**.	type_is(animal, X) :- (can_fly(X); type_is(mammal, X)).

Table 3.1: Connectives in Logical English

Five things are worthwhile noting here. First, because I restrict the head of every rule to be a single positive literal[2], and the body to a conjunction (possibly of complex disjunctions), there will always be at most one occurrence of "if" in a any LE sentence.

Second, "'and" will always be preceded by a comma or semi-colon when it plays the role of a logical connective. This will help the automated interpreter distinguish logical "and" from cases when "and" is a prepositions (line in "Water is made from hydrogen and oxygen.").

Third, in ordinary usage, the expression "either... or..." usually implies that both disjuncts cannot be the case together – e.g., a coin falls either heads or tails. In logic however, the usual "or" operator is inclusive. Logical English resolves this

[2]A literal is an atomic formula or its negation. Literals do not have logical connectives like "and", "or", or "if". They are the equivalent of an LE formula.

ambiguity by always assuming the exclusive disjunction is meant. For inclusive “or”, the additional “or both” must be added immediately after a disjunction, as shown in the example below:

```
% exclusive
A animal canFly if
either
  the animal is a bird
or
  the animal is a bat.

% inclusive
A person is content if
either
  the person is rich
or
  the person is healthy
or both.
```

I implement the “exclusive or” connective using the “weak negation” operator (also known as negation as failure or NAF) in Prolog. Discussion of negation is omitted here and presented in Section 3.8.

Fourth, a further issue with “either... or...” arises in cases where one of the disjuncts is itself a conjunction. Consider:

```
A person is content if
either
  the person is healthy
or
  the person is rich , and
  the person is loved.

A person is content if
either
  the person is healthy
or
  the person is rich;
and
the person is loved.
```

The first sentence says that one way for the person to be content is to be healthy. The second sentence stipulates that the person needs to be both healthy and loved, with the “loved” condition being a requirement regardless of which part of the disjunction turns out to be true. In Logical English we will use either a semi-colon or a full stop

to demarcate the end of a disjunction. The word “either” demarcates the beginning of the first disjunct, and first occurrence of the word “or” demarcates the end of the first disjunct and the beginning of the second. The end of the second disjunct is marked by “;” or “.”.

The above rules are sufficient for an automated interpreter to disambiguate disjunctions, but the formulation might still be confusing for casual readers of the LE program. The pink boxes above demonstrate how indentation can be used in the LE editor to improve human readability. Indentation is used in legal drafting with positive results (21).

Finally, although disjunctions can be complex, they cannot be nested arbitrarily many times. That is to say, every disjunct in Logical English is either a literal, or a conjunction of literals. Sentences such as the one below will not be permitted:

```
A cat is Schrodinger's cat if
either
  the cat is in a box and
  either
    the cat is dead
  or
    the cat is alive
  or both
or
  the cat is Schrodinger's pet.
```

This restriction makes writing an interpreter simpler, without taking away much (if any) expressive power from Logical English. Expressive power is not lost because there is usually a way to paraphrase a statement to avoid nesting. The example above can be re-written as:

```
1  A cat is schrodingersCat if
2  the cat is in a box , and
3  either
4    the cat is dead
5  or
6    the cat is alive.
7
8  A cat is schrodingersCat if
9  the cat is a pet to Schrodinger.
```

3.6 Infix and prefix notation

Formal logic uses parentheses to group similar concepts together, while (written) English makes use of punctuation, prepositions, and context. Consider the sentences:

```
John gives a book to Mary.
gives(john, mary, book).
```

The first sentence uses infix notation and is more easily understood by English speakers than the second. The second sentence is easier for computers to process as the parentheses clearly separate the predicate name from the arguments, and group the arguments together.

To aid readability, Logical English uses infix notation. The mechanisms through which infix notation at the LE-level is converted to Prolog were discussed in Section 1.3. The key is distinguishing verbs, which become predicate names, from noun phrases (e.g., a book, the GoodBook), which become predicate terms.

3.7 Reserved syntax

General reserved syntax

Apart from logical connectives, the LE lexicon (which is included in every LE program) includes determiners, prepositions, the phrases "that" and "so that" (for nested predicates), and the word "not" as a constant used in strong negation.

Importantly, the lexicon also includes facilities to deal with simple arithmetic. Numbers (including dates) are treated like constants. Simple arithmetic operations can be invoked with words like "equals", "greater than", "smaller than", "before" and "after" (for dates), "plus", and "minus".

Special Prolog predicates

The LE lexicon also reserves a small number of phrases for expressing some frequently used, predefined Prolog predicates. A good example is the *findall(+Tempalte, :Goal, -Bag)* predicate which finds all groundings of "Template" that can prove "Goal", and places them in the list "Bag". This is actually another example of a meta-predicate which takes in a predicate as its "Tempalte" argument. If we want to make use of *findall/3* in LE, we need some reserved syntax which maps to it. For example, the fragment:

```
1  ...a listOfPayments is a list containing
2  any payment such that
3  a party owes a counterparty a payment on a date...
```

will translate to:

```
1    ...findall(Payment,
2    owes(Party, Counterparty, Payment, Date),
3    ListOfPayments),
4    ...
```

3.8 Strong and weak negation

There are two types of negation in logic – strong and weak. For a proposition "P", weak negation corresponds to "it cannot be shown that P" and is represented as \+P. Strong negation corresponds to "it is not the case that P" and is represented as $\neg P$.

Most logic programming implementations, including Prolog, make the so-called "closed world assumption". Given a database of rules and facts, the assumption states that all facts which are not present in the database, and cannot be derived from the current rules, are false. The closed world assumption effectively reduces the strong negation to a weak negation with the rule $\neg P \ if \ $ \+P.

However, strong and weak negation are not the same. Specifically, strong negation is monotonic while weak negation is not. Consider:

```
It cannot be shown that there is a teacup orbiting the Sun.
It is not the case that 2+2=5.
```

The statement "there is a tea cup orbiting the Sun" can change its truth-value if we find evidence for it (e.g., by going to the Sun and seeing an indestructible teacup). On the other hand, there is no amount of evidence that can make $2+2 = 5$ true.

In legal texts, both negations are frequently used. Consider the sentences:

```
1. The plaintiff is innocent.  (positive fact)
2. The plaintiff is not innocent. (strong negation)
3. A plaintiff is innocent until proven otherwise. (weak negation)
```

Any of the three sentence can appear in legal texts and be required in the right circumstances. Thus, Logical English should be able to express both strong and weak negation. Consider the LE translation of sentences 1-3:

```
1  Plaintiff is innocent.
2  Plaintiff is not innocent.
3  Plaintiff is innocent if
4  it cannot be shown that
5  plaintiff is not innocent.
6
7  Contradiction if
8  a person is innocent and
9  the person is not innocent.
```

In Prolog this will become:

```
1  is_(innocent, plaintiff).
2  is_(not, innocent, plaintiff).
3  is_(innocent, plaintiff) :-
```

```
4     \+is_(not, innocent, plaintiff).
5   contradiction :-
6     is_(innocent, X), is_(not, innocent, X).
```

The above solution treats the strong negation of a predicate as a new predicate which has the same name but one more argument place taken by the constant "not". The rule from line 7 of the pink box is an instance of the general rule of excluded middle (something either is the case or it is not the case). The rule of excluded middle is implicit in every LE program. Whenever the constant "contradiction" is derivable form our program, we know that there is an inconsistency and the program needs to be changed. An advantage of Prolog as low-level language is that we can trace the contradiction back to the top of the proof tree using the *trace/0* predicate. This will make remedying contradictions significantly easier.

Kowalski and Sadri present another useful way to link strong and weak negation (26). Consider:

```
1   A plaintiff is innocent if
2   it cannot be shown that
3   the plaintiff is not innocent.
```

which translates to:

```
1   type_is(innocent, Plaintiff) :-
2     \+type_is(not, innocent, Plaintiff).
```

This rule ensures that the positive predicate is the default – *type_is(innocent, Plaintiff)* will always be provable unless *type_is(not, innocent, Plaintiff)* can be proven. Default assumptions are often used in law, with the classic example being "innocent until proven guilty". They are used to shift the burden of proof on the most appropriate party. As another example consider the two rules:

```
1   An obligation is satisfied on a date if
2   it cannot be shown that
3   the obligation is not satisfied on the date.
4
5   An obligation is satisfied on a date if
6   the obligation is an action ,and
7   the obligation happens on the date.
```

If an obligation has "obligor" and a "beneficiary", the first rule places the burden of proof on the beneficiary and the second on the obligor. During litigation, in the first case, the beneficiary will have to try (and fail) to prove that "the obligation is

not satisfied on the date" and vice versa. This shows that the interplay of strong and weak negation can have legal implications, and needs to be carefully considered when legal texts are being translated to Logical English.

3.9 Default reasoning, rules and exceptions

In logic programs rules are inviolable, while in everyday English, most rules have exceptions or qualifications which can override or make the rules inapplicable in certain circumstances. This section defines and discusses three distinct ways in which rules can be become inapplicable.

Exceptions and rebutting arguments

Consider the sentences:

```
All birds can fly.
Penguins are birds but cannot fly.
Jojo is a penguin.
```

English speakers will agree that the statements above are consistent, with the first statement being a rule which happens to have some exceptions. However, if translated to predicate logic, the statements will be inconsistent.

$canfly(X) \leftarrow bird(X).$
$bird(X) \leftarrow penguin(X).$
$\neg canfly(X) \leftarrow penguin(X).$
$penguin(jojo).$

The problem is that we can derive both $canfly(jojo)$ and $\neg canfly(jojo)$, which is a contradiction. We can get around this problem by specifying exceptions to the first rule as conditions:

$$canfly(X) \leftarrow bird(X) \wedge \neg penguin(X).$$

However, this will quickly get cumbersome, as more relevant exceptions become available – for example, if we start talking about ostriches, emus, and other flightless birds, we need to keep adding conditions:

$$canfly(X) \leftarrow \\ bird(X) \wedge \neg penguin(X) \wedge \neg ostrich(X) \wedge \neg emu(X)...$$

The convenience of natural language is that exceptions are only mentioned when they become relevant, allowing for simpler statements of rules. Logical English should preserve this feature.

I follow the approach of Kowalski and Sadri, using the interplay between strong and weak negation to handle exceptions in LE (26). There is no strong negation in Prolog, but something similar can be achieved, as shown in the previous section. We can re-write the statements above so they are consistent even when expressed in predicate logics:

```
1  canfly(X) :- bird(X), \+canfly(not, X).
2  bird(X) :- penguin(X).
3  canfly(not, X) :- penguin(X).
4  penguin(jojo).
```

Here we assume that *canfly(X)* is the default predicate, which holds unless *canfly(not, X)* is explicitly added. More generally, all we needed to do to allow for exceptions to a default rule is to add as an extra condition a weakly negated predicate which takes the same arguments (with an additional "not" as first argument) as the predicate in the head of the rule. Later on, we can have multiple definitions of this predicate which encapsulate the various exceptions.

In the language of argumentation theory (27) the rule about birds flying is called an argument and the exceptions to it are called rebuttals. For example, the fact that Jojo is a penguin is a rebuttal to the argument that Jojo can fly because it directly proves the opposite – that Jojo cannot fly.

Undermining arguments

The rule:

```
1  canfly(X) :- bird(X), \+ canfly(not, X).
```

from the previous subsection hints at a different way through which arguments can be attacked. Specifically, to attack the rule that all birds can fly, we don't always need to prove that certain birds cannot fly. Instead, we can just undermine the rule by arguing that it is not applicable in the current situation.

