

An Integrated Development Environment (IDE) for Computable Contracting

Edgar Tsang¹

BSC Computer Science

Supervisor: Prof. Christopher D. Clack

Submission date: 1st May 2025

¹**Disclaimer:** This report is submitted as part requirement for the BSC Computer Science degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

The growing interest in the automation of contracts has led to the expanding field of Computable Contracts. In turn, this has led to the development of numerous Computable Contracting (CC) languages to represent contracts in a computable manner; however, the absence of a unified drafting environment has limited their accessibility and adoption. This dissertation presents the CC-IDE, a novel Integrated Development Environment specifically designed for drafting, managing, and analysing computable contracts across different CC-languages. The CC-IDE introduces a language-onboarding pipeline based on BNF grammars, automating parser generation and contract processing with minimal configuration, in turn streamlining CC-language development. Furthermore, it implements a Grammar Agnostic Semantics (GAS) framework, enabling the flexible mapping of legal semantics onto any CC-language and facilitating analytical tools such as contract simulations, visualisations, and semantic-based analyses. Through an interactive drafting interface featuring syntax highlighting, auto-formatting, real-time error detection, and automated suggestions, the CC-IDE improves both the efficiency and accuracy of contract drafting. The system was verified through comprehensive unit, system, and characterisation testing, demonstrating its correctness, robustness, and extensibility. Ultimately, this dissertation delivers a foundational proof-of-concept for future CC-IDEs, promoting the wider adoption of computable contracts and setting the groundwork for further innovation down this avenue.

Acknowledgements

I would like to convey a deep appreciation to my supervisor, Professor Christopher D. Clack, for the guidance and support that he has provided throughout this project. Exploration and innovation does not come easy, but it is definitely easier when pushed in the right direction.

Furthermore, I would like to thank my family who have supported me throughout my journey in the past 3 years. They are the foundation to my accomplishments and have been with me every step of the way.

Lastly, I would like to thank my friends, who have been incomparably supportive and have made my time at UCL truly unforgettable. They inspire me each day and have made my experience all the more worthwhile.

Contents

1	Introduction	6
1.1	The Problem Space	6
1.2	The Proposed Solution - the CC-IDE	7
1.2.1	Objectives	7
1.3	Dissertation Structure	7
2	Background	8
2.1	Natural Language Contracts	8
2.1.1	Contract components	8
2.2	Computable Contracts	9
2.2.1	Defining Computable Contracts	9
2.3	Computable Contracting (CC) Languages	10
2.3.1	Controlled Natural Languages	10
2.3.2	Semantics in Computable Contracts	11
2.3.3	Existing Computable Contracting (CC) Languages	12
2.4	Parsing	13
2.4.1	The Parse Tree (PT)	13
2.4.2	Parser Generators	14
3	An Integrated Development Environments for Computable Contracts	15
3.1	Integrated Development Environments (IDEs)	15
3.1.1	Critical Analysis of Common IDEs and their Characteristics	15
3.1.2	What are the common features of an IDE?	18
3.2	Understanding IDEs for Computable Contracts	20
3.2.1	The IDE Developer, Designer and the Lawyer	20

3.2.2	Designing a CC-IDE	21
3.3	Computable Contract IDE (CC-IDE)	22
4	Constructing the IDE	25
4.1	A Basic Text Editor	25
4.1.1	Developing the GUI	25
4.2	Integrating Computable Contracting (CC) Languages	27
4.2.1	Pipeline Design	27
4.2.2	Parsing with Tree-Sitter	28
4.2.3	BNF Parsing and Tree-Sitter Grammar Creation	28
4.2.4	The Challenge of Writing Valid BNFs	29
4.2.5	Resolving Ambiguities — Symbol Prefixes to write Complete Grammars . .	29
4.2.6	The Complete Pipeline	30
4.3	Universal set of IDE Features	32
4.3.1	Syntax Highlighting and Formatting	33
4.3.2	Error Detection	35
4.3.3	Auto Suggestion	35
4.3.4	Feature Benefit and Summary	36
5	Applying Semantics	37
5.1	Semantics	37
5.1.1	Symbols	38
5.1.2	The Set of Legal Semantics	38
5.1.3	Mapping Semantics onto Grammars	38
5.2	Translation — When Semantics are Indirect	40
5.2.1	Representing Time	40
5.2.2	Class Template	40
5.2.3	Try-Except	41
5.2.4	Extending GAS	41
5.2.5	Time as an Example	42
5.3	Analyses, Visualisations, and Simulations — Benefitting the Lawyer	42
5.3.1	Condition Simulation	42

5.3.2	Event Simulation	43
5.3.3	Identity Generation	44
5.3.4	Definition Substitution	44
5.3.5	Time Inconsistency Detection	45
5.3.6	Abstracted Contract Visualisations	45
5.4	CC-IDE Summary	46
6	Verification, Validation and Evaluation	48
6.1	Verification	48
6.1.1	Unit Testing	49
6.1.2	System Testing	49
6.1.3	Characterisation Testing	49
6.2	Validation	50
6.3	Critical Evaluation	51
6.3.1	Grammar Restriction	51
6.3.2	Semantic Framework Constraints (GAS)	51
6.3.3	Adoption within Legal Industry	52
6.3.4	Test Coverage and User Testing	52
6.3.5	System Extensibility	53
6.4	Future Work	53
6.4.1	Open Opportunity	53
6.4.2	Reusable Clause Templates	53
6.4.3	Security	53
6.4.4	Shared Access and Version Control	54
6.4.5	Automation of Operational Aspects	54
6.4.6	Plugin System	54
6.4.7	CC-languages for Automation	54
7	Conclusion	55
	Bibliography	57
	Appendices	61
A	CC-IDE Main Page	61

B	CC-IDE BNF Editor	62
C	Metadata in the CC-IDE	63
D	BNF-Parser BNF Grammar	65
E	Example Function from Visitor Class	67
F	Main CC-IDE Text Processing Code	68
G	OutputCSS function	71
H	Autosuggestion Generation Function	72
I	TimeOverride Class Example	73
J	Full Set of Symbol Prefixes and Usage	76
K	Condition Simulation in CC-IDE	78
L	Event Simulation in CC-IDE	79
M	Event Simulation Logic and Code	80
N	Unit Tests - Parser Pipeline	86
O	Unit Tests - Symbol Prefix Mapping	89
P	Discount Language BNF	90
Q	Testing Overview	93
R	Semantically Extended CoLa Grammar	96

Chapter 1

Introduction

Legal contracts are a crucial part of our lives, playing a role in formalising agreements and “safeguarding the rights and interests of parties involved” [1]. Traditionally, contracts are drafted by lawyers who leverage their understanding of the law to write legally enforceable contracts. In the pursuit of automation and greater efficiency, modern developments have led to the invention of Computable Contracts, where contracts are designed to be understood and analysable by a computer.

Computable Contracts are “contracts, expressed in a high-level, structured language that can be (a) readily generated, understood and used by humans and (b) easily and simultaneously understood (or processed) by computers.” [2] They are a form of written contracts which offer enhanced clarity, reliability, and efficiency through its computability, benefiting both legal professionals and the public who rely on such agreements.

1.1 The Problem Space

The increasing interest in Computable Contracts has led to the development of numerous Computable Contracting (CC) languages, such as Lexon [3], Logical English [4], and CoLa [5]. Each of these languages offer unique benefits, ranging from code generation to logical reasoning to deontic modelling. However, despite the growing variety of CC-languages, a critical gap remains: the lack of a dedicated drafting environment tailored to the unique needs of drafting Computable Contracts.

Generic text editors are unsuitable for drafting in CC-languages, as they lack the necessary supporting tools — such as syntax validation, semantic checking, and structured drafting guidance — forcing users to rely on manual formatting and increasing the risk of drafting errors. They do not check the syntax of contract text, nor do they guide lawyers through the drafting process. This leads to a drafting process that is both cumbersome and error-prone.

Instead, lawyers would significantly benefit from a consolidated environment offering integrated features such as contract analytics [2] and drafting assistance tools [6, 7]. An integrated development environment (IDE), similar to those used by software developers (e.g., Visual Studio Code [8]), could introduce substantial gains in drafting efficiency, accuracy, and overall productivity, ultimately leading to higher-quality legal documents.

1.2 The Proposed Solution - the CC-IDE

This dissertation aims to develop a novel Integrated Development Environment (IDE) for Computable Contracting (CC), the CC-IDE: a purpose-built IDE for drafting, managing, and analysing computable contracts across a diverse range of CC-languages. The goal of the CC-IDE is to provide a unified development environment for CC-languages equipped with features commonly found in programming IDEs — such as syntax highlighting and error detection — to make CC-languages more accessible and usable for lawyers and legal professionals.

Beyond drafting support, the CC-IDE also seeks to offer a streamlined integration pipeline for language developers, enabling the incorporation of new and existing CC-languages in a simple and flexible manner. Through reducing the manual effort involved in language integration, the CC-IDE aims to provide language developers with an efficient and user-friendly platform that transforms the typical design lifecycle of CC-languages.

Ultimately, the CC-IDE looks to provide a comprehensive contract drafting environment, with supporting tools and analytical features, such that the benefits of CC-languages and Computable Contracts can be leverage to produce higher-quality, more precise legal contracts.

1.2.1 Objectives

1. Create an Integrated Development Environment (IDE) for Computable Contracts
 - Enable a user to write, edit, and save Computable Contracts
 - Enable the integration of existing and novel Computable Contracting (CC) Languages
 - Provide common IDE Features tailored towards Computable Contracting to the user
2. Provide Legal Semantics to CC-Languages
 - Enable mapping of legal semantics onto any CC-Language
 - Leverage legal semantics to provide Contract Analytics, Visualisations and Simulations

1.3 Dissertation Structure

Chapter 2 explores the background context surrounding computable contracting, CC-languages, and parser generators. This includes a review of literature within the field of computable contracting, along with an exploration of the different designs and implementations of CC-languages.

Chapter 3 will conduct an analysis of existing IDEs, their common features, and their applicability within a CC-IDE. Furthermore, a set of fundamental requirements regarding a successful CC-IDE will be presented.

Chapter 4 and 5 feature the design and implementation of a model CC-IDE, highlighting the development process of such a software tool, along with exploring the challenging notion of generalisation for CC-languages. These chapters explores how each requirement of a CC-IDE can be practically achieved, presenting functional results.

Chapter 6 sets out the verification, validation, and evaluation of the presented tool. Initially, a test plan is set out regarding the tool, before the results of the tests are exhibited. An evaluation of the CC-IDE is then carried out against the set of requirements set out previously. Furthermore, critical evaluation of the CC-IDE is also carried out, considering both technological and social aspects. Lastly, the chapter also provides possible future work regarding a CC-IDE.

Chapter 7 then concludes the dissertation, showcasing the key achievements of this dissertation in achieving a comprehensive CC-IDE.

Chapter 2

Background

This chapter aims to understand what constitutes a contract, both in natural language, and in a computable form. It defines Computable Contracts and explores the notion of computability. It then dives into what constitutes a Computable Contracting Language and the different approaches taken towards developing Computable Contracts. The benefits of parsing towards Computable Contracts is then outlined with an exploration of parser generators.

2.1 Natural Language Contracts

Contracts have existed since ancient civilisations, used to hold parties accountable for their actions and obligations — what they should or must do. From employment contracts to rental agreements to End-User License Agreements (EULAs), contracts are used ubiquitously across industries. Their historical prevalence led to the formation of contract law – a legal framework designed to govern and enforce these binding agreements. Consequently, due to their intricate nature and legal nuances, these agreements are typically drafted by lawyers who leverage their legal expertise to ensure a contract is legally sound and enforceable.

2.1.1 Contract components

Within the scope of this dissertation (with particular relevance to computability) an overview of the main components that build up a contract is summarised by Cummins and Clack [2]:

Contract objects and their definitions — Clarification of the meaning of key terms used in the contract

Rights and obligations of the parties (deontic aspects) — What is expected of the parties to fulfil the contract

Operational aspects — How the contract obligations are to be fulfilled

Temporal aspects — Aspects of time for fulfilment of the obligations

Events — Situations that may or should arise during the contract period

Contract states — Scenarios involving the parties, their fulfilment of obligations (or not), the operational and temporal aspects, and events

These different components within a contract are used as building blocks to define the promises

and exchanges made between two parties, serving as a formal representation of their agreement. By defining these components, clarity is provided to the notions expressed within contracts, setting out the different segments of a contract in a structured and explicit manner. Existence of such components, however, do not define the quality of a contract — it’s enforceability and robustness. Instead, this is dependent on the expertise of the drafting lawyer. How sound the promise is, how explicit the obligations are, the coverage of events and edge cases are just some of the benchmarks that a lawyer has to consider in creating a high quality contract.

2.2 Computable Contracts

As mentioned, Computable Contracts offer enhanced clarity, reliability, and efficiency over traditional natural language contracts. However, the infancy of the field introduces varying definitions on what a Computable Contract is and what it must entail.

2.2.1 Defining Computable Contracts

In a general sense, when building a definition of Computable Contracts, we refer first to Harry Surden’s definition of “Data-Oriented Contracts” [9]:

“A data-oriented contract is one in which the parties have expressed one or more terms or conditions of their agreement in a manner designed to be processable by a computer system.”

Surden mentions the ability for a computer system to process an agreement, likened towards the idea of semantics being provided to a contract text [9]. The use of “data-oriented” focuses on how contractual obligations are expressed in the form of highly structured data, whereby each component of a contract is semantically labelled and “understandable” by a computer — processable in some computational sense. This data-oriented approach is supported by the Stanford Computable Contracts Initiative project page [10], stating “When contract terms and conditions are represented in highly structured data, computers are able to process them automatically with high accuracy...” This builds up half of our definition, whereby a contract can be considered “computable” if it has semantic meaning - by the way of highly structured data - to a computer. In particular, a contract should be able to extract the components of a contract mentioned in section 2.1.1. Some standpoints believe that semantic meaning involves the subjection of a contract to partial or full automation [5].

Conversely, in contrast to modern day “Smart Contracts” on blockchain platforms, which are instead self-executing immutable programs with no room for interpretation [11], Computable Contracts should remain as a traditional legal instrument — written in a human-readable form. This readability ensures clarity and enforceability of the contract, remaining accessible for the parties involved, and providing a natural language understanding of the contract’s purpose and intent [12]. This point is emphasised by Cummins and Clack, stating that a contract should be “readily generated, understood, and used by humans”.

Aside from readability, an equivalent expressiveness to traditional legal contracts is also desired for Computable Contracts, allowing for contextual and legal nuance [13]. Contracts have to be understood and analysed within the broader context in which they operate — context that is often external to the contract and not necessarily be computable. Take, for example, a force majeure clause: These clauses refer to exceptional acts, events, or circumstances beyond the control of parties — such as natural disasters or pandemics — that can relieve a party of its obligations [14]. The defining feature of such clauses is the reliance on uncertainty, specifically that these events are unknown and unforeseen until they happen — *post hoc*.

More formally, this is known as Vagueness, a fundamental challenge in legal drafting, where terms like “reasonable efforts” or “best endeavours” are deliberately used to preserve flexibility amid

future uncertainties [15, 9]. Surden notes that contracts often specify obligations with varying degrees of discretion and abstraction, while Clack highlights that, unlike computer scientists who define semantics *a priori* and *ante hoc*, lawyers typically defer semantic interpretation *post hoc*, allowing meanings to emerge contextually during disputes [16].

Contracts remain as Contracts

As a consequence, Computable Contracts must retain their core contract characteristics, even when computable. Law is inherently pragmatic, and as such, the design of Computable Contracts must reflect this principle. They should retain the capacity to support ambiguity and interpretation, accommodating flexibility and legal nuance. More broadly, Computable Contracts should be equivalent to traditional contracts in their expressibility of legal notions, with computability instead serving as an extension of an existing human-readable and legally valid contract.

2.3 Computable Contracting (CC) Languages

Driving these Computable Contracts forward are Computable Contracting (CC) Languages. As the field of Computable Contracts continues to grow, there exists a parallel development of CC-languages, specialised languages used for writing Computable Contracts, with the aim to bridge the gap between legal theory and computational analysis. In particular, CC-languages are designed to express legal concepts, such as rules, regulations, and contracts, in a computable manner. They aim to make legal agreements machine-readable, enabling automation, verification, and improved clarity.

However, each language approaches the relationship between legal language and computation differently. These CC-languages can often be categorised into two different buckets, those similar to programming languages (e.g Prolog [17], Catala [18], CSL by Deon Digital [19]), and those similar to natural language (Lexon [3] and Logical English [4], L4 [20]). The languages similar to programming are designed to be formal languages which follow programming syntax and concepts to encode legal contracts and rules. On the other hand, the languages similar to natural language aim to reduce complexity of legal text while maintaining legal precision. They intend to be directly readable by a human and understood similar to regular contracts that do not employ a CC-language — with an equivalent legal expressivity.

2.3.1 Controlled Natural Languages

These languages similar to natural language are often constrained by a Controlled Natural Language (CNL), subsets of our natural language (in our case, the English that we speak and write) confined to fixed structures and arrangements. Kuhn [21] details a CNL as “A controlled natural language is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax, and/or semantics, while preserving most of its natural properties.” It’s main goal is to “reduce or eliminate ambiguity and complexity” [22], achieved through a formal notion of consistency, the evaluation of whether a text follows the defined syntax (any contradictions or ambiguities are detected and flagged as inconsistent).

The restrictive syntax and lexicon of a CNL is often defined through an underlying grammar — which we will assume to be context-free — denoting the syntactic rules to follow, such as sentence structures and use of certain operators or words. Formally, these grammars can be represented and written using a Backus-Naur Form (BNF) [23, 24], a mathematical notation to represent a strict set of syntax and rules, desired by developers due to their readability and simplicity. Though conflicts may arise (discussed in Section 4.2.4), the BNF remains a flexible and easily modifiable format — allowing for quick alterations and extensions, advantageous characteristics for dynamic

language development. These rules can then be followed to generate well-formed sentences within the CNL, with the derivation representable as a Parse Tree (PT).

The benefit of a CNL is in its retention of human-readability, while providing a representation (the Parse Tree) that can be analysed and understood in a structured manner. In the context of Computable Contracts, the PT breaks down contract text into clearly defined grammar components. This offers transparency and clarity into the structure and syntax of the contract — distinguishing the obligations, events, and conditions — similar to how a lawyer mentally parses a contract, extracting the same components from a contract using their legal expertise.

Contract components can be identified more easily by a computer, in turn enabling interactivity with and integration into a computer system (a task difficult to achieve with a natural language contract without the use of Large Language Models (LLMs) ¹). Likewise, the aforementioned evaluation of consistency can provide assurances of structural correctness, mitigating the risk of human errors.

2.3.2 Semantics in Computable Contracts

While syntactic correctness is necessary for computability, it still requires an explicit set of associated semantics (the associated meaning). The semantic layer of a contract, the actual legal substance of the agreement, typically requires interpretation by the Lawyer who builds a mental model of the contract based on their legal expertise and surrounding context. On the other hand, a computer, without any associated semantic information, will consider all sentences and syntactic constructs as semantically null (and equivalently so), deriving no contextual meaning from the contract. Crucially and resultantly, to achieve computability and be considered a “Computable Contract”, a CNL must associate some form of semantics with the syntax.

However, the idea of semantics is not straightforward either, Clack [13] mentions a legal contract having two aspects, operational aspects and non-operational aspects — one relating to precise actions that can be automated, and the other relating to items that can not. While smart contracts focus on completely operational aspects, being a complete and sole automation of such aspects, legal contracts are very likely to have non-operational aspects, aspects that cannot be quantified or automated. Vagueness is one key example.

Combining Syntax and Semantics

Conceptually, a model of integrating semantics has been explored by Grigg [25], proposing the idea of a “Ricardian Contract”, a document that must be “readable by people and parsable by programs”. Ricardian Contracts are made to be readable and legally binding. Intended to be for legally enforceable financial contracts, the concept could be expanded to general computable contracts.

One application of Ricardian contracts revolves around clause automation or digitisation, along with equipping all clauses with tags and metadata to be extracted and used by a computer system [26]. This follows a markup style language to highlight the components that are to be used in calculations and computations. Similarly, Smart Contract Templates (SCT) [13], leverages a JSON style parameter based markup of a contract to incorporate semantics into a contract. A key benefit of the markup style approach is the direct mapping of semantics onto contract components within the contract text, although this process is often tedious and requires a deeper understanding of the evaluation process.

¹The recent development and widespread adoption of large language models (LLMs) has led to more accurate and effective contract recognition and understanding. However, significant challenges remain, including the risk of hallucinated outputs, inconsistencies in analysis, and the substantial carbon footprint associated with training and deploying these models. Due to the complexity of this issue, it will not be covered in this dissertation.

Another approach involves the combination of contract and code into a single artefact. Goodenough and Salkind [27] mention that one approach to Computable Contracts is where a “natural language representation of the contract could co-exist along with the computer program representation of the contract.” Although one form of this approach is the inclusion of code blocks alongside contract text (Which is not dissimilar to a markup language; and used for Catala [18]), Clack [16] describes a more interesting methodology to this approach, whereby a merged combination of a CNL and Domain Specific Language (DSL) — a purpose driven computer language — are used to represent a contract as “a single artefact expressing both contractual obligations and the automated implementation of those obligations.” In this model (Used by the aforementioned CNL based languages), the natural language text serves as both the human-readable contract itself and the underlying machine-readable code. The benefit of such a model is that a contract can be written and understood similar to any other natural language contract, but at the same time enable computation of contract aspects.

2.3.3 Existing Computable Contracting (CC) Languages

Advancements have been made to develop a merged CNL and DSL for Computable Contracts, where legal documents can be drafted in a form both human-readable and machine-understandable. These languages integrate code into a natural language form, allowing for a contract to be drafted in natural language with a direct translation into a computable representation. Multiple explorations have been conducted into existing CC-languages that leverage both a CNL and DSL [13, 5, 2, 28].

A plethora of works exist including: “Lexon” digital contracts [3], Logical English [4], Controlled Components Contract Language (CoLa) [5], the work done at Stanford University’s CodeX Centre for Legal Informatics under the Stanford Computable Contracts Initiative [10], L4 from the SMU Centre for Computational Law [20], the Domain Specific Language (DSL) Marlowe (although writing is done through a drag-and-drop interface) [29], the CNL Attempto Controlled English (ACE) – although not yet used for contracts [30], and the combination of a CNL and deontic logic from Pace and Rosner [31].

Contract Drafting with Existing CC-languages

As noted by Fattal [5] and Chong [28], these different languages and works are promising for generating computable contracts, with the ability to encapsulate a wide range of different complex contract examples. However, a clear gap remains in the development of a shared set of tools for these languages. Each language approaches computable contracting using its own toolchain:

- L4 adopts a spreadsheet-like drafting model, where clauses, conditions and sub-conditions are laid out in rows and hierarchy is indicated by fixed-width indentation, enabling authors to see clause structure at a glance [20].
- Lexon has a web-based compiler that lets you write, compile and test contracts live in the browser [3]. It then translates the contract into solidity, a lower level language.
- CoLa is implemented in Miranda, carrying out lexing and parsing to generate an Abstract Syntax Tree (AST). This tooling is delivered as a CLI [5].
- Logical English provides a VS Code extension (called “le-ui”) backed by a Language Server, offering smart syntax highlighting, template-aware code completion and real-time error detection within the IDE.

Semantics within Existing CC-languages

Furthermore, within these works, embedding of semantics (of the DSL) into the CNL is carried out differently – done on a case-by-case basis for each language. One such form is to embed semantics through the keywords or terminology of the CC-language. In the case of Lexon [3], there exists 91 keywords which have a fixed semantic meaning within the language – relating to different functions, types, and operators – and are used in the compilation process to provide machine interpretability. CoLa [5] on the other hand, uses non-terminals to define semantic concepts, highlighting contract components from a rule-based standpoint – with higher level concepts like clauses, statements and conditions represented as rule symbols.

2.4 Parsing

Parsing is a foundational process for extracting both structure and semantics from Computable Contracts. It is carried out by a parser, a tool that analyses the syntactic structure of text using a pre-defined set of rules (a formal grammar specification) from a stream of lexical tokens[32] and generates an output in the form of a Parser Tree (PT). In the context of CNL based CC-languages, the set of rules is the underlying BNF grammar.

2.4.1 The Parse Tree (PT)

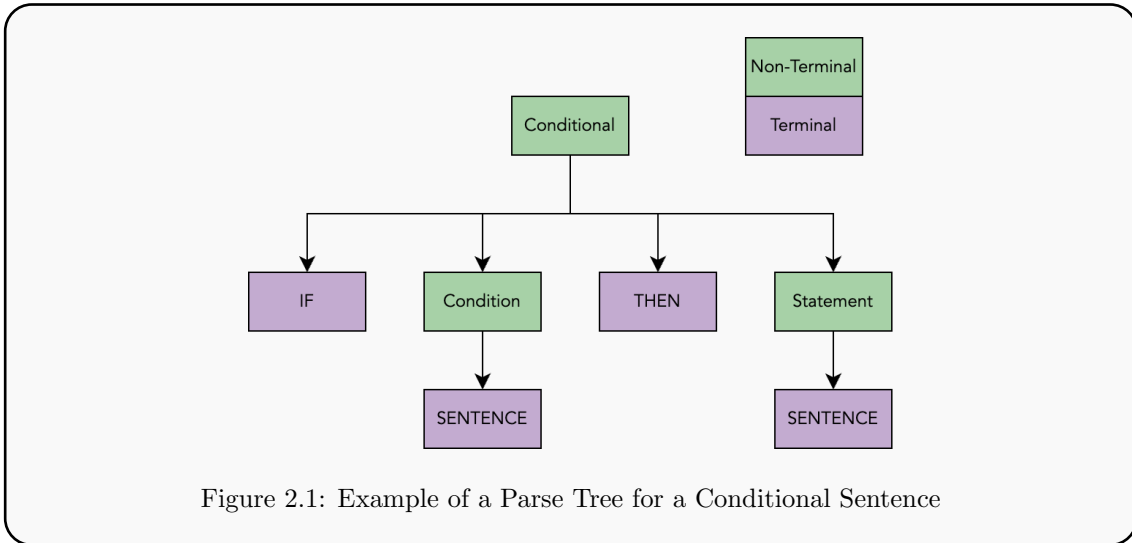


Figure 2.1 illustrates a PT for a simple conditional sentence. In this hierarchical model, internal nodes denote non-terminal grammar symbols, while leaf nodes correspond to terminal tokens. Its tree structure enables traversal via standard algorithms — Depth-First Search (DFS) or Breadth-First Search (BFS), enabling systematic inspection of clauses, statements, and conditions. In Computable Contracting, this PT provides a computable representation that facilitates downstream semantic analysis, validation routines and automations. Traditionally, PTs are often derived via a custom-written parser. Developments in this field have led to the emergence of effective and efficient tools that automatically generate parsers for any grammar (e.g ANTLR4 [33] and Tree-Sitter [34]) — Parser generators.

2.4.2 Parser Generators

Parser generators automatically synthesise parser implementation code from grammar specifications, supplying the lexing and parsing routines needed to construct a parse tree (PT) that precisely mirrors the language’s formal syntax. By eliminating the manual effort and error-proneness of hand-coded parsers, they ensure accuracy to the intended syntactic structure and significantly accelerate the development of parsers. In the case of Computable Contracts, this process can be leveraged to automatically develop a parser for a CC-language. Two widely used parser generators are ANTLR and Tree-sitter.

ANTLR

ANTLR is a parser generator that generates Top-Down LL(*) Parsers from developer-defined grammars [33]. Its parsers guarantee deterministic, precise recognition and automatically construct a complete parse tree for any valid input. Upon encountering lexical or syntactic inconsistencies—such as malformed tokens or rule violations—ANTLR’s built-in error-reporting mechanism immediately halts parsing and emits diagnostic messages, thereby enabling rigorous validation of the underlying structural definitions. Furthermore, by generating a visitor class alongside the parser, ANTLR affords a straightforward, node-centric traversal API, allowing developers to implement custom processing or semantic analysis for each parse-tree node with a boilerplate skeleton code.

