INTERNATIONAL COMPUTATIONAL LAW JOURNAL

CODEX SOURCE ARCHIVE

What Would an Integrated Development Environment for Law look like?

Michael Jeffery

Computational Law Report · Release 2.0 · 2020-04-01

Reader edition prepared by The Research University under CC BY 4.0. This cover was added; the original PDF pages follow unchanged. Original authorship, publication identity and source links are preserved. This independent republication is not sponsored or endorsed by Stanford University, Stanford Law School, CodeX or MIT.

Volume 1 · Issue 2 · Systems, Rules and Public Infrastructure

Original source and license evidence are recorded on the article landing page.

MIT Computational Law Report

What Would an Integrated Development Environment for Law look like?

Michael Jeffery

Published on: Apr 02, 2020

License: Creative Commons Attribution 4.0 International License (CC-BY 4.0)

As a lawyer specialising in corporate and commercial law, I have spent a large proportion of the last 15 years in Microsoft Word drafting contracts and other legal documents. Microsoft Word is undeniably a fantastic product and .DOCX files are likely to remain the default format for sharing and editing legal and business documents for the foreseeable future. However, when it comes to the initial drafting of legal documents in Microsoft Word, there are some inefficiencies that would ideally be addressed.

What are the Problems?

A lawyer's workflow when drafting a new document will typically start with finding the closest existing document or template that meets the criteria for the particular task - usually from a centrally managed precedent database or document management system. The base document will inevitably need further tailoring, which is normally addressed through:

- manually updating terminology to suit the specific requirements of the transaction;
- copying and pasting text from other documents - which will often also require time modifying fonts and other style settings;
- manually checking that the clauses used are the most up-to-date versions (including that they incorporate any updates required to comply with recent changes to law);
- manually drafting custom clauses - which will inevitably result in incidental amendments to other sections of the document; and
- finally, reviewing and re-reviewing the whole document multiple times to ensure that all legal issues have been addressed, that clauses are drafted in a clear, precise and unambiguous manner, that defined terms have been used consistently and that the document does not contain formatting or typographical errors.

Some of these inefficiencies can be partially addressed through better precedent management processes and the use of document automation systems. However, since Microsoft Word does not really have any built-in tools to assist with the typical legal drafting workflow process (other than spelling and grammar correction), there is a case to be made that:

- Microsoft Word may not be the ideal application for lawyers that draft long and complicated legal documents; and
- perhaps lawyers should be looking at the systems and processes used by software engineers as inspiration for the development of a better legal drafting tool.

I am certainly not the first person to have had these thoughts.[1] In fact, I think that any lawyer who dips their toe into the computer science world would have the same epiphany - that the way we currently draft legal documents is incredibly antiquated and there is room for serious improvement!

Similarities between Drafting and Coding

At the outset of this paper, I must emphasise that I am not a computer scientist or software engineer. My programming experience is mostly limited to learning sufficient Python to build automated contract packages and interviews on top of the open source Docassemble framework.[2]

But coming from a legal background, when learning to code, the striking similarities between drafting legal documents and writing code were immediately obvious. Both:

- require significant attention to detail;
- have language, syntax and form rules that need to be consistently applied; and
- at a fundamental level, are just rules and conditional logic reduced to text.

Programmers normally use an application called an *integrated development environment* (**IDE**) to facilitate the authoring, compiling and debugging of code. Most of the popular IDEs (and even more basic code editors) provide tools for:

- **colour coded syntax highlighting** - to assist with readability and to separate different types of commands, variables, words and symbols;
- **automatic error detection** - based on the specific programming language being used;
- **predictive auto-complete features** - providing suggestions as text is typed; and
- **the ability to seamlessly pull code from hosted repositories** - allowing for code to be easily shared and ensuring that up-to-date versions are used.

These are all features that would seem to be equally applicable to the workflow that lawyers go through when drafting legal documents.

So, why don't lawyers use an IDE? In my opinion, we probably should be.