For concreteness, let us consider a simplified version of the British nationality act. Suppose there are two ways for a person to become a British citizen – either on the basis of birth, but only if they were born after the year 1980, or on the basis of marriage:

1. A person is a British citizen if they are born in the UK
   and it cannot be shown that rule 1 does not apply to the person.
2. A person is a British citizen if they have been married to a
   British citizen for at least 3 years.
3. Rule 1 does not apply to a person if the person is born before 1980.

Suppose that Jojo claims to be a British citizen on the grounds of the first rule. We can attack his claim on the grounds of the third rule by producing evidence that he was born in 1965. The important difference here is that our attack does not conclusively disprove Jojo's claim to Britishness as Jojo can still qualify under rule 2. Our attack just shows that rule 1 is not applicable in the current case.

The method for dealing with exceptions outlined in the previous section works equally well for the case of undermining arguments. Indeed, the Prolog implementation for both cases is the same. The general form of logical rules which can be violated is:

```
P if Q and it cannot be shown that S.
```

Overriding rules and rule hierarchies

A related but separate concept is the idea that some rules might be "switched" on and off. In contract law, parties frequently make use of templates. Templates are contracts with unalterable, standard clauses. The standardisation brings clarity and predictability to how the contracts are interpreted but reduces flexibility. An example of a contract template is the ISDA Master Agreement (see Chapter 5). The only words that can be altered in an ISDA Agreement are the names of the signatories and the date of the start of the agreement.

One way to allow parties some negotiating flexibility, while still keeping the benefits of standardisation, is for templates to list several variations of the same clause and let the signatories decide which ones they wish to apply. Effectively, clauses can be switched on or off, depending on the specific circumstances.

Consider a contract template which has the following clause variations, adapted from (19):

> CLAUSE X: At the request of the Borrower to be given on July 1 2014, the Lender will advance $1000 to the borrower on **July 2, 2014**.
>
> CLAUSE XI: At the request of the Borrower to be given on July 1 2014, the Lender will advance $1000 to the borrower on **July 9, 2014**.

Clause XI gives the lender a week longer to transfer the loan than Clause X. If we wish to translate the contract template in Logical English, we need to state both clauses but allow some mechanism for parties to switch these clauses on or off at will. We do so by giving each clause an identifier, and then stating which identifiers are "on". The translation will look similar to the following:

```
1  ClauseX: Lender pays Borrower 1000 pounds
2    on 20140702 if
3  Borrower requests 1000 pounds from lender
4    on 20140701 ,and
5  ClauseX isOn.
6
```

```
ClauseXI: Lender pays Borrower 1000 pounds
  on 20140709 if
Borrower requests 1000 pounds from lender
  on 20140701 ,and
ClauseXI isOn.

ClauseXI isOn.
```

The tags “ClauseX” and “ClauseXI” are used for the convenience of the reader and disappear at the Prolog level where it is enough to just have an extra condition.

```
pays(lender, borrower, 1000, 20140702) :-
  requests(borrower, lender, 1000, 20140701),
  isOn(clauseX).

pays(lender, borrower, 1000, 20140709) :-
  requests(borrower, lender, 1000, 20140701),
  ison(clauseXI).

isOn(clauseXI).
```

The fact in the last line of the grey box can be used in the proof of ClauseXI. Because there is no equivalent fact for ClauseX, the clause cannot be proved and is effectively switched off.

The above three methods for handling exceptions in Logical English can be used to explicitly build hierarchies of rules – something which is often left implicit in legal contracts (see Chapter 5).

3.10 Tense

Cormacain notes that well-drafted legislation should be written in the present tense because “we should write the law as if it will be applied right now” (21). Using a uniform tense will make Logical English programs easier to interpret by readers. It will also make LE easier to implement, as we will not need to worry about verb conjugations, and will not have to define explicit relationships between predicate names like “is”, “was”, and “will be”. For these reasons, Logical English is always written in the present tense.

3.11 Reification, and nominalisation, and meta-predicates

The use of reification in logic takes place when predicates are treated as terms. This is useful because it allows us to talk about properties of predicates. For example, the

sentences:

```
Mary wins the race.
Mary wants to win the race.
```

can be formalised in Prolog as:

```
1  wins(mary, race).
2  wants(mary, wins(mary, race)).
```

The predicate *wins/2* is reified in the second sentence and occurs as a term in *wants/2*. When a predicate carries information about other predicates, rather than about individuals, it is called a meta-predicate. Meta-predicates can be very expressive. For example, sentence stating that Mary achieves everything she wants:

```
1  achieves(mary, Something) :-
2    wants(mary, Something).
```

carries information about all one-place predicates which apply to Mary.

In Logical English reification can be dealt with in two ways – using nominlasation or using meta-predicates

Nominalisation

In natural language, nominalisation is when a verb or another part of speech is converted to a noun. Consider the English sentences:

```
1. Bank lends Corp 100 pounds at 5% monthly interest.
2. All loans carry maximum interest of 10%.  (reification of 1)
```

The lending relationship from the first sentence is treated as a noun in the second. This allows us to succinctly talk about the properties of all lending relationships.

Nominalisation brings the benefits of reification at the Logical English level without using meta-predicates at the Prolog level. So, for example, we can paraphrase sentences 1-3 in LE in the following way:

```
1  % using "extends a loan" in place of "lends"
2  Bank extends Loan01 to Corp.
3  Loan01 hasValue of 100.
4  Loan01 is denominated in GBP.
5  Loan01 is a loan.
6  Loan01 hasMonthlyInterest of 5 percent.
7
8  % this holds of all entities of type "loan"
```

```
9  A thing hasMaximumMonthlyInterest of 10 percent if
10 the thing is a loan.
```

At the Prolog level this becomes:

```
1  extends(bank, loan01, corp).
2  hasvalue(loan01, 100).
3  type_is(denominated, loan01, gbp).
4  type_is(loan, loan01).
5  hasMonthlyInterest(loan01, 5, percent)
6
7  hasMaximumMonthlyInterest(Thing, 10, percent) :-
8    type_is(loan, Thing).
```

Notice that at the LE level, sentence 1 was split into four separate statements (lines 2-6 of pink box). Each new statement was shorter and more explicit than sentence 1, with the underlying predicates having fewer arguments than the English source. Notice also that we did not need to use meta-predicates at the Prolog level. Nominalisation effectively is a style of writing in LE rather than a formal grammatical feature. The general strategy for "nominalising" relationships is the following:

- We find a common noun that intuitively represents the relationship we are considering. For example:

  ```
  pays -> makes a payment
  delivers -> makes a delivery
  decides -> makes a decision
  presents -> makes a presentation
  ```

- Each particular relationship we wish to discuss gets an identifier (like "payment01", or "decision05").
- We can then "attach" additional properties to individual relationships by using binary predicates: *Payment01 hasValue of 100 pounds.*; *Payment01 is denominated in GBP.* This follows Kowalski's discussion of binary vs n-ary predicates (28).

Meta-predicates in LE

Nominalisation is a useful way to deal with reification; however, it is not always sufficiently expressive. Consider the sentences about Mary again.

```
Mary wins the race.
Mary wants to win the race.
```

With nominalisation, they can be translated into LE so:

```
1  Mary wants Victory01 for the Race.
2  Mary achieves Victory01.
```

Or like even so:

```
1  Mary wantsVicotry in the Race.
2  Mary achievesVictory in the Race.
```

Neither of the two translations preserves the semantic relationship between “wants” and “wins” from the source sentences. If we wish to preserve it, we can use some reserved syntax. In LE, the phrases “that” and “so that” signal that the words to follow should be treated both as a predicate and as a term in a meta-predicate. The meta-predicate is defined by the words before the “so that” or “that”. Consider:

```
1  Mary wishes that Mary wins the Race.
2  Mary wins the Race.
```

In Prolog this will become:

```
1  wishes(mary, wins(mary, race)).
2  wins(mary, race).
```

This syntax is naturally useful for sentences with adverb clauses. Consider again the example of Mary notifying her landlord:

```
Mary notifies her landlord that she's terminating her tenancy.
```

In LE this becomes:

```
1  Mary notifies MarysLandlord that
2   Mary terminates MarysTenancyAgreement.
```

And Prolog:

```
1  notifies(mary, marysLandlord,
2    terminates(mary, marysTenancyAgreement)).
```

3.12 Using types

Computer languages like C++ and Java are strongly typed. This means that every time the programmer declares a variable, she will need to specify whether this variable is a string, an integer, etc. Two of the main advantages of using types are efficiency and program readability. Efficiency improves when the compiler knows

immediately how much memory to allocate for each variable rather than having to guess from context. Also, typed code is more human readable, as each variable introduction carries additional information about the types of data the variable can hold.

LE is a typed language, with every common noun, except the word "thing", introducing a new type. Variables introduced with a given common noun can refer only to individuals of a certain type. At the Prolog level, types will be represented and enforced by the predicate *type_is(type, Individual)*. For example:

```
A person likes an animal if
the animal is cute.
```

Will translate to the Prolog:

```
likes(Person, Animal) :-
  is_(cute, Animal),
  type_is(animal, Animal),
  type_is(person, Person).
```

Notice that the Prolog has two additional conditions not present in the LE (lines 3-4 of the grey box). These conditions get added by the interpreter to ensure that types are enforced. So for example, we will not be able to prove that *likes(john, teddyBear)*. because even though the Teddy Bear is cute, it is not an animal.

Type conditions are added for every common noun except for the noun "thing". "thing" is a reserved word that stands in for a generic variable, so that:

```
A thing is an animal
if the thing is a bird.
```

will become:

```
type_is(animal, X) :-
  type_is(bird, X).
```

Types allow us to build type hierarchies which categorise individuals. In the example above, we stated that "bird" is a sub-type of "animal". Thus, individuals can have multiple types.

The above rule can be stated more naturally in LE with the following construction:

```
A bird is an animal.
```

At lower levels of translation, this construction will expand back to *A thing is an animal if the thing is a bird*.

3.13 Three uses of the verb "is"

In Logical English, the verb "is" can be used in three different ways:

- To explicitly link adjectives to nouns – *The Hat is red*.
- To create type hierarchies – *A bird is an animal*.
- To express relationships – *Mary is the mother of John*; *Mary is late for the Meeting*.

All three uses are treated differently at the Prolog level. In the first case, when "is" is followed by a non-capitalised word without a determiner, at the Prolog level we use the predicate *is_/2*, like in *is_(red, hat)*. We have already seen that the second case translates down to a *type_is/2* predicate. The third use is more interesting. The constructions [first common noun + is + second common noun + ...] and [first common noun + adjective + ...] are used to name relationships between individuals. The first noun/adjective after the verb is interpreted as the constant name of the relationship (not as a variable). In Prolog:

```
1  relat_is(mother, mary, john).
2  relat_is(late, mary, meeting).
```

In effect, *relat_is/n* is a family of predicates with arbitrary arity greater than 2. *relat_is/n* introduces some flexibility at the LE level. For example, the sentence:

```
Mary is late for the meeting.
```

can be translated to LE in two ways:

```
1  Mary is late for the Meeting.
2  Mary isLate for the Meeting.
```

depending on what the writer finds more convenient.

Chapter 4

A Prolog interpreter for a Simplified Logical English

In this chapter, I present an extensible interpreter which can automatically translate Simplified Logical English sentences into executable Prolog code. Simplified Logical English (SLE) has fewer features than the Logical English described in Chapter 1. Specifically, SLE does not incorporate reference resolution using ordinals, and it does not support syntax for predefined Prolog predicates like *findall/3* and operators like weak negation. SLE does support the basic arithmetic operations with natural numbers. The SLE interpreter is designed to be extensible, so that the missing features can be introduced in future development work.

4.1 Description

The core of the interpreter is the predicate *parse(Input, Output)* which takes an SLE string, analyses it, converts it to a list of Prolog rules, and updates the global environment by asserting those rules. The "Output" argument, which is optional, contains the list of Prolog rules resulting from the analysis of the input string.