Tree-Sitter

Tree-Sitter is a parser generator that generates Bottom-Up GLR parsers renowned for their robustness and flexible error-recovery mechanisms [34]. Traditional parsers employ a “fail-fast” strategy: upon encountering malformed input (e.g. missing punctuation or a typo), they immediately raise an error and halt all further analysis. In contrast, Tree-Sitter replaces each erroneous segment with an “ERROR” node in the parse tree and continues parsing subsequent tokens. However, edge cases do exist: In some cases, Tree-Sitter will try a best case attempt, jumping over large text spans in search of a valid subtree.

Equally important is Tree-Sitter’s incremental-parsing capability. Rather than rebuilding the entire parse tree after each edit — a time and resource intensive process for lengthy texts — it restricts reparsing to only the modified fragments, efficiently splicing updated subtrees back into the existing structure. This targeted approach significantly reduces redundant computation, providing near instantaneous parsing and a more responsive editing experience. Finally, similar to ANTLR’s visitor-class traversal, a Tree-Sitter parse tree can be navigated directly as a conventional n-ary tree, simplifying downstream analysis and tooling.

Chapter 3

An Integrated Development Environments for Computable Contracts

This chapter explores Integrated Development Environments (IDEs) more closely, critically analysing their features along with generalising common characteristics. The context of computable contracts is then applied onto IDEs, ideating a computable contract IDE (CC-IDE), before setting out a list of requirements of characteristics that could constitute a CC-IDE.

3.1 Integrated Development Environments (IDEs)

An IDE is a type of software application designed to make programming more efficient. As opposed to a generic text editor, whereby a programmer will write their code with little to no analysis and compile it via the command line afterwards, an IDE provides an easier and more responsive way to write code. Code is analysed as it is written, providing the programmer with instant feedback. It consolidates different aspects of program writing, providing tools and functionality (with common features highlighted in Section 3.1.2) to help the developer write clear and maintainable code, contributing to a more productive environment [35][36]. Due to their effectiveness, IDEs are used throughout different industries by programmers for software development, data science, and more. In order to understand the advantages of an IDE, we must first analyse existing IDEs, looking at their features and benefits.

3.1.1 Critical Analysis of Common IDEs and their Characteristics

There are two types of IDEs: Language Specific IDEs — e.g. IntelliJ IDEA for Java and JVM based languages [37] or PyCharm for Python [38] — and General Purpose IDEs — e.g. Visual Studio Code (VSC) [8] or VIM [39]. These IDEs all aim to offer a streamlined coding process, enhancing the coding experience for developers through a plethora of supporting features. The key difference is that language specific IDEs focus on one programming language, providing an advanced suite of features tailored to that language, whilst a general purpose IDE is compatible with multiple languages, employing plugin-based extensions to provide language-specific features.

IntelliJ IDEA

IntelliJ IDEA by JetBrains is widely recognised as a powerful and versatile development environment [40]. It is primarily designed for code development in Java and Kotlin, both of which run on the Java Virtual Machine. It was one of the first tools to provide refactoring (automatic code restructuring without affecting functionality [41]) support [42], and provides a comprehensive suite of features for its languages directly out of the box — such as syntax highlighting and code auto-completion.

A key advantage of the IntelliJ IDEA is that upon installation, a majority of features are fully integrated and can be used directly without extra configuration. Configuration can at times be tedious and confusing, and as such, automated configuration saves users time and effort. IntelliJ IDEA also has a graphical interface which is beneficial for learning, along with keyboard shortcuts which provides added efficiency [43]. Furthermore, there exists an ecosystem of additional plugins, extending the IDE towards other languages and tools, making the IDE more flexible and comprehensive. Developers do not always focus on a single language — take a website for example which may use JavaScript for the frontend and Java for the backend. This requires flexibility within the IDE to handle multiple languages.

However, with IntelliJ IDEA, a point of criticism is the abundance of options and features, which may be difficult to learn and unintuitive to use, affecting productivity gains as users may not be able to use the IDE in the designed manner. Furthermore, the interface may be difficult to navigate as it contains a number of features which may not be relevant to a starting user, for example the debugging and logging features, which are likely relevant only to advanced programmers.

PyCharm

PyCharm is a popular Python development environment also developed by JetBrains. The features are similar to IntelliJ IDEA, however, it does not have the same compatibility for other languages. It is instead focused exclusively on Python, with a focus on data science and web development, and providing a plethora of Python specific tools, such as refactoring, web framework support, and Jupyter Notebook integration, alongside general IDE features such as autocomplete and syntax highlighting. These tools are crucial to Python development, and PyCharm wraps it into a single development environment, making it accessible and easy to use. As a result, users often find PyCharm to have a better user experience for Python development, as opposed to a more general IDE which may require configuration to use. A downfall of PyCharm is similar to that of IntelliJ, whereby, while the abundance of features are beneficial and immediately available, it may distract a user from their code, and overcomplicate the development process.

Visual Studio Code

Visual Studio Code (VSCode) is another popular IDE, with a focus on a plugin based system. By itself, without extensions, it is a generic text editor. However, compatibility with different languages can be achieved through extensions and plugins. As a result, it can be configured to handle multiple languages and contains a plethora of supporting tools and features that can be integrated such as formatters, linters, and more. More broadly, the collection of aiding features is referred to as IntelliSense [8]. Aside from the benefits gained from the graphical interface, similar to IntelliJ and PyCharm, the customisability also stands out. By following a plugin system, users have greater control over the IDE, making it easier to prevent an overabundance of functionalities. However, this is a double edged sword as a user will have to know exactly which extensions to use and install and features may not be as comprehensive as that within a language specific IDE — though this is dependent on the configuration, and if chosen correctly, the different extensions can match language specific IDEs like IntelliJ and PyCharm to a high degree.

Vim

Vim is similar to VSCode in the manner of a plugin system, but is exclusively a text editor, utilising keyboard shortcuts as opposed to a graphical interface. Vim leverages a key-bind system to allow for productivity shortcuts, matching different keys and written commands to different functions. Once mastered, this can allow for fast and efficient text editing, which is highly beneficial in writing code. Nevertheless, this also leads to a steep learning curve, requiring developers to memorise functionality within the editor, whilst not providing a menu or toolbar to use. Alongside memorisation, Vim is also difficult to setup in regards to extensions and plugins, and requires a user to actively build the system that they desire. Unlike VSCode, where plugins are managed automatically upon installation, Vim will require a user to manually manage plugin installation, configure plugins, and ensure that they are integrated correctly. As such, this also adds an overhead in maintenance and troubleshooting unlike other IDEs which typically handle setup for the developers.

Overleaf

Overleaf, an online LaTeX editor may not be a traditional IDE, as LaTeX can be considered as a markup language as opposed to a programming language, however, Overleaf can be considered a strong example as to how an IDE looks like for document writing — equivalent to drafting contracts. An interesting concept within Overleaf is the split between the editing of the LaTeX document on the left, and the output document post processing on the right hand side — seen within Figure 3.1. In IDEs for code, as the end goal is not a document but to compile and run the code, there is no desire for an entire output page alongside the coding page — although this can exist within VSCode with the right extensions. In the context of Computable Contracts, however, this concept would be highly desirable, as it would enable a lawyer to have a live preview of the final contract, beneficial in checking for errors and correctness.

Overleaf is also intriguing as it supports real-time collaboration with multiple users. It also contains numerous beneficial features that would be found within a text editor like Microsoft Word such as text formatting, document history, and spelling check [44].

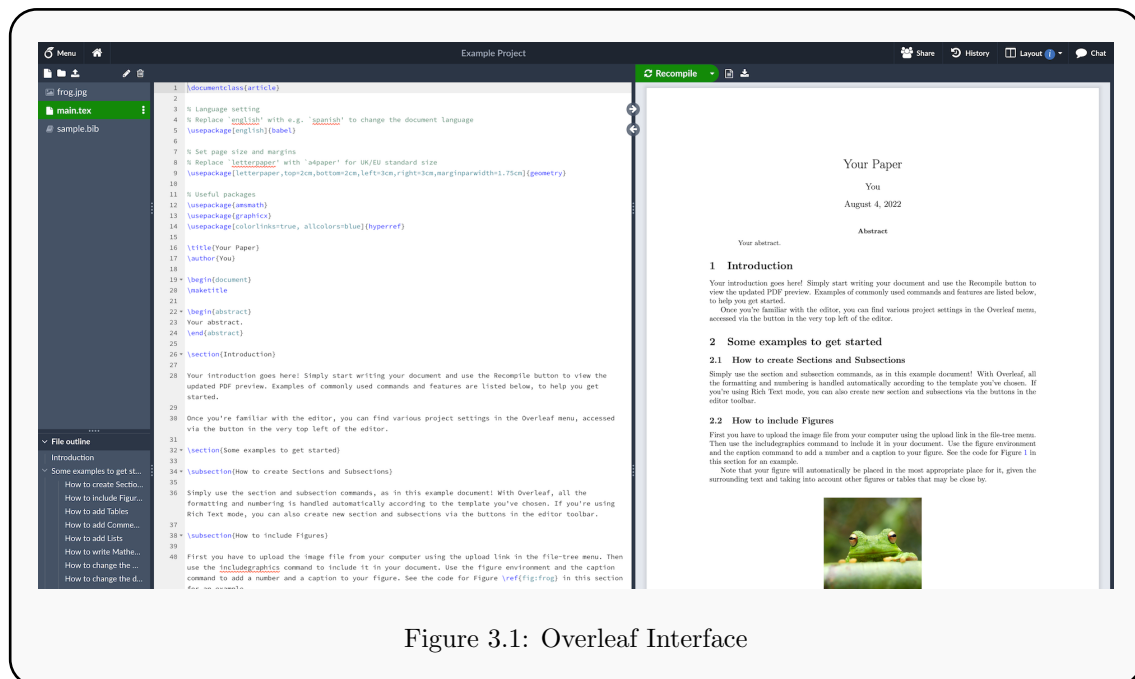


Figure 3.1: Overleaf Interface

Language Workbench

IDEs are also not limited to coding languages, and may also be for creating languages themselves. Language Workbench is a term coined by Martin Fowler in 2005 [45], describing a software development tool geared towards building Domain Specific Languages (DSLs). DSLs are languages that are designed for a specific task or domain, speeding up the development process within that domain. The rationale behind language workbenches is to empower domain experts to design and implement a language to express complex ideas more naturally than would be possible with general-purpose programming languages.

Two common language workbenches are Racket [46] and JetBrains Meta Programming System (MPS) [47]. Within Racket, you are able to define a new DSL quickly using a set of predefined basic operations, building macros to create new functions and expressions. JetBrains MPS on the other hand is focused on building fully fledged DSLs defining not just the structure of the language, but also the constraints, behaviour and compilation. It focuses on enabling a complete DSL creation, whereby a developer would define each element of a language in detail. An interesting concept within is projectional editing, whereby a user could interact with the DSL in non-traditional ways, such as tables or trees. Notably, JetBrains MPS contains both a DSL developer as well as an IDE for these DSLs upon completion.

3.1.2 What are the common features of an IDE?

As defined in the AWS IDE page[48], and seen in the aforementioned IDEs, modern IDEs often include the following features: Code Editing Automation, Syntax Highlighting, Intelligent Code Completion / Suggestion, Refactoring Support, Local Build Automation, Compilation, Testing, and Debugging. Often within an IDE, this set of code editing features is achieved through a combination of language-specific tools.

A main set of features and their implementation will be analysed below — specifically syntax highlighting, code auto suggestion, error detection, and reformatting. Understanding how these could be applied within a CC-IDE to benefit the Lawyer first requires an understanding of what these features entail and how they are achieved for a programming language:

Syntax Highlighting: Users experience improved readability of their code, with visual cues differentiating keywords and syntactical elements (Figure 3.2 illustrates keyword highlighting within the VSCode IDE). These visual provide instant feedback to the user, enabling users to quickly identify and correct syntax errors [49].

Syntax Highlighting employs a language-specific Lexer to initially break the source code into a list of tokens. Each token, representing a lexical unit (e.g. keyword, identifier, operator) is then mapped to a specific color or style based on the provided theme [8, 50]. In VSCode, this process leverages regular expressions defined within a specialised grammar for token recognition. This is typically performed incrementally — often using Tree-Sitter — within modern IDEs, optimising it such that only the modified region of the source code is re-analysed.

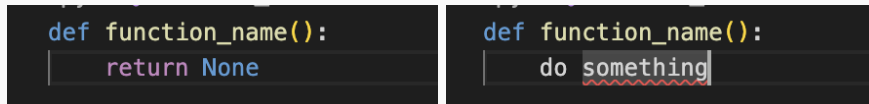


Figure 3.2: VSCode Syntax Highlighting and Error Detection

Auto Suggestion: Users benefit from prompts that aid them in writing code by offering suggestions on how to complete an expression or word. This accelerates the coding process through

minimising errors and reducing the reliance on the memorisation of names and expressions, which may be difficult in large programs and projects.

Auto Suggestion primarily utilises the Language Server Protocol (LSP). The IDE communicates with a dedicated language server to provide language-specific insights and suggestions. This language server — implemented by the developers of a language — analyses the code, leveraging the information it has of the enclosing program, to provide a list of likely proceeding text options. This is presented to the user as a picklist [8] (Figure 3.3 illustrates a picklist within the VSCode IDE, providing smart code suggestions). While this process is seamless from the user’s perspective, the underlying implementation of a language server involves a significant amount of intricate and often tedious work, requiring the consideration of the vast majority of use cases within a language.

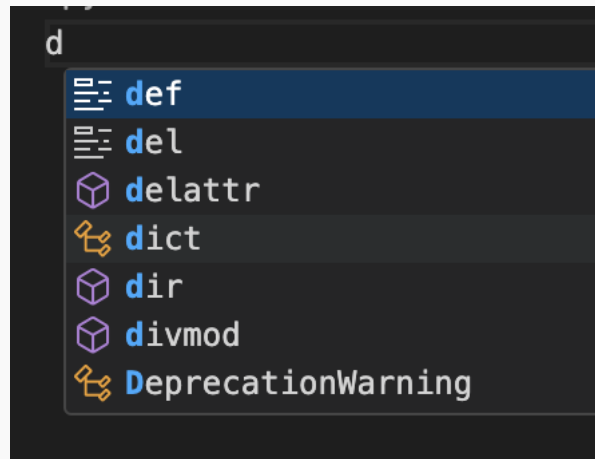


Figure 3.3: VSCode Auto Suggestion

Error Detection / Debugging: Users receive immediate feedback on mistakes within their code as it is being written. Commonly, this is achieved through a visual cue highlighting the error, bringing it to the user’s attention. Users can then efficiently locate and rectify the error.

Error Detection runs a parser that adheres to the grammar of the programming language. The parser analyses the code to determine if its structure and syntax is consistent with the grammar. When a construct within the code violates the grammar, it is then flagged back to the user within the IDE. (Figure 3.2 shows an error underlined in red) Beyond syntax errors, IDEs and language servers may also perform semantic analysis to detect errors in logic [51].

Auto Formatting: Users experience improved readability of their code through structural fixes and application of formatting. It delivers consistency in code structure and automates time-consuming formatting tasks [52]. Auto-formatting focuses on indentations, paragraph structure, and the output style as opposed to syntax highlighting which focuses on just the colouring of text.

Source code is initially processed and broken down via a parser into an Abstract Syntax Tree (A concise version of a Parse Tree), which represents the hierarchical structure of the code. AST transformations are then applied to modify the format and structure of the code. Code formatting often requires a separate reformatting process — delegating to tools such as the Python Black code formatter which automatically adjusts indentations, spacing, line lengths, and other stylistic elements based on predefined rules. (Figure 3.4 and Figure 3.5 displays how a piece of long and unstructured code could be refactored for readability)

The benefits of these features are proven to benefit programmers in writing code — e.g. an investigation by Sarkar [53] presents that the presence of syntax highlighting significantly reduces task completion time as well as reducing context switches. Another study by Debuse and Lawley

```
retstr += "\t" * (len(store_arr) + 1) + '<b style="color:#01422f;">' + node.text.decode("utf-8") + "</b>" + "\n"
```

Figure 3.4: Unformatted Code within VSCode

```
retstr += (  
    "\t" * (len(store_arr) + 1)  
    + '<b style="color:#01422f;">'  
    + node.text.decode("utf-8")  
    + "</b>"  
    + "\n"  
)
```

Figure 3.5: Formatted Code using Black within VSCode

[54] claim that introductory programming can be taught using a complex commercial IDE, with most IDE aspects improving their understanding and productivity — although the study is not completely controlled and may suffer from biases such as the participants having previously used the IDE.

Within this section, we have not analysed build automation, compilation, testing, and debugging, as they are highly language specific and can vary drastically. Take Java and Python during a debugging process for example, Java is primarily a compiled language, and as a result, can catch most syntax and possibly semantic errors during the compilation *before a program is run*. Python on the other hand is interpreted, and will typically only realise errors during runtime *after a program is run*. Practices differ massively between languages, and as such, these features are difficult to generalise.

3.2 Understanding IDEs for Computable Contracts

Each IDE has its own advantages and disadvantages. When looking specifically at an IDE for computable contracts — the CC-IDE — it is crucial to identify how an IDE could be built around the drafting of computable contracts as opposed to development of software. We must identify the advantageous components of the different IDEs, whilst ensuring avoidance of common pitfalls when considering a foundation for an effective and comprehensive CC-IDE.

3.2.1 The IDE Developer, Designer and the Lawyer

Firstly, the notion of a legal expert within the CC-IDE has to be divided further into two categories. The Designer and the Lawyer. By the Designer, we mean the person responsible for configuring the legal grammar, as well as the formatting options and analyses. The Designer may be a Lawyer with a coding background, or on the contrary, a programmer with legal expertise — however, it is becoming ever more common for law firms to have an in-house development team. By contrast, the Lawyer is any person who uses the CC-IDE for contract drafting purposes. The Lawyer may be a lawyer or legal professional in charge of drafting contracts.

This distinction is necessary as conceptually the CC-IDE is divided into two different interfaces: One concerning the development and design of legal grammars, and the other concerning the use of these grammars. The Designers are expected to have a better understanding of code and CC-languages, allowing them to be directly engaged with the grammar, creating the language which gets used within the CC-IDE. Furthermore, as every CC-language is different, they are also responsible for documenting their own grammar along with making design choices and leveraging the tools that we provide within the CC-IDE to maximise the effectiveness of the language in expressing their legal sub-domain.

The design of the CC-IDE starts with the tools provided by the IDE developers — in this case, the writer of this dissertation. It then progresses to the usage of these tools by the Designer to create and design the grammar of their CC-language. Thereafter, the Lawyer will use the language within the IDE to draft contracts more efficiently and comprehensively. It is upon these layers of dependencies that the IDE is built, particularly with a focus on automation and abstraction to make the process of both developing legal grammars and drafting contracts easier.

3.2.2 Designing a CC-IDE

Looking first at the difference between building a language specific or general purpose IDE, the desire for the integration of different existing and new CC-languages requires a focus on the IDEs capable of integrating multiple programming languages — i.e. VSC, Vim and IntelliJ. All of them incorporate a plugin approach, whereby programming languages can be installed and integrated to the IDE via downloadable extensions. This concept of extensibility should be reflected in the CC-IDE, where different CC-languages can be applied and used.

Language Development & Contract Drafting

Similarly, the JetBrains MPS also displays an interesting solution, whereby a DSL can be both created and used within the same application, creating a pipeline from the designer of the DSL to the user. With CC-languages, this would mean being able to both define the language and use it immediately within the same interface. This is a significant benefit towards the accessibility of CC-languages, encapsulating both language development and contract drafting without additional tooling work. There is no current known implementation of this; it would be highly beneficial for a combination of the development or extension of CC-languages and their immediate application.

IDE Features: Ubiquitous throughout these different IDE options is the different code support features, namely syntax highlighting, code autocomplete and suggestion, and error detection. These features help boost productivity and efficiency when developing code, analysing the code as it is written and providing developers with instant feedback, all while aiding readability. The CC-IDE could provide the same benefits towards contract drafting through leveraging similar features for CC-languages.

In this case, syntax highlighting could be applied to key contract components or key words (Both lexical and syntactic elements), providing visual cues to contract components, flagging them within a contract such that they can be extracted without having to understand the full context of a contract. Error detection would benefit in detecting contract inconsistencies, with additional visual cues to denote where an error has occurred. Code suggestion could be likened towards contract component suggestion, providing the next component that would fit within the contract at a point of development. As opposed to requiring the Lawyer to memorise the legal structure, they would instead be prompted with the next component, especially beneficial towards new Lawyers and those unfamiliar with the in-house style. Overall, these features would make the process of contract drafting more efficient and productive.

Automatic Integration of Languages: Looking towards PyCharm, language specific IDEs are advantageous in how they provide “out of the box” directly available features to their coding

language. If a set of directly available features could be immediately used for any CC-language upon application, this would be hugely beneficial to not just the Lawyer but also the Designer, providing them with supporting tools and features without extensive configuration (unlike Vim or VSCode). However, an overabundance of features and tools should be avoided as it may distract the Lawyer from their main tasks, especially those not beneficial to a Lawyer. A restricted set of features would prevent over complication of the interface.

User Interface: Looking towards the interface of the CC-IDE, the interface of Overleaf would be highly beneficial if tailored towards computable contracts, providing both a contract development side as well as an output side. This gives the Lawyer a comprehensive and focused drafting experience on one side, whilst outputting a well formatted and organised document on the other side. It allows the Lawyer to write in a more rapid and unstructured way, while formatting the contract automatically, providing a polished outcome of their contract. This could provide productivity benefits as the Lawyer could write more efficiently through the elimination of manual formatting tasks.

Existing CC-IDEs: In terms of existing IDEs for computable contracts, none are prominent and used within the field. There has been no consolidation of tools for different CC-languages, and no tool for developing CC-languages either. As explored in Section 2.3.3, each CC-language has their own extension or application: Lexon has an online compiler and can generate Solidity code, but it doesn't have the productivity features of an IDE. Logical English has a VS Code extension but cannot be extended as a language. CoLa has a separate parser and lexer but these have to be redeveloped each time the language is changed. There is no unifying solution providing a development environment for drafting in different CC-languages, and especially not one that allows for editing and extending a CC-language.

Translation and Representation: Likewise, in most cases, it is also difficult to use the contract in a way different to the design of the CC-language, providing users with only the designed outcome and no usable representation — take Lexon for example, it can be compiled to Solidity, but a user would not be able to compile it to another lower level language. It would be beneficial for each contract in a CC-language to be generated in some known and easily manipulatable representation such that it could be used in other ways.

Contract Analytics: An additional point that could be beneficial specifically for a CC-IDE would be the use of contract analytics. Research [28] has been conducted into the analysis of contracts, particularly concepts like consistency, completeness, and complexity. Providing the Lawyer with these different analyses at the time of writing would be advantageous, allowing them to understand how their contract may perform, as well as adherence to the desired structure given by a CC-language (its consistency).

3.3 Computable Contract IDE (CC-IDE)

Through an examination of IDEs and how different components could be used towards a CC-IDE, it is possible to then form a set of rudimentary requirements that would constitute a successful CC-IDE, along with supporting reasoning. In particular, the CC-IDE serves to benefit both the Lawyer and the Designer, providing a platform for the two to interact. The Lawyer has a focus on drafting contracts. They require the use of CC-languages (possibly more than one), and have an end goal of a comprehensive and complete contract. Contrarily, the Designer has a focus on developing CC-languages as well as supporting tools. They aim to create languages that provide productivity benefits to the Lawyer, leveraging the semantic benefit of computable contracts to achieve different analyses, visualisations and simulations. A successful CC-IDE would accommodate both parties in achieving their respective tasks and roles.

- *(RQ1) Be able to write, save, and edit computable contracts*

A trivial requirement. In its most basic form, an IDE is a glorified text-editor, and the

foundation should remain as such. A CC-IDE must provide, at the bare minimum, a basic text editor for drafting and modifying contracts.²

- (RQ2) *Extensibility for different Computable Contracting (CC) languages*

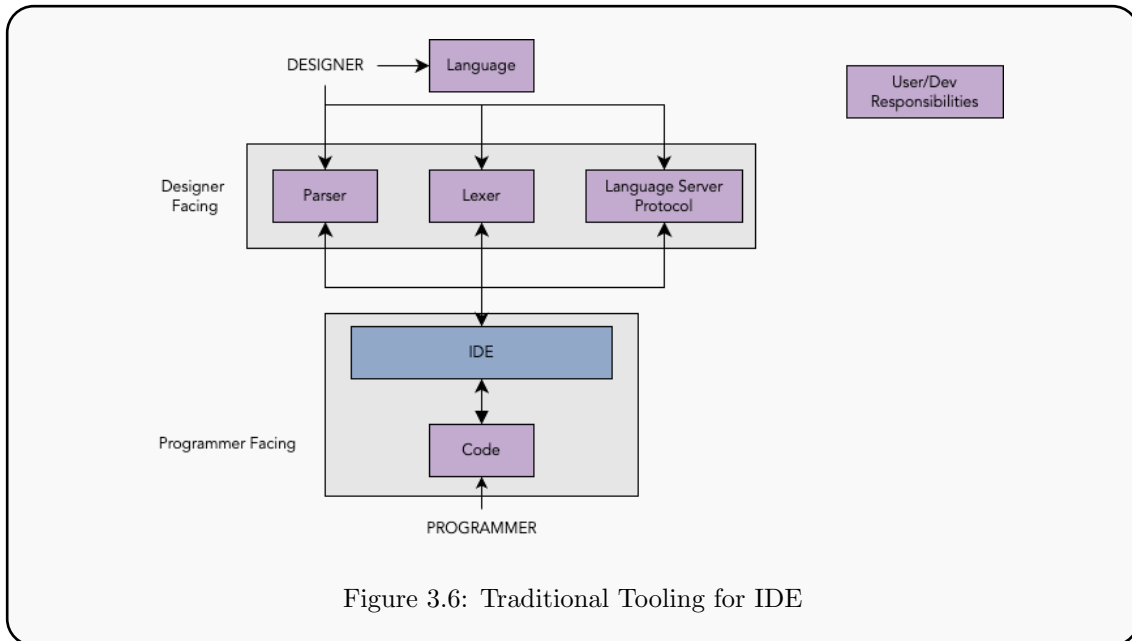
This is an interesting and difficult requirement applicable to both the Lawyer and the Designer. The Lawyer has a desire to write in different CC-languages, while a Designer has a desire to implement said CC-languages. The focus of the IDE is to enable this use of different CC-languages, and in doing so, there must be some way of integrating and applying CC-languages within the IDE — in a simplified manner.

This is challenging as there needs to be a system that can convert a standard representation of a CC-language and enable it for use, without requiring the Designer to manually create a complete DSL and extension for the language — a time consuming and cumbersome task. Resultantly, a pipeline must exist for any CC-language to be applied within the IDE for immediate use by the Lawyer, such that there can be a parallel development and use of a CC-language.

- (RQ3) *Common IDE Features are applied interactively onto computable contracts*

The preference of an IDE over a generic text editor are attributed to the plethora of IDE features that are provided. In particular, the inclusion of syntax highlighting, auto-suggestion, error detection, and auto formatting would be beneficial in promoting greater productivity and efficiency towards contract drafting.

While static analysis of a completed contract is beneficial and is easily achievable for computable contracts, contracts will instead often be incomplete throughout the drafting process, requiring analysis of the contract as it is being written. As a result, a challenging aspect to this requirement is that these features have to be provided interactively, similar to a standard IDE. The solution cannot be static and should be applicable to inconsistent and unfinished contracts.