For improved readability, I also define the prefix operator *translate*. The operator can be applied to strings of text. In queries, *translate* has the effect of invoking the *parse/2* predicate with an anonymous "Output" argument. The grey box below shows how Prolog will answer queries using both the *parse/2* predicate and the *translate* operator.

```
?- parse("A thing is an animal if the thing is a person.",Output).
Output = [(type_is(animal,_7010):-type_is(person,_7010))].

?-  translate "A thing is an animal if the thing is a person.".
true.
% "Output" is anonymous and the resulting rules are not displayed
```

Figure 4.1 is a schematic representation of how the simple interpreter works. For simplicity, a single sentence is presented. The interpreter can process individual sentences as well as blocks of text containing multiple sentences separated by full stops.

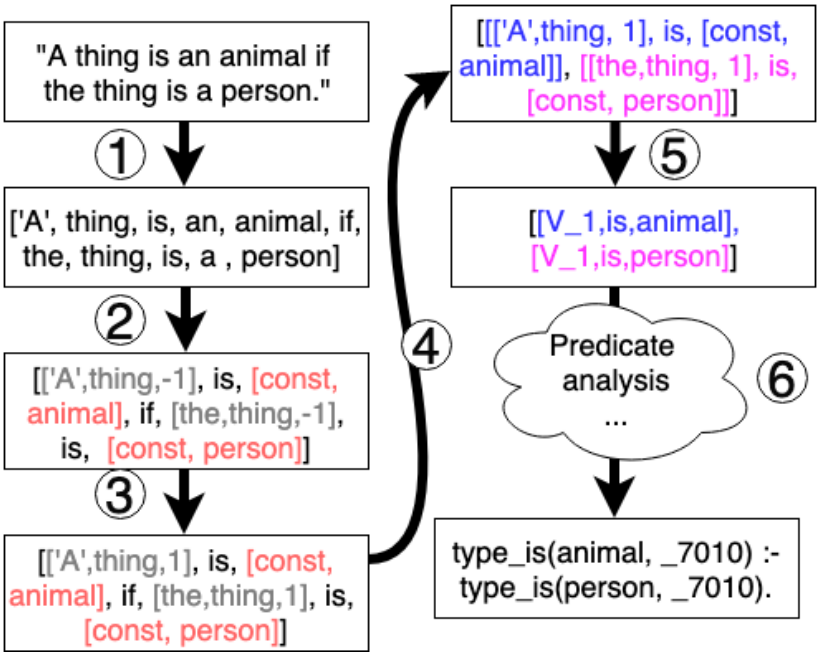


Figure 4.1: Workflow of the simple LE interpreter

Step 1 converts an LE sentence to a Prolog list of atoms. Capitalisation in atoms is preserved (note the 'A'), as capital letters in Logical English may be used to signal constants. Capitalised determiners, however, are correctly recognised as such.

In step 2, we identify variables and constants. At the SLE level, variables are introduced by determiner-common noun pairs. At this step, variables are represented by lists with three elements – *[determiner, common noun, variable index]*. The variable index is used in the next steps for resolving definite determiners.

Constants are introduced by a capitalised word or with the "is a common noun" construction. In step 2, they are represented by lists of two elements (the fist element is the word "const" and the second is the constant's name). Thus, "a person" in our example is treated as a constant because it is preceded by the word "is".

In step 3, we count the first occurrence of each new variables and update the index. Terms which begin with an indefinite determiner ("a", "an", "some") are assumed to be new variables, with each new variable receiving an incremented index. For example, because *['A', thing, 1]* is the first new variable encountered, its index is set to 1.

Variables which begin with a definite determiner are assumed to be repeat occurrences of variables already introduced earlier in the same sentence. In the case of *[the, thing, 1]*, the interpreter looks for the first occurrence a variable structure

containing the common noun “thing”, and attaches the same index to the repeat variable. All variable indices are updated on the same pass though the sentence and no backtracking is required. If we wish to add a facility for resolving variable references with ordinals, here would be the appropriate place for it.

Step 4 checks whether the sentence being analysed has the form *HEAD.* or *HEAD if BODY.* In the latter case, it separates the head from the body around the word “if”. This means that the interpreter allows for at most one occurrence of the word “if” per sentence. Note that if the sentence is a fact (i.e., *HEAD.*), variables are disallowed.

The interpreter has a facility to parse multiple “and” and “or” conditions. A semi-colon is used to demarcate the scope of these conditions whenever the scope is ambiguous.

Step 5 converts the lists representing variables and constants to atoms. Note that here variables are capitalised, while constants begin with a lower-case letter. Thus, we have made the transition from the SLE syntax to the Prolog syntax. The analysis and construction of predicate arguments is completed, and we turn to constructing the predicate symbol.

Step 6 is responsible for analysing predicate symbols and adding logical connectives. The terms identified in the previous steps are extracted to a separate list, and it is assumed that the remaining words form part of the predicate. The simple interpreter assumes there are only two kinds of predicates – those introduced with the verb “is”, corresponding to the *type_is/2*, *is_/2* and *relat_is/n*, and all other predicates. Predicate symbols from the second group are constructed by simply appending words. For example, “Max runs away” will be translated to “runs_away(max)”.

Step 6 is one place where the simple interpreter can be extended to accommodate more expressive predicates. Specifically, the checks for reserved syntax for various predefined Prolog predicates can take place here.

Once all predicate symbols are identified, a list structure is created for each predicate. The list contains the predicate symbol as a first element, followed by the terms. The predicate structures are added as elements to a larger list, and logical connectives are inserted at appropriate places. For example, towards the end of the predicate analysis step, Prolog might produce the following structure:

```
1  [[is_, slow, 'V_1'], ":-", [is_, turtle, 'V_1']]
```

Finally, the actual predicate is produced through a series of atom-to-string conversions and string concatenations:

```
1    "is_(slow, V_1) :- is_(turtle, V_1)"
```

The string is then converted to a rule using the bi-directional *term_string/2* predicate, and it is added to the current database using the *assert/1* predicate.

4.2 A simple example

To demonstrate how the SLE interpreter works in practice, consider a simple example – defining what an "ancestor" means using Logical English.

Consider the following text:

> An ancestor is a parent or (recursively) the parent of an ancestor (i.e., a grandparent, great-grandparent, great-great-grandparent, and so forth) (29).
>
> John is the father of Molly who is the mother of Janet.
>
> Is John an ancestor of Janet? Who are the ancestors in the example?

We are interested in paraphrasing the above definition and fact into SLE so that we can answer the questions posed at the end.

The following SLE sentences define the ancestor relationship and state the fact about John, Molly, and Janet:

```
A person is an ancestor of another person2 if
the person is a parent of the person2.

A person is an ancestor of another person2 if
the person is an ancestor of a person3 and
the person3 is a parent of the person2.

A person is a parent of another person2 if
the person is a mother of the person2 or
the person is a father of the person2.

John is a father of Molly.
Molly is the mother of Janet.
```

Note also that SLE makes use of "person1", "person2", etc. because the simple interpreter does not yet have the facility to deal with ordinals. Instead, we use different variable names to distinguish variable references. Note also that "another" is treated as an indefinite determiner.

At the Prolog level, we automatically interpret the Logical English text by passing it as a string in a query with the *translate* operator:

```
?- translate "A person is an ancestor of another person2 if
the person is a parent of the person2. A person is
an ancestor of another person2 if the person is an ancestor of
a person3 and the person3 is a parent of the person2.
John is a father of Molly. A person is a parent of
another person2 if the person is a mother of
the person2 or the person is a father of the person2.
```

```
8  Molly is the mother of Janet.".
9
10 true.
```

After entering the query, Prolog will respond with *true.*, meaning the string was accepted and assimilated.

If we wish to inspect the exact rules which went into the Prolog database, we can use the *parse/2* predicate, whose second argument unifies with the list of translated rules:

```
1  ?- parse( "A person is ..." , ListOfRules).
2
3  ListOfRules = [
4       relat_is(father,john,molly),
5
6       relat_is(mother,molly,janet),
7
8       (relat_is(ancestor,_18120,_18122):-
9         relat_is(parent,_18120,_18122)),
10
11      (relat_is(ancestor,_20946,_20948):-
12        relat_is(parent,_20954,_20948),
13        relat_is(ancestor,_20946,_20954)),
14
15      (relat_is(parent,_24422,_24424):-
16        relat_is(mother,_24422,_24424) ;
17        relat_is(father,_24422,_24424))
18 ].
```

“ListOfRules” lists the Prolog sentences in the order in which they are processed. Notice that facts are processed before rules, regardless of how the sentences in the source SLE string were ordered. Notice also that rules which have a recursive condition always place that condition last. These two features of the interpreter reduce the chance that Prolog loops when asked a query.

Looping in Prolog occurs when a rule keeps being available for selection for resolution and the empty clause is never reached. For example the below database and query:

```
1  p :- p.
2  ?- p.
3  | % Prolog loops until stack limit is exceeded
```

will cause an infinite loop because `:-p` can always be resolved with `p:-p`. LE interpreters should avoid looping conditions.

Finally, once the LE text has been converted to rules and added to a Prolog database, we can ask the questions we wanted answered in the first place. However, the questions are not themselves formulated in SLE. Neither SLE nor LE yet have the facility to deal with questions. This is an important area for future development.

```
?- relat_is(ancestor, Ancestor, Descendant).
Ancestor = molly,
Descendant = janet ;
Ancestor = john,
Descendant = molly ;
Ancestor = john,
Descendant = janet ;
false.
```

Prolog will answer the question of who is an ancestor of whom by listing all possible ancestor-descendant pairs, and then stopping. This demonstrates the ways in which LE can be used to automate legal analysis.

Chapter 5

A case study: The ISDA Master Agreement

This chapter shows that Logical English is expressive enough to formalise real-world financial contracts. I translate key parts of the ISDA Master Agreement – a standard and widely used financial contract – in both LE and Prolog.

5.1 Financial background

A derivative is a financial contract whose value is determined by the performance of an underlying asset, contract, index or rate. A common example is the GBP interest rate swap agreed between Party A and Party B. The swap is built around a sum of money called a notional. Party A agrees to transfer to Party B a fixed percentage, say 5%, of the notional sum, say 100 pounds, at future dates 1, 2, and 3. Party B agrees to transfer Party A a floating percentage of the notional at those same dates. The floating percentage owed by Party B on a given date is equal to the interest rate set by the Bank of the England on that date. Thus, the value of the swap depends on the underlying Bank rate. Party A will benefit if the Bank sets a rate above 5%, and vice versa. Figure 5.1 shows one possible example of how the swap transaction may unfold. In it, Party B ends up making £1 at the end of the transaction, as it pays a total of £14 but receives £15.

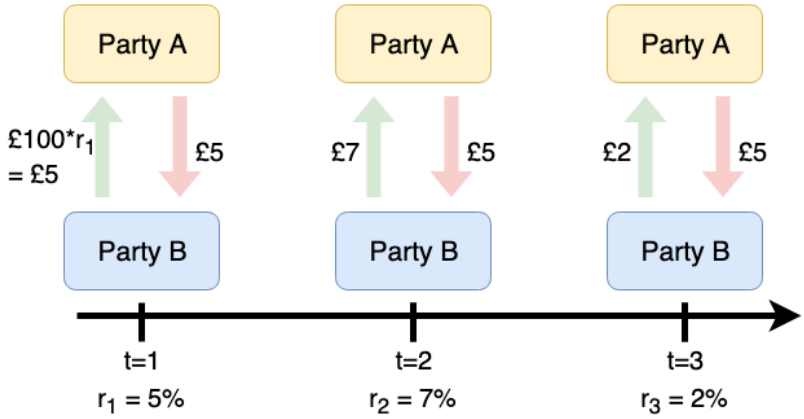


Figure 5.1: An example of a swap transaction with three payment dates

When two parties negotiate a derivative contract directly, without the involvement of an intermediary, such a stock exchange, their contact is said to be over the counter (OTC). The ISDA Master Agreement is a document which sets out some standard terms to apply to all OTC derivatives between two parties. The standardisation makes writing, interpreting, and following a given contract substantially cheaper and easier.