Applying these features for **ALL** CC-languages within the CC-IDE adds an additional layer of complexity. IDEs typically rely on language-specific tooling, and cannot provide these features without the Designer of a language integrating a suite of supporting tools. These tools have to be written from scratch and integrated within the IDE – for each individual

²It is important to note that the focus of this CC-IDE is not on the interface but on the fundamental concepts behind CC-language integration — particularly when syntax and semantics are unknown. As a result, a high aesthetic quality of the user interface is NOT a requirement.

language. The process highlighted in Figure 3.6 shows the different tools implemented by the Designer: The Designer not only has to handle the design of a language, but also develop and maintain a well-defined parser, a linter, and a language server in order to achieve these features — which demands considerable effort, especially as the complexity of the language grows. While it may be argued that such tooling is often already implemented during compiler development for a programming language, this is not necessarily true for CC-languages. CC-languages are often intended as modelling tools rather than for direct execution, meaning traditional compiler pipelines — and the associated tooling — are frequently unwarranted or completely absent.

- *(RQ4) Semantics for **all** CC-languages*

As hinted in the definition of computable contracts, there must be the ability for a computer to semantically understand a computable contract. As such, there must exist a way to embed semantics within a CC-language in the CC-IDE, such that it can be extracted and used computationally. This is a demanding requirement, as semantics are complex as explained in section 2.3.2. There is no universal solution to applying semantics to computable contracts. Furthermore, the desire within the CC-IDE to be able to apply semantics to **ALL** CC-languages, compounds the difficulty, as this solution must be flexible towards different CC-languages. As the computability of contracts from the semantics underlies the entire IDE, the lack of some semantic system would mean the IDE has no benefit over a generic text editor towards computable contracts.

A successful CC-IDE would take a flexible approach towards semantics — whether following a combination of a CNL and DSL or using a markup form, and accommodate its application to different CC-languages.

- *(RQ5) Analyses, Visualisations, and Simulations for CC-languages*

Aside from just language and contract development, a CC-IDE would benefit heavily from tailored analyses, visualisations and simulations on a contract. A CC-IDE should set itself apart from other IDEs via its customisation towards computable contracts. Although the application of semantics, as expressed in the previous requirement, are what make a contract computable, there must then be a use of the semantics in order to drive forward the benefits of a computable contract. These semantics should be leveraged to benefit the Lawyer, providing greater insights into their contract.

Different languages approach this differently, with some looking towards contract automation, and others looking towards contract analyses. What is necessary here is to exemplify the ability to leverage the semantics of a computable contract, and use it in a computable form, demonstrating the benefits of computable contracts. In the case of a successful CC-IDE, the semantics of a computable contract would be leveraged through different analytical purposes, such as through different contract analyses, visualisations and simulations. Additionally, these analyses should be applicable to computable contracts in **ALL** CC-languages.

Generalisation is a big theme within the IDE, and is the largest complexity to solve. Languages are inherently different (although some components may be shared), and as such have to be handled within the CC-IDE. As language syntax and semantics are unknown ahead of time, the task of ensuring compatibility with each language is markedly difficult. As a result, the aim of this dissertation to set out a framework and system that attempts to solve the generalisation problem above all else, placing an emphasis on flexibility to incorporate as many existing and novel CC-languages as possible.

Chapter 4

Constructing the IDE

This chapter focuses on the foundation of the CC-IDE. Initially, it explores the technology stack of the CC-IDE and how this facilitates both an interactive contract drafting process and a comprehensive CC-language development process. It then dives into the integration of CC-languages, the challenges behind it, and the solutions within the CC-IDE to provide contract parsing and processing capabilities. Lastly, it focuses on the implementation of different IDE features within the CC-IDE, particularly the challenge in providing a universal set of features compatible with all integrated CC-languages.

4.1 A Basic Text Editor

The CC-IDE is built around the concept of a generic text editor — the foundation of any IDE. Developed as a web application inspired by the Overleaf interface, it uses Python for its cross-platform compatibility and robust ecosystem, with Flask providing the web application framework for the user interface [55]. The frontend employs HTML for structure, CSS for styling, and JavaScript for interactive functionality, creating a modular system that allows for flexible contract output formatting and visualisation while maintaining an integrated pipeline between drafting and processing components. The structure of this web-application forms the foundation of the IDE, serving as a motherboard onto which additional functionalities can be added. Furthermore, the interpreted nature of Python is leveraged for semantic benefit as displayed further in the project (Section 5.2).

4.1.1 Developing the GUI

This CC-IDE’s technology stack delivers an interactive and well-formatted environment through which a Lawyer can draft contracts and a Designer can develop CC-languages. A focus is placed on simplicity and functionality, simple in the number of features and the CC-IDE’s usability, while functional in providing the necessary tools and features. Likewise, it focuses on modularity and extensibility of the back end, enabling integration of different CC-IDE components such as the main parsing engine, while being easily extensible.

Shown in Figure 4.1 is the main interface of the CC-IDE, containing a dual-view editing page. The logic of the main editor of the CC-IDE is as follows: User interaction happens in the frontend, whereby contracts are drafted in the main editor (On the left). The frontend sends a request to the backend for contract parsing and processing. The backend receives the request and processes the contracts. The processed output is then updated in the Parsed Output (on the right). This process happens instantly and interactively, providing a seamless contract drafting experience.

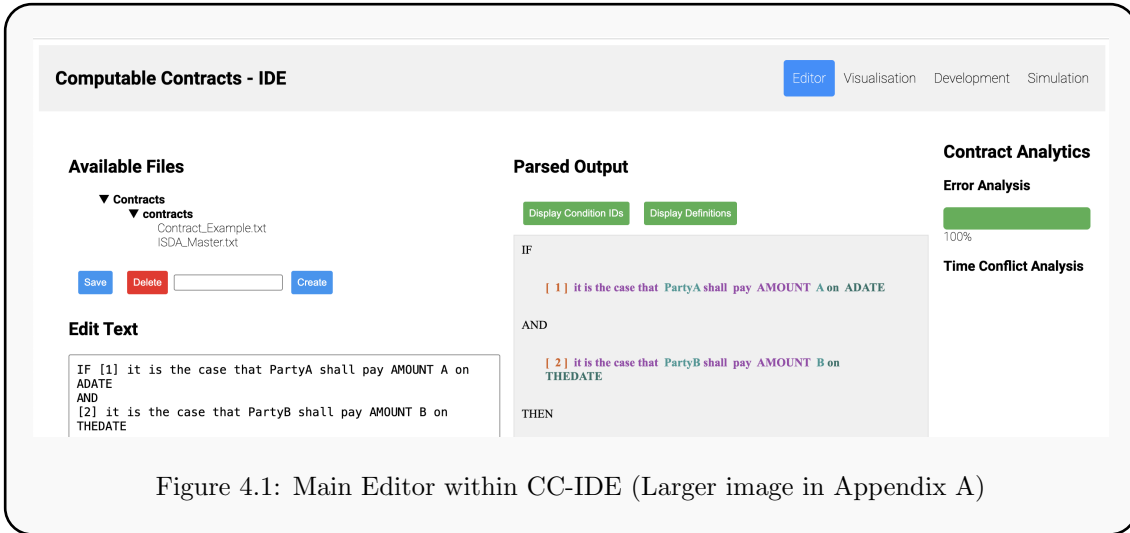


Figure 4.1: Main Editor within CC-IDE (Larger image in Appendix A)

Although a trade-off is that pinpointing specific output-text relationships (e.g. contract errors – described in Section 4.3.2) may be challenging, the dual view is ultimately beneficial as different IDE features (such as auto-formatting and syntax highlighting) are included, with the layout helping the Lawyer focus on a rapid drafting process without distractions from formatting tasks.

The core editor supports all essential text editing tasks, achieving the same functionality as a generic text editor. In addition, the editor also incorporates multiple analytical and supporting functionality which are displayed in Figure 4.1 (On the right hand side). These essential tasks and supporting functionality are outlined below in Table 4.1. Overall, the CC-IDE delivers a comprehensive contract drafting experience, supporting the Lawyer in this process.

Task	Description	
Write and Edit Contracts	The Lawyer can draft contracts within the text editor and view it immediately within the output.	✓
Contract File System	A file system has been implemented such that the Lawyer can save, load and delete completed (or uncompleted) contracts.	✓
Support for IDE Features	Utilising the formatting flexibility of HTML and CSS, IDE features (Implementation explained in Section 4.3.1) are integrated into the output.	✓
Support for Contract Analytics	Connected to the backend contract processing system, the Lawyer is provided with different analyses that refresh automatically as the contract is written.	✓
Display Condition IDs and Definitions	Through a breakdown of the contract structure, beneficial contract details can be displayed. (IDs of conditions & Substitution of definitions)	✓

Table 4.1: Outline of CC-IDE Text Editing Functionality

An interesting concept also explored and implemented within the IDE is Metadata. Its use in providing contextual information on grammars and contracts (Appendix C outlines the role of metadata within the CC-IDE) has widespread applications and benefits within an IDE, e.g. how contracts should only be viewed when a compatible CC-language is active, as incorrect parsing would otherwise occur. Within the CC-IDE, Metadata provides the ability to denote what language a contract is written in, and then subsequently allow the CC-IDE to restrict visible contracts based on the current applied language. Additional metadata should be considered in future CC-IDEs.

The CC-IDE is a substantial project, designed with a highly modular architecture to support

extensibility and interactivity. A variety of individual components form the CC-IDE, each contributing to a different section of the CC-IDE to create a comprehensive and cohesive development environment. Not desirable all components can be covered in this dissertation. An abstracted Unified Modelling Language (UML) Diagram outlining the structure of the application is featured in Figure 4.2, summarising the components of the IDE: As can be seen from the extensions of the visualisation class, the CC-IDE is designed to be added upon.

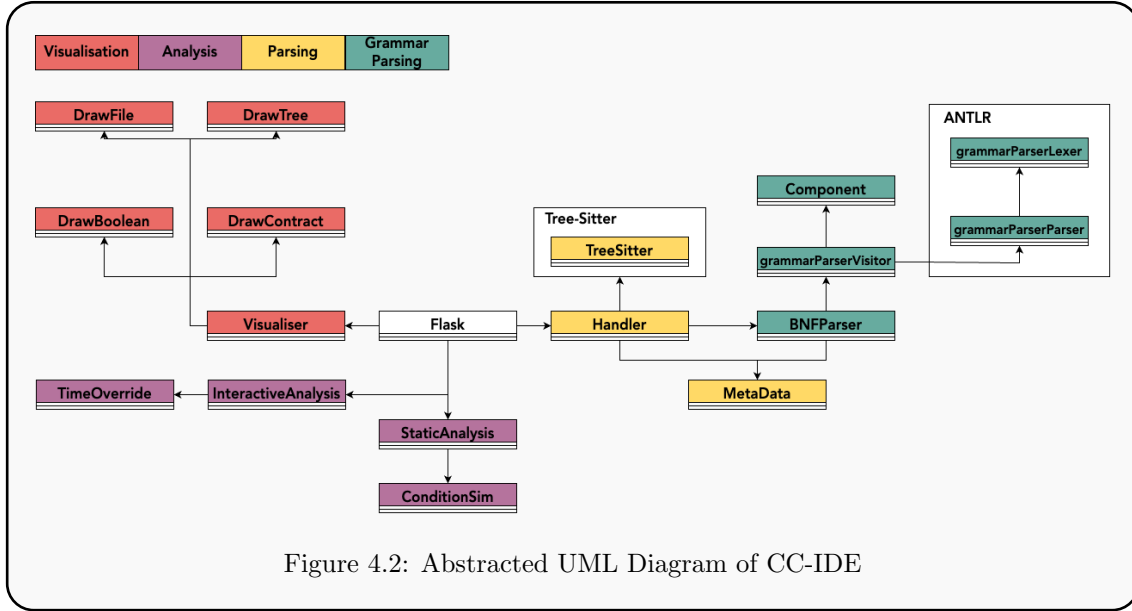


Figure 4.2: Abstracted UML Diagram of CC-IDE

4.2 Integrating Computable Contracting (CC) Languages

The CC-IDE is designed to support the integration of both new and existing CNL-based CC-languages under a single IDE framework, providing a universal set of features and tools. Within the CC-IDE, the goal is to transform the CC-languages (defined through a BNF as mentioned previously) into a fully usable CC-language with practical IDE support – i.e contract parsing and subsequent processing – providing the backend to the contract editing interface described in the previous section. However, the goal to integrate all CC-languages is not an easy task to achieve.

4.2.1 Pipeline Design

It is important for us to be able to parse contracts with different CC-languages, generating PTs for these languages, as this enables downstream computability and analysis of the contract. As contracts are often drafted in stages and follow an iterative process, it is therefore also necessary that the CC-IDE is resilient enough to handle incomplete or inconsistent contracts without failing. Otherwise fully completed contracts would be necessary to provide an output — interrupting what should be a fluid and flexible process. The result is the desire for a pipeline within the CC-IDE, able to handle the following:

1. Take the BNF representation of various CC-languages and generate a usable parser from each one.
2. Use a parsing technique that can handle imperfect, inconsistent contract drafts

4.2.2 Parsing with Tree-Sitter

Tree-Sitter, as seen in section 2.4.2, is a compelling solution, capable of generating parsers that handle inconsistent contracts. In particular, the benefit of Tree-Sitter parsers lies in its error recovery and incremental parsing — two uncommon features within traditional parsers.

These two features are crucial for iterative and interactive contract drafting. Error recovery prevents lapses in contract correctness from affecting the parsing process, providing a smooth drafting experience. Incremental parsing provides the system with a PT in a continuous and uninterrupted manner, reducing computational overhead and preventing the risk of perceptible latency. In the context of the CC-IDE, these features converge to deliver a drafting experience that is both resilient to human error and responsive to iterative refinement, thereby aligning with the iterative and precise nature of legal contract authoring.

However, the use of Tree-Sitter presents an additional challenge. Creating a Tree-Sitter grammar adds complexity for the Designer as the grammar must be written in the Tree-Sitter Domain Specific Language (DSL). Unlike the simplicity of a BNF, the Tree-Sitter DSL is unintuitive, requiring familiarity with JavaScript and additional syntax.

```
<def> ::= <simple-def>
      | <simple-def> AND <def>
      ;
```

Figure 4.3: BNF Form of a CoLa Definition

```
def: $ => choice(
    $.simple_def
    ,seq($.simple_def, 'AND', $.def)
),
```

Figure 4.4: Tree-Sitter Form of a CoLa Definition

A side-by-side comparison of the CoLa definition component presented in conventional BNF (Figure 4.3) and its equivalent Tree-Sitter DSL form (Figure 4.4). The juxtaposition highlights the noticeably different syntax, with the Tree-Sitter grammar necessitating a more verbose and explicit syntax inherent to its DSL [56]. Tree-Sitter’s DSL replaces BNF’s concise operators ($—$, $*$, $+$, $?$) with explicit function calls — such as *choice* for alternatives and *seq* for sequences, introducing syntactic overhead. The Designer may already face a steep learning curve when trying to develop CC-language syntax, and adding another programming layer only compounds the challenge. Automating the conversion from BNF to Tree-Sitter grammars could significantly reduce this burden and simplify the process for Designers.

4.2.3 BNF Parsing and Tree-Sitter Grammar Creation

Within the CC-IDE, a central innovation is the development of an automated BNF-to-Tree-Sitter DSL converter. This tool serves as a crucial bridge to enable Designers to leverage the power of Tree-Sitter parsing without needing to learn and master its intricate DSL.

The conversion pipeline comprises of two stages. First, a custom ANTLR4 parser—generated from a BNF-specific grammar (see Appendix D)—ingests the Designer’s BNF and produces a parse tree delineating rules, alternatives, and terminals. Second, a custom visitor class (in “tools/antlr/grammarParserVisitor.py” — and an example function in Appendix E) traverses that PT and emits the equivalent Tree-Sitter DSL, mapping each BNF construct to its corresponding function calls. The translation includes (inter alia):

- Wrapping alternative expansions (separated by “|” in BNF) within Tree-Sitter’s *choice()* function.
- Converting implicit sequences of BNF elements into explicit *seq()* function calls.

- Mapping BNF repetition (*, +) and optionality (?) symbols to their Tree-Sitter counterparts (repeat, repeat1, optional).
- Ensuring correct referencing of rules and definition of terminals according to the Tree-Sitter DSL structure (Symbols in the form \$.symbol).

The final output of this process is an automatically generated, fully compliant Tree-Sitter grammar, produced directly from the Designer’s BNF input ³. By encapsulating Tree-Sitter’s DSL intricacies within a BNF-converter, we eliminate manual translation overhead, reduce the potential for error, and lower the Designer’s barrier to CC-language development — allowing them to focus on the linguistic aspects of the CC-language in a familiar BNF. The resulting grammar can then be used to instantiate a Tree-Sitter parser, thereby enabling the error-recovery and incremental-parsing benefits described previously.

4.2.4 The Challenge of Writing Valid BNFs

A “Valid BNF” grammar must be not only syntactically correct but also compatible with the CC-IDE. Key compatibility issues include symbol repetition and, more critically, ambiguity. Contract parsing demands deterministic results, but legal grammars often contain structural ambiguities — different to semantic ambiguities. For example, in a financial contract grammar with the expression *base + stocks * dividend*, ambiguity arises when rules don’t specify operator precedence:

```
<expression> ::= <addition> | <multiplication> | <num>;
<addition> ::= <expression> + <expression>;
<multiplication> ::= <expression> * <expression>;
```

This creates two possible interpretations: *(base + stocks) * dividend* or *base + (stocks * dividend)*.

Tree-Sitter resolves such ambiguities through:

1. Precedence values: Using `prec(number, rule)` to prioritise rules with higher values
2. Associativity directives: Using `prec.left([number], rule)` or `prec.right([number], rule)` to control how expressions with equal-precedence operators are grouped

4.2.5 Resolving Ambiguities — Symbol Prefixes to write Complete Grammars

As Tree-Sitter grammar generation is abstracted to the Designer, we introduced a lightweight extension to the BNF editor to allow inline specification of precedence and associativity. Specifically, reserved prefix symbols —optionally followed by integer literals — are placed immediately before the definition operator (`::=`) in the BNF. During ANTLR-based BNF parsing, these prefixes are extracted and mapped to the corresponding Tree-Sitter annotations (`prec`, `prec.left`, `prec.right`), with any following digits interpreted as explicit precedence values. This augmentation preserves the standard BNF authoring experience while fully exposing Tree-Sitter’s conflict-resolution functionality — achieving it in a simplified manner.

The syntax is as follows:

```
<name> [Key] [Val] ::= expansion | expansion2;
```

³Additionally, comments are enabled within the BNF input via starting a line with: `# COMMENT` or `// COMMENT`, or doing a block (Multi-line) comment: `/* COMMENT */` This enables Designers to document their languages inline.

With the notion that the BNF rule:

```
<name> P6 ::= expansion | expansion2;
```

Evaluates to the Tree-Sitter rule:

```
name: $ => precedence(6, choice(
    expansion,
    expansion2
));
```

By introducing simple symbol prefix annotations in the BNF, the Designer can declaratively specify numeric precedence and associativity without restructuring their grammar production rules. This mechanism resolves conflicts at the grammar level, guaranteeing that each grammar-contract pairing yields a single, deterministic parse tree. Deterministic parsing is essential for consistent semantic analysis, since divergent parse trees can lead to inconsistent semantic reasoning about a contract. Moreover, this lightweight extension not only unlocks Tree-Sitter’s conflict-resolution features but also lays the groundwork for richer, grammar-driven semantic specifications (see Section 5.1).

Currently, the Tree-Sitter functions listed below have been incorporated into the CC-IDE BNF editor:

Functionality	Key	Example Syntax
Precedence	P	P6::=
Dynamic Precedence	PD	PD4::=
Left Precedence	L	L::=
Right Precedence	R	R6::=
Tokens	T	T::=
Immediate Tokens	TI	IT::=

4.2.6 The Complete Pipeline

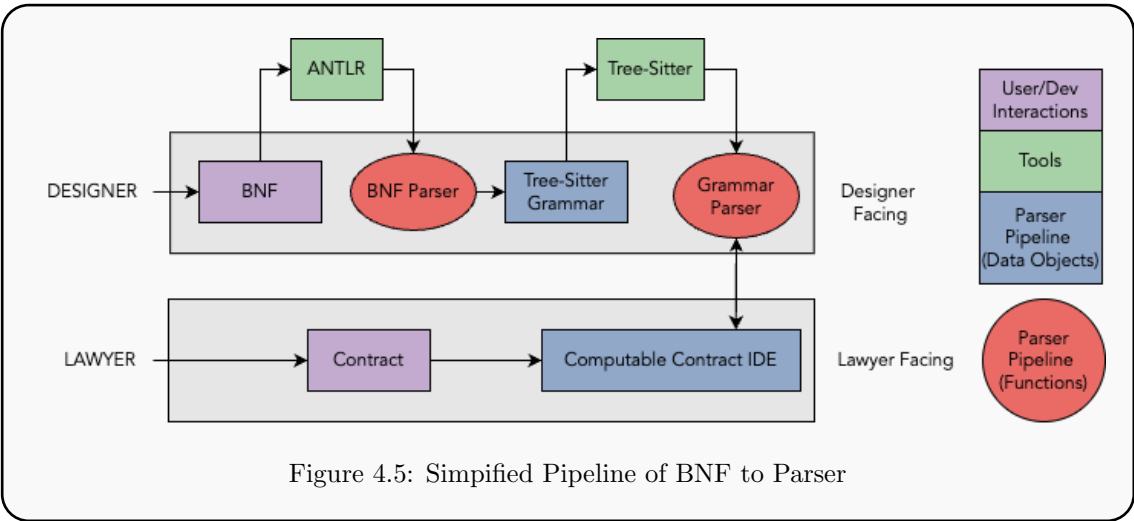


Figure 4.5: Simplified Pipeline of BNF to Parser

The previous sections highlight the components to a fundamental element of the CC-IDE’s architecture. A fully automated, designer-centric language-onboarding pipeline (illustrated in the Designer Facing segment of Figure 4.5). By reducing integration to a single action by the Designer — provision of a CC-language grammar in standard BNF — this pipeline abstracts and consolidates the traditional, multi-stage tooling workflow (Highlighted previously in Figure 3.6) into four phases:

BNF Parsing: The supplied BNF is ingested by a custom BNF-parser, decomposing the CC-language grammar into a PT representing production rules, alternative expansions, and terminal symbols. Additionally, Symbol Prefix mappings are extracted.

Grammar Conversion: A customer visitor class, implemented via ANTLR4’s visitor pattern, traverses the BNF-derived PT and emits an isomorphic grammar in the Tree-Sitter DSL.

Parsing-Engine Integration: The generated Tree-Sitter grammar is dynamically loaded as the CC-IDE’s core parsing engine. This component — equipped with error-recovery and incremental parsing capabilities — serves as the main contract processor, enabling downstream syntactic analysis.

Contract Parsing: Upon drafting a contract in the newly integrated CC-language, the text is parsed by this engine to produce a contract PT – available within the backend. This structural representation underpins all subsequent IDE functionalities, including syntax highlighting, error diagnostics, formatting routines, and semantic analyses (Highlighted in subsequent sections).

By encapsulating parser generation and integration within an automated pipeline, the CC-IDE ensures that Designer can remain focused on just the formal specification of their CC-languages. The system then guarantees consistent and immediate parsing support, providing interactive tooling across all integrated grammars.

Within the CC-IDE, the location for the Designer to input a BNF is under the Developer page (Figure 4.6), whereby a Designer would write or paste the BNF into the editor. Once the “Save & Apply” button is pressed, the process to transform the BNF and apply the CC-language begins. In the back-end, contracts will be parsed using this CC-language, generating a PT for analysis and integration with other CC-IDE components. After processing, results are reflected in the user interface, where Lawyers can write contracts with language-specific features.

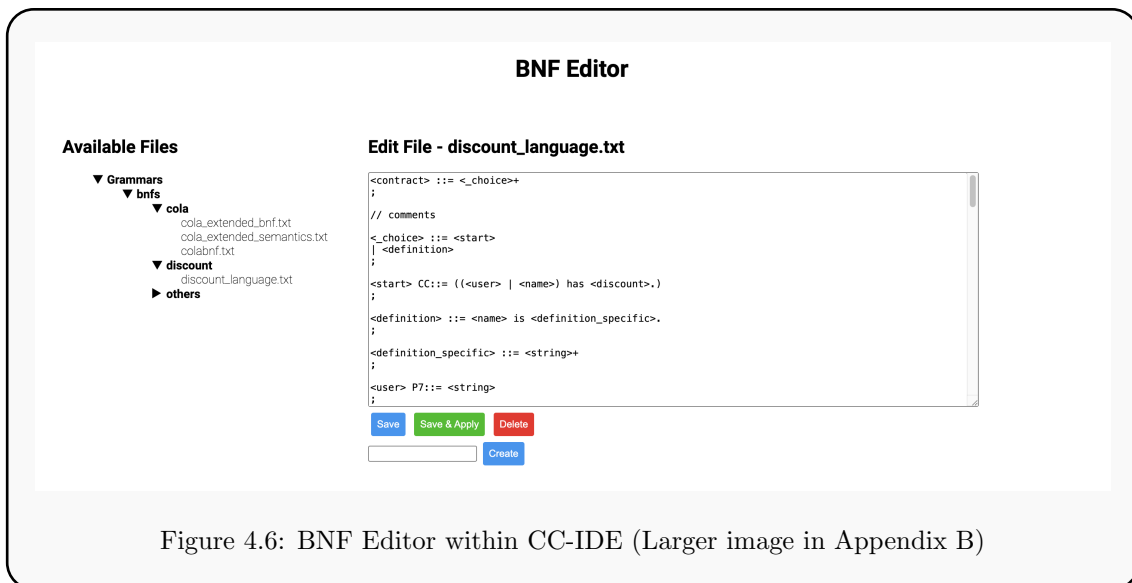


Figure 4.6: BNF Editor within CC-IDE (Larger image in Appendix B)

4.3 Universal set of IDE Features

The parsing process transforms contract text into a parse tree (PT) that reflects grammar rules, making the contract computationally comprehensible by organising it into syntactic elements (nodes representing rules/non-terminals and tokens/terminals). This structured representation enables rich IDE features for any integrated CC-language.

Instead of traditional visitor patterns requiring custom methods per node type, the CC-IDE implements a generic traversal strategy using Depth-First-Search with a dictionary-driven mechanism for uniform node processing. Feature support comes from a central configuration dictionary mapping each node type to its associated settings (highlighting styles, formatting rules). During PT traversal, the algorithm applies these configurations to each visited node. The following steps are performed:

1. **Lookup:** Use the node’s type as a key to retrieve its configuration entry from the dictionary.
2. **Dispatch:** Extract the prescribed action or style information (for instance, a particular colour or formatting function).
3. **Application:** Apply the retrieved metadata to the node —rendering syntax highlighting, enforcing formatting conventions, or invoking other IDE features — before proceeding with the traversal.

The results of these dictionary-driven actions (Explained more closely in Appendix F) become immediately visible within the parsed-output view on the main contract editing page of the CC-IDE. Importantly, the management of this configuration dictionary is conducted visually through the IDE’s graphical interface rather than programmatically via code. Utilising the dedicated formatting interface illustrated in Figure 4.7, Designers can intuitively associate desired visual styles — such as colours or text formatting options (discussed further in Section 4.3.1) — with individual node types defined by their grammar. This visual configuration mechanism removes the necessity for Designers to manually implement traversal logic or write feature-specific code, significantly simplifying the integration and ongoing maintenance of IDE visual features as grammars evolve.