The ISDA Agreement comprises two legal documents – a Master Agreement and a Schedule. The text of the Master Agreement cannot be changed. Once signed, it sets out the general terms of all currently active and future derivative contracts between the signatory parties. The parties cannot enter into derivative contracts outside the Master Agreement.

Even though the text of the Master Agreement cannot be changed, some modification to specific terms is possible. These modification can be specified in the Schedule document. If the parties elect to use a Schedule, the terms set out in it supersede any conflicting terms from the Master Agreement.

The Master and Schedule stipulate only generic terms and do not reference any specific derivatives. When two parties want to enter into a specific derivative contract, referred to as a Transaction in the ISDA text, they sign a document called a Confirmation. In our example, the swap agreement is a single Transaction which stipulates payments to be made at dates 1, 2, and 3. Each new Transaction will require a new Confirmation but will be bound by the same Master Agreement and Schedule, as shown in Figure 5.2.

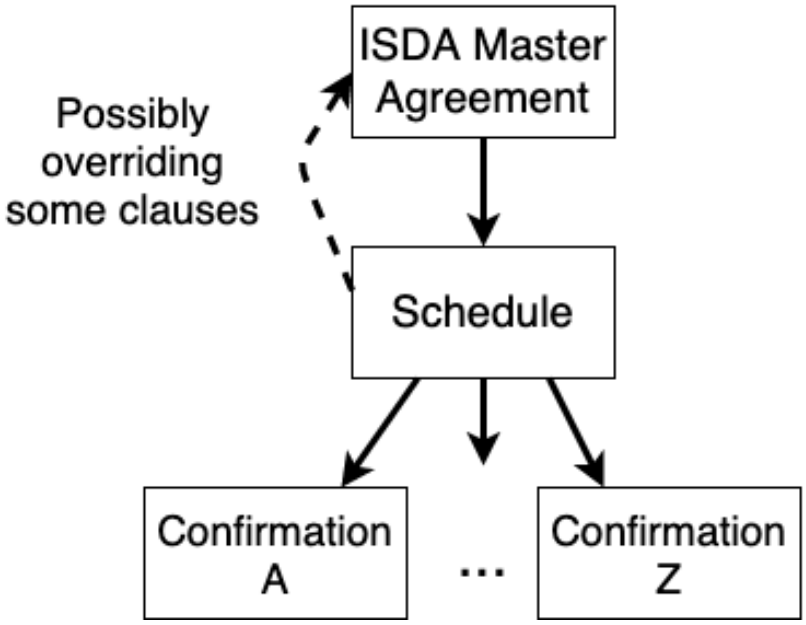


Figure 5.2: Structure of ISDA Agreement and related documents

In the remainder of this chapter, I will go through the first three sections of an ISDA Master Agreement, signed between two imaginary entities, Bank and Corp, on 22 March 2011. The full text of the agreement is taken from the US Securities and Exchange Commission website (30).

5.2 Preamble

The preamble to the ISDA Master Agreement states:

> [Contract] ...dated as of March 22, 2011. Bank and Corp have entered and/or anticipate entering into one or more transactions (each a "Transaction") that are or will be governed by this 2002 Master Agreement, which includes the schedule (the "Schedule"), and the documents and other confirming evidence (each a "Confirmation") exchanged between the parties or otherwise effective for the purpose of confirming or evidencing those Transactions. This 2002 Master Agreement and the Schedule are together referred to as this "Master Agreement".

This can be translated into the Logical English clauses:

```
1  Bank is a party. Corp is a party.
2  A thing is a counterparty if
3    the thing is a party.
4  IsdaAgreement is an agreement between Bank and Corp.
5  IsdaAgreement is an agreement between Corp and Bank.
6  IsdaAgreement commences on 2011/03/22.
```

And the Prolog clauses:

```
1  type_is(party, bank). type_is(party, corp).
2  type_is(counterparty, X) :-
3    type_is(party, X).
4  relat_is(agreement, isdaAgreement, bank, corp).
5  relat_is(agreement, isdaAgreement, corp, bank).
6  commences(isdaAgreement, 20110322).
```

The first three clauses (lines 1-3 in both grey and pink boxes) are type assignments. They introduce the crucial entities “Bank” and “Corp” and start building a type hierarchy. The two subsequent clauses together state that the ISDA agreement is a symmetric relationship between Bank and Corp. The symmetry is implied in the source text, but needs to be stated explictly in the Logical English.

Note that “isdaAgreement” here does not name a separate class of agreements, but, in fact, is the name of the specific agreement signed between Bank and Corp on 22 March 2011. This works well because, by law, there can only ever be a single ISDA agreement between two parties. We give it a name, so we can refer to it later.

The preamble also states that the ISDA Master agreement comprises the agreement itself, the Schedule and every Confirmation of a transaction. Practically, separate documents will be represented in separate files and sourced from the main file using the Prolog *:-include(File).* directive. Included files can be specified with the aid of reserved syntax corresponding to *:-include(File)*.

5.3 Section 1: Interpretation

The ISDA Master Agreement continues:

> (a) Definitions. The terms defined in Section 14 and elsewhere in this Master Agreement will have the meanings therein specified for the purpose of this Master Agreement.
>
> (b) Inconsistency. In the event of any inconsistency between the provisions of the Schedule and the other provisions of this Master Agreement, the Schedule will prevail. In the event of any inconsistency between the provisions of any Confirmation and this Master Agreement, such Confirmation will prevail for the purpose of the relevant Transaction.
>
> (c) Single Agreement. All Transactions are entered into in reliance on the fact that this Master Agreement and all Confirmations form a single agreement between the parties (collectively referred to as this ”Agreement”), and the parties would not otherwise enter into any Transactions.

Part (a) is an example of source text that can be omitted in the LE formulation. As we are not using pre-defined ontologies or dictionaries, the meanings of legal

terms will be explicitly defined in the LE program anyway, so there is no danger of misunderstanding.

Part (b) is trickier because it is in effect a meta-logical statement about which rules should take precedence over which others. One way to incorporate this into Logical English will be to follow the strategy outlined in Section 3.9. Every clause from the Master and Schedule can be tagged with an identifier (say, Master001, Schedule034, etc.) and turned "on" or "off" using the *ison/1* predicate.

A drawback of this approach, which can be addressed in future iteration of Logical English, is that we need to manually identify and resolve conflicts. The rule in part (b), on the other hand, is more general – it covers every possible conflict without having to identify any specific ones.

Part (c) states that no transactions between the signatory parties can exit outside the current ISDA agreement. In LE:

```
% rule 1
A transaction is governed by IsdaAgreement if
the transaction is a transaction between
  a party and a counterparty , and
IsdaAgreement is an agreement between
  the party and the counterparty , and
the transaction completes on a first date , and
IsdaAgreement commences on a second date , and
the first date is after the second date.
```

Or in Prolog:

```
% rule 1
relat_is(governed, Transaction, isdaAgreement) :-
  type_is(transaction, Transaction),
  relat_is(transaction, Transaction, Party, Counterparty),
  relat_is(agreement, isdaAgreement, Party, Counterparty),
  completes(Transaction, Date1),
  commences(isdaAgreement, Date2),
  Date1 #> Date2.
```

Rule 1 states that all transactions between Bank and Corp which complete after the current ISDA agreement is signed are governed by the agreement. The rule is important because it links the individuals IsdaAgreement, Bank, and Corp.

Transactions are symmetric, so we need to state explicitly in LE that:

```
A transaction is a transaction between a party and
  a counterparty if
```

```
3  the transaction is a transaction between
4    the counterparty and the party.
```

In Section 3.12 I mentioned that an automatic interpreter enforces types at the Prolog level by appending additional type conditions for every new common noun mentioned in the LE rule. For the sake of readability, I omit non-essential type conditions in the Prolog code in this chapter.

5.4 Section 2: Obligations

Section 2 of the agreement introduces the concept of obligations.

5.4.1 General Obligations

> (a) General Conditions.
>
> (i) Each party will make each payment or delivery specified in each Confirmation to be made by it, subject to the other provisions of this Agreement.

Part (a)(i) states that any payment or delivery specified in any transaction is obligatory. In LE:

```
1  % rule 2
2  An action is an obligation for a party if
3  a transaction specifies the action , and
4  the transaction is a transaction between the party and
5    a counterparty , and
6  the action is directed from the party to
7    the counterparty , and
8  either
9    the action is a payment or
10   the action is a delivery ;
11 and
12 it cannot be shown that
13 the action is not an obligation for the party.
```

In Prolog:

```
1  % rule 2
2  relat_is(obligation, Action, Party) :-
3    specifies(Transaction, Action),
4    relat_is(transaction, Transaction, Party, Counterparty),
5    relat_is(directed, Action, Party, Counterparty),
```

```
6     relat_is(governed, Transaction, isdaAgreement),
7     (
8      type_is(payment, Action ) ;
9      type_is(delivery, Action)
10     ),
11     \+(type_is(payment, Action ),
12        type_is(delivery, Action)), % exclusive or
13     \+relat_is(not, obligation, Action, Party).
```

In this case, the obligation is treated as a relationship between an action and an agent, and the action is treated as in individual rather than a predicate – an example of the nominalisation strategy discussed in Section 3.11.

Note the condition in lines 12-13 of the pink box. It makes use the interplay between weak and strong negation to handle exceptions to Rule 2. This will become useful when translating section 2(c) of the ISDA Agreement. 2(c) talks about situations where payments, which would otherwise be obligations, stop counting as such under certain conditions.

Finally, note the condition in lines 6-7 of the pink box, which talks about the direction of specific payments. Unlike transactions and contracts, payments are not symmetric – one party has to pay another. Again, this lack of symmetry is implicit in the source but needs to be stated explicitly in the LE translation.

With obligations defined, we naturally need to define rules which specify when a given obligation is satisfied or broken:

```
1  % rule 3
2  An action is satisfied on a date if
3  the action is an obligation , and
4  the action is due on a second date , and
5  the action happens on the second date , and
6  the second date is before the first date.
7
8  % rule 4
9  An action is an obligation if
10 the action is an obligation for a party.
11
12 % rule 5
13 An obligation is broken on a date if
14 it cannot be shown that
15 the obligation is satisfied on the date.
```

In Prolog:

```
% rule 3
relat_is(satisfied, Action, Date) :-
  relat_is(obligation, Action, Party),
  relat_is(due, Action, Date2),
  happens(Action, Date2),
  Date2 #<= Date.

% rule 4
type_is(obligation, Action) :-
    relat_is(obligation, Action, Party).

% rule 5
relat_is(broken, Obligation, Date1) :-
  \+relat_is(satisfied, Obligation, Date1).
```

As per the source text, Rule 3 states that an obligation is satisfied only when a payment or delivery is made exactly on the due date. Specifically, an early payment is not admissible. This is standard in financial transactions, where the value of securities can change drastically from day to day due to fluctuations in interest rates, exchange rates, and other economic variables. Furthermore, Rule 3 assumes that once an obligation is satisfied, it remains satisfied on all future dates.

Rule 4 is an auxiliary rule (not explicitly mentioned in the source text) which links a type name to a relationship name. This allows us to talk about obligations without mentioning who the obligation falls on. Rule 4 is useful in certain cases. For example, the statement "It is obligatory that payments are made on time" expresses an obligation but makes no commitments about who it falls on to satisfy this obligation.

Rule 5 states that obligations which are not clearly satisfied on a date should be considered broken on that date.

Continuing with the agreement, part (a)(ii) can be interpreted as specifying what it means for a payment or delivery to happen.

> (a) General Conditions.
>
> (ii) Payments under this Agreement will be made on the due date for value on that date in the place of the account specified in the relevant Confirmation or otherwise pursuant to this Agreement, in freely transferable funds and in the manner customary for payments in the required currency. Where settlement is by delivery (that is, other than by payment), such delivery will be made for receipt on the due date in the manner customary for the relevant obligation unless otherwise specified in the relevant Confirmation or elsewhere in this Agreement.

In Logical English:

```
% rule 6
A payment happens on a date if
the payment is valued an amount on the date , and
the payment is denominated in a currency , and
the payment is directed from a party
  to a counterparty , and
the party transfers the amount
  to an account on the date , and
the party transfersInCustomaryWay
  the amount  for the currency , and
the party transfersInFreelyTransferableFunds
  the amount , and
the account is correctlySpecified by the counterparty
  for a transaction for the payment , and
the transaction specifies the payment , and
the transaction is governed by IsdaAgreement.

% rule 7
A delivery happens on a date if
the delivery is directed from a party
  to a counterparty , and
a transaction specifies the delivery , and
the transaction is governed by IsdaAgreement , and
either
  the party makes the delivery for a receipt , and
  the party makesInCustomaryWay the delivery
or
  the party makesInPreSpecifiedWay the delivery.
```

In Prolog:

```
% rule 6
happens(Payment, Date) :-
  type_is(payment, Payment),
  relat_is(valued, Payment, Amount, Date),
  relat_is(denominated, Payment, Currency),
  relat_is(directed, Payment, Party, Counterparty),
  transfers(Party, Amount, Account, Date),
  transfersInCustomaryWay(Party, Amount, Currency),
  transfersInFreelyTransferableFunds(Party, Amount),
  relat_is(correctlySpecified, Account, Counterparty,
    Transaction, Payment),
```

```
12    specifies(Transaction, Payment),
13    relat_is(governed, Transaction, isdaAgreement).
14
15    % rule 7
16  happens(Delivery, Date) :-
17    type_is(delivery, Delivery),
18    relat_is(due, Delivery, Date),
19    relat_is(directed, Delivery, Party, Counterparty),
20    specifies(Transaction, Delivery),
21    relat_is(governed, Transaction, isdaAgreement),
22    (
23      (
24       type_is(receipt, Receipt),
25       makes(Party, Delivery, Receipt),
26       makesInCustomaryWay(Party, Delivery, customary)
27      );
28    makesInPreSpecifiedWay(Party, Delivery)
29    ). % exclusive or negation omitted for clarity
```

Note that some conditions, like what it means for a payment to be made in a customary way for a currency, are not defined here but will likely be defined either elsewhere in the document or in supporting documents. It is also possible that the definition of a "customary way" of paying is left the discretion of the parties. This would be an example of vagueness, as discussed in Section 3.2.

The last subsection of the general conditions specifies some situations in which an action which normally constitutes on obligation stops counting as such.

> (a) General Conditions.
>
> (iii) Each obligation of each party under Section 2(a)(i) is subject to (1) the condition precedent that no Event of Default or Potential Event of Default with respect to the other party has occurred and is continuing,(2) the condition precedent that no Early Termination Date in respect of the relevant Transaction has occurred or been effectively designated and (3) each other condition specified in this Agreement to be a condition precedent for the purpose of this Section 2(a)(iii).

In effect section 2(a)(iii) provides exceptions for the default rule about obligations.

```
1  % rule 8
2  An action is not an obligation for a party if
3  a transaction specifies the action , and
4  the action is directed from the party to
```

```
5     a counterparty . and
6   the transaction is governed by IsdaAgreement , and
7   the action is due on a date , and
8   it cannot be shown that
9   EventOfDefault holds for the counterparty
10    on the date , and
11  it cannot be shown that
12  PotentialEventOfDefault holds for the counterparty
13    on the date , and
14  it cannot be shown that
15  EarlyTermiantionDate holds for the transaction
16    on the date.
17
18  % partial rule 9
19  EarlyTermiantionDate holds for a transaction on a date
20    if ...
```

Rule 8 enables us to prove directly that something is not an obligation, allowing us to deal with exceptions to Rule 2.

Notice the use of constant names in lines 9 and 12 of the pink box. Confusingly, the contract treats events of default as both an event and a state that a party can be in over time (i.e. a fluent): "Potential Event of Default with respect to the other party **has occurred and is continuing**".

In the LE translation, I make the choice to treat these special "events" as states (with names like "EeventOfDefault") which either hold or not for a given party and date. I do this to simplify the structure of the LE rule, making it shorter and more self contained. An alternative approach would be to specify rules which say that an event of default happens on a date and triggers a state of default, which state then holds for all future dates unless something terminates it.

The partial Rule 9 is a placeholder for the clause:

```
1  holds(earlyTerminationDate, Transaction, Date) :- ...
```

This is needed because the text says that "early termination" counts as a special condition not only if it has happened, but if the parties have agreed on some definite future early termination date. It is up for legal interpretation what happens in the case where early termination is agreed on Monday, with the termination date set for Wednesday, and a payment scheduled for Tuesday. It can be argued that the Tuesday payment is still an obligation, but all payments after that are not. However, it can also be argued the payment on Tuesday stops being an obligation as soon as it is clear (i.e. on Monday) that the parties want to terminate the transaction early. The specific definition of Rule 9 will depend on the legal interpretation.

5.4.2 Change of Account

The obligation section continues with a provision of how a party may announce a change to their designated account:

> (b) Change of Account. Either party may change its account for receiving a payment or delivery by giving notice to the other party at least five Local Business Days prior to the Scheduled Settlement Date for the payment or delivery to which such change applies unless such other party gives timely notice of a reasonable objection to such change.

This is another example of rule with an exception – an account can be changed in advance, but not if the counterparty objects. In Logical English:

```
% rule 10
A an account is correctlySpecified by a party
  for a transaction for a payment if
the transaction specifies the payment , and
the transaction is governed by IsdaAgreement , and
the payment is directed from the party to
  the counterparty , and
the payment is due on a date , and
either
  the account is theDefaultAccount for
    the party for the transaction
or
  the party makes a notification to the counterparty
    on a second date , and
  the second date is before the first date , and
  the second date is separated by
    an amountOfLocalBusinessDays from
    the first date , and
  the amountOfLocalBusinessDays is
    greater than or equal to 5 , and
  it cannot be shown that both
    the counterparty reasonablyObjects to
      the notification on a third date , and
    the third date is before the first date.
```

In Prolog:

```
% rule 10
relat_is(correctly, specified, Account, Party,
    Transaction, Payment) :-
  specifies(Transaction, Payment),
  relat_is(governed, Transaction, isdaAgreement),
```

```
6     relat_is(directed, Payment, Party, Counterparty),
7     relat_is(due, Payment, Date),
8     (
9      relat_is(default, account, Party, Transaction) ;
10     (
11      makes(Notification, Party, Account, Counterparty,
12        Date2, Notification),
13      Date2 #< Date,
14      relat_is(separated, Date2, Date,
15        AmountOfLocalBusinessDays),
16      AmountOfLocalBusinessDays #> 5,
17
18      \+(reasonablyObjects(Counterparty, Notification,
19           Date3),
20         Date3 #> Date2)
21      )
22    ),
23    \+(/* conjunction of the two disjuncts, as
24          exclusive ``or'' syntax is used */).
```

Rule 10 is complicated and requires closer examination. First, note that the rule contains a complex disjunction (starting from line 9 in pink box). The second disjunct (lines 13-22 of pink box) is not a simple sentence, but rather a conjunction. At the Prolog level, the conjuncts comprising the left disjunct are placed in parentheses to ensure correct priority of logical operations. At the LE level this is achieved with reserved syntax (like the use of ";"). I also make use of spacing to improve the readability of the LE rule.

Line 19 of the pink box contains another example of special syntaxs – "it is not the case that both". This phrase indicates that the two LE formulas which follows should be placed in a conjunction, and then the entire conjunction should be weakly negated.

Finally, note that the definition of "a reasonable objection" is not given – another example of vagueness. The definition can be left to lawyers' discretion or it can be provided in supplementary documents later.

This LE rule is not particularly pleasant to read, but it *can* be read, understood, and compared to the source text.

5.4.3 Netting of Payments

Next in the obligations section, we have a complicated set of rules describing netting of payments. First, a rule states that under certain conditions, multiple payments

from the same transactions need to added together, with only the net amount actually being transferred between parties:

> (c) Netting of Payments. If on any date amounts would otherwise be payable:—
>
> (i) in the same currency; and
>
> (ii) in respect of the same Transaction,
>
> by each party to the other, then, on such date, **each party's obligation to make payment of any such amount will be automatically satisfied and discharged** and, if the aggregate amount that would otherwise have been payable by one party exceeds the aggregate amount that would otherwise have been payable by the other party, **replaced by an obligation upon the party by which the larger aggregate amount would have been payable** to pay to the other party the excess of the larger aggregate amount over the smaller aggregate amount.

Note the two bolded phrases (my emphasis). They state that if multiple payments are matched by currency, date, and transaction ID, they will cancel each other out, with any outstanding amount becoming an obligation for the relevant party. The obligations arising from the individual payments are "discharged" and (if an amount is still outstanding) replaced by a single new obligation. The presence of the word "will" implies the the netting of payments is not optional but mandatory.

Confusingly, the word "will", the word will is the only hint that netting is required, and the clause can easily be misinterpreted to simply allow netting, rather than mandate it. This is a good example of how "legalese" can be just as inscrutable to the uninitiated as computer code.

Defining netting requires arithmetic operations – e.g., adding and subtracting payment amounts. It would be useful to take advantage of some of Prolog's predefined predicates like *[]*, *findall/3*, and *sum_list/2*. At the LE level this requires using some reserved syntax. Given how important the predefined Prolog predicates are for the netting rules, I start with the Prolog code and then "translate upwards" to Logical English.

The strategy I take is the following. I start by defining the concept of an aggregate payment. This is needed to collect information on all payments going from one party to the other for a given date, transaction, etc. Once each party's aggregate payment amounts are calculated, we can subtract the smaller amount from the larger to determine the resulting size and direction of the net payment (i.e. who owns how much to whom, on net). Finally, we need some additional rules which state that the net payment creates a new obligations and discharges the obligations resulting from the individual underlying payments.

In what follows, payments are nominalised – i.e. treated as individuals. Different payments are referred to by their IDs, like "p1", "p2", etc. Individual payments are then given various attributes, like amount, currency, and so on. For example,

“p1”can have an amount of “100” and be denominated in “GBP”.

```
% rule 11
% useful to collect needed attributes together
owes(Party, Counterparty, Payment, Currency,
     Transaction, Date) :-
    type_is(payment, Payment),
    specifies(Transaction, Payment),
    relat_is(denominated, Payment, Currency),
    relat_is(due, Payment, Date),
    relat_is(directed, Payment, Party, Counterparty),
    relat_is(governed, Transaction, isda).

% rule 12
% ListOfPayments is a list of the IDs of relevant
% payments for a given party, ccy, date, transaction
relat_is(aggregatePayment, ListOfPayments, Party,
         Currency, Transaction, Date) :-

  type_is(party, Party),

  % compiles a list of payment IDs given
  % characteristics
  findall(Payment,
            (
             owes(Party, _, Payment, Currency,
             Transaction, Date)
            ),
             ListOfPayments).

% rule 13
% AggregateAmount is the sum of all payments from list
relat_is(aggregateamount, AggregateAmount,
             ListOfPayments, Date) :-
  findall(Amount,
    (
     member(Payment, ListOfPayments),
     relat_is(valued, Payment, Amount, Date)
    ),
     ListOfAmounts),
     sum_list(ListOfAmounts, AggregateAmount).
```

Rules 11-13 define the concept of aggregate payment via two main features – the "ListOfPayments" identifies an aggregate payment as a list of the IDs of constituents; "AggregateAmount" is a total monetary size. The *owes/6* is a convenient helper predicate which collects together some useful attributes of a given payment.