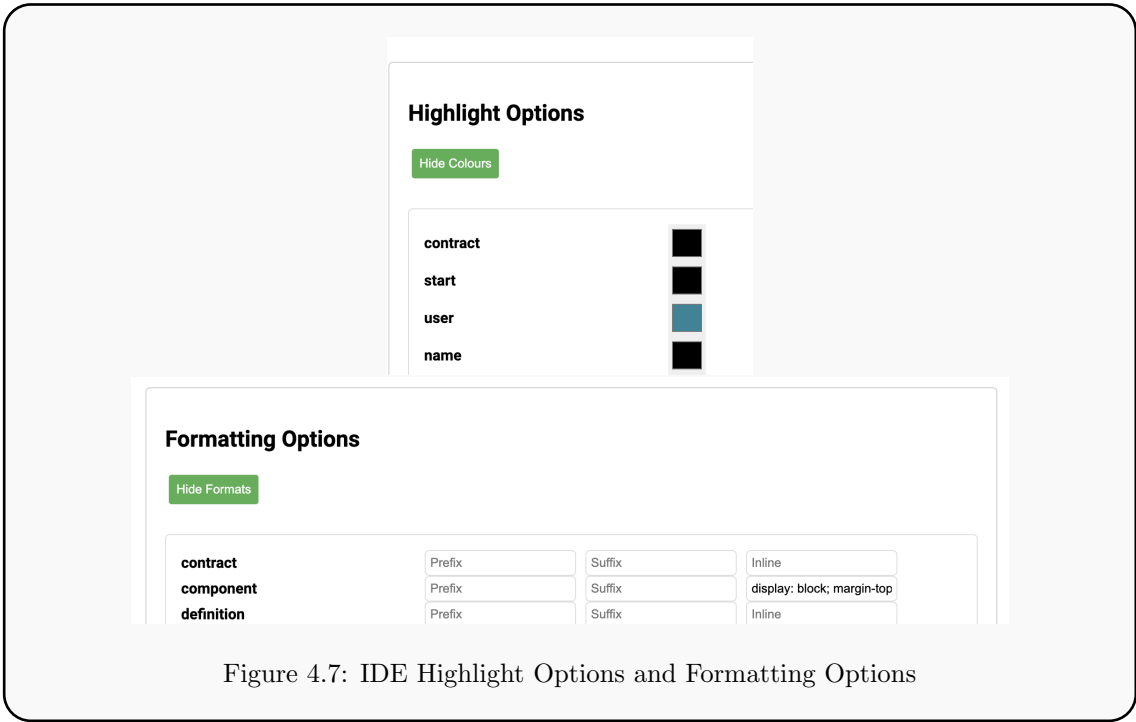


Figure 4.7: IDE Highlight Options and Formatting Options

Within the CC-IDE, this approach to implementing the IDE features diverges from traditional methods, prioritising simplicity and automation. The goal is to significantly reduce the Designer’s workload in integrating and using CC-languages, while still delivering an equivalent IDE support. The subsequent sections explain how different IDE functionalities are achieved.

4.3.1 Syntax Highlighting and Formatting

Within the CC-IDE, both syntax highlighting and formatting are achieved through leveraging the generated PT and the visual flexibility of HTML/CSS. The core technique involves translating the contract’s hierarchical PT structure into a corresponding structure of nested HTML `<span>` elements within the CC-IDE’s parsed output of the contract text. Each `<span>` element represents a specific node (a grammar rule) from the PT, and is assigned a CSS class name derived directly from its corresponding node type (the name of the grammar rule, e.g. *simple-statement*). User defined CSS rules then control colour (for syntax highlighting) and layout/decoration (for autoformatting), eliminating the need for a separate dedicated lexer or AST-based tooling mentioned in traditional contexts (Section 3.1.2).

Syntax Highlighting: Standard CSS rules are generated to target the specific class names. By assigning colour properties to these classes, different grammatical elements (both syntactic rules and terminals) defined in the language (keywords, clauses, literals, etc.) are rendered in distinct colours, providing immediate visual cues — a benefit to the Lawyer reading the parsed output. This PT-based method ensures highlighting can be applied universally to any grammar integrated into the CC-IDE. Figure 4.8 illustrates the streamlined process, enabling the Designer to select colours through an HTML colour picker.

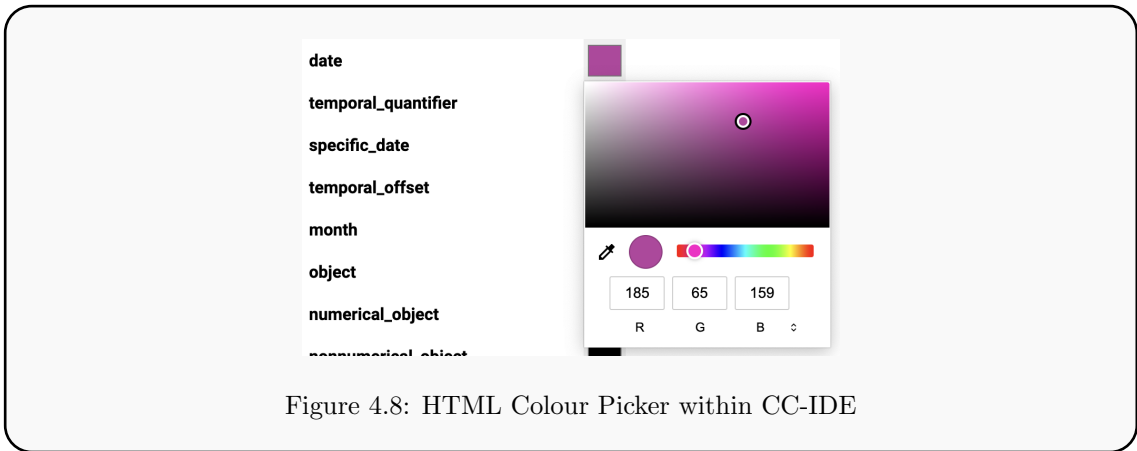


Figure 4.8: HTML Colour Picker within CC-IDE

Formatting: The same CSS mechanism is used for applying structural formatting. Properties like `display: block`, `margin`, `padding`, or `text-indent` (And additionally, whatever is feasible via CSS) can be assigned to classes associated with specific grammar rules to visually organise the contract’s presentation. Furthermore, CSS `::before` and `::after` pseudo-elements can be used to automatically insert decorative text (like prefixes or suffixes) around specific elements, enhancing readability. The interface of inputting of this formatting style is displayed in Figure 4.9.



Figure 4.9: Formatting Input within CC-IDE

Automated CSS Generation from Designer Configuration

A central feature of this system is the automation of CSS generation based on user preferences. Through the CC-IDE interface (as shown in Figure 4.7), the Designer can specify colours and formatting options for various node types within their grammar. This configuration data, represented as dictionaries (e.g., highlights and formatting), is processed by a backend Python function, `outputCSS` (shown in Appendix G). The function dynamically generates the required CSS rules according to the Designer’s input and writes them to a dedicated stylesheet (`static/parsing-formatting/parse.css`). Figure 4.10 visually represents the configuration process. Additionally, the system provides the flexibility to conceal terminals or non-terminals using the `display: none` property in CSS, as opposed to the `display: block` seen in the examples below. This showcases how CSS styling can be leveraged to create tailored contract-style outputs with precision and flexibility.

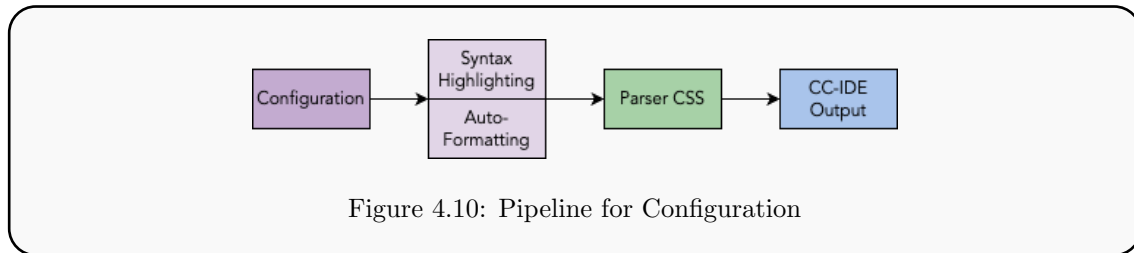


Figure 4.10: Pipeline for Configuration

This automated process then generates CSS rules (found in `static/parsing-formatting/parse.css`) to be used within the CC-IDE parsed output:

```
.simple_statement { color: #4071d4; font-weight: bold; display: block; margin: 2rem; }  
.date { color: #b9419f; font-weight: bold; }
```

The final result, where the contract text is displayed with syntax highlighting and formatting, is shown in Figure 4.11. This demonstrates how the CC-IDE achieves syntax highlighting and auto-formatting directly from the grammar and user configurations, without the need for complex, language-specific rendering code. The visual cues enhance the Lawyer’s experience, while the streamlined and automated formatting process contributes to greater efficiency in contract drafting.

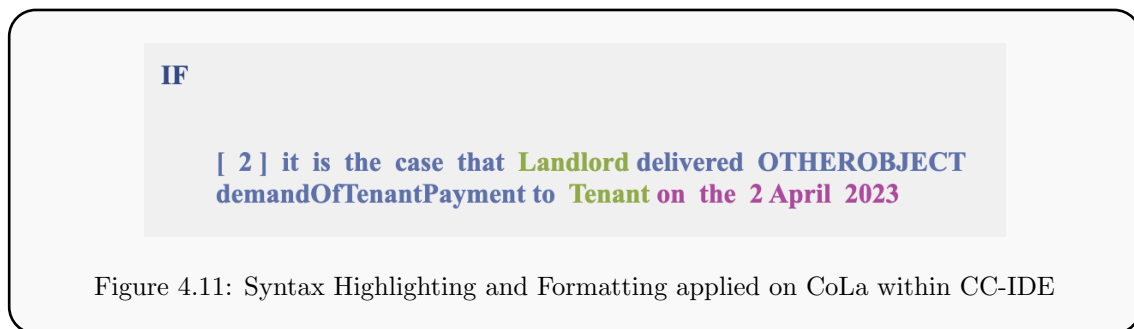


Figure 4.11: Syntax Highlighting and Formatting applied on CoLa within CC-IDE

Tooltips and Node Type

Additionally, hovering over any text segment in the CC-IDE Parsed Output displays a tooltip showing the name of the underlying grammar rule (e.g., “statement”, “date”). This utilises the HTML title attribute to provide immediate identification of the text’s component type. A possible future extension could be the customisation of tooltip text, done through a similar configuration system and used by a Designer to provide language-specific information – similar to a docstring in Python.

4.3.2 Error Detection

Building directly on the Parse Tree (PT) traversal and styling mechanisms used for syntax highlighting and formatting, the CC-IDE also implements real-time syntax error detection. This leverages a key capability of Tree-sitter: its ability to generate a PT even for partially invalid input, marking sections that violate the active CC-language with designated ERROR nodes.

During the standard PT traversal process (the same DFS used for styling), the CC-IDE identifies these ERROR nodes. Upon encountering an ERROR node, the system applies a specific visual style to alert the user directly within the editor where a syntax issue has arisen. This involves styling the content associated with the error — i.e. the direct children of the ERROR node — using a distinct red, bold text format (as illustrated in Figure 4.12) which overrides the Designer’s highlight options. This approach distinctly highlights the error, making it immediately apparent to the Lawyer.

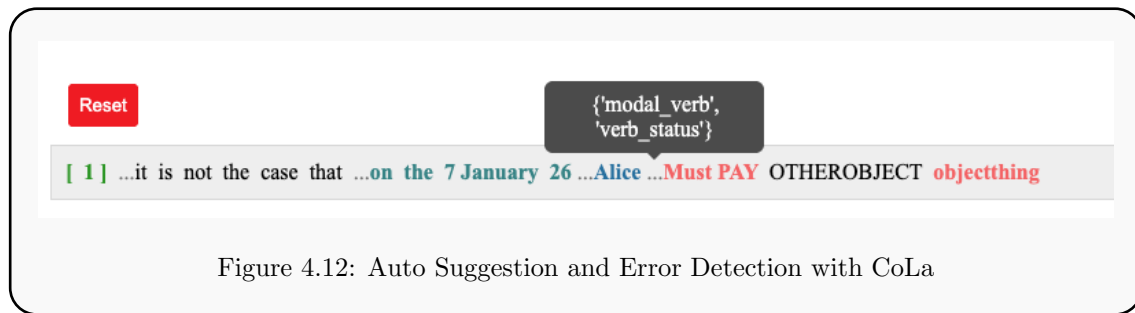


Figure 4.12: Auto Suggestion and Error Detection with CoLa

4.3.3 Auto Suggestion

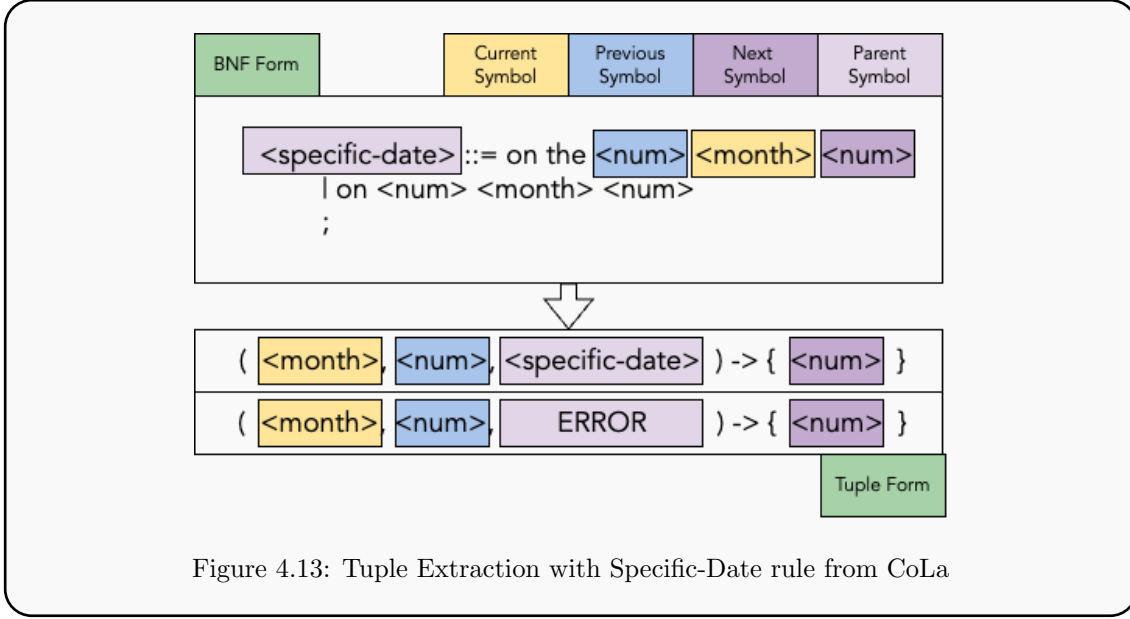
Context-aware auto-suggestions (code completion) are a hallmark of traditional IDEs but typically require a complex Language Server Protocol (LSP) implementation for each language. The CC-IDE simplifies this by generating suggestions through pre-processing the structure of the BNF grammar itself, eliminating the need for a full LSP implementation. An LSP implementation is often cumbersome, requiring the definition of intricate language-specific rules, managing language servers, and maintaining complex communication protocols between the client and server.

In contrast, the CC-IDE’s approach streamlines the process by analysing the BNF during the initial BNF parsing stage. In this phase, the CC-IDE analyses each grammar rule from the BNF-parser-generated PT to extract valid sequences of symbols (terminals and non-terminals). This process constructs a map of “context tuples” that define valid symbol relationships, including those in error contexts. Each tuple represents valid succeeding symbols that may follow a given symbol within a specific context. The mapping for each tuple is as follows:

```
(current symbol, previous sibling symbol, parent symbol) # Tuple  
-> {set of valid next sibling symbol} # Mapping
```

Figure 4.13 visualises how a BNF is analysed to extract such tuples — further details of the underlying code is laid out in Appendix H. Two versions of the tuple are extracted: the original and one in which an ERROR symbol substitutes the parent symbol. This design allows the system to offer auto-suggestions even when a valid pair of sibling nodes is parsed in an erroneous context. The “Next Symbol,” although depicted as a single option in the example, actually comprises all possible alternatives derived from the relevant grammar rules.

This mechanism generates a list of subsequent items to be suggested to the user. Within the CC-IDE parsed output, these suggestions are presented as a popup, which can be accessed by clicking the “...” button that appears in appropriate contexts, as shown in Figure 4.12.



While this method provides valuable, context-aware auto-suggestions that assist the Lawyer in contract drafting, it is essential to recognise that it simulates, rather than fully replicates, the functionality of a traditional Language Server Protocol (LSP). Although it simplifies the process by generating suggestions based on the grammatical structure, it may not account for all language-specific intricacies or edge cases that a fully realized LSP would address – such as when the end of a rule is reached and the subsequent symbol follows a different rule⁴. Overall, this approach offers a significant reduction in development effort compared to constructing and maintaining a comprehensive LSP for each CC-language, providing a simplified alternative.

4.3.4 Feature Benefit and Summary

The key benefit of this integrated IDE feature system lies in its automation and efficiency. By requiring only a valid BNF grammar and minimal configuration, it eliminates the need for external tooling and instantly equips any CC-language with a powerful suite of IDE features (syntax highlighting, auto-formatting, error detection, auto-suggestion). This directly enhances the contract drafting process for lawyers, providing a comprehensive toolkit within a single interface. Ultimately, the CC-IDE fosters a unified environment for the entire lifecycle of computable CC-languages — from development and testing to drafting — representing a significant advancement over fragmented traditional approaches.

⁴Due to how the code is constructed, changes to implement this is limited to the code of the tuple extraction component. The opening symbol of a subsequent *rule* should be added to the set of next symbols. Due to the systems modularity, no change is required to the other components of the auto-suggestion system.

Chapter 5

Applying Semantics

The preceding sections highlight the CC-IDE’s capabilities in enabling the development and use of CC-languages. In this chapter, we explore challenge of semantics within Computable Contracts, developing a solution to apply semantics onto any CC-language (both direct and translationally). We then explore why these semantics are beneficial and how they can be leveraged to benefit the contract drafting process.

5.1 Semantics

By parsing contracts against CC-languages, the CC-IDE ensures structural correctness while enabling syntax-driven features. However, the semantic challenge remains. In the previous section, we have defined a pipeline for integrating CC-languages, however, this focuses on the CNL portion without incorporating a DSL – the defining feature of the combined artifact model. Resultantly, the CC-language provided suffers from a lack of embedded legal contract semantics.

This highlights a fundamental challenge: Although the Designer can easily define grammar rules representing contractual components (e.g., *< clause >*, *< condition >*) and present them within the syntax, these syntactic labels are not automatically understood semantically by the system. The CC-IDE, based purely on the BNF, knows the structure named *< clause >* exists, but doesn’t inherently know that it represents a legal clause. This meaning is only explicit to the designer or inferred by a human reader.

Therefore, a crucial step towards incorporating semantics within the CC-IDE framework is to establish a mechanism for formally identifying key contractual components in an explicit and computationally extractable manner. The system needs to be able to recognise that a specific piece of text, parsed under a certain rule, corresponds to a defined contractual component (like an “clause”, “condition”, “definition”, etc.), whereby this named component has some actionable outcome within the CC-IDE. This moves the component beyond its role as just a named syntactic element, forming the foundation for any subsequent semantic analysis or computation.

Granted Attribute Grammars (AGs) could be considered as an approach to define semantics directly within the grammar, AGs are inconvenient and complex. Implementing AGs requires the Designer to possess programming skills to correctly code semantic rules and define attributes, and an understanding of AG evaluation is necessary. This can often lead to intricate dependencies between attributes and potentially challenging debugging processes, particularly as the grammar and its associated semantic rules increase in complexity.

5.1.1 Symbols

To bridge the gap and enable the application of computable legal semantics within CC-languages, the CC-IDE introduces the concept of Grammar-Agnostic Semantics (GAS). This approach reshapes the traditional idea of a DSL by focusing on a shareable set of domain-specific semantics rather than what is achieved with AGs, avoiding the coupling of semantics to a single fixed syntax (A similar notion named “Semantic Interoperability” is present in medicine, denoting standardised vocabularies and data formats [57]). The key focus of GAS ensuring semantic invariance across diverse grammars.

An example to highlight the need for flexibility can be displayed through conditionals. As shown in Table 5.1, various syntactic forms can express the same underlying conditional logic. While these different CC-languages define different syntactic structures, and have varying levels of expressivity, they have the same semantic importance.

Example Syntax	Semantics
If A, then B.	Conditional (if a then b)
If (A):	
B	
B must occur if A	

Table 5.1: Different Syntaxes, One Semantic Concept (Conditional)

GAS allows them to share the same underlying semantic interpretation within the CC-IDE. The fundamental principle of GAS is to define a standardised set of core legal semantic concepts relevant to computable contracts (highlighted in Section 2.1.1). This predefined semantic layer exists independently of any specific grammar. The CC-IDE then enables Designers to map these common semantic concepts onto their BNF grammars, applying it onto their syntactic structures. GAS provides crucial flexibility: different legal domains (e.g., finance, real estate, employment) can utilise grammars with tailored syntax and jargon, yet achieve semantic interoperability for analysis and tooling within the CC-IDE by mapping to the shared semantic framework.

The notion of a common semantic translation could be likened towards traditional compilation approaches seen in programming languages or even smart contracts (like Solidity/Vyper compiling to EVM bytecode). The similarity lies in how source code in these different languages typically translate into a the same central executable format (like bytecode or assembly) based on operational semantics. However, the difference is prominent in how GAS focuses instead on a semantic representation, and not a direct lower language translation. It maps nodes in the source language’s parse tree directly to predefined semantic tags, allowing the CC-IDE to understand the semantic role of different contract parts across diverse syntaxes for purposes like analysis, feature support, and potentially validation against semantic rules, directly within the IDE.

5.1.2 The Set of Legal Semantics

To practically realise the mapping of semantics within the CC-IDE, an extension of the Symbol Prefix mechanism described in Section 4.2.5 is employed. This implementation provides a direct mechanism for Designers to annotate the intended semantic role of their grammar rules within the BNF definition itself — attributing legal semantic components onto specific symbols, allowing contracts in that grammar to be analysed within the IDE. The IDE predefines a set of core legal semantic types (detailed in Table 5.2 and it’s usage explained in the Appendix J).

5.1.3 Mapping Semantics onto Grammars

The semantic mapping mechanism, triggered by the symbol prefixes is implemented during the automated conversion from the Designer’s BNF grammar into the Tree-Sitter grammar. This

process leverages Tree-Sitter’s *alias* function. According to the Tree-Sitter documentation, *alias* “causes the given rule to appear with an alternative name in the syntax tree” [56]. This is used to transform the Designer’s symbol into a symbol in the semantic set seen in Table 5.2.

Prefix	Symbol Transformation	Semantic Meaning
CC::=	clause	A major building block of a contract, and it divides the contract into different deontic components. Within this system, each clause can hold one set of conditions and two sets of statements, the Consequent (What happens if the condition is true) and the Alternative (What happens if the condition is false).
C::=	condition	A logical requirement that must be satisfied (true or false) for the associated clause to be triggered or made valid. Acts as a pre-requisite for the clause.
S::=	statement	A statement asserts a fact, obligation, or action. It is the outcome or consequence of a condition being met, representing what must occur or hold true. It represents the Consequent and Alternative.
D::=	definition	A definition assigns meaning to a term or phrase used within a contract. It reduces ambiguity by clarifying terminology and providing a precise context.
B::=	bracket	Used for grouping expressions and controlling the evaluation order of conditions. Helps ensure that the logical operators are evaluated correctly.
A::=	and_expression	Represents a logical AND. All conditions within the expression must be true for the expression to evaluate to true.
O::=	or_expression	Represents a logical OR. At least one condition within the expression has to be true for the expression to evaluate to true.

Table 5.2: Semantic Symbol Transformations

A short example illustrates this process. Consider this BNF snippet where CC::= tags rules as belonging to the **clause** semantic symbol:

```

<contract> ::= <payment_block> | <termination_block>;

<payment_block> CC::= Payment Clause { <clause_content>+ };
<termination_block> CC::= Terminal Clause { <clause_content>+ };

<clause_content> ::= if <condition> then <statement>;

```

When the CC-IDE’s BNF to Tree-Sitter converter encounters rules prefixed with CC::=, in this case, `< payment_block >` and `< termination_block >`, it modifies how these rules are represented and referenced within the output grammar. Specifically, it applies the *alias* function to map the original rule name to the target semantic symbol name, such that in a syntax tree, it would appear as this name. This conversion would take place as such:

```

contract: $ => choice(
  alias($.payment_block, $.clause)
  , alias($.termination_block, $.clause)
);

```

This aliasing provides the foundation for GAS within the IDE. Downstream tools (those operating on the PT of the contract) within the CC-IDE (such as analyses, visualisations, and simulations) now operate on PTs containing standardised symbols, known to developers regardless of the underlying grammar — provided that the semantic mapping is carried out correctly (Defining a

Non-Terminal with a Semantic Symbol is the same as using GAS, however, if used in conjunction with a prefix, it will be overwritten). They can query the PT for known types (such as a `clause`) and do not need to know about the specific syntactic variation. Furthermore, syntactic flexibility remains as rules with aliases still operate in the desired manner: only the name would change within the syntax tree, the underlying syntactic rule stays the same. Beneficially — as shown in the example, a Designer could also apply the semantic symbol onto syntactically distinct rules (such as for different types of clauses), whilst ensuring they are all recognised and processed as the common semantic symbol.

The GAS method provides a more succinct method of mapping semantics as opposed to AGs, streamlining the mapping task through the application of predefined prefixes to BNF rules. This process lowers the technical barrier and requires less specialised knowledge than writing full AG rules or defining a complete DSL. GAS effectively provides a mechanism to connect different legal grammars onto a set of common semantics, establishing a functional and computable CC-language.

5.2 Translation — When Semantics are Indirect

Not all contract concepts have a direct one-to-one mapping with a semantic symbol. Take time for example; whilst it can be embedded within the grammar and declared as a symbol, it is difficult for the CC-IDE to decide how to translate a time expression — even if in a common and easily understandable format. Table 5.3 highlights different common expressions of time. Each denotes the same time, demonstrating the variety in construction. Even simple representations of dates are different between countries, with the American time format MM/DD/YYYY varying from the international time format DD/MM/YYYY. When the day is below or equal to 12, the two time formats cannot be differentiated. When the scope is broadened to languages and dialects, this range of representation widens even further. As such, within the IDE, a framework for translation is provided.

Time Expression	Denoted Time
21-02-2025	Day: 21
21/02/25	Month: 02
21st of February 2025	Year: 2025

Table 5.3: Different Expressions of the same Date

5.2.1 Representing Time

To ensure consistency and enable computation with temporal aspects, the CC-IDE employs Python’s standard datetime object [58] as its internal representation for all temporal values. This provides precision, allowing for reliable date/time comparisons and calculations (crucial for analyses relating to time, such as time inconsistency analysis), and establishes a standardised format that can be expected by backend analysis tools and developers interacting with the system. While different CC-languages will display dates and times in various textual forms, all internal logic operates on these datetime objects. However, as a result, a system has to be provided to translate textual time to datetime objects.

5.2.2 Class Template

The CC-IDE addresses this by allowing the Designer to provide a custom time parsing and formatting class, overriding the methods of a predefined “Time” class - provided in Figure 5.1. Through this class, the Designer provides translations from time representations within their language to a datetime object (*evaluate_time_str* and *evaluate_time_tree*), along with the contrary of a datetime object translation back to a textual representation (*return_time_str*), facilitating bidirectional

```

from datetime import datetime

class TimeBase:
    def evaluate_time_str(self, time_str) -> datetime:
        return datetime.now()

    def evaluate_time_tree(self, time_tree) -> datetime:
        return datetime.now()

    def return_time_str(self, time_datetime) -> str:
        month = time_datetime.month
        day = time_datetime.day
        year = time_datetime.year
        return f"{day}-{month}-{year}"

class TimeOverride(TimeBase):
    def __init__(self):
        pass

```

Figure 5.1: Time Override Class

translation. Within the developer page of the IDE, the Designer can upload a Python file with the overridden class, containing the required translation methods. The interpreted nature of Python allows for the CC-IDE to be refreshed, integrating the class into the program. These methods can then be verified and subsequently used within the IDE to facilitate an internal representation of temporal aspects within a contract. An example implementation of the overridden class is provided in Appendix I.