In Logical English, Rules 11-13 become:

```
% rule 11
A party owes a counterparty a payment in a currency
  for a transaction on a date if
the transaction specifies the payment , and
the payment is denominated in the currency , and
the payment is due on the date , and
the payment is directed form the party to
  the counterparty , and
the transaction is governed by isdaAgreement.

% rule 12
A listOfPayments is an aggregate payment from a party
  in a currency on for a transaction on a date if
the listOfPayments is a list containing
any payment such that
  the party owes a counterparty the payment in
    the currency for the transaction on the date.

% rule 13
An aggregateAmount is an aggregateamount for
  a listOfPayments on a date if
a listOfAmounts is a list containing
any amount such that
  a payment is contained in the listOfPayments, and
  the payment is valued the amount on the date ;
and
the aggregateAmount equals summation over
  the listOfAmounts.
```

Note the special syntax which maps to *findall/3* and *sum_list/2*. Also note the semi-colon in line 25 of the pink box, which serves to mark the end of the *findall/3* goal.

Once we have the concept of aggregate payment, we are ready to define a net payment.

```
% rule 14
% NetPayment is a list all relevant payments IDs
% Amount is the net amount to be paid by the party
```

```
4  % with bigger aggregate obligations
5  relat_is(netpayment, NetPayment, Party, Counterparty,
6           Currency, Transaction, Amount, Date) :-
7
8      % needed to instantiate some arguments
9      relat_is(agreement, isdaAgreement, Party, Counterparty),
10
11
12     % calculate aggregate payment for party
13     relat_is(aggregatePayment, ListOfPayments1,
14              Party, Currency, Transaction, Date),
15     relat_is(aggregateamount, A1, ListOfPayments1, Date),
16
17     % calculate aggregate payment for counterparty
18     relat_is(aggregatePayment, ListOfPayments2,
19              Counterparty, Currency, Transaction, Date),
20     relat_is(aggregateamount, A2, ListOfPayments2, Date),
21
22     % assumption is that Party unifies with whoever owns
23     % some money on net
24     A1 #>= A2,
25     append(ListOfPayments1, ListOfPayments2, NetPayment),
26     Amount #= A1 - A2.
```

The “NetPayment” term collects the IDs of all individual payments involved in the calculation (on both the party and counterparty sides); the “Amount” is the net amount, and the “Party” is the entity who owes the larger aggregate amount (and so has to make a net payment).

In Logical English:

```
1  % rule 14
2  A netPayment is a netPayment form a party to a
3    counterparty in a currency for a transaction for
4    a netAmount on a date if
5  isdaAgreement is an agreement between the party and
6    the counterparty , and
7  a first listOfPayments is an aggregatePayment for
8    the party in the currency for the transaction on
9    the date , and
10 a first aggregateAmount is an aggregateamount for
11   the first listOfPayments on the date , and
12 a second listOfPayments is an aggregatePayment for
```

```
13     the counterparty in the currency for the transaction
14     on the date , and
15   a second aggregateAmount is an aggregateamount for
16     the second listOfPayments on the date , and
17   the first aggregateAmount is larger than the
18     second aggregateAmount , and
19   the netPayment is the concatenation of the first
20     listOfPayments and the second listOfPayments , and
21   the netAmount equals the first aggregateAmount minus
22     the second aggregateAmount.
```

Note the specialised syntax in lines 17-22 of the pink box.

Finally, we need auxiliary rules to establish how net payments create new obligations, and discharge old ones.

```
1  % rule 15
2  relat_is(not, obligation, Payment, Party) :-
3      owes(Party, Counterparty, Payment, Currency,
4           Transaction, Date),
5      relat_is(netPayment, NetPayment, Party, Counterparty,
6               Currency,Transaction, _, Date),
7      member(Payment,  NetPayment).
8
9  % rule 16
10 relat_is(not, obligation, Payment, Counterparty) :-
11     owes(Counterparty, Party, Payment, Currency,
12          Transaction, Date),
13     relat_is(netPayment, NetPayment, Party, Counterparty,
14              Currency,Transaction, _, Date),
15     member(Payment,  NetPayment).
16
17 % rule 17: additional obligation definition
18 relat_is(obligation, NetPayment, Party) :-
19     owes(Party, Counterparty, Payment, Currency,
20          Transaction, Date),
21     relat_is(netPayment, NetPayment, Party, Counterparty,
22              Currency,Transaction, _, Date),
23     member(Payment,  NetPayment).
24
25 % rule 18
26 happens(NetPayment, Currency, Transaction, Date) :-
27     relat_is(netPayment, NetPayment, Party, Counterparty,
```

```
28                Currency, Date, Transaction, Amount),
29      transfers(Party, Amount, Account, Date),
30      transfers(customary, way, Party, Amount, Currency),
31      transfers(freely, transferable, funds, Party, Amount),
32      relat_is(correctly, specified, Account, Counterparty,
33                Transaction, NetPayment),
34      relat_is(governed, Transaction, isdaAgreement).
35
36  % rule 19
37  happens(NetPayment, Date) :-
38      type_is(currency, Currency),
39      type_is(transaction, Transaction),
40      happens(NetPayment, Currency, Transaction,  Date).
41
42  % rule 20
43  relat_is(due, NetPayment, Date) :-
44      type_is(payment, Payment),
45      member(Payment, NetPayment),
46      relat_is(due, Payment, Date).
```

Rules 15-16 state that the individual payments which constitute a net payment are not obligations. The strong negation is used, which means that we can directly prove *relat_is(not, obligation, ...)*. The two rules are very similar. The only difference is that Rule 15 references the party making a net payment, and Rule 16 references the counterparty receiving a net payment. Both rules are needed as both parties see some of their obligations automatically discharged.

Rule 17 enables us to establish that net payments themselves become obligations. Finally, rules 18-20 specify what it means for a net payment to "happen", and assign a due date to every net payment (same as the due date of the constituents).

A fully worked out Prolog example of how payment netting works can be found in file *netPayment.pl* in the code archive. Once all the rules are specified, along with some additional facts about payments, we can query the program to test it. For example, we can ask about the current obligations of Bank:

```
1  %?- relat_is(obligation, Payments, bank).
2  %@ Payments = [p1, p2, p3] ;
3  %@ false.
```

Or we can ask what obligations have been satisfied on a particular date:

```
% which obligations are satisfied
%?- relat_is(satisfied, Obligation, 20110503).
%@ Obligation = [p1, p2, p3] ;
%@ false.
```

In Logical English, the key rules will state:

```
% rule 15
A payment is not an obligation for a party if
the party owes a counterparty the payment in
  a currency for a transaction on a date , and
a netPayment is a netPayment from the party to
  the counterparty in the currency for the
  transaction for an amount on a date , and
the payment is contained in the netPayment.

% rule 16
A payment is not an obligation for a counterparty if
the counterparty owes a party the payment in
  a currency for a transaction on a date , and
a netPayment is a netPayment from the party to
  the counterparty in the currency for the
  transaction for an amount on a date , and
  the payment is contained in the netPayment.

% rule 17
A netPayment is an obligation for a party if
the party owes a counterparty a payment in
  a currency for a transaction on a date , and
the netPayment is a netPayment for the party to
  the counterparty in the currency for
  the transaction for an amount on the date , and
the payment is contained in the netPayment.

/* remaining rules are omitted for brevity */
```

Notice that what was expressed with a simple paragraph in the source text gave rise to ten Logical English rules. On one hand, this shows how efficient natural language can be. On the other hand, however, it shows how many logical relationships are often left implicit in legal texts. This can be a problem if the clauses in a contract are not standardised, or if the person agreeing to them is not familiar with legal English. It is a benefit of Logical English that all relationships between entities need to be made explicit.

One last thing that needs to be addressed is the clarification in the ISDA agreement that payments can be netted together across multiple transactions if those transactions are specifically designated by the parties:

> [From 2(c)] The parties may elect in respect of two or more Transactions that a net amount and payment obligation will be determined in respect of all amounts payable on the same date in the same currency in respect of those Transactions, regardless of whether such amounts are payable in respect of the same Transaction. The election may be made in the Schedule or any Confirmation by specifying that "Multiple Transaction Payment Netting" applies to the Transactions identified as being subject to the election (in which case clause (ii) above will not apply to such Transactions). If Multiple Transaction Payment Netting is applicable to Transactions, it will apply to those Transactions with effect from the starting date specified in the Schedule or such Confirmation, or, if a starting date is not specified in the Schedule or such Confirmation, the starting date otherwise agreed by the parties in writing. This election may be made separately for different groups of Transactions and will apply separately to each pairing of Offices through which the parties make and receive payments or deliveries.

Essentially, if parties choose to bundle transactions together, these bundles will be governed by the rule specified in 2(c) and not by any of the previous rules. Transaction bundles then present an exception to the netting rules (which are themselves exceptions to ordinary payments!). We handle section 2(c) by modifying Rule 12 with an exception condition:

```
1  % Rule 12*
2  relat_is(aggregatePayment, ListOfPayments, Party,
3           Currency, Transaction, Date) :-
4
5    type_is(party, Party),
6
7    findall(Payment,
8              (
9               owes(Party, _, Payment, Currency,
10              Transaction, Date)
11             ),
12              ListOfPayments),
13
14   \+ relat_is(designated, Transaction, Bundle).
```

In LE:

```
% rule 12*
A listOfPayments is an aggregatePayment from a party
in a currency for a transaction on a date if
... , and
it cannot be shown that
the transaction is designated in a bundle.
```

The new condition ensures that transactions which are part of bundles (represented as lists in Prolog) will not be bound by Rule 12* but treated as exceptions.

We then need to introduce a new aggregate payment rule which can apply to groups (i.e. bundles) of designated transactions, rather than a single transaction.

```
1  % rule 21
2  relat_is(designatedAggregatePayment, ListOfPayments,
3           Party, Currency, Date,
4           ListOfTransactions) :-
5
6      type_is(designatedGroup, ListOfTransactions),
7
8      findall(Payment,
9              (
10                 member(Ttransaction, ListOfTransactions),
11                 owes(Party, _, Currency,
12                      Transaction, Date, Payment)
13             ),
14              ListOfPayments).
```

Rule 21 differs from Rule 12 due to an the additional *type_is/2* predicate (line 6 of grey box) which identifies a given *list* of transactions as a designated group. The aggregate payment is then the list of any payment in any of the transactions form the designated group (see lines 9-10).

In LE:

```
% rule 21
A listOfPayments is a designatedAggregatePayment
  from a party in a currency on a date
  for a listOfTransactions if
the listOfTransactions is a designatedGroup , and
the listOfPayments is a list containing
any payment such that
  a transaction is contained in
    the listOfTransactions , and
```

```
10      the party owes a counterparty the payment
11        for the currency for
12        the transaction on the date.
```

Finally, we need a new rule for designated net payments. This will be identical to Rule 14 with the exception that the "aggregatePayment" constant argument will be replaced by "designatedAggregatePayment". Because of this similarity, I omit an explicit statement of the rule here.

Note that although party offices and effective designation dates were mentioned in the source text, they were not directly relevant in the netting rules. These attributes are instead used to further specify groups of transactions. In the rules just discussed this specification is not needed, as we can already precisely identify groups by the IDs of their constituent transactions.

5.5 Conclusion

In this chapter, I demonstrated how key financial relationships can be expressed in Logical English. The chapter already provides sufficiently many examples to give the reader a flavour of how the entire ISDA Master Agreement could be translated. For brevity's sake, I conclude the chapter here.

Chapter 6

User testing

The current project focuses on showing how LE can be used to translate legal contracts into executable programs. In real-world applications, interested parties and lawyers should be able to personally verify that LE programs correspond to the source text.

The best way to check readability is to show Logical English text to its intended audience. This chapter discusses two experiments which provide some evidence in support of LE's readability.

6.1 User testing in the CNL literature

In his PhD thesis, Kuhn devotes a chapter to evaluating the user-friendliness of Attempto Controlled English, and whether the language is easier to understand than another popular CNL (13). He reports on experiments in usability and understandability.

Understandability experiments are relevant to my project. In Kuhn's case, these experiments consisted of showing participants a picture, called an ontograph, which represents some small knowledge base, and asking whether sentences written in ACE are true of this picture (13). An example of one of Kuhn's experiments is reproduced in Figure 6.1 below.

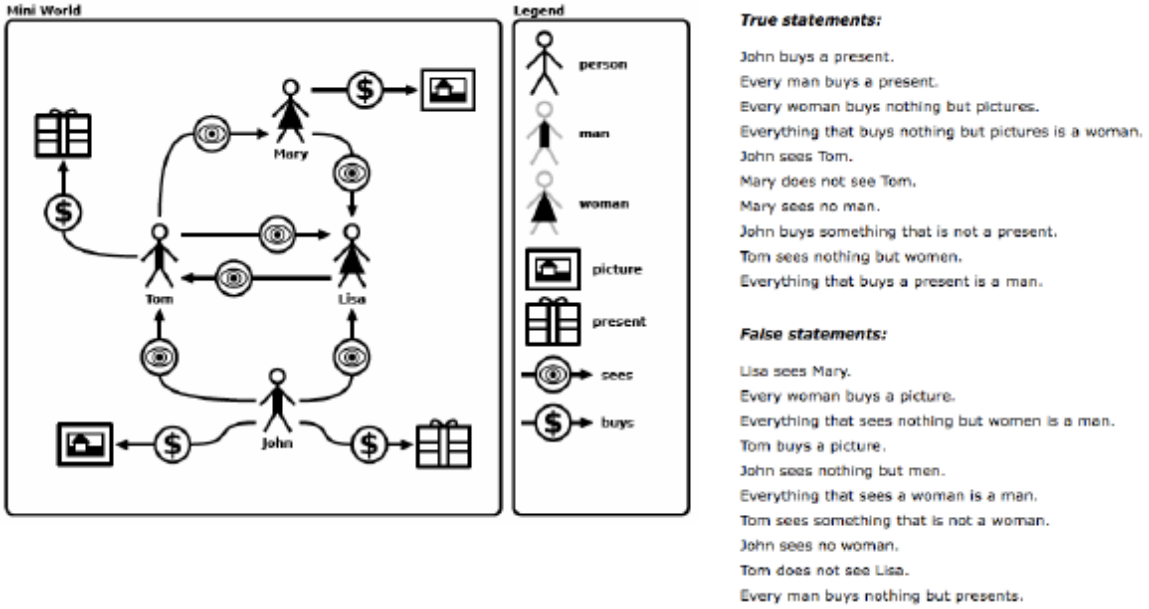


Figure 6.1: A reproduction of one of Kuhn's ontograph experiments

Ontographs are a useful way to measure user comprehension for Logical English. Their major limitation is that they can unambiguously represent only a small set of simple relationships. Specifically, there would not be an intuitive way to represent financial obligations (such as those contained in the ISDA agreement) as pictures. Thus, ontographs cannot be applied to the real-world contexts with which LE is primarily concerned. In those cases, eliciting subjective opinions from subject-matter experts might be more informative.

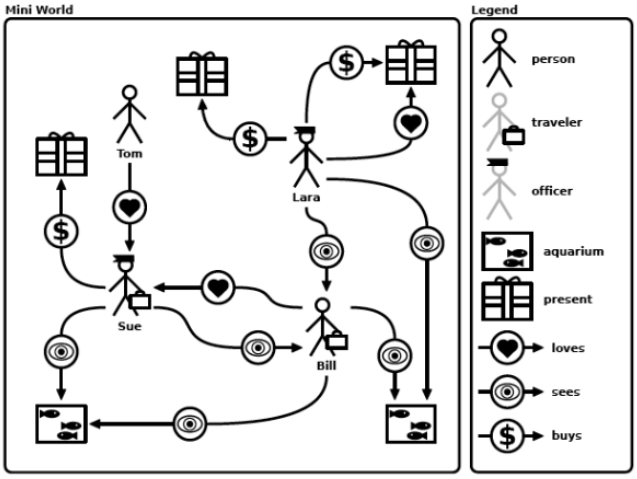
SUS is a "quick and dirty" questionnaire for measuring system usability (31). It asks users to signal how much they agree with statements like "I think that I would like to use this system frequently". Their responses are on a scale from 1 (=strongly disagree) to 5 (=strongly agree). I used a questionnaire adapted from SUS to check the usability of Logical English in the domain of legal contracts.

6.2 LE user testing

Ontograph experiment: LE vs Prolog

I conducted two experiments. The first experiment consisted of showing five participants with no logic programming background an ontograph with twelve statements – see Figure 6.2. Six of the statements were written in Logical English and six in Prolog. Minimal explanation of the ontograph was given, and participants were asked to decide whether the statements are true or false. Their answers were scored by giving a point for correct answers and deducting a point for incorrect ones – so that random guessing on average carried 0 points, the maximum possible score was 6, and the minimum -6. I compared the average score for LE sentences to the score for

the Prolog sentences.



ID	Question	TRUE/FALSE
1	An person is loved by a numberOfPeople if the person is an officer , and the numberOfPeople is greater than 2.	
2	buys(bill, X) :- sees(bill, X), type_is(aquarium, X).	
3	A thing is a person if the thing loves a numberOfThings and the numberOfThings is smaller than or equal to 1.	
4	Lara sees a numberOfPeople if the numberOfPeople is greater than 2.	
5	A thing is a present if the thing is bought by a person.	
6	A person is an officer if the person sees a thing.	
7	loves(X, Y) :- type_is(traveller, X), type_is(person, Y).	
8	type_is(aquarium, X) :- bought_by(_, X).	
9	A thing is a person if either the thing loves another thing or the thing is loved by another thing or both.	
10	sees(bill, X) :- type_is(aquarium, X).	
11	sees(X, 1, travellers) :- type_is(officer, X).	
12	loved_by(sue, lara).	

Figure 6.2: Experiment 1: Readability of Logical English vs Prolog
Adapted from: http://attempto.ifi.uzh.ch/site/docs/ontograph/

Over the five participant in this experiment, the LE questions carried an average of 2.4 points, and the Prolog questions carried 1.2 points. For full results see Appendix B.

The sample size is very small, but the results provide tentative support for assumption that LE sentences are easier to understand than Prolog code, at least for non-logicians. Perhaps surprisingly, the average score on the Prolog questions was positive, suggesting that participants were able to make some intuitive sense of the Prolog code.

The ISDA Agreement experiment

The second experiment consisted in showing two qualified lawyers three excerpts from the ISDA Master Agreement, with the source text and corresponding LE translations placed side by side (see Annex B for the excerpts). The two participants were asked to read the experts and indicate how strongly they agree or disagree with four statements. The statements and the two participants' answers are shown in Figure 6.3.

Questionnaire						Answers	
	Strongly disagree		Neither agree nor disagree		Strongly agree	P1	P2
1. I needed some support to fully understand the Logical English text.	1	2	3	4	5	4	3
2. I could verify that the Logical Englsih sentences accurately capture the legal meaning of the source text.	1	2	3	4	5	4	4
3. I found the Logical English sentences more difficult to read than the source text.	1	2	3	4	5	4	5
4. With a little support from an expert, I will feel comfortable signing contracts written in Logical English.	1	2	3	4	5	4	4

Figure 6.3: Experiment 2: Expert judgement on how applicable LE is to legal texts translations

Both participants needed some support when interpreting the LE sentences initially (see question 1). However, they could both eventually verify that that the LE text carried the same logical meaning as the corresponding source text (see question 2). This is important as users' ability to verify Logical English programs against source text is a key goal of my project. Finally, even though both participants found the LE sentences more difficult to read than legal English (question 3), they agreed that, with some support, they would feel comfortable signing contracts written in Logical English (question 4). Overall, the two subject-matter experts' responses to Logical English were positive, suggesting that the language could indeed be applied to the area of law.

Chapter 7

Conclusion

I conclude with an evaluation of my achievements against my stated goals (see Chapter 1), and I discuss areas for future work.

7.1 Evaluation

This project was about developing initial standards for a controlled language called Logical English. There were three main goals – that Logical English should be easy to read for English speakers; that the language should be automatically translatable to executable code, and the that the language should be expressive enough to capture relationships found in legal contracts.

Achievements

I presented the rules for writing and reading Logical English in Section 1.3. The motivation for these rules, along with examples of LE sentences were given in Chapter 3.

Against the three overarching objectives, I consider the project a qualified success. First, as demonstrated by examples throughout the report, and by the results of the two small user tests (see Chapter 6), Logical English can be read and understood by English speakers with no programming background.

Second, the grammar described in Section 1.3 is deterministic, and the syntax rules are described in sufficient detail to show that automatic translation is possible. More importantly, the interpreter presented in Chapter 4 demonstrates how a slightly simplified version of Logical English can be automatically translated to Prolog code.

Finally, Chapter 5 translates key sections of a popular financial contract to both Logical English and Prolog code. This demonstrates that LE is expressive enough to be applied in legal contexts. Furthermore, few papers in the Controlled Natural Language literature present end-to-end translation (i.e. from English through CNL

to executable code) using real-world documents.[1] In this sense, Chapter 5 represents a contribution to the literature.

Areas for improvement

As mentioned, my project is a *qualified* success. I now discuss three short-term objectives which I ideally should have delivered but did not. First, although the grammar rules for LE are presented in enough detail to give the reader a very good idea of how to read the language, I did not produce exhaustive documentation, listing all the reserved syntax and formally defining every grammar rule.

Second, I did not have time to develop an interpreter which can automatically parse all Logical English sentences. The interpreter presented in Chapter 4 is not powerful enough to include ordinal words and the specialised syntax needed for (among other things) the expression of weak negation.

Finally, although I showed that Prolog translations of LE programs can be effectively analysed using Prolog queries, I did not demonstrate a way for LE programs to be queried directly, using questions formulated in LE.

7.2 Future work

Future work should focus on making Logical English more expressive and intuitive, while at the same time keeping LE's relatively simple grammar, and preserving LE's deterministic semantics[2]. I identify three main areas for longer-term development:

First, rules could be developed to allow adjectives and adverbs to be written as separate words in LE sentences. This will improve both readability and expressive power. Currently, for example, "walksQuickly" is treated as a single predicate, but semantically, the action "walks" is separate from the manner in which it is performed, "quickly". It would be an improvement if this separation existed in Logical English. Adverbs can modify predicates either as meta-predicates, *quickly(walks(john))*, or as additional arguments, *walks(john, quickly))*.

Second, more thought needs to be given to the treatment of meta-predicates. Currently, meta-predicates can only appear in LE sentences in the from of adverb clauses (as in "Mary wishes that Mary isHome"). There is no general way to treat any predicate as a meta-predicate. A possible way forward is developing a system for tagging predicates and using their tags as arguments of meta-predicates. For example, "Fluent1" could be the tag for "Mary isHome". And "Fluent1 isTerminated" could correspond to *isTerminated(isHome(mary))*.

[1] One other example of this is the Flood and Goodenough translation of a simple loan agreement (19).

[2] This refers to the fact that LE sentences have a unique and explicit logical interpretation.