5.2.3 Try-Except

Integrating user-provided code, such as the Python file containing the custom TimeOverride class, carries inherent risks. The uploaded file might be missing, syntactically incorrect, fail to define the required class, or have other issues preventing a successful import. A direct import attempt without error handling could lead to failure of the CC-IDE. To prevent such failures and ensure system robustness, the CC-IDE employs a try...except mechanism when loading the overridden time class. This approach is illustrated in Figure 5.2.

```

try:
    from time_override import TimeOverride
except ImportError:
    from time_base import TimeOverride

```

Figure 5.2: Try Except to Enable Time Override Class

If this import fails for any reason (e.g., file not found, syntax errors), the error is caught, and the system safely falls back to importing a default TimeOverride implementation from the time_base.py module, preventing failure of the CC-IDE system.

5.2.4 Extending GAS

Resultantly, we extend the GAS system with the prefix mappings in Table 5.4, including symbol transformations for temporal aspects and allowing them to be mapped onto grammars. It's usage

is highlighted in Appendix J.

Prefix	Symbol Transformation	Semantic Meaning
TT::=	time	The time object itself. It is the container for the time text, which is translated into a datetime object.
TTH::=	time_holder	Holds all aspects related to time, enabling evaluation beyond the time object itself. Used for declaring ranges and applying operators.
TTCA::=	time_and	Links two times together to create a continuous range where both times are included.
TTCO::=	time_or	Links two times together to create a range where either time can satisfy the condition.
TTB::=	time_before	Declares that an event should occur before a specified time.
TTA::=	time_after	Declares that an event should occur after a specified time.
TTO::=	time_on	Declares that an event should occur exactly on a specified time.

Table 5.4: Time-related Semantic Symbol Transformations

5.2.5 Time as an Example

Although this mechanism is used within the CC-IDE to handle translations for Time (And then used for time inconsistency analysis shown in Section 5.3.5), it is not limited to Time. For example, it could be used to support calculations on non-decimal currencies used in Madagascar and Mauritania. The example of Time shows, within this dissertation, how code from a Designer could be integrated and used within a CC-IDE, opening up the CC-IDE to numerous other integrations. One alternative approach with this mechanism could be an extension ecosystem, whereby Designers upload code for their respective languages with the goal of adding additional tools and features.

5.3 Analyses, Visualisations, and Simulations — Benefiting the Lawyer

The semantic mapping of symbols within the CC-IDE enables contract text to be broken down into structured, computable data. This data can then be utilised to generate a range of analyses, visualisations, and simulations, offering lawyers deeper insights into their contracts. This functionality is designed to be applicable across all integrated CC-languages, leveraging the universal set of semantic concepts provided by GAS, ensuring that such analyses and insights are available for any CC-language that adheres to the semantic framework.

5.3.1 Condition Simulation

One significant application of contract semantics is a simulation tool for testing the logical performance of a contract. This tool allows Lawyers to verify if combinations of conditions and their satisfaction lead to the intended outcomes (statements).

Mechanism: Within the CC-IDE, this is implemented as a boolean condition simulation. PT nodes mapped to the `condition` semantic symbol are treated as input flags (boolean variables). Nodes mapped to the `statement` semantic symbol are treated as results. The simulation engine analyses the logical structure connecting these conditions (utilising the mapped `and_expression` and `or_expression` nodes) and evaluates the outcome statements based on user-toggled true/false values assigned to the input condition flags.

Benefit: This enables Lawyers to interactively test contract logic, exploring different scenarios (including edge cases), and visually confirm (as shown in Figure 5.3) that the clauses perform as expected under various condition states, verifying the contract’s performance.

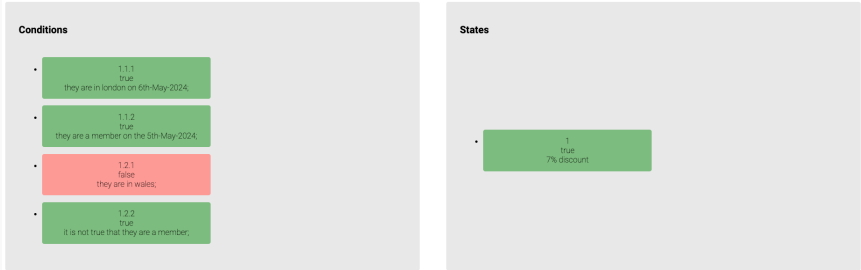


Figure 5.3: Condition Simulation within CC-IDE after Condition Toggling
(Full size Before and After in Appendix K)

5.3.2 Event Simulation

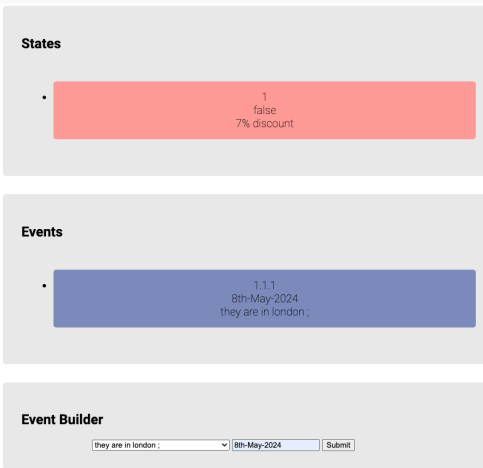


Figure 5.4: Event Simulation after Building New Event(Full size image in Appendix L)

To incorporate temporal aspects and extend condition simulation to account for events over time, the CC-IDE introduces a novel Dynamic Event Simulation Tool. This feature allows Lawyers to verify whether specific events occurring at defined times lead to the intended contractual outcomes.

Mechanism: Within the CC-IDE, event simulation operates similarly to condition simulation; however, instead of toggling boolean flags, the Lawyer is provided with an Event Builder interface (shown in Figure 5.4). Through this tool, users can select conditions and specify the times at which they are fulfilled. The system then evaluates the relevant outcome statements against the defined temporal conditions (Using the respective time translations provided in the TimeOverride class) and displays the resulting effects. The implementation details are provided in Appendix M.

Benefit: This tool enables Lawyers to interactively test contract logic by simulating different event sequences over time, providing visual confirmation (Figure 5.3) of whether clauses perform

as intended. By supporting dynamic exploration of temporal scenarios, the Event Simulation Tool introduces a new dimension to computable contracts, treating time as a computable contract factor.

5.3.3 Identity Generation

Given that the CC-IDE can semantically distinguish between clauses and conditions within a contract, it is also capable of systematically generating identifiers to enhance readability for the Lawyer. Conditions are automatically labelled with hierarchical identifiers. These identifiers are generated based on the position of each clause and the sequence of conditions within it. Deeper levels of nesting result in additional decimal points, providing a clear and scalable method for referencing specific conditions. By leveraging its understanding of the contract’s structure, the CC-IDE can automatically produce hierarchical identifiers for key components, significantly improving both readability and referencing.

Member has 7 % discount and 5 % discount if

(1.1.1) they are in london on 6th - May - 2024

and

(1.1.2) they are a member on the 5th - May - 2024

or

(1.2.1) they are in wales

and

(1.2.2) it is not true that they are a member

Figure 5.5: An example contract with automatically generated identities

Mechanism: Since the system can identify nodes mapped as “clause” and “condition”, it traverses the Parse Tree to systematically assigned hierarchical identifiers. These identifiers reflect the nesting structure, based on the sequence of clauses and the order and depth level of conditions within them (e.g., (1.1.1), (1.2.2)). Greater nesting depth corresponds to deeper levels of the identifier, maintaining a consistent and logical reference system.

Benefit: As illustrated in Figure 5.5, these automatically generated identifiers provide precise external references for documentation, aiding comprehension and navigation of contract structure for the Lawyer. ⁵

5.3.4 Definition Substitution

Contracts frequently introduce defined terms early in the text, requiring the reader to recall these definitions throughout the document — a task that can be challenging, particularly in lengthy contracts. The CC-IDE addresses this issue by leveraging the semantic mapping of ‘definition’ nodes to provide support.

⁵It is noted, however, that cross-referencing truth values would require explicit support within the BNF of the CC-language. Additionally, traditionally, the use of an identification system is for clauses rather than conditions. Although the implementation in this CC-IDE generates identifiers for conditions (Allowing it to be easily referenced linked to the conditions within the simulations mentioned previously), it could easily be changed to include clauses, however, a formal identification system would have to be designed.

Mechanism: The CC-IDE includes a feature that allows terms identified as definitions (via nodes mapped to the "definition" semantic category) to be interactively expanded or viewed. Lawyers can substitute definitions directly into the contract text using the "Display Definitions" button available on the main CC-IDE interface, as shown in Figure 4.1. This functionality is implemented using regular expressions, which replace target terms with their corresponding definitions.

Benefit: As displayed in Figure 5.6, this feature enhances readability and comprehension by providing immediate access to definitions at the point of use. It reduces the risk of misinterpreting defined terms and allows users to toggle definition expansion on or off according to their preferences, thereby improving flexibility and user control.⁶

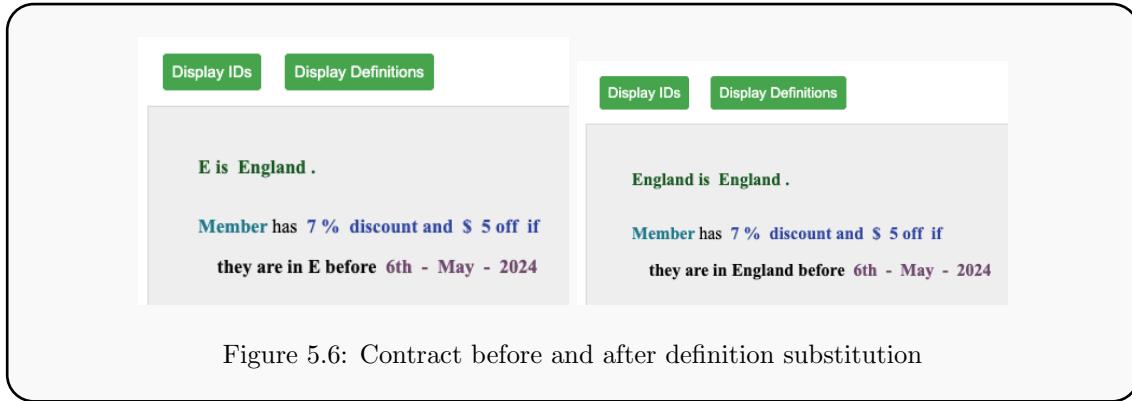


Figure 5.6: Contract before and after definition substitution

5.3.5 Time Inconsistency Detection

Through the translation of temporal expressions within contracts into computable time objects, the CC-IDE enables automated detection of temporal inconsistencies. This analysis is performed automatically and is found on the contract analysis section of the main editor interface.

Mechanism: The system uses semantic mappings to identify temporal constraints (nodes mapped to "After", "Before", "On" symbols). It then uses the Designer's custom TimeOverride class (discussed previously) to parse the associated text into standard datetime objects. With these computable time objects and their relationships (Before, After and On), the IDE can perform logical comparisons to detect inconsistencies, such as specifying an event must occur after Date X but also before Date Y, where Y is earlier than X. The system can handle combined ranges (like "Between" derived from "After" and "Before").

Benefit: This analysis automatically highlights potentially contradictory or impossible temporal constraints within the contract, alerting the Lawyer to logical temporal flaws that might be missed in a manual review. It flags the identity of the condition, allowing the Lawyer to efficiently locate and address the issue.

5.3.6 Abstracted Contract Visualisations

The semantic understanding embedded within the CC-IDE enables the generation of simplified, abstracted views of a contract, focusing on core contract logic rather than full contract text.

Mechanism: By identifying and extracting only the nodes mapped to key semantic categories (like "clause", "statement", "condition"), the IDE can construct visualisations that emphasise the contract's logical flow and structure. This involves tree diagrams, flowcharts, or structured summaries (An example is seen in Figure 5.7).

⁶However, within this implementation, it does not target individual names, substituting all names instead. A possible solution for individual names is extending the tooltips previously mentioned, altering it to display definitions.

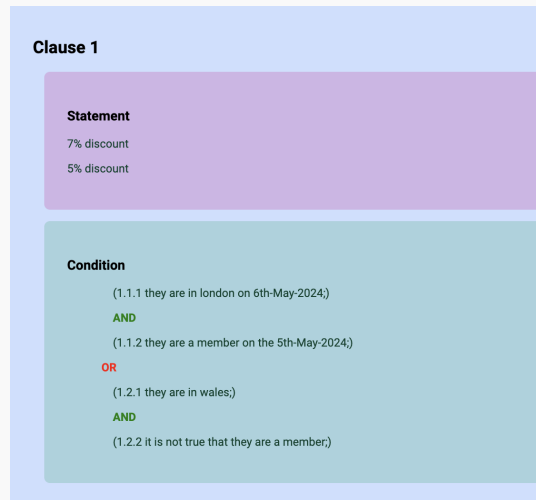


Figure 5.7: Example Abstracted Contract Visualisation

Benefit: These abstracted views help the Lawyer to quickly comprehend the essential components and logical relationships within a contract, abstracting away boilerplate or less critical text (“templated sections”). This aids understanding, review, and communication about a contract’s core components – in turn benefiting the Lawyer in drafting higher *quality* contracts.

5.4 CC-IDE Summary

Having developed the CC-IDE to support the integration of diverse CC-languages and extended it to incorporate computable legal semantics, we now provide a structured overview of the CC-IDE’s architecture and functionality. The CC-IDE’s core features are organised across the following pages:

Editor - Main Page

This serves as the primary workspace for lawyers drafting contracts. The layout features a text editor panel on the left, where contracts can be written and edited. Concurrently, the right panel displays a real-time parsed and styled representation of the contract text based on the current active CC-language. This output incorporates automated syntax highlighting and auto-formatting according to the configuration set by the language Designer. Furthermore, this output also benefits from auto-suggestion and error detection. Functionality for generating hierarchical condition identifiers and expanding defined terms is available, along with a framework for incorporating analyses, providing deeper insights into a contract’s structure and content. This framework is extensible, allowing for future offerings by other developers.

Visualisation - Visualisation Page

This page provides alternative representations of a parsed contract to facilitate understanding and analysis. It is designed for the addition of various visualisation methods such that contracts can be examined in a multitude of ways. This enables users to choose a view tailored to their needs,

making it easier to identify and extract key components, logical structures, or potential issues within the contract. Currently implemented visualisations include:

- Tree view - Displays the Concrete Syntax Tree (Parse Tree) generated by the Tree-sitter parser, presenting the direct syntactic structure.
- Hierarchical view - Presents the contract in a hierarchical structure, showing explicit relationships and rules.
- AST Hierarchical view - Offers a simplified and abstracted hierarchical view, focusing on Lawyer-chosen core contract components.
- Boolean view - Extracts and visualises the logical structure of the contract, removing template text.

Development - BNF Development Page

This page encapsulates the dedicated environment for language Designers to create, configure, and extend CC-languages used within the CC-IDE — leveraging the BNF-to-Tree-Sitter grammar conversion pipeline. The key tools involved are:

- BNF Editor: Enables the Designer to write, edit, and manage BNF grammar rules. It integrates the CC-IDE’s symbol prefix system to enable Designers to map their grammar rules onto the CC-IDE’s predefined semantic symbols — following the GAS system.
- Styling Configuration (Colour Picker and Formatter): Provides an interface for Designers to configure the visual presentation of their grammar. They can pick colour themes for syntax highlighting as well as configure auto-formatting rules to be used in the main editor output panel.
- Time Class Upload: Enables the Designer to upload a custom Python class (TimeOverride) containing temporal logic specific to their language, ensuring accurate temporal representation and analysis.

Simulation - Simulation Page

The simulation page focuses on testing the performance of a contract. It allows lawyers to interactively explore the links between different conditions and resultant outcomes (statements) as defined semantically within the contract. It allows a lawyer to stress test the contract logic and identify whether or not it achieves the right purpose, simulating the events and outcomes of a contract.

- Condition Simulator: This tool allows the Lawyer to systematically exploring different combinations of fulfilled and unfulfilled conditions, validating that the logical structure behaves as intended. This mode is particularly useful for identifying missing dependencies, unintended outcomes, or redundancies within contract logic.
- Event Simulator: Extending beyond static condition testing, the Event Simulator introduces temporal dynamics. Lawyers can define when specific conditions are fulfilled using the “Event Builder” interface, simulating the impact of events occurring over time. This allows for dynamic testing of the contract’s behaviour in response to time-based scenarios, enabling detection of issues that only arise through the interaction of temporal factors.

The entire CC-IDE codebase can be found through this GitHub link: <https://github.com/20tsange1/CC-IDE>

Chapter 6

Verification, Validation and Evaluation

This chapter focuses on the verification and validation of the CC-IDE system, assessing the correctness of the developed software and confirming that it meets its intended functional requirements. Following this, a critical evaluation of the system is presented, offering an analysis of the CC-IDE’s design and implementation. Finally, potential future developments are outlined, highlighting steps that could be taken to extend the CC-IDE beyond its current research prototype toward a commercially viable solution.

6.1 Verification

Verification of the CC-IDE will be structured across several layers of testing to ensure correctness, reliability, and functional coverage of the system. The testing plan is designed to validate not only technical accuracy, but also cross-language integration and semantic consistency, aligned with the goals of supporting diverse CC-languages through the Grammar-Agnostic Semantics (GAS) framework. The testing plan is organized into the following stages:

Unit Testing: Targeted unit tests will be conducted on core internal components, focusing particularly on the BNF-to-Tree-Sitter grammar converter and the GAS symbol prefix mapping mechanisms. These tests will verify that individual modules operate correctly and that grammar rules and semantic annotations are processed into valid Tree-Sitter grammars.

System Testing: A purpose-built CC-language, designed to incorporate all major semantic categories and temporal constructs, will be developed to exercise the full functionality of the CC-IDE. Using this language, we will perform end-to-end testing, verifying the correct operation of drafting, parsing, visualization, analysis, and simulation features within the platform.

Characterisation Testing: To evaluate cross-language compatibility, we will integrate two existing CC-languages, Fattal’s CoLa and Chong’s Extended CoLa grammar, into the CC-IDE. By parsing existing contracts written in these languages and validating the availability of IDE features and semantic mappings, we will confirm the system’s ability to meaningfully support diverse syntactic structures.

Through this multi-layered approach, the CC-IDE will be rigorously tested not only for technical and functional correctness, but also for its compatibility and robustness in supporting a diverse range of CC-languages. The testing plan is designed to systematically address the key concepts introduced in the preceding sections, ensuring comprehensive verification of the platform’s functionality and its ability to meet the demands of Lawyers and Designers.

6.1.1 Unit Testing

Unit Testing within the CC-IDE is concentrated on the BNF-to-Tree-Sitter grammar converter — a critical component in enabling language integration. The testing was divided as follows:

1. **Standard BNF Conversion:** Tests were designed to verify the translation of standard BNF syntax into syntactically valid and equivalent Tree-Sitter grammars. By covering the complete range of standard BNF constructs and validating their conversion to the Tree-Sitter DSL, these tests confirm that the converter operates correctly across diverse grammar specifications. The corresponding tests are located within the project’s codebase under `”/tests/test_bnf_parser_”` and are detailed in Appendix N.
2. **Symbol Prefix Mapping (GAS Implementation):** A separate suite of tests validates the mechanism for GAS symbol prefix mapping. These tests confirm that BNF rules annotated with semantic prefixes correctly generate the appropriate *alias* functions within the Tree-Sitter grammar, linking the Designer’s syntax to the CC-IDE’s predefined semantic symbols. The associated tests are found in `”/tests/test_symbol_prefix_mapping.py”` and documented in Appendix O.

The successful completion of these unit tests verifies that the core components of the language processing pipeline generate Tree-Sitter grammars accurately and reliably. A total of 38 test cases were developed, systematically covering a wide range of syntax patterns, alternatives, and edge cases within standard BNF (Along with semantic mappings). Furthermore, these tests form a reusable test suite, intended to be executed with each future software update to ensure continued correctness and stability of the conversion process.

6.1.2 System Testing

To perform full system testing of the CC-IDE, a purpose-built CC-language, “discount_language”, was developed. This language was intentionally designed to be syntactically simple while semantically comprehensive, ensuring that all major semantic categories and temporal constructs supported by the IDE — including GAS mappings — were exercised. Using this language, a suite of example contracts was authored within the drafting interface to test the editor’s behaviour, including real-time syntax checking, auto-formatting, and semantic validation.

Through the development and application of “discount_language” (full grammar available in Appendix P), comprehensive end-to-end testing of the CC-IDE was achieved. Explicit testing of the involved functions and features proves difficult, as it is unfeasible to write static test cases in every case (especially for interface related tests). As such, a form of empirical testing was utilised, observing whether a task could be carried out successfully. Categorised by the relevant CC-IDE pages, a total of 80 distinct scenarios and functions were tested (detailed in Appendix Q), covering the full workflow of contract creation and analysis. This testing process simulated the roles of both the Lawyer and the Designer, exercising all primary user functionalities within the platform.

While the checklist format is unlikely to capture every possible edge case due to the inherent complexity of the CC-IDE, it successfully covers the majority of intended interactions and system behaviours. Consequently, this testing approach provides strong empirical evidence of the platform’s overall stability and correctness. Every scenario tested was verified to complete successfully, allowing us to conclude that the CC-IDE operates correctly as a fully integrated system.

6.1.3 Characterisation Testing

Characterisation testing — testing of legacy code, in this case, languages and contracts — was conducted for the CC-IDE. This evaluation assessed the system’s robustness in syntactic integration,

semantic extension, and contract processing across diverse CC-languages (Not just purpose-built ones).

1. **Fattal’s CoLa Grammar[5]**: The original CoLa grammar, authored by Fattal, was successfully imported and processed through the CC-IDE pipeline with minor modification. Legacy contracts written in this version were loaded into the drafting interface, parsed, and visualised correctly, thereby validating the backward compatibility of the CC-IDE and integration with existing CC-languages.
2. **Chong’s Extended CoLa Grammar[28]**: Further verification was carried out using Chong’s enhanced version of the CoLa grammar. The CC-IDE accurately processed both the grammar and an additional suite of extended contracts, reinforcing the pipeline’s resilience to diverse syntaxes.

Both grammars, available within the CC-IDE as “cola.bnf” and “cola.extended.bnf”, and their respective contracts, are fully operational within the editor environment. Semantic mapping, facilitated by the GAS framework, then enabled the CoLa grammar to be used for analyses, visualisations, and simulation processes. Although minor modifications were necessary to fully realize semantic compatibility, the success of the mappings highlights the CC-IDE’s ability to adapt CC-languages to generate full computable contracts. (The semantically extended grammar is available as “cola.extended.semantics” and detailed in Appendix R)

6.2 Validation

Looking back on the requirements laid out in Section 3.3, we can use these to validate whether the goals and intended purpose of the CC-IDE has been achieved.

- *(RQ1) Be able to write, save, and edit computable contracts* ✓ Through the CC-IDE’s graphical user interface, comprehensive text editing capabilities are provided, including drafting, editing, saving, loading, and deleting contract files. These functionalities mirror those of standard text editors, but have been purposefully adapted for contract drafting workflows, as detailed in Section 4.1. This ensures that users can manage computable contracts throughout their full lifecycle within the platform.
- *(RQ2) Extensibility for different CC-languages* ✓ The integration pipeline designed for CC-languages, outlined in Section 4.2, has been demonstrated to operate successfully through the incorporation of two distinct languages: Fattal’s CoLa and the “Discount Language” developed for this project. Each language was fully integrated and semantically mapped within the CC-IDE, allowing parsing and analysis to be carried out in these languages, in turn providing empirical evidence of the system’s extensible and language-agnostic design.
- *(RQ3) Common IDE Features are applied interactively onto computable contracts* ✓ As explained in Section 4.3, essential IDE features such as syntax highlighting, error checking, and auto-formatting have been implemented within the CC-IDE using a flexible, grammar-driven approach. This enables these features to be universally applied to any integrated CC-language without requiring language-specific manual configurations, ensuring consistent user experience across diverse contracts and CC-languages.
- *(RQ4) Semantics for CC-languages* ✓ The implementation of Grammar Agnostic Semantics (GAS) and the Symbol Prefix mapping mechanism, detailed in Section ??, enables semantics to be dynamically attached to any CC-language grammar within the IDE. This foundational

feature allows analyses, simulations, and visualizations to be meaningfully executed regardless of the underlying language, substantially extending the CC-IDE’s functionality beyond simple syntax processing, enabling the computability of Computable Contracts.

- *(RQ5) Analyses, Visualisations, and Simulations for CC-languages* ✓ As discussed in Section 5.3, the CC-IDE provides a suite of advanced tools that extend the practical utility of computable contracts. Analytical tools detect structural and temporal inconsistencies, visualisation tools present complex contract logic in intuitive formats, and simulation tools enable users to predict and verify contract behaviour. These components collectively fulfil the requirement for actionable interaction with Computable Contracts – building a full computable contracting integrated development environment.

As a result, the CC-IDE fulfils all objectives and requirements defined in this dissertation, achieving a comprehensive set of functional and extensible capabilities for computable contract drafting, validation, and simulation. Furthermore, it offers a concrete and scalable foundation for the development of future CC-IDEs capable of supporting a wide range of CC-languages.

6.3 Critical Evaluation

The CC-IDE is designed as a proof-of concept, illustrating what an IDE for computable contracting could entail and the functionality that it may contain. As an exploration rather than a commercial product, it is not intended to be perfect or faultless. Consequently, several design choices, involving technological and conceptual factors can be critically evaluated and discussed. However, it must also be noted that as there are limited commercial solutions, it is difficult to compare the CC-IDE against any existing tools.

6.3.1 Grammar Restriction

The CC-IDE’s reliance on a BNF to define language syntax provides standardisation and compatibility with the tools within the CC-IDE. While suitable in the CC-IDE for defining the context-free grammars of CC-languages, implementing context-sensitive syntactic rules and more complex legal logic directly within the BNF is challenging and may require future extensions to the system. The current CC-IDE design prioritises standardisation and generalisation over maximising language design flexibility. While it results in greater syntactic variety, it may therefore limit logical difference between languages.

Exploring alternative grammatical frameworks could offer new avenues for more effectively encapsulating the CNL of CC-languages. One such approach is the use of Attribute Grammars (As previously hinted), which extend traditional context-free grammars by associating additional semantic information (attributes) with grammar rules. If Attribute Grammars were to be integrated with the existing GAS system, while retaining their ability to propagate and compute attributes, the result would be a significantly more expressive language framework. This would enable richer modelling of contextual relationships, dependencies, and semantic constraints within contracts. However, such an enhancement would also introduce greater complexity into the CC-language development process, requiring more sophisticated tooling for attribute definition and propagation.

6.3.2 Semantic Framework Constraints (GAS)

Furthermore, the notion of semantics within the CC-IDE are limited to the GAS system, which, while exploring a number of key contract components, contains only a subset of what a lawyer may

consider to be contract components. A fundamental discussion point arises regarding the GAS framework. Does this approach enable the development of semantically diverse CC-languages? or does it only offer syntactic diversity in a way that ultimately conforms to the same underlying conceptual model defined within GAS? There is an inherent conflict between enabling generalisation and broad language support, and defining a malleable semantic foundation to accurately model the nuances within the legal domain is no easy task.