Finally, LE sentences can represent logic program clauses but not reactive rules. I argued that this is sufficient to translate legal contracts into Logical English. However, it will not be sufficient if we want complete contract automation. During the lifetime of a given contract, certain actions need to be performed. For example, a tenant needs to pay rent to the landlord on the first day of every month. Most of these actions can be automated – a computer can automatically transfer the rent mo ney between accounts on the given date. However, as mentioned in Section 3.4, logic program clauses are "static" and cannot perform actions. For this, we need the facility to express rules of the form *if CONDITIONS then DO ACTIONS*.

Appendix A

Ethics checklist

My project seeks to develop a computer language to aid in contract automation. There are two ethical issues to consider – the extent to which people can interpret Logical English correctly, and the extend to which an LE interpreter can correctly translate LE sentences to executable code.

First, when a person signs a contract, they must be competent and capable of understanding what the contract entails. Logical English is designed to be clear and unambiguous; however, it is a new language which is yet to be fully tested. Future developers of Logical English will need to consult closely with legal professionals to ensure that the first contracts written in LE do not have any unintended legal consequences.

Second, even if the Logical English in a given contract is unambiguous, a bug in the interpreter can result in an unintended translation to executable code. In the case of fully automated contracts, this can lead to actions being executed which have not been agreed by the signatories. LE interpreters will require extensive user testing, as well as safeguard mechanisms to reduce and manage the risk of faulty programs.

Though my project involved an element of user testing, I did not collect any personal data from my participants, and my experiments had no ethical implications for them.

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		X
Does your project involve the use of human embryos?		X
Does your project involve the use of human foetal tissues / cells?		X
Section 2: HUMANS		
Does your project involve human participants?	X	
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from "Human Embryos/Foetuses" i.e. Section 1)?		X
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		X
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		X
Does it involve processing of genetic information?		X
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		X
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		X
Section 5: ANIMALS		
Does your project involve animals?		X
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		X
If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		X
Could the situation in the country put the individuals taking part in the project at risk?		X
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		X
Does your project deal with endangered fauna and/or flora /protected areas?		X
Does your project involve the use of elements that may cause harm to humans, including project staff?		X
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		X
Section 8: DUAL USE		
Does your project have the potential for military applications?		X
Does your project have an exclusive civilian application focus?	X	
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		X
Does your project affect current standards in military ethics – e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		X
Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?		X
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		X
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?		X
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		X
SECTION 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		X
Will your project use or produce goods or information for which there are data protection, or other legal implications?		X
SECTION 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		X

Figure A.1: How LE relates to spoken English and Prolog

Appendix B

Experiments

Experiment 1: Full results

ID	Question	TRUE/FALSE	Answers 1	2	3	4	5	CORRECT
1	An person is loved by a numberOfPeople if the person is an officer , and the numberOfPeople is greater than 2.		TRUE	FALSE	FALSE	FALSE	TRUE	FALSE
2	buys(bill, X) :- sees(bill, X), type_is(aquarium, X).		FALSE	TRUE	TRUE	TRUE	FALSE	FALSE
3	A thing is a person if the thing loves a numberOfThings and the numberOfThings is smaller than or equal to 1.		TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
4	Lara sees a numberOfPeople if the numberOfPeople is greater than 2.		FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
5	A thing is a present if the thing is bought by a person.		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
6	A person is an officer if the person sees a thing.		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
7	loves(X, Y) :- type_is(traveller, X), type_is(person, Y).		FALSE	TRUE	FALSE	TRUE	FALSE	FALSE
8	type_is(aquarium, X) :- bought_by(_, X).		FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
9	A thing is a person if either the thing loves another thing or the thing is loved by another thing or both.		FALSE	FALSE	FALSE	FALSE	TRUE	TRUE
10	sees(bill, X) :- type_is(aquarium, X).		TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
11	sees(X, 1, travellers) :- type_is(officer, X).		FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
12	loved_by(sue, lara).		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
		SCORE LE	2	4	2	0	4	2.40
		SCORE Prolog	4	-2	0	0	4	1.20

Figure B.1: How LE relates to spoken English and Prolog

Experiment 2: ISDA Agreement extracts

Source Legal Text	Source Legal Text	Source Legal Text
Single Agreement. All Transactions are entered into in reliance on the fact that this Master Agreement and all Confirmations form a single agreement between the parties (collectively referred to as this "Agreement"), and the parties would not otherwise enter into any Transactions.	Each party will make each payment or delivery specified in each Confirmation to be made by it, subject to the other provisions of this Agreement.	Payments under this Agreement will be made on the due date for value on that date in the place of the account specified in the relevant Confirmation or otherwise pursuant to this Agreement, in freely transferable funds and in the manner customary for payments in the required currency.
Logical English Translation	**Logical English Translation**	**Logical English Translation**
A transaction is governed by IsdaAgreement if the transaction is a transaction between a party and a counterparty , and IsdaAgreement is an agreement between the party and the counterparty , and the transaction completes on a first date , and IsdaAgreement commences on a second date , and the first date is after the second date.	An action is an obligation for a party if a transaction specifies the action , and the transaction is a transaction between the party and a counterparty , and the action is directed from the party to the counterparty , and either the action is a payment or the action is a delivery ; and it cannot be shown that the action is not an obligation for the party.	A payment happens on a date if the payment is valued an amount on the date , and the payment is denominated in a currency , and the payment is directed from a party to a counterparty , and the party transfers the amount to an account on the date , and the party transfersInCustomaryWay the amount for the currency , and the party transfersInFreelyTransferableFunds the amount , and the account is correctlySpecified by the counterparty for a transaction for the payment , and the transaction specifies the payment , and the transaction is governed by IsdaAgreement.

Figure B.2: How LE relates to spoken English and Prolog

Bibliography

[1] R. A. Kowalski, "Legislation as logic programs." Springer, Berlin, Heidelberg, 1992. [Online]. Available: http://link.springer.com/10.1007/3-540-55930-2{_}15 pages 2

[2] N. A. Vonneuman, "Layman E. Allen. Symbolic logic: A razor-edged tool for drafting and interpreting legal documents. Yale law journal, vol. 66 (1957), pp. 833–879. - Layman E. Allen and Gabriel Orechkoff. Toward a more systematic drafting and interpreting of the internal r," *Journal of Symbolic Logic*, vol. 29, no. 1, mar 1964. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0022481200117189/type/journal{_}article pages 2

[3] B. Niblett, Science Research Council (Great Britain), and European Economic Community. Scientific and Technical Research Committee., *Computer science and law*. Cambridge University Press, 1980. [Online]. Available: https://books.google.co.uk/books?id=3Cc9AAAAIAAJ{&}redir{_}esc=y pages 2

[4] M. Wong, "Computable contracts – Internet of Agreements." [Online]. Available: http://internetofagreements.com/2017/12/16/meng-wong-legalese/ pages 2

[5] G. Governatori, F. Idelberger, Z. Milosevic, R. Riveret, G. Sartor, and X. Xu, "On legal contracts, imperative and declarative smart contracts, and blockchain systems," *Artificial Intelligence and Law*, vol. 26, no. 4, dec 2018. [Online]. Available: http://link.springer.com/10.1007/s10506-018-9223-3 pages 2

[6] M. Triska, *The Power of Prolog*, 2019. [Online]. Available: https://www.metalevel.at/prolog/clpz pages 7

[7] P. Blackburn and J. Bos, *Representation and inference for natural language : a first course in computational semantics*. Center for the Study of Language and Information, 2005. pages 8

[8] R. Kowalski, "History of Logic Programming," jun 2014. pages 8, 9

[9] T. Kuhn, "A Survey and Classification of Controlled Natural Languages," *Computational Linguistics*, vol. 40, no. 1, mar 2014. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/COLI{_}a{_}00168 pages 9

[10] T. Gao, "Controlled Natural Languages and Default Reasoning," may 2019. [Online]. Available: http://arxiv.org/abs/1905.04422 pages 9, 10, 11

[11] "About ASD-STE100." [Online]. Available: http://asd-ste100.org/about.html pages 9

[12] "Attempto Project." [Online]. Available: http://attempto.ifi.uzh.ch/site/ pages 10

[13] T. Kuhn, "Controlled English for Knowledge Representation." [Online]. Available: https://www.academia.edu/9107030/Controlled{_}English{_}for{_}Knowledge{_}Representation pages 10, 55

[14] P. Clark, W. R. Murray, P. Harrison, and J. Thompson, "Naturalness vs. Predictability: A Key Debate in Controlled Languages." Springer, Berlin, Heidelberg, 2010. [Online]. Available: http://link.springer.com/10.1007/978-3-642-14418-9{_}5 pages 12, 15

[15] P. Clark, J. Thompson, P. Yeh, S.-Y. Chaw, K. Barker, V. Chaudhri, P. Harrison, J. Fan, B. John, B. Porter, and A. Spaulding, "Capturing and answering questions posed to a knowledge-based system," in *Proceedings of the 4th international conference on Knowledge capture - K-CAP '07*. New York, New York, USA: ACM Press, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1298406.1298419 pages 12

[16] R. Schwitter, "Specifying and Verbalising Answer Set Programs in Controlled Natural Language," apr 2018. [Online]. Available: http://arxiv.org/abs/1804.10765 pages 13

[17] LSP Working Group, "Developing A Legal Specificaion Protocol: Technological Considerations and Requirements (Forthcoming)," 2018. pages 13

[18] M. Sergot, F. Sadri, and R. Kowalski, "The British Nationality Act as a logic program," *doc.ic.ac.uk*. [Online]. Available: http://www.doc.ic.ac.uk/{~}rak/papers/BritishNationalityAct.pdf pages 13, 17

[19] M. D. Flood and O. R. Goodenough, "Contract as Automaton: The Computational Representation of Financial Agreements — Office of Financial Research," 2017. [Online]. Available: https://www.financialresearch.gov/working-papers/2015/03/26/contract-as-automation/ pages 13, 18, 27, 60

[20] Lord Bingham of Corhnhill KG, "The Rule of Law." [Online]. Available: https://www.cpl.law.cam.ac.uk/sir-david-williams-lectures/rt-hon-lord-bingham-cornhill-kg-rule-law pages 14

[21] R. Cormacain, "Legislative Drafting and the Rule of Law," *Institute of Advanced Legal Studies*, 2017. [Online]. Available: https://sas-space.sas.ac.uk/6693/ pages 14, 21, 28

[22] R. Stim, "Contract Man: ambiguity." [Online]. Available: https://contractman.blogspot.com/2010/04/ambiguity.html pages 16

[23] S. &. S. P. Stimmel, "Ambiguity In Contracts-What Do The Courts Do?" [Online]. Available: https://www.stimmel-law.com/en/articles/ambiguity-contracts-what-do-courts-do pages 17

[24] T. Bench-Capon and F. Coenen, "Isomorphism and legal knowledge based systems," *Artificial Intelligence and Law*, 1992. pages 17

[25] R. Kowalski and F. Sadri, "Integrating Logic Programming and Production Systems in Abductive Logic Programming Agents." Springer, Berlin, Heidelberg, oct 2009. [Online]. Available: http://link.springer.com/10.1007/978-3-642-05082-4{_}1 pages 18

[26] ——, "Logic Programs with Exceptions," *New Generation Computing*, 1991. pages 24, 25

[27] J. L. Pollock, "Defeasible Reasoning," *Cognitive Science*, vol. 11, no. 4, 1987. pages 26

[28] R. Kowalski, *Logic for Problem Solving*. Elsevier Science Publishing, 1979. pages 29

[29] "Wikipedia: Ancestor." [Online]. Available: https://en.wikipedia.org/wiki/Ancestor pages 36

[30] International Swaps and Derivatives Association, Inc. pages 40

[31] J. Brooke, "System Usability Scale (SUS)." [Online]. Available: https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html pages 56