Even when comparing the high-level perspective of contract components (as outlined in Section 2.1.1) and the extraction of lower-level elements (detailed in Table 5.2), inconsistencies in naming conventions and conceptual frameworks are already apparent. These divergences further highlight the absence of a standardised semantic model for contracts. It would be highly beneficial to develop a set of interoperable, universally accepted contract concepts capable of spanning both legal nuance and computational requirements – whilst maintaining malleability. While initiatives such as Stanford’s CodeX project [10] have hinted at the need for such a framework, no comprehensive formalisation currently exists.

6.3.3 Adoption within Legal Industry

Another point of discussion is the adoption of CC-IDEs within the legal industry. Although the CC-IDE is a tool designed to help lawyers in writing contracts, lawyers may prefer to stick to their established methods of contract drafting, favouring the flexibility of free-form natural language. Introducing a requirement to work within the structure of a legal grammar could be restricting, and represents a significant change to a lawyer’s workflow. The CC-IDE must offer compelling and clear advantages over traditional methods for the perceived benefits to outweigh the need to learn and adapt to a new system. In this case, it is difficult to argue that the CC-IDE provides a paradigm in contract drafting, although it is certainly a step towards a more efficient and productive solution.

Several factors influence this assessment, including the system’s ability to scale to larger and more complex grammars, its potential integration with ontologies for enhanced semantic and syntactic completeness, and its capacity to support the drafting of comprehensive, real-world contracts. The commercial viability of the CC-IDE would ultimately depend on its performance in large-scale, high-stakes environments, where usability, scalability, and semantic depth are critical. These challenges must be considered for a CC-IDE to transition from a research prototype to a practical commercial tool.

6.3.4 Test Coverage and User Testing

Achieving complete test coverage for a system like the CC-IDE is inherently challenging. The interactive and dynamic nature of contract drafting makes it difficult to develop purely static or automated tests that capture the full range of possible user behaviours. While empirical testing was conducted to verify core functionalities, this approach primarily focuses on functional correctness rather than overall usability and user experience. A more comprehensive evaluation strategy would involve leveraging established methods from Human-Computer Interaction (HCI) research, focusing on how real users engage with the CC-IDE. Specifically, techniques such as heuristic evaluation could be employed, using Nielsen’s 10 usability heuristics as a structured framework for identifying usability problems [59].

Additionally, it would be beneficial for future work to include the development of a full grammar, as this dissertation does not focus on language design and as a result, the discount language fails to capture all the nuances that a CC-language may require. Instead, it may be beneficial for a future project to focus on designing a CC-language specifically for formal contract drafting, and extending the GAS system to capture the legal and linguistic nuances of the language.

6.3.5 System Extensibility

It is important to acknowledge that the set of legal concepts currently defined within the CC-IDE, along with the analyses and visualisation tools presented, represent only a subset of a much larger potential. The current implementation demonstrates some of the key possibilities, but does not yet encompass the full breadth of possibility available towards computable contracting applications. However, it must also be noted that the CC-IDE was intentionally designed to be an extensible framework, serving as an architectural structure where new tools and features can be added. Similar to the plugin systems of modern IDEs, the core structure is designed to allow new semantic concepts, analytical tools, and integrated features to be added.

6.4 Future Work

6.4.1 Open Opportunity

The benefit of the CC-IDE and the system of Grammar Agnostic Semantics is that it is made to be extended, and to be explored in different ways. This is only a subset of what can be explored, and there exists many other avenues to explore. This dissertation focuses more on simulation and a use of the semantics in a analytical and visual way, however, there is a massive opportunity to use it for automation purposes. We have explored concepts that can be extended such as the overriding of time modules, which could be done to mathematical aspects or operational aspects. Furthermore, the GAS system that we have designed with the semantics may not necessarily be the most comprehensive system, covering only what a small subset of contract components. It may be that legal experts prefer other descriptive sets of components. There is no requirement to follow the exact structure and implementation as set out within this dissertation. Instead, it is hoped that this system provides a valuable starting point towards CC-IDEs, inspiring future projects exploring this idea of computable contracting and CC-language development.

6.4.2 Reusable Clause Templates

To support more efficient and standardised contract drafting, the CC-IDE could be extended to incorporate a database of reusable clause templates. Users drafting a contract could browse or search the template database, selecting pre-defined clauses to insert into their contracts. This would significantly streamline the drafting process, promote consistency across contracts, and reduce drafting errors by reusing legally vetted content. Such a system could also support dynamic updating: if legal requirements change or best practices evolve, updates to a clause template in the database could automatically propagate to new contracts — saving administrative overhead.

6.4.3 Security

Within the Metadata of the contracts, a form of security has been hinted — recording the (deterministic) hashes of contracts. This serves as a form of accountability and security, ensuring that contracts have not been altered outside of the CC-IDE. However, this approach was not fully explored and could not be covered within the scope of this dissertation. A deeper exploration of security would benefit the CC-IDE, preventing adversarial attacks and safeguarding possibly sensitive information. Notions such as encryption, access control, and authentication are examples of security that could be leveraged within a future CC-IDE.

6.4.4 Shared Access and Version Control

Writing contracts is an iterative process (especially during contract negotiation), taking multiple versions before achieving the final contract. For this, a version control system, like Github, could provide greater accountability and reflect changes to a contract in a more structured and organised way (as opposed to emails or paper drafts). In some cases, this may also require more than one party editing the contract. For this purpose, it would be beneficial to have a document sharing system, similar to Google Docs or Overleaf, allowing these parties to work on the same document concurrently, preventing misalignment and miscommunication.

6.4.5 Automation of Operational Aspects

As mentioned, this CC-IDE focuses more on analysis and visualisation as opposed to automation. It is often the case that contracts contain some form of operational aspect that relates to a real world action. Although not true in every case, some such actions can be completely automated, carried out without the a human. A future CC-IDE could focus on incorporating an automation system, possibly through translation of semantics into a lower level language to be executed — similar to smart contracts. As opposed to direct translation, it could leverage the GAS system for translation from any CC-language to a set of lower level languages. Another idea on parallel lines is cross-language translation, whereby a contract written in one CC-language can be computationally translated to another. This could allow for reusability of contracts across legal domains, while another key application could be the translation across languages.

6.4.6 Plugin System

Currently, the CC-IDE follows an implicit plugin system, requiring developers to extend the CC-IDE through the source files. Instead, an approach that could be explored is to leverage a similar concept to the overriding of time modules, allowing the Designer to develop and upload analyses and visualisations specific to their language (or even for general use). This could take on the form of a more explicit plugin system, similar to VSC and Vim, allowing the Designer to customise and personalise the IDE for their CC-language. This is beneficial as it would remove limitations from the development of the tools within the CC-IDE, allowing it to adapt with time and new CC-languages.

6.4.7 CC-languages for Automation

The use of Large Language Models (LLM) for contract drafting with the use of a CC-IDE (Or at least its different analyses and simulations) could be an interesting avenue to explore. In this case, an LLM drafting a contract could benefit significantly from a CC-language as it constrains the behaviour of the LLM to verifiably correct legal language with known structures — preventing atypical behaviour. The use of contract analytics (such as consistency) could then be used for a feedback loop, such that the LLM can constantly refine and rewrite the contract until it reaches a certain quality. This would enable an automation of the entire contract drafting process; such that a Lawyer no longer has to be directly involved.

Chapter 7

Conclusion

Within this dissertation, the concept of a novel Computable Contracting Integrated Development Environment (CC-IDE) has been thoroughly explored, presenting a successful realisation of a software tool designed to unify contract drafting and CC-language development within a single platform. Addressing the problem of tool generalisation and language integration with regards to CC-languages, the CC-IDE enables syntactic variation in CC-languages while maintaining a shared, extensible semantic foundation. This key innovation ensures compatibility with a universal set of drafting and analysis tools, significantly enhancing usability and scalability.

A set of five generic requirements for a CC-IDE were initially proposed in Section 3.3 based on a detailed analysis of existing IDEs and its applicability to Computable Contracts, focusing particularly on how drafting support and language development could be intertwined within a single environment. Grounded in these requirements, the development of a proof-of-concept CC-IDE system was then explored. the foundation of the CC-IDE is built in Section 4.1, setting out how a modular and extensible architecture can be established using a Python web-application technology stack, enabling an interactive and dynamic drafting environment.

In order to solve the challenge of language integration, the subsequent section, Section 4.2 focuses on the creation of a complete language-onboarding pipeline, reducing it to the declaration of a CC-language's BNF grammar. This is achieved through a Symbol Prefix system for the creation of complete CNLs and an automated Tree-Sitter parser generation for producing the parsing engine that can effectively handle partial contract parsing. Furthermore, common IDE features — such as syntax highlighting, auto-formatting, and auto-suggestion — which often require significant manual overhead to configure and develop, were generalised for use across any CC-language through dictionary-driven styling and dynamic configuration interfaces. Pre-processing of grammars then enabled context-aware drafting support. This simplification explored in Section 4.3 provides Lawyers with a comprehensive toolkit for contract drafting.

Another major contribution of the CC-IDE is the development of Grammar Agnostic Semantics (GAS), devised in Section 5.1. It focuses on a flexible semantic framework, providing a set of universal contract semantics that can be mapped onto any CC-language through an extension of the Symbol Prefix system. Moreover, it also supports indirect translation of semantics such as temporal aspects, achieved through uploading an overridden Python class. In combination, this semantic mapping facilitates a range of advanced contract analysis tools within the CC-IDE, including condition and event simulation, abstracted logic views, and a range of static and dynamic analyses. By providing semantics applicable to, and more importantly, extractable from any CC-language, the computability of contracts is significantly enhanced, enabling consistent analysis, simulation, and interpretation of contracts across a range of legal subdomains.

Ultimately, this dissertation demonstrates that a CC-IDE is both feasible and practical. It presents the CC-IDE as a comprehensive drafting and development environment that supports Lawyers and

Designers alike. By providing standardised tooling for any compliant CC-language with minimal configuration, the CC-IDE lowers the barrier to adopting and using Computable Contracts. Furthermore, it serves as a foundation for future research and development in computable contracting tools, offering a blueprint for future development software that can bridge the gap between traditional drafting practices and computational analysis. It is hoped that the concepts introduced will inspire further advancements in the field, and contribute meaningfully to the evolution of Computable Contracting technologies.

Bibliography

- [1] Maria Battle. *Contract Law: Elements of a Binding Contract*. Commercial Law Firm — London, UK — Corporate Solicitors — EM Law, Sept. 2022. URL: <https://emlaw.co.uk/contract-law-elements-of-a-binding-contract/>.
- [2] John Cummins and Christopher D Clack. “Transforming Commercial Contracts through Computable Contracting”. In: *Journal of Strategic Contracting and Negotiation* 6 (Feb. 2022), p. 205556362110725. DOI: 10.1177/20555636211072560.
- [3] Lexon Labs LCC. *Lexon*. lexon.org, 2017. URL: <https://lexon.org/> (visited on 04/07/2025).
- [4] Robert Kowalski. *Logical English*. Nov. 2020. URL: <https://www.doc.ic.ac.uk/~rak/papers/LPOP.pdf>.
- [5] Simon Fattal. “CoLa: A Controlled Natural Language for Computable Contracting”. MA thesis. May 2021.
- [6] Michael Jeffery. *What Would an Integrated Development Environment for Law look like?* MIT Computational Law Report, Apr. 2020. URL: <https://law.mit.edu/pub/whatwouldanintegrateddevelopmentrelease/2>.
- [7] Matthew Roach. “Toward a new language of legal drafting”. In: *CoRR* abs/1507.05081 (2015). URL: <http://arxiv.org/abs/1507.05081>.
- [8] Microsoft. *Visual Studio Code*. Visualstudio.com. URL: <https://code.visualstudio.com/>.
- [9] Harry Surden. “Computable Contracts”. In: *University of California Davis Law Review* 46 (2012), p. 629.
- [10] Stanford Law School. *Stanford Computable Contracts Initiative — Stanford Law School*. Stanford Law School, 2019. URL: <https://law.stanford.edu/projects/stanford-computable-contracts-initiative/> (visited on 04/11/2025).
- [11] Konstantinos Christidis and Michael Devetsikiotis. “Blockchains and Smart Contracts for the Internet of Things”. In: *IEEE Access* 4 (2016), pp. 2292–2303. DOI: 10.1109/access.2016.2566339.
- [12] Stephen Newman. “THE INFLUENCE OF PLAIN LANGUAGE AND STRUCTURE ON THE READABILITY OF CONTRACTS”. In: *Obiter* 31 (Sept. 2021). DOI: 10.17159/obiter.v31i3.12339. (Visited on 04/15/2023).
- [13] Christopher D Clack, Vikram Bakshi, and Lee Braine. “Smart Contract Templates: foundations, design landscape and research directions.” In: (Aug. 2016).
- [14] *Force majeure*. Practical Law. URL: [https://uk.practicallaw.thomsonreuters.com/3-107-5776?transitionType=Default&contextData=\(sc.Default\)](https://uk.practicallaw.thomsonreuters.com/3-107-5776?transitionType=Default&contextData=(sc.Default)).
- [15] Kenneth Adams. *Know Your Enemy Sources of Uncertain Meaning in Contracts*. 2016. URL: <https://www.michbar.org/file/barjournal/article/documents/pdf4article2969.pdf> (visited on 04/08/2025).
- [16] Christopher D Clack. “Languages for Smart and Computable Contracts”. In: *Oxford University Press eBooks* (Apr. 2022), pp. 269–304. DOI: 10.1093/oso/9780192858467.003.0013. (Visited on 04/07/2025).
- [17] Alain Colmerauer. “An introduction to Prolog III”. In: *Communications of the ACM* 33 (July 1990), pp. 69–90. DOI: 10.1145/79204.79210. URL: <http://alain.colmerauer.free.fr/alcol/ArchivesPublications/Prolog3/acmprolog3e.pdf> (visited on 03/29/2019).

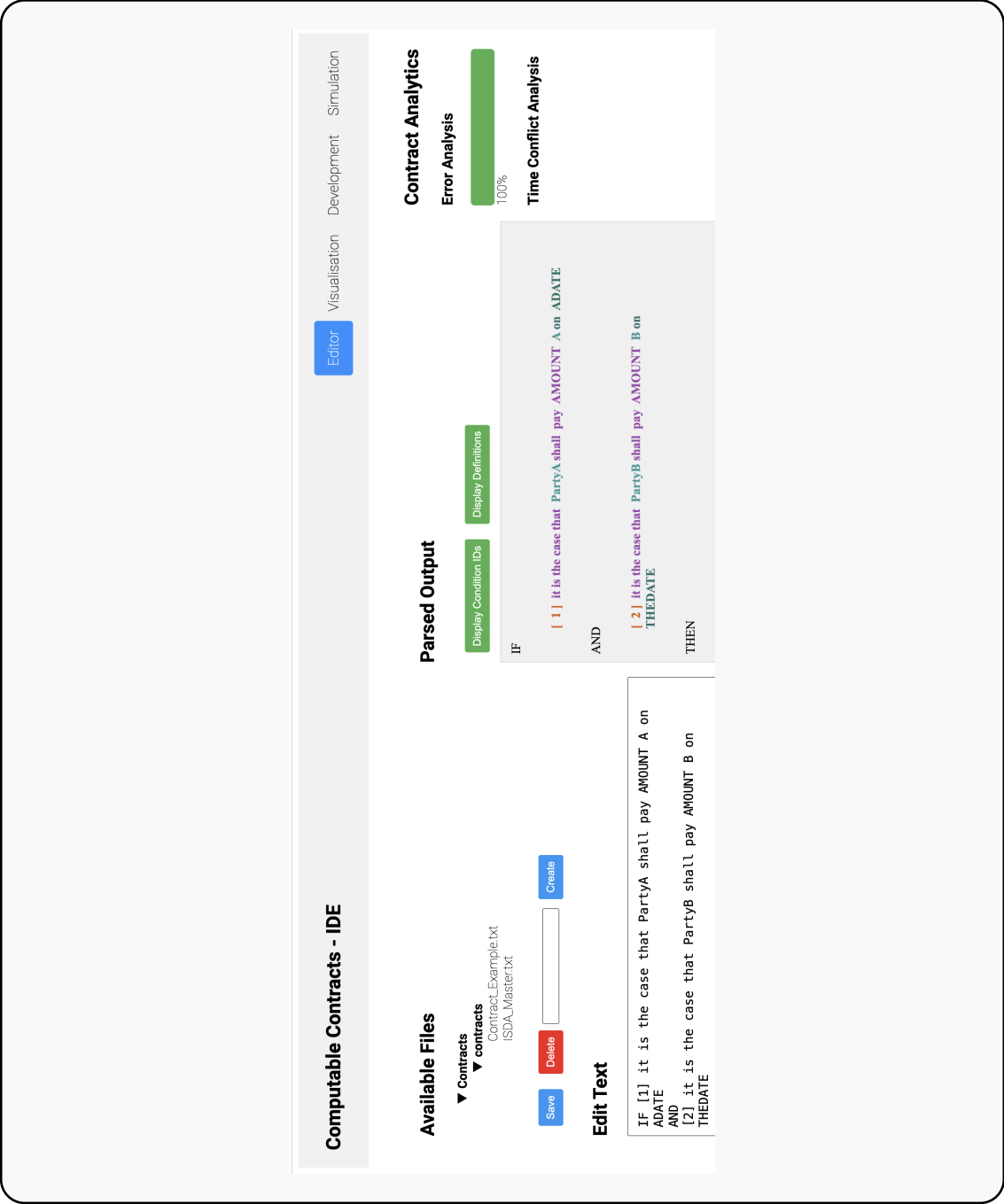
- [18] *Catala*. Catala-lang.org, 2025. URL: <https://catala-lang.org/> (visited on 04/07/2025).
- [19] *Deon Digital CSL Platform Documentation — Deon Digital CSL Documentation 5.0.0 documentation*. Deondigital.com, 2025. URL: <https://docs.deondigital.com/latest/> (visited on 04/11/2025).
- [20] SMU CCLAW. *L4 Documentation — L4 documentation*. Readthedocs.io, 2023. URL: <https://l4-documentation.readthedocs.io/en/23.12.04/> (visited on 04/07/2025).
- [21] Tobias Kuhn. “A Survey and Classification of Controlled Natural Languages”. In: *Computational Linguistics* 40 (Mar. 2014), pp. 121–170. DOI: 10.1162/coli_a_00168.
- [22] *Controlled natural language*. Wikipedia, Mar. 2024. URL: https://en.wikipedia.org/wiki/Controlled_natural_language.
- [23] John W Backus. “The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference.” In: *IFIP Congress* (Jan. 1959), pp. 125–131. (Visited on 04/07/2025).
- [24] Peter Naur. *A COURSE OF ALGOL 60 PROGRAMMING*. 1961. URL: https://archive.computerhistory.org/resources/text/algol/ACM_Algol_bulletin/1064048/frontmatter.pdf (visited on 04/07/2025).
- [25] Ian Grigg. *The Ricardian Contract*. iang.org, Aug. 2004. URL: https://iang.org/papers/ricardian_contract.html.
- [26] Jurij Lampic. *Ricardian Contracts: A Smarter Way to Do Smart Contracts?* Medium, Feb. 2022. URL: <https://medium.com/%40jurij.lampic/ricardian-contracts-a-smarter-way-to-do-smart-contracts-b6743752bb8b> (visited on 04/08/2025).
- [27] Oliver Goodenough and Susan Salkind. *Computable Contracts and Insurance: An Introduction*. MIT Computational Law Report, Aug. 2022. URL: <https://law.mit.edu/pub/computablecontractsandinsuranceanintroduction/release/1> (visited on 04/07/2025).
- [28] Yoong Xin Chong. “An Extension to CoLa: Consistency, Completeness and Complexity Analysis”. MA thesis. Apr. 2024.
- [29] Pablo Lamela and Simon Thompson. “Marlowe: financial contracts on blockchain”. In: (Nov. 2018).
- [30] Norbert E Fuchs and Rolf Schwitter. “Attempto Controlled English (ACE)”. In: *arXiv (Cornell University)* (Jan. 1996). DOI: 10.48550/arxiv.cmp-lg/9603003. (Visited on 04/23/2025).
- [31] Gordon J Pace and Michael Rosner. “A Controlled Language for the Specification of Contracts”. In: *Lecture notes in computer science* (Jan. 2010), pp. 226–245. DOI: 10.1007/978-3-642-14418-9_14. (Visited on 04/23/2025).
- [32] Kartik. *Parsing — Set 1 (Introduction, Ambiguity and Parsers)*. GeeksforGeeks, Feb. 2017. URL: <https://www.geeksforgeeks.org/introduction-of-parsing-ambiguity-and-parsers-set-1/>.
- [33] T. J. Parr and R. W. Quong. “ANTLR: A predicated-LL(k) parser generator”. In: *Software: Practice and Experience* 25 (July 1995), pp. 789–810. DOI: 10.1002/spe.4380250705. (Visited on 01/17/2023).
- [34] Max Brunsfeld. *Introduction - Tree-sitter*. Github.io, 2018. URL: <https://tree-sitter.github.io/tree-sitter/index.html> (visited on 04/07/2025).
- [35] Codecademy Team. *What Is an IDE?* Codecademy, 2019. URL: <https://www.codecademy.com/article/what-is-an-ide>.
- [36] *Integrated development environments - Translators and facilities of languages - OCR - GCSE Computer Science Revision - OCR*. BBC Bitesize. URL: <https://www.bbc.co.uk/bitesize/guides/z6x26yc/revision/6>.
- [37] JetBrains. *IntelliJ IDEA*. JetBrains, 2001. URL: <https://www.jetbrains.com/idea/>.
- [38] JetBrains. *PyCharm*. JetBrains, 2010. URL: <https://www.jetbrains.com/pycharm/>.
- [39] Vim. *welcome home : vim online*. www.vim.org. URL: <https://www.vim.org/>.

- [40] Melanie. *IntelliJ IDEA: Why is it the best IDE for you?* Data Science Courses — DataScientest, Mar. 2024. URL: <https://datascientest.com/en/intellij-idea-why-is-it-the-best-ide-for-you>.
- [41] Alexander Gillis. *What Is Refactoring (Code Refactoring)?* SearchAppArchitecture, Sept. 2021. URL: <https://www.techtarget.com/searchapparchitecture/definition/refactoring>.
- [42] Martin Fowler. *bliki: Post-IntelliJ*. martinowler.com, 2025. URL: <https://martinfowler.com/bliki/PostIntelliJ.html> (visited on 04/11/2025).
- [43] David M. Lane et al. “Hidden Costs of Graphical User Interfaces: Failure to Make the Transition from Menus and Icon Toolbars to Keyboard Shortcuts”. In: *International Journal of Human-Computer Interaction* 18 (May 2005), pp. 133–144. DOI: 10.1207/s15327590ijhc1802_1.
- [44] Muhammad Waqas Riaz. *The Benefits of Using MS Word for Academic Writing and Research Projects*. www.linkedin.com, Feb. 2023. URL: <https://www.linkedin.com/pulse/benefits-using-ms-word-academic-writing-research-muhammad-waqas/>.
- [45] Martin Fowler. *bliki: Language Workbench*. martinowler.com, Sept. 2008. URL: <https://martinfowler.com/bliki/LanguageWorkbench.html> (visited on 04/09/2025).
- [46] *Racket*. racket-lang.org. URL: <https://racket-lang.org/>.
- [47] *MPS: The Domain-Specific Language Creator by JetBrains*. JetBrains. URL: <https://www.jetbrains.com/mps/>.
- [48] Amazon Web Services. *What is an IDE? IDE Explained - AWS*. Amazon Web Services, Inc. URL: <https://aws.amazon.com/what-is/ide/>.
- [49] Marin Levrel. *Advantages of IDE: How Does It Enable Faster Development — Qodo*. Qodo, Mar. 2024. URL: <https://www.qodo.ai/blog/advantages-of-ide-how-does-it-enable-faster-development/>.
- [50] Visual Studio Code. *Syntax Highlight Guide*. code.visualstudio.com. URL: <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide>.
- [51] Ankit87. *Error detection and Recovery in Compiler*. GeeksforGeeks, Apr. 2018. URL: <https://www.geeksforgeeks.org/error-detection-recovery-compiler/>.
- [52] Sorawee Porncharoenwase. *UW PLSE — The Science of Code Formatting (Part I)*. Uwplse.org, Mar. 2024. URL: <https://uwplse.org/2024/03/04/code-formatting.html> (visited on 04/09/2025).
- [53] Advait Sarkar. “The impact of syntax colouring on program comprehension.” In: *PPIG* (Jan. 2015), p. 8.
- [54] Justin C. W. Debusse and Meredith Lawley. “Benefits and drawbacks of computer-based assessment and feedback systems: Student and educator perspectives”. In: *British Journal of Educational Technology* 47 (Dec. 2014), pp. 294–301. DOI: 10.1111/bjet.12232.
- [55] Flask. *Welcome to Flask — Flask Documentation (3.0.x)*. Palletsprojects.com, 2010. URL: <https://flask.palletsprojects.com/en/stable/>.
- [56] *The Grammar DSL - Tree-sitter*. Github.io, 2025. URL: <https://tree-sitter.github.io/tree-sitter/creating-parsers/2-the-grammar-dsl.html> (visited on 04/13/2025).
- [57] Pan American Health Organization. *Introduction to Semantic Interoperability*. 2021. URL: https://iris.paho.org/bitstream/handle/10665.2/55417/PAHOEIHIS21023_eng.pdf.
- [58] Python Software Foundation. *Datetime — Basic Date and Time Types — Python 3.7.2 Documentation*. Python.org. URL: <https://docs.python.org/3/library/datetime.html>.
- [59] Jakob Nielsen. *10 Heuristics for User Interface Design*. Nielsen Norman Group, Apr. 1994. URL: <https://www.nngroup.com/articles/ten-usability-heuristics/>.

Appendices

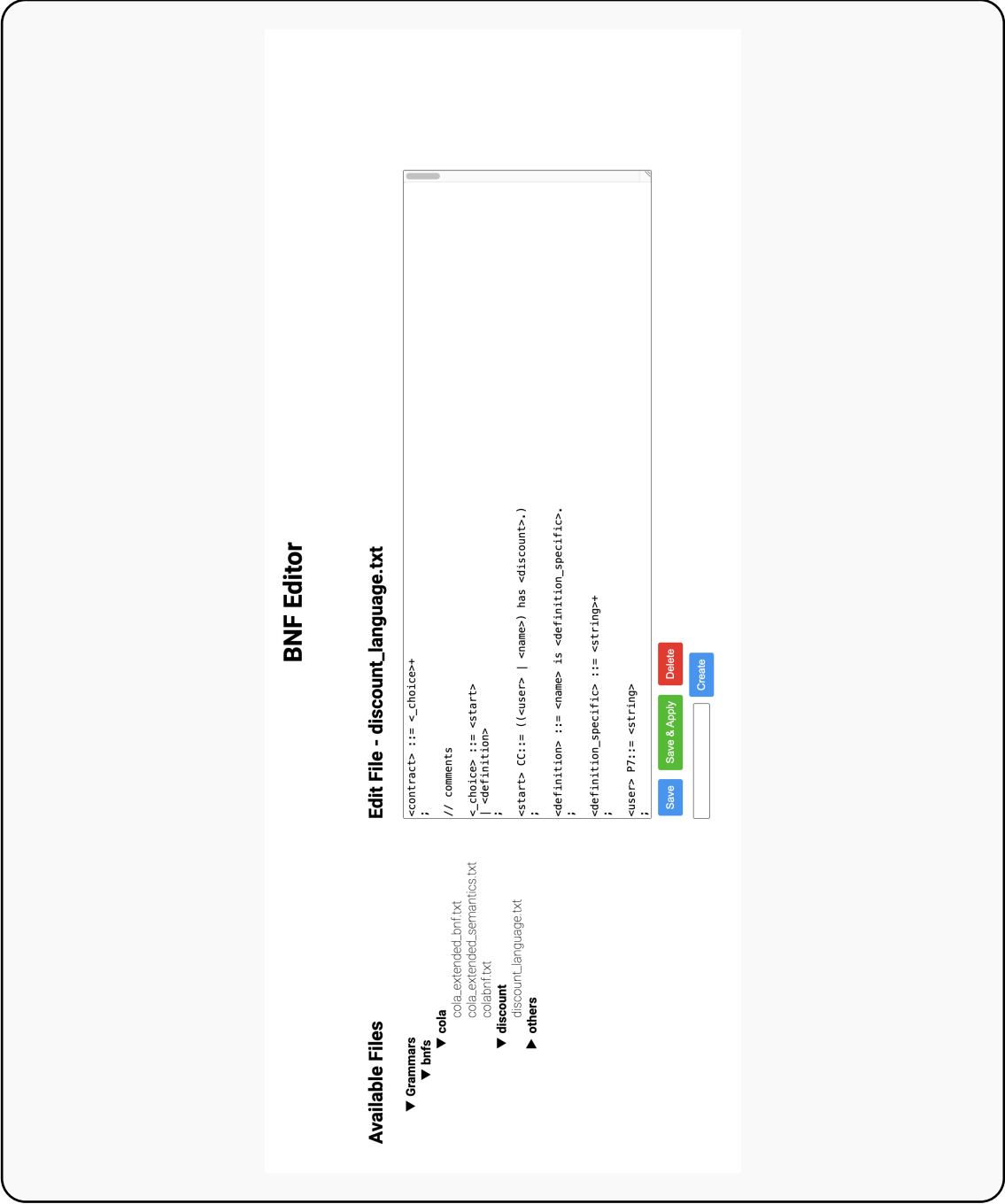
A CC-IDE Main Page

This is a full image of the CC-IDE main page, such that it can be seen more clearly and closely. The left hand side focuses on contract drafting and file management, while the right hand side focuses on contract outputs and analyses.



B CC-IDE BNF Editor

This is a full image of the CC-IDE BNF Editor. It displays the use of the Symbol Prefix system (With symbol *< start >* mapping to “clause”).



C Metadata in the CC-IDE

The section details the implementation of Metadata within the CC-IDE and the information that is held. These pieces of data can be found in the code under folders “contracts/meta” and “bnfs/meta”. The MetaData class in “tools/tool-auxiliary” details the code for implementing Metadata.

```
# Metadata Plan
```

```
## Reasoning
```

```
Each piece of information that is saved requires metadata, allowing us to provide a
↳ structured and standardised way of understanding the information. It is data that
↳ describes data. In our case, this would be used for grammar tracking, date and time
↳ tracking, and more.
```

```
This is important for accountability and integrity. It allows us to track when a piece of
↳ information has last been changed, along with whether what exists now is true to it's
↳ previous version.
```

```
---
```

```
## BNF Metadata
```

```
### Format
```

```
*Filename* - Same as the BNF name
```

```
*Structure* - Metadata_Name: Information
```

```
*To be updated* - Whenever a grammar is saved / applied.
```

```
*To be used* - When retrieving information regarding a grammar, when saving a contract
```

```
We will maintain two structures, one regarding the BNF itself, and another regarding the
↳ the BNF currently being used in the IDE.
```

```
### Information to be held - BNF
```

1. Grammar Name
2. Grammar Version (Previous + 1)
3. Creation Date
4. Update Date

```
Stored in the folder bnfs/meta, each BNF Grammar has their own MetaData file corresponding
↳ to the CC-language name.
```

```
### Information to be held - Grammar currently in use
```

1. Grammar Name
2. Grammar Version
3. Applied Date

```
Stored in the folder bnfs, only one such file exists at any time. Important upon startup
↳ for the previously used CC-language to be reapplied.
```

```
---
```

```
## Contract Metadata
```

```
### Format
```

Filename - Same as the contract name

Structure - Metadata_Name: Information

To be updated - Whenever a contract is saved

To be used - To ensure integrity of the saved file and for accountability.

Need to store a hash of the file, make sure it hasn't been altered.

Information to be held - Contract

1. Contract Name
2. Contract Version
3. Grammar Used
4. Grammar Version
5. Creation Date
6. Save Date
7. Hash of Contract

Stored in the folder contracts/meta. Each contract has their own MetaData file

↔ corresponding to the contract name.

D BNF-Parser BNF Grammar

This details the grammar used for parsing Backus Naur Form. The most updated version will be held within the “tool/tool-auxiliary/antlr” folder as “grammarParser.g4”.

```
// Grammar for parsing grammars
grammar grammarParser;
gram: component+
    ;

component: symbol otter expansion SEMICOLON
    | comment
    ;

comment: ('#' | '//') WORD*
    | '/*' WORD* '*/'
    ;

expansion: arg (SPLITTER arg)*
    ;

arg: entersymbol+
    ;

entersymbol: checksymbol optionalsymbol
    | checksymbol repeatsymbol
    | checksymbol repeat1symbol
    | checksymbol
    ;

optionalsymbol: QUESTION
    ;

repeatsymbol: STAR
    ;

repeat1symbol: PLUS
    ;

bracketsymbol: OPENBRACKET expansion CLOSEBRACKET
    ;

checksymbol: bracketsymbol
    | symbol
    | strings
    | regex
    ;

symbol: CHEVRON WORD ICHEVRON
    ;

otter: OTTER
    | WORD? OTTER
    ;
```

```

strings: QUOTE (WORD | othersymbol)* QUOTE
        | SINGLEQUOTE (WORD | othersymbol)* SINGLEQUOTE
        | WORD
        ;

```

```

othersymbol: OPENBRACKET
            | CLOSEBRACKET
            | SPLITTER
            | CHEVRON
            | ICHEVRON
            | QUESTION
            | PLUS
            | STAR
            | SEMICOLON
            | COLON
            | SLASH
            ;

```

```

regex: SLASH WORD SLASH
      ;

```

```

OTTER : '::~=' ;
OPENBRACKET: '(' ;
CLOSEBRACKET: ')';
SPLITTER: '|';
CHEVRON: '<' ;
ICHEVRON: '>' ; // inverted chevron
QUOTE: '"';
SINGLEQUOTE: '\'';
QUESTION: '?';
PLUS: '+';
STAR: '*';
SEMICOLON: ';';
COLON: ':';
SLASH: '/';

```

```

INT : [0-9]+ ;
WORD: ~[ \t\n?"'<>:|;()/]+;
ID: [a-zA-Z_][a-zA-Z_0-9]* ;
WS: [ \t\n\r\f]+ -> skip ;

```

E Example Function from Visitor Class

visitArg is an example function from the Visitor Class defined by Antlr. It exemplifies how the Parse Tree (PT) of the CC-language BNF grammar is manipulated into a Tree-Sitter grammar.

```
# Visit a parse tree produced by grammarParserParser#arg.
def visitArg(self, ctx:grammarParserParser.ArgContext):
    """
    arg: entersymbol+
    """
    arg_internal_arr = []
    # It will be evaluated further down, you just need to combine them.
    for i in range(ctx.getChildCount()):
        arg_internal_arr.append(self.visit(ctx.getChild(i)))

    arg_string = ""

    # Sequences
    if len(arg_internal_arr) <= 1:
        arg_string += '\n, '.join(arg_internal_arr)
    else:
        arg_string += "seq(" + ', '.join(arg_internal_arr) + ")"

    return arg_string
```

F Main CC-IDE Text Processing Code

This piece of code is used to form the HTML nested node structure. Additional, it is used to flag errors.

```
def nodeAddText(self, node, finalarr):
    if node:
        # In the case of no children, this means that the node is a terminal (leaf)
        → node.
    if node.children:
        # Sets up the css structure + Hover text
        finalarr.append(f'<span title="{node.type}" class="{node.type}">')
    else:
        text = node.text.decode("utf8")

        # Attribute describing if node is an error node
        if node.is_error:
            colour = "#f76f6f"
            finalarr.append(f'<b style="color:{colour};">')

        if node.parent:

            if node.type in self.node_types or node.type in
            → self.node_types.values() and node.type != text:
                finalarr.append(f'<span class="{node.type}">{text} </span>')
            else:
                finalarr.append(f'{text} ')
        else:
            finalarr.append(f'{text} ')
```

This function corresponds to wrapping up an HTML block, ensuring that it is closed.

```
def nodeAddTextEnd(self, node, finalarr):
    if node.children:
        if node:
            finalarr.append(f'</span>')
    else:
        if node.is_error:
            finalarr.append(f'</b>')
```

This function calculates the depth of the condition or clause, ensuring that we correctly generate identifiers.

```
def nodeEvalIDStart(self, node, finalarr):
    if node.type == "clause":
        self.global_count_arr[0] += 1
        self.global_count_arr.append(0)
    elif node.type == "bracket":
        self.global_count_arr[-1] += 1
        self.global_count_arr.append(0)
    elif node.type == "condition":
        self.global_count_arr[-1] += 1
        # Add it to front of condition
        identity = '.'.join([str(i) for i in self.global_count_arr])
        self.internal_id[node.id] = identity
```

```

        return identity
    return ""

```

This function is an auxiliary function used to aid the calculation in the previous depth calculation (Stack structure).

```

def nodeEvalIDEnd(self, node, finalarr):
    if node.type == "clause":
        self.global_count_arr.pop()
    elif node.type == "bracket":
        self.global_count_arr.pop()

```

This function is used to extract definitions from the definition symbol, such that it can be used for definition substitution.

```

def nodeEvalDef(self, node, finalarr):
    if node.type == "definition":
        name = ""
        definition = ""
        for c in node.children:
            if c.type == "name":
                name = c.text.decode("utf8")
            elif c.type == "definition_specific":
                definition = c.text.decode("utf8")
        self.definitions[name] = definition
    return True
return False

```

The following function drives the auto-suggestion mapping functionality, generating the popup with regards to the suggested list of symbols.

```

def nodeAutoSuggestion(self, node, finalarr):
    """
    Doing the AutoSuggestion Mapping

    Checking your current node, previous node, and parent node.
    Then using that to see if it is a valid triplet within the language.
    If it is, we can then provide a suggestion for the next node.
    """

    prev_sibling = ""
    parent = ""

    if node.prev_sibling:
        prev_sibling = node.prev_sibling.type
    if node.parent:
        parent = node.parent.type

    key = (node.type, prev_sibling, parent)

    if key in self.mapper and parent == "ERROR":

        hashed = hash(key)
        # Created a popup class within css.

```

```
finalarr.append(f'<span class="popup" style="color:{"grey"}"
→ onclick="popupFunction(\'{hashed}\')">...<span class="popuptext"
→ id="{hashed}">{self.mapper[key]}</span></span>')
```

This is the base function, used for a depth-first-search traversal of the parse tree. It utilises the Tree-Sitter cursor object, and explores the tree while applying the different aforementioned functions.

```
def exploreNodes(self, cursor, finalarr, reached):

    node = cursor.node

    flag = True

    self.nodeAddText(node, finalarr)

    identity = self.nodeEvalIDStart(node, finalarr)

    self.nodeEvalDef(node, finalarr)

    # Checks if IDs should be displayed
    if reached and identity != "":
        finalarr[-1] += "(" + identity + ") "

    if cursor.goto_first_child():
        while flag:
            self.exploreNodes(cursor, finalarr, reached)
            flag = cursor.goto_next_sibling()
        cursor.goto_parent()

    self.nodeEvalIDEnd(node, finalarr)

    self.nodeAddTextEnd(node, finalarr)

    self.nodeAutoSuggestion(node, finalarr)

    return cursor
```

G OutputCSS function

OutputCSS is the main function used to output the formatting and styling information of a grammar. It writes the code into a designated CSS file “static/parsing-formatting/parse.css”, used exclusively for formatting the contract output. For each node type, it outputs the configured styling options.

```
def outputCSS(highlights, formatting):
    """
    Function for outputting colours and formatting options into a css file.
    Leverages the use of CSS before and after for prefix and suffix.
    """

    with open("static/parsing-formatting/parse.css", "w") as file:
        formats = ""
        for node in highlights.keys():

            node_name_actual = node
            if node in current_app.config["handler"].node_types:
                node_name_actual = current_app.config["handler"].node_types[node]

            if node in formatting:
                prefix = formatting[node]["prefix"]
                suffix = formatting[node]["suffix"]
                inline = formatting[node]["inline"]
                if prefix:
                    formats += f".{node_name_actual}::before {{ {prefix} }}\n"
                if suffix:
                    formats += f".{node_name_actual}::after {{ {suffix} }}\n"
                formats += f".{node_name_actual} {{ {'color: ' + highlights[node] + \n"
                ↪ '; font-weight: bold;' if highlights[node] != '#000000' else ''} \n"
                ↪ {inline} }}\n"
            else:
                formats += f".{node_name_actual} {{ {'color: ' + highlights[node] + \n"
                ↪ '; font-weight: bold;' if highlights[node] != '#000000' else ''} \n"
                ↪ }}\n"

        file.write(formats)

    return True
```

H Autosuggestion Generation Function

This function carries out the tuple extraction described for autosuggestion. It uses the tuple of (Current Symbol, Previous Symbol, Parent Symbol) to create a list of possible proceeding symbols.

```
def autosuggestion(self, grammar_name):
    # Make a mapping system based on Current, Prev, Parent
    mapper = {}

    for node in self.nodes:
        for i in range(len(node.children)):
            for j in range(len(node.children[i])-1):
                child = node.children[i][j]

                childprev = ""
                childnext = ""

                if j > 0:
                    childprev = node.children[i][j-1]
                if j < len(node.children[i]) - 1:
                    childnext = node.children[i][j+1]

                key = (child, childprev, node.name)
                key2 = (child, childprev, "ERROR")

                if key not in mapper:
                    mapper[key] = set()
                if key2 not in mapper:
                    mapper[key2] = set()

                mapper[key].add(childnext)
                mapper[key2].add(childnext)

    with open(f"text-files/{grammar_name[:-4]}/nodemappings.txt", "w") as file:
        for i in mapper.items():
            arr = list(i[0])
            arr.extend(list(i[1]))
            for item in arr:
                file.write(item + ",")
            file.write("\n")

    return
```

I TimeOverride Class Example

This is an example of an overridden TimeOverride class, used for the “discount language”. In particular, this translates the time from both a string and a tree, along with providing the reverse of a string from a Python datetime object. This enables the use of temporal aspects within the CC-IDE such as event simulation.

```
from datetime import datetime

from time_base import TimeBase

class TimeOverride(TimeBase):
    def __init__(self):
        return

    #
    # =====
    # Evaluating Time from a String
    # =====
    #
    def evaluate_time_str(self, time_str) -> datetime:
        time = time_str.split("-")
        date_formats = [
            "%Y-%B-%d",          # Format: 2042-March-21st
            "%d-%B-%Y",          # Format: 21st-March-2042
            "%Y-%m-%d",          # Format: 2042-03-21
            "%d-%m-%Y",          # Format: 21-03-2042
        ]
        # Clean the input date string by removing unwanted suffix like 'st', 'nd',
        # → 'rd', 'th'
        time_str = time_str.replace('st', '').replace('nd', '').replace('rd',
        # → '').replace('th', '')

        for date_format in date_formats:
            try:
                # Attempt to parse the date using the given format
                return datetime.strptime(time_str, date_format)
            except ValueError:
                continue

        # If no format matched, return None or raise an error
        raise ValueError(f"Unrecognized date format: {time_str}")

    def month_to_val(self, month):
        month_dict = {
            "January": 1,
            "February": 2,
            "March": 3,
            "April": 4,
            "May": 5,
            "June": 6,
            "July": 7,
            "August": 8,
            "September": 9,
            "October": 10,
            "November": 11,
            "December": 12
        }
        if month in month_dict:
```

```

        return month_dict[month]
    else:
        if 0 < int(month) <= 12:
            return str(int(month))
        else:
            raise ValueError(f"Unrecognised month format: {month}")

def day_to_val(self, day):
    day_dict = {
        '1st': 1, '2nd': 2, '3rd': 3, '4th': 4, '5th': 5, '6th': 6, '7th': 7,
        '8th': 8, '9th': 9, '10th': 10, '11th': 11, '12th': 12, '13th': 13,
        '14th': 14, '15th': 15, '16th': 16, '17th': 17, '18th': 18, '19th': 19,
        '20th': 20, '21st': 21, '22nd': 22, '23rd': 23, '24th': 24, '25th': 25,
        '26th': 26, '27th': 27, '28th': 28, '29th': 29, '30th': 30, '31st': 31
    }

    if day in day_dict:
        return day_dict[day]
    else:
        if 0 < int(day) <= 31:
            return str(int(day))
        else:
            raise ValueError(f"Unrecognised day format: {day}")

#
# =====
# Evaluating Time from a Tree
# =====
#

def evaluate_time_tree(self, time_tree) -> datetime:
    year = ""
    month = ""
    day = ""
    for c in time_tree.children:
        if c.type == "year":
            year = c.text.decode("utf-8")
        elif c.type == "month":
            month = self.month_to_val(c.text.decode("utf-8"))
        elif c.type == "day":
            day = self.day_to_val(c.text.decode("utf-8"))
    date_format = "%Y-%m-%d"
    return datetime.strptime(f"{year}-{month}-{day}", date_format)

def month_to_string(self, month):
    month_dict = {
        1: "January",
        2: "February",
        3: "March",
        4: "April",
        5: "May",
        6: "June",
        7: "July",
        8: "August",
        9: "September",
        10: "October",
        11: "November",
        12: "December"
    }

```

```

        return month_dict[month]

def day_to_string(self, day):
    day_dict = {
        1: '1st', 2: '2nd', 3: '3rd', 4: '4th', 5: '5th', 6: '6th', 7: '7th',
        8: '8th', 9: '9th', 10: '10th', 11: '11th', 12: '12th', 13: '13th',
        14: '14th', 15: '15th', 16: '16th', 17: '17th', 18: '18th', 19: '19th',
        20: '20th', 21: '21st', 22: '22nd', 23: '23rd', 24: '24th', 25: '25th',
        26: '26th', 27: '27th', 28: '28th', 29: '29th', 30: '30th', 31: '31st'
    }
    return day_dict[day]

#
# =====
# Returning Time from a String
# =====
#
def return_time_str(self, time_datetime) -> str:

    month = self.month_to_string(time_datetime.month)
    day = self.day_to_string(time_datetime.day)
    year = time_datetime.year

    return f"{day}-{month}-{year}"

```

J Full Set of Symbol Prefixes and Usage

As there are a large number of Symbol Prefixes within the GAS system, this serves to document the different prefixes along with how they can be used. However, it should also be noted that the “discount_{language}” provides the clearest example of usage, depicting the relationships of symbols.

Symbol: CC::=

Mapping: clause

Semantic Meaning: A major building block of a contract, and it divides the contract into
→ different deontic components. Within this system, each clause can hold one set of
→ conditions and two sets of statements, the Consequent (What happens if the condition
→ is true) and the Alternative (What happens if the condition is false)

Usage: Highest Level, used to denote the beginning of contract section. Contains

→ "condition" and "statement". Inclusion of "else" denotes that there are two sets of
→ statements.

Symbol: C::=

Mapping: condition

Semantic Meaning: A logical requirement that must be satisfied (true or false) for the
→ associated clause to be triggered or made valid. Acts as a pre-requisite for the
→ clause.

Usage: Must be independent from "statement" and "definition". Can contain "bracket".

→ "bracket" denotes that there exists a set of sub-conditions, these sub-conditions
→ should be under "bracket".

Symbol: S::=

Mapping: statement

Semantic Meaning: A statement asserts a fact, obligation, or action. It is the outcome or
→ consequence of a condition being met, representing what must occur or hold true. It
→ represents the Consequent and Alternative

Usage: Must be independent from "condition" and "definition". Evaluated as a standalone.

Symbol: D::=

Mapping: definition

Semantic Meaning: A definition assigns meaning to a term or phrase used within a contract.
→ It reduces ambiguity by clarifying terminology and providing a precise context.

Usage: Must be independent from "condition" and "statement". Evaluated as a standalone.
→ Should contain the word "IS" as the 2nd element.

Symbol: B::=

Mapping: bracket

Semantic Meaning: Used for grouping expressions and controlling the evaluation order of
→ conditions. Helps ensure that the logical operators are evaluated correctly.

Usage: Should be a holder of "condition"s. It can contain "and_expression" and
→ "or_expression". "and_expression" and "or_expression" denote a sequence of conditions
→ and should follow a flat structure.

Symbol: A::=

Mapping: and_expression

Semantic Meaning: Represents a logical AND. All conditions within the expression must be
→ true for the expression to evaluate to true.

Usage: Should chain "condition"s together. Follow a structure like <condition>
→ (<and_expression> <condition>)*

Symbol: O::=

Mapping: or_expression

Semantic Meaning: Represents a logical OR. At least one condition within the expression
→ has to be true for the expression to evaluate to true.

Usage: Should chain "condition"s together. Follow a structure like <condition>
→ (<or_expression> <condition>)*

Symbol: E::=

Mapping: else

Semantic Meaning: Represents the existence of alternatives within the clause. Used to
↪ separate the sets of statements.

Usage: It's placement is not necessary, but has to be after the first set of statements to

↪ evaluate correctly. i.e. <statement> <else_expression> <statement> and not <statement>
↪ <statement> <else_expression>

Symbol: TT::=

Mapping: time

Semantic Meaning: The time object itself. It is the container for the time text which is
↪ translated into a datetime object.

Usage: The symbol that gets translated, this should encapsulate the text for time within a

↪ language. e.g. <day> <month> <year> This will then be translated via the overridden
↪ class.

Symbol: TTH

Mapping: time_holder

Semantic Meaning: Holds all aspects related to time, such that it can be evaluated beyond
↪ just the time object itself. Used for declaring ranges and for using operators.

Usage: Contains the time symbol as well as the operators and range declarators. Should

↪ follow the structure (<time_after> | <time_before>) <time> ((<time_or> | <time_and>)
↪ (<time_after> | <time_before>) <time>)?

Symbol: TTCA

Mapping: time_and

Semantic Meaning: For linking two times together to create a range. Used for creating
↪ ranges between two times.

Usage: Comes in between two sets of times. Should be within the time_holder.

Symbol: TTCO

Mapping: time_or

Semantic Meaning: For linking two times together to create a range. Used for creating
↪ ranges outside two times.

Usage: Comes in between two sets of times. Should be within the time_holder.

Symbol: TTB

Mapping: time_before

Semantic Meaning: For declaring that a condition should happen before a certain time.

Usage: Used as a boolean operator for a time symbol within the time_holder. i.e.

↪ <time_before> <time>

Symbol: TTA

Mapping: time_after

Semantic Meaning: For declaring that a condition should happen after a certain time.

Usage: Used as a boolean operator for a time symbol within the time_holder. i.e.

↪ <time_after> <time>

Symbol: TTO

Mapping: time_on

Semantic Meaning: For declaring that a condition should happen on a certain time.

Usage: Used as a boolean operator for a time symbol within the time_holder. i.e. <time_on>

↪ <time>

K Condition Simulation in CC-IDE

These two pictures show the Condition Simulation tab within the CC-IDE, whereby a Lawyer can toggle boolean flags relating to a contract’s conditions in order to simulate its performance. The two images show the toggling of a set of conditions and it’s subsequent effect.

The contract in question is as follows (Written in the “discount language”):

```
Member has 7% discount if [they are in london on 6th-May-2024; and they are a member on  
→  the 5th-May-2024;] or [they are in wales; and it is not true that they are a member;].
```

Conditions

1.1.1
false
they are in london on 6th-May-2024;

1.1.2
false
they are a member on the 5th-May-2024;

1.2.1
false
they are in wales;

1.2.2
true
it is not true that they are a member;

States

1
false
7% discount

Before

Conditions

1.1.1
true
they are in london on 6th-May-2024;

1.1.2
true
they are a member on the 5th-May-2024;

1.2.1
false
they are in wales;

1.2.2
true
it is not true that they are a member;

States

1
true
7% discount

After

L Event Simulation in CC-IDE

This picture shows the Event Simulation tab within the CC-IDE, whereby a Lawyer can choose a relevant condition and create an event that fulfils it (with respect to time). This allows for dynamic analysis of a contract’s performance. States refer to the statements within the contract, events are the ones created by the lawyer, and Event Builder is the interface for creating the events that occur.

States

•

1

false

7% discount

Events

•

1.1.1

8th-May-2024

they are in london ;

Event Builder

they are in london ;

8th-May-2024

Submit

M Event Simulation Logic and Code

Event simulation within the CC-IDE utilises all Semantic components laid out within the GAS system. Resultantly, it demonstrates how the application of GAS allows for a generalisation of tools and features, with event simulation being just one such facet. An interesting piece of code relates to the mapping of functions based on the node types. By using the node types as keys and then placing a function as the value, this allows us to create a generalised function (An alternative to if statements) beneficial when searching the parse tree, and done so in a scalable manner.

```
try:
    from time_override import TimeOverride
except ImportError:
    from time_base import TimeOverride

from datetime import datetime

class Condition:
    def __init__(self, identifier, sentence, negation, time_cond=None):
        self.identifier = identifier
        self.sentence = sentence
        self.flag = False
        self.negation = negation
        self.time_cond = time_cond

    def __repr__(self):
        return self.identifier + " --- " + str(self.flag)

    def evaluate(self):
        if self.negation:
            return not self.flag
        else:
            return self.flag

class Statement:
    def __init__(self, identifier, sentence, if_else, and_or):
        self.identifier = identifier
        self.sentence = sentence
        self.conditions = []
        self.if_else = if_else
        self.and_or = and_or  # AND = TRUE, OR = FALSE

    def __repr__(self):
        return self.identifier + " --- " + str(self.evaluate())

    def evaluate(self):
        flag = False

        if self.and_or:
            # When it is an AND
            flag = True
            for c in self.conditions:
                flag = flag and c.evaluate()
        else:
            # When it is an OR
            flag = False
            for c in self.conditions:
                flag = flag or c.evaluate()
```

```

        if self.if_else:
            return not flag
        else:
            return flag

class EventSim:
    def __init__(self):
        # Global dictionary of conditions
        self.conditions = {}
        # Global dictionary of statements and definitions
        self.state_def = {}
        # Array of Past Events - Not necessarily chronologically in the past
        self.past_events = []

        # Mapping for functions, based on node types
        self.mapper = {
            "statement": self.s_d_function,
            "statement_specific": self.ss_function,
            "definition": self.s_d_function,
            "condition": self.c_function,
            "else": self.e_function,
            "bracket": self.b_function,
            "and_expression": self.a_function,
            "or_expression": self.o_function,
            "negation": self.n_function,
            "time_holder": self.t_function,
        }

        self.and_or = True
        self.negation = False

        # For extracting time
        self.time = None
        self.time_convert = TimeOverride()

        # For unique identifier
        self.global_count = 1

        # For unique identifier
        self.global_count_arr = [0]

    def eval_count(self):
        return '.'.join([str(i) for i in self.global_count_arr])

    def find_clauses(self, tree, clause_arr):
        # cursor = tree.walk()
        # First find the clauses. BFS first

        root = tree.root_node

        self.explore_clauses(root, clause_arr)

    def explore_clauses(self, node, clause_arr):
        for c in node.children:
            if c.type == "clause":
                # Start with two arrays, first is for conditions, second for THEN,
                # ↪ add third for ELSE
                clause_arr.append([], [])
                # Default is AND, then flips if necessary

```

```

        self.and_or = True

        # Increment count
        self.global_count_arr[0] += 1

        self.global_count_arr.append(0)

        # Completely recursive.
        self.explore(c, clause_arr)

        self.global_count_arr.pop()

        # Have to the ands and ors at the clause level because it's not
        ↪ nested within the statement

        # First set of statements
        for child in clause_arr[-1][1]:
            child.and_or = self.and_or

        # Second set of statements (ELSE)
        if len(clause_arr[-1]) > 2:
            for child in clause_arr[-1][2]:
                child.and_or = self.and_or
                child.if_else = True
        else:
            self.explore_clauses(c, clause_arr)

# AND Function
def a_function(self, c, clause_arr):
    self.and_or = True

# OR Function
def o_function(self, c, clause_arr):
    self.and_or = False

# BRACKET Function
def b_function(self, c, clause_arr):

    # Storing previous level operator
    old_and_or = self.and_or

    # Increment count
    self.global_count_arr[-1] += 1
    self.global_count_arr.append(0)

    clause_arr.append([[], []])
    self.explore(c, clause_arr)
    state = Statement(
        str(c.id),
        c.text.decode("utf8"),
        if_else=False,
        and_or=self.and_or,
    )
    state.conditions = clause_arr.pop()[0]
    clause_arr[-1][0].append(state)

    # Pop count
    self.global_count_arr.pop()

    # Restoring previous level operator
    self.and_or = old_and_or

```

```

# HELPER Function for extracting text without time
def extract_text(self, c, arr):
    if c.type == "time_holder":
        return
    else:
        if len(c.children) > 0:
            for child in c.children:
                self.extract_text(child, arr)
        else:
            arr.append(c.text.decode("utf8"))

# CONDITION Function
def c_function(self, c, clause_arr):
    self.negation = False
    self.time = None

    # Increment count
    self.global_count_arr[-1] += 1

    self.explore(c, clause_arr)

    str_arr = []
    self.extract_text(c, str_arr)

    cond = Condition(
        self.eval_count(),
        # c.text.decode("utf8"),
        ' '.join(str_arr),
        self.negation,
        self.time,
    )

    # Adding to array of conditions for that clause
    clause_arr[-1][0].append(cond)
    # Adding to dictionary, easier to access
    self.conditions[cond.identifier] = cond

# STATEMENT DEFINITION Function
def s_d_function(self, c, clause_arr):
    # This is evaluated too early. - Now it is evaluated fine but the text is
    ↪ off.
    self.explore(c, clause_arr)
    """
    Realistically, you should be using statement as a more general one, and not
    ↪ nest conditions under it.
    """

    state = Statement(
        str(self.global_count),
        c.text.decode("utf8"),
        len(clause_arr[-1]) == 3,
        and_or=self.and_or,
    )
    self.global_count += 1
    clause_arr[-1][-1].append(state)
    state.conditions = clause_arr[-1][0]

# ELSE Function
def e_function(self, c, clause_arr):
    clause_arr[-1].append([])

```

```

# NEGATION Function
"""
Need to think about how to use negations, are all conditions starting off false
↪ or?
"""
def n_function(self, c, clause_arr):
    self.negation = True

# TIME Function
"""
For extracting time from condition (Should be made to condition & statement
↪ level)
"""
def t_function(self, c, clause_arr):
    self.time = self.time_convert.time_analyser(c)

# STATEMENT_SPECIFIC Function
def ss_function(self, c, clause_arr):
    return c.text.decode("utf8")

def explore(self, node, clause_arr):
    for c in node.children:
        if c.type in self.mapper:
            self.mapper[c.type](c, clause_arr)
        else:
            self.explore(c, clause_arr)

def event_simulation(self, parse_tree):
    # Resetting the simulation
    self.conditions = {}
    self.state_def = {}
    self.past_events = []
    self.global_count = 1
    self.global_count_arr = [0]

    clause_arr = []
    self.find_clauses(parse_tree, clause_arr)
    for clause in clause_arr:

        for s in clause[1]:
            self.state_def[s.identifier] = s

        if len(clause) > 2:
            for e in clause[2]:
                self.state_def[e.identifier] = e

def evaluate_condition(self, identity, time_str):

    time_obj = datetime.now()

    if time_str != "":
        time_obj = self.time_convert.evaluate_time_str(time_str)

    self.past_events.append((identity,
        ↪ self.time_convert.return_time_str(time_obj),
        ↪ self.conditions[identity].sentence))

    time_comparison = self.evaluate_condition_time(identity, time_obj)

    if time_comparison:

```

```

        self.conditions[identity].flag = True
    else:
        self.conditions[identity].flag = False

# Function for evaluating time against the condition time
def evaluate_condition_time(self, identity, time_obj):
    # Getting the time tuple
    tc = self.conditions[identity].time_cond

    if tc == None:
        return True
    else:
        # Checking if it is before, after, or on the date.

        if len(tc) == 2:
            val, op = tc

            if op == "o":
                if time_obj == val:
                    return True
            elif op == "a":
                if time_obj > val:
                    return True
            elif op == "b":
                if time_obj < val:
                    return True

        elif len(tc) == 5:

            val1, op1, condition, val2, op2 = tc

            if op1 != op2:
                if condition == "and":
                    # Inclusive
                    # <A t B>
                    if min(val1, val2) <= time_obj <= max(val1, val2):
                        return True
                elif condition == "or":
                    # t A> or <B t
                    if min(val1, val2) >= time_obj or max(val1, val2) <=
                        ↪ time_obj:
                        return True

    return False

def toggle_condition(self, identity):
    self.conditions[identity].flag = not self.conditions[identity].flag

def evaluate_state_def(self, identity):
    return self.state_def[identity]

```

N Unit Tests - Parser Pipeline

This details the unit tests used in verifying the BNF to Tree-Sitter grammar conversion. In particular, it focuses on testing all the syntactic notions of BNFs.

```
def test_single_rule(self):
    self.assertEqual(rulebuild('<rule> ::= <expansion>;'), 'rule: $ =>
↳ ($.expansion\n),')

def test_single_rule_optional(self):
    self.assertEqual(rulebuild('<rule> ::= <expansion>;'), 'rule: $ =>
↳ (optional($.expansion)\n),')

def test_single_rule_repeat(self):
    self.assertEqual(rulebuild('<rule> ::= <expansion>;'), 'rule: $ =>
↳ (repeat($.expansion)\n),')

def test_single_rule_repeat1(self):
    self.assertEqual(rulebuild('<rule> ::= <expansion>;'), 'rule: $ =>
↳ (repeat1($.expansion)\n),')

def test_alternative_rule(self):
    self.assertEqual(rulebuild('<rule> ::= <expansion> | <expansion2>;'), 'rule: $ =>
↳ choice(\n\t$.expansion\n\t,$.expansion2\n),')

def test_alternative_rule_optional(self):
    self.assertEqual(rulebuild('<rule> ::= <expansion>? | <expansion2>;'), 'rule: $
↳ => choice(\n\t optional($.expansion)\n\t,$.expansion2\n),')

def test_alternative_rule_repeat(self):
    self.assertEqual(rulebuild('<rule> ::= <expansion>* | <expansion2>;'), 'rule: $
↳ => choice(\n\t repeat($.expansion)\n\t,$.expansion2\n),')

def test_alternative_rule_repeat1(self):
    self.assertEqual(rulebuild('<rule> ::= <expansion>+ | <expansion2>;'), 'rule: $
↳ => choice(\n\t repeat1($.expansion)\n\t,$.expansion2\n),')

def test_alternative_rule_multi_optional(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion> | <expansion2>)?;'), 'rule: $
↳ => (optional(choice(\n\t$.expansion\n\t,$.expansion2\n))\n),')

def test_alternative_rule_multi_repeat(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion> | <expansion2>)*;'), 'rule: $
↳ => (repeat(choice(\n\t$.expansion\n\t,$.expansion2\n))\n),')

def test_alternative_rule_multi_repeat1(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion> | <expansion2>)+;'), 'rule: $
↳ => (optional(choice(\n\t$.expansion\n\t,$.expansion2\n))\n),')

def test_alternative_rule_multi_optional_within(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion>? | <expansion2>)?;'), 'rule:
↳ $ => (optional(choice(\n\toptional($.expansion)\n\t,$.expansion2\n))\n),')

def test_alternative_group(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion> | <expansion2>)
↳ <expansion3>;'), 'rule: $ => (seq(choice(\n\t
↳ $.expansion\n\t,$.expansion2\n), $.expansion3)\n),')
```

```

def test_alternative_group_2(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion> | <expansion2>) |
↳ <expansion3>;'), 'rule: $ => choice(\n\tchoice(\n\t
↳ $.expansion\n\t,$.expansion2\n)\n\t,$.expansion3\n),')

def test_alternative_group_optional(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion> | <expansion2>)? |
↳ <expansion3>;'), 'rule: $ => choice(\n\toptional(choice(\n\t
↳ $.expansion\n\t,$.expansion2\n))\n\t,$.expansion3\n),')

def test_alternative_group_repeat(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion> | <expansion2>)* |
↳ <expansion3>;'), 'rule: $ => choice(\n\t
↳ repeat(choice(\n\t$.expansion\n\t,$.expansion2\n))\n\t,$.expansion3\n),')

def test_alternative_group_repeat1(self):
    self.assertEqual(rulebuild('<rule> ::= (<expansion> | <expansion2>)? |
↳ <expansion3>;'), 'rule: $ => choice(\n\t
↳ optional(choice(\n\t$.expansion\n\t,$.expansion2\n))\n\t,$.expansion3\n),')

def test_alternative_group_optional_within(self):
    self.assertEqual(rulebuild('<rule> ::= ((<expansion> | <expansion2>)? |
↳ <expansion3>)?;'), 'rule: $ => (optional(choice(\n\t
↳ optional(choice(\n\t$.expansion\n\t,$.expansion2\n))\n\t,$.expansion3\n))\n),')

def test_rule_special_precedence(self):
    self.assertEqual(rulebuild('<rule> P0::= "hello";'), 'rule: $ =>
↳ prec(0,("hello"\n)),')

def test_rule_special_dynamic_precedence(self):
    self.assertEqual(rulebuild('<rule> PD0::= hello;'), "rule: $ =>
↳ prec.dynamic(0,('hello'\n)),")

def test_rule_special_left_precedence(self):
    self.assertEqual(rulebuild('<rule> L::= "hello world";'), 'rule: $ =>
↳ prec.left(("hello world"\n)),')

def test_rule_special_right_precedence(self):
    self.assertEqual(rulebuild('<rule> R::= hello world;'), "rule: $ =>
↳ prec.right((seq('hello', 'world')\n)),")

def test_rule_special_left_precedence_2(self):
    self.assertEqual(rulebuild('<rule> L0::= hello "world";'), 'rule: $ =>
↳ prec.left(0,(seq(\'hello\', "world")\n)),')

def test_rule_special_right_precedence_2(self):
    self.assertEqual(rulebuild('<rule> R0::= "hello"*;'), 'rule: $ =>
↳ prec.right(0,(repeat("hello")\n)),')

def test_single_rule_terminal_expansion(self):
    self.assertEqual(rulebuild('<rule> ::= "hello";'), 'rule: $ => ("hello"\n),')

def test_single_rule_terminal_expansion_2(self):
    self.assertEqual(rulebuild('<rule> ::= hello;'), "rule: $ => ('hello'\n),")

def test_single_rule_terminal_phrase_expansion(self):
    self.assertEqual(rulebuild('<rule> ::= "hello world";'), 'rule: $ => ("hello
↳ world"\n),')

def test_single_rule_terminal_phrase_expansion_2(self):

```

```

self.assertEqual(rulebuild('<rule> ::= hello world;'), 'rule: $ => (seq('hello',
↪ 'world')\n),')

def test_single_rule_terminal_phrase_mix(self):
self.assertEqual(rulebuild('<rule> ::= hello "world";'), 'rule: $ =>
↪ (seq(\'hello\', "world")\n),')

def test_single_rule_terminal_repeat(self):
self.assertEqual(rulebuild('<rule> ::= "hello"*;'), 'rule: $ =>
↪ (repeat("hello")\n),')

def test_single_rule_terminal_repeat_2(self):
self.assertEqual(rulebuild('<rule> ::= "hello"+;'), 'rule: $ =>
↪ (repeat1("hello")\n),')

def test_single_rule_terminal_optional(self):
self.assertEqual(rulebuild('<rule> ::= "hello"?;'), 'rule: $ =>
↪ (optional("hello")\n),')

```

O Unit Tests - Symbol Prefix Mapping

This details the unit tests used in verifying the Symbol Prefix mapping system.

```
def test_symbol_none(self):
    symbol_replacer = BNFParse('','','')
    parsed_output = parsing('<rule> ::= "hello";')
    rules = parsed_output[0]
    replace_arr = parsed_output[1]
    output_rules = symbol_replacer.buildGrammar(rules, replace_arr)
    self.assertEqual(output_rules, 'rule: $ => ("hello"\n),\n\n')

def test_symbol_no_replace(self):
    symbol_replacer = BNFParse('','','')
    parsed_output = parsing('<rule> C::= "hello";')
    rules = parsed_output[0]
    replace_arr = parsed_output[1]
    output_rules = symbol_replacer.buildGrammar(rules, replace_arr)
    self.assertEqual(output_rules, 'rule: $ => ("hello"\n),\n\n')

def test_symbol_no_replace_alternative(self):
    symbol_replacer = BNFParse('','','')
    parsed_output = parsing('<rule> S::= "hello" | <world>;')
    rules = parsed_output[0]
    replace_arr = parsed_output[1]
    output_rules = symbol_replacer.buildGrammar(rules, replace_arr)
    self.assertEqual(output_rules, 'rule: $ =>
↳ choice(\n\t"hello"\n\t,$.world\n),\n\n')

def test_symbol_replace(self):
    symbol_replacer = BNFParse('','','')
    parsed_output = parsing('<rule2> ::= <rule>;\n<rule> S::= "hello";')
    rules = parsed_output[0]
    replace_arr = parsed_output[1]
    output_rules = symbol_replacer.buildGrammar(rules, replace_arr)
    self.assertEqual(output_rules, 'rule2: $ => (alias($.rule,
↳ $.statement)\n),\n\nrule: $ => ("hello"\n),\n\n')

def test_symbol_replace_alternative(self):
    symbol_replacer = BNFParse('','','')
    parsed_output = parsing('<rule2> ::= <rule> | <rule> "hello";\n<rule> S::=
↳ "hello";')
    rules = parsed_output[0]
    replace_arr = parsed_output[1]
    output_rules = symbol_replacer.buildGrammar(rules, replace_arr)
    self.assertEqual(output_rules, 'rule2: $ => choice(\n\talias($.rule,
↳ $.statement)\n\t,seq(alias($.rule, $.statement), "hello")\n),\n\nrule: $ =>
↳ ("hello"\n),\n\n')

def test_symbol_similar(self):
    symbol_replacer = BNFParse('','','')
    parsed_output = parsing('<rule2> ::= <ruler>;\n<rule> S::= "hello";')
    rules = parsed_output[0]
    replace_arr = parsed_output[1]
    output_rules = symbol_replacer.buildGrammar(rules, replace_arr)
    self.assertEqual(output_rules, 'rule2: $ => ($.ruler\n),\n\nrule: $ =>
↳ ("hello"\n),\n\n')
```

P Discount Language BNF

This is the BNF for the Discount Language, a custom CC-language designed for the CC-IDE. It contains all of the semantic mappings, carried out through both the use of Symbol Prefixes as well as direct Symbol names. While it is unlikely to serve as a full contract CC-language, it can still be considered a form of a CC-language due to its incorporation of contract components. Importantly, the Discount Language played a large role to comprehensively test the CC-IDE and its features.

```
<contract> ::= <_choice>+
;

<_choice> ::= <start>
| <definition>
;

<start> CC ::= ((<user> | <name>) has <discount>.)
;

<definition> ::= <name> is <definition_specific>.
;

<definition_specific> ::= <string>+
;

<user> P7 ::= <string>
;

<name> ::= <string>
;

<negation> N ::= (it is not true that)
;

<discount> ::= <discount_specific> ("and" <discount_specific>)* (if <conditional> (<else>
↪ <discount_specific> ("and" <discount_specific>)*))?)?
;

<discount_specific> S ::= <num>% (discount | off)?
| $<num> off
;

<else> ::= "else"
;

<conditional_upper> B ::= <bracketopen> <conditional> <bracketclose>
;

<bracketopen> ::= [
;

<bracketclose> ::= ]
;

<conditional> ::= <conditional_upper>
| <conditional_and>
| <conditional_or>
| <condition_n>
;

<conditional_and> ::= (<conditional_upper> | <condition_n>) ( <and_connect>
↪ (<conditional_upper> | <condition_n>))+
```

```

;

<and_connect> A ::=
and
;

<conditional_or> ::= (<conditional_upper> | <condition_n> ) ( <or_connect>
↳ (<conditional_upper> | <condition_n>)) +
;

<or_connect> O ::=
or
;

<condition_n> C ::= <negation>? (<string>)+ <time_holder>? <semicolon>
;

<time_holder> ::= <_pre_time>
| <_pre_time> (<time_and> | <time_or>) <_pre_time>
;

<_pre_time> ::= (<time_before> | <time_after> | <time_on>) <time>
;

<time_and> ::= "and"
;

<time_or> ::= "or"
;

<time_before> ::= "before" "the"?
;

<time_after> ::= "after" "the"?
;

<time_on> ::= "on" "the"?
;

<time> ::= <day>"-"<month>"-"<year>
| <year>"-"<month>"-"<day>
;

<day> ::= "1st" | "2nd" | "3rd" | "4th" | "5th" | "6th" | "7th" | "8th" | "9th" | "10th" |
↳ "11th" | "12th" | "13th" | "14th" | "15th" | "16th" | "17th" | "18th" | "19th" |
↳ "20th" | "21st" | "22nd" | "23rd" | "24th" | "25th" | "26th" | "27th" | "28th" |
↳ "29th" | "30th" | "31st" | <_numSingle><_numSingle>
;

<month> ::= "January" | "February" | "March" | "April" | "May" | "June" | "July" |
↳ "August" | "September" | "October" | "November" | "December" |
↳ <_numSingle><_numSingle>
;

<year> ::= <_numSingle><_numSingle><_numSingle><_numSingle>
;

<string> ::= /[a-zA-Z]+/
;

<num> ::= /[0-9]+/
;

```

```
<_numSingle> ::= /[0-9]/  
;  
  
<semicolon> ::= ","  
;
```

Q Testing Overview

This checklist contains the different criteria and functional requirements that had to be achieved. As the CC-IDE is a broad and multi-layered software, it is difficult to conduct a full test of functionality other than through empirical testing.

1. Main Editor Page

(a) Contract Drafting

- i. Able to write contracts ✓
- ii. Able to edit contracts ✓
- iii. Able to save contracts ✓
 - A. Unable to save with the same file name ✓
- iv. Able to load contracts ✓
 - A. Unable to load nonexistent contracts ✓
- v. Able to delete contracts ✓
 - A. Unable to delete nonexistent contracts ✓

(b) Parsed Output

- i. Contract text is visible ✓
 - A. Contract text is automatically formatted ✓
 - B. Contract text contains syntax highlighting ✓
 - C. Contract text indicates errors ✓
- ii. Autosuggestions are provided ✓
- iii. Identifiers are generated ✓
 - A. IDs are in correct order ✓
 - B. IDs have correct nesting ✓
- iv. Definitions are substituted ✓
 - A. Definitions are substituted within the text ✓
 - B. Definitions only substitute the word as a standalone and not if in a larger word ✓

(c) Contract Analysis

- i. Error Analysis provides a consistency measure ✓
 - A. When errors are introduced, the consistency measure will reduce ✓
- ii. Time Conflict analysis will detect if time ranges are inconsistent or invalid ✓
 - A. When errors are introduced, the identifier will be displayed ✓

2. Visualisation Page

(a) Tree View

- i. The contract is correctly represented in a tree structure ✓
- ii. The tree contains all terminals and non-terminals ✓
- iii. The relationships are clear and obvious ✓

(b) Hierarchical View

- i. The contract is displayed in the correct file-like structure ✓
- ii. The terminals and non-terminals are displayed hierarchically ✓
- iii. The correct indentations are placed ✓
- iv. Terminals are in green, non-terminals in black ✓

(c) Abstracted Hierarchical View

- i. The contract is displayed in the correct file-like structure ✓

- ii. The terminals and non-terminals are displayed hierarchically ✓
 - iii. The correct indentations are placed ✓
 - iv. The visualisation options work ✓
 - A. The options fold out correctly ✓
 - B. It is possible to check the desired non-terminals and terminals ✓
 - C. Only the selected ones are displayed ✓
 - D. Any unselected ones are abstracted ✓
 - (d) Boolean View
 - i. The contract is split into its clauses ✓
 - A. Boilerplate text is removed ✓
 - B. Statements are isolated ✓
 - C. Alternatives are isolated ✓
 - D. Conditions are isolated ✓
 - E. Conditions have the correct boolean structure with the correct indentation based on depth ✓
3. BNF Editor Page
- (a) BNF Editor
 - i. Creating CC-Languages
 - A. Able to write and edit BNFs within the Edit File box ✓
 - B. Able to save, apply, delete, and create new BNFs ✓
 - C. Able to load previously created BNFs ✓
 - ii. Applying CC-Languages
 - A. CC-Language is correctly applied within main interface ✓
 - B. Semantic Mappings (GAS) are carried out correctly, verifiable in grammar ✓
 - C. Precedence rules are applied correctly ✓
 - D. Invalid grammars are noted and reflected to the user ✓
 - E. Errors are thrown when the BNF is invalid ✓
 - F. Errors are thrown when issues arise within the BNF ✓
 - iii. Highlight Options
 - A. Able to show highlight options ✓
 - B. Terminals and Non-terminals appear correctly ✓
 - C. Colouring system works as expected, allowing you to drag a colour slider ✓
 - D. Able to save colours ✓
 - E. Colours reflected correctly in main editor ✓
 - iv. Formatting Options
 - A. Able to show formatting options ✓
 - B. Terminals and Non-terminals appear correctly ✓
 - C. CSS System works correctly, allowing you to input CSS rules ✓
 - D. Able to save rules ✓
 - E. Rules reflected correctly in main editor ✓
 - v. Time Override
 - A. Accepts python files as input ✓
 - B. Allows file to be uploaded ✓
 - C. Renames the python file into the correct `time.py` file ✓
 - D. Validation button works correctly, flagging if the translations are accurate ✓
 - E. Errors are thrown if the translations do not work ✓
 - F. Errors are thrown if the file does not work ✓
4. Simulation Page

(a) Condition Simulation

- i. Conditions are correctly identified and displayed ✓
- ii. Statements are correctly identified and displayed ✓
- iii. Conditions can be toggled ✓
- iv. Statements are correctly evaluated ✓
- v. The correct order and evaluation of boolean conditions are carried out ✓

(b) Event Simulation

- i. Statements are correctly identified and displayed ✓
- ii. Events can be built ✓
 - A. Time can be added ✓
 - B. Condition can be chosen ✓
- iii. Events are displayed correctly ✓
- iv. Statements are evaluated correctly with respect to the events ✓

R Semantically Extended CoLa Grammar

Furthermore, it is necessary to extend an existing grammar with the semantic notions, such that it can be proven that the GAS system is grammar agnostic. In this case, we map the main contract components onto CoLa, such that it can be used for some of the analyses. However, as CoLa is not designed around the GAS system, it can be noted that applying all semantic notions is a difficult task and requires manipulation of the original grammar.

```
<contract>
::= <component> (C-AND <component>)*
;

<component>
CC::= <component-definition>
    | <conditional-definition>
    | <component-statement>
    | <conditional-statement>
;

<component-definition>
::= <definition> (AND <definition>)*
;

<definition>
::= <ID> <subject> IS <subject>
    | <ID> <subject> EQUALS <numerical-expression>
;

<numerical-expression>
L::= <num>
    | <numerical-object>
    | <numerical-expression> <operator> <numerical-expression>
;

<operator>
::= PLUS
    | MINUS
    | TIMES
    | DIVIDE
;

<conditional-definition>
::= <component-definition> IF <component-condition>
    | IF <component-condition> THEN <component-definition>
    | <component-definition> IF <component-condition> <else> <component-definition>
    | IF <component-condition> THEN <component-definition> <else> <component-definition>
;

<component-statement>
::= <statement> (AND <statement>)* (OR <statement>)*
;

<conditional-statement>
::= <component-statement> IF <condition>
    | IF <component-condition> THEN <component-statement>
    | <component-statement> IF <component-condition> <else> <component-statement>
    | IF <component-condition> THEN <component-statement> <else> <component-statement>
;

<else>
::= (ELSE | OTHERWISE)
```

```

;

<statement>
R ::= <ID> <holds>? <subject> <modal-verb> <verb> <object> <receiver> <date>
    | <ID> <holds>? <subject> <date> <modal-verb> <verb> <object> <receiver>
    | <ID> <holds>? <date> <subject> <modal-verb> <verb> <object> <receiver>
    | <ID> <holds>? <subject> <verb-status> <object> <receiver> <date>
;

<component-condition>
R ::= <condition> (<or> <condition>)*
    | <condition> (<and> <condition>)*
;

<and> ::=
AND
;

<or> ::=
OR
;

<condition>
R ::= <ID> <holds>? <subject> <verb-status> <object> <receiver> <date>
    | <ID> <holds>? <subject> <date> <verb-status> <object> <receiver>
    | <ID> <holds>? <date> <subject> <verb-status> <object> <receiver>
    | <ID> <holds>? <subject> <modal-verb> <verb> <object> <receiver> <date>
    | <ID> <holds>? <boolean-expression>
;

<boolean-expression>
::= <subject> <verb-status> <comparison> <subject>
;

<ID>
R ::= [ <num> ]
    | [ <num> ( <num> ) ]
;

<holds>
::= it is the case that
    | it is not the case that
;

<subject>
::= <string>
;

<verb>
::= deliver
    | pay
    | charge
    | refund
;

<verb-status>
::= delivered
    | paid
    | charged
    | refunded
;

```

```

<comparison>
::= less than
   | <equal>
   | <more-than>
;

<equal>
::= equals
   | equal to
;

<more-than>
::= more than
   | greater than
;

<modal-verb>
::= <obligation>
   | may
   | is forbidden to
;

<obligation>
::= shall
   | must
;

<date>
::= <specific-date>
   | on ANYDATE
   | on SOMEDATE <subject>
   | on THEDATE <subject>
   | <temporal-quantifier> <num> <month> <num>
   | <temporal-quantifier> SOMEDATE <subject>
   | <temporal-quantifier> THEDATE <subject>
   | <temporal-offset> <temporal-quantifier> SOMEDATE <subject>
   | <temporal-offset> <temporal-quantifier> THEDATE <subject>
   | <temporal-quantifier> <temporal-offset> <temporal-quantifier> SOMEDATE <subject>
   | <temporal-quantifier> <temporal-offset> <temporal-quantifier> THEDATE <subject>
;

<temporal-quantifier>
::= before
   | after
;

<specific-date>
::= on the <num> <month> <num>
   | on <num> <month> <num>
;

<temporal-offset>
::= <num> day
   | <num> week
   | <num> year
   | <num> days
   | <num> weeks
   | <num> years
;

<month>
::= January

```

```

| February
| March
| April
| May
| June
| July
| August
| September
| October
| November
| December
;

<object>
::= <numerical-object>
   | <nonnumerical-object>
;

<numerical-object>
::= <pounds> <num>
   | <dollars> <num>
   | <euros> <num>
   | AMOUNT <subject>
;

<nonnumerical-object>
::= SOMECURRENCY <string>
   | REPORT <string>
   | NAMEDOBJECT <string>
   | OTHEROBJECT <string>
;

<pounds>
::= GBP
   | POUNDS
   | quid
;

<dollars>
::= USD
   | DOLLARS
   | buck
;

<euros>
::= EUR
   | EUROS
;

<receiver>
::= to <subject>
;

<string> ::= /[a-zA-Z]+/
;

<num> ::= /[0-9]+/
;

```