Features Required for a Legal IDE

If an IDE for legal drafting were to be developed, what would this potentially look like and what features could be incorporated into the IDE to improve legal drafting efficiency? The rest of this paper shares my views on this topic from a lawyer's perspective.

File Format and Language

Microsoft Word is a WYSIWYG (what you see is what you get) editor. It allows the user to visualise their final document (in its printed form) while a document is being written or edited - which is useful and important for many use cases (including legal drafting). However, the inefficiencies associated with legal drafting in Microsoft Word stem from the fact that DOCX files are editable files - but they are not programmable files (or at least that is what the vast majority of lawyers think). If a change

occurs in one section of a document that impacts on another section, in most cases the relevant change needs to be made manually (rather than being linked and controlled through code). Lawyers often find that they need to make the same types of incidental changes over and over again (meaning that many of the modifications being made are the result of legal or grammatical rules that could be broken down, codified and executed automatically in a format that allows both text and code).

The vast majority of lawyers are unaware that .DOCX files can be controlled through code (in the form of macros, conditional logic within mail merge fields and through code written in the Visual Basic programming language). However, for me personally, none of these methods have ever felt like a natural and logical part of the legal drafting process. This is primarily because the inputs and code are often inserted and displayed through different windows (or hidden fields), so it is difficult for the code, and the logic within the code, to be read and reviewed as part of, and in conjunction with, the legal text and the logic within the legal text.

Hybrid file formats (where code and natural language text are displayed together) do exist. In the scientific community, written reports (particularly those containing complex calculations) are often written and shared using a Jupyter Notebook[3], RMarkdown[4] or Julia Markdown[5]. These are all interactive formats that allow for text to be written within a report (with basic formatting) and combined with code blocks (using the Python, R and Julia programming languages). The code blocks within a report can be executed and used to run calculations, pull data from other sources, display graphs and tables, and automatically update parts of the report. All of these formats could be used to hack together interactive legal documents (and an example is given later in this paper using Julia). However, these formats have been designed primarily for use in situations where the interactive component of the relevant document or report is a graph, plot or calculation. They can be used to control and update document text - but the method of doing so is quite syntactically complex and unnatural when compared to normal writing or legal drafting.

In the legal sector, there have been a few projects that have developed new programmable contract formats (such as OpenLaw[6] and the Accord Project[7]). The focus of these projects have been to develop interactive open source formats suitable for generating smart contracts for use on blockchain platforms. While traditional paper based contracts may not be considered as smart and innovative - interactive and programmable formats would be equally useful (if not more so) in the day-to-day drafting of traditional paper based contracts and legal documents.

Programmability

There is often debate about whether or not lawyers should learn to code. Finding lawyers who can code is rare, but I would argue that the programming skills required for interactive document design and

drafting are very different than the skills required to develop software. For legal drafting (or legal programming) - the focus is linguistic - rather than mathematical - but the core concepts are the same.

Set out in the table below are the key programming concepts that could be taken into account when designing an IDE for legal programming and how they could be used to improve efficiency. The table also includes examples of how these programming concepts are already used by lawyers when preparing legal documents - albeit that these computations are currently done by lawyers in their heads.

Programming concept	**Application to legal drafting**
Variables	Details like party names, addresses, monetary figures and other defined terms are used throughout legal documents. The ability to change a variable once (with that update then automatically propagating across the rest of the document) is likely to save significant time and reduce the risk of human error in legal drafting. Most legal agreements contain a section with a dictionary of defined terms. If variables are defined within a document (through the code), this would also help to ensure that the relevant defined terms are used consistently throughout the drafting and that all defined terms are included within the document's dictionary section.

Conditional logic (if-else statements)	Conditional logic is a normal part of legal drafting. The lawyer will need to include different types of clauses to satisfy the particular transaction requirements (e.g. if a loan agreement is secured, then include the relevant security clauses). Conditionality also often applies in the context of grammatical changes in a document (e.g. if there are multiple vendors in a transaction, references to vendor, vendor is and vendor has may need to be updated to vendors, vendors are and vendors have). The ability to code these changes into a clause would greatly assist in the future re-usability of the relevant document (or parts of the document). In an ideal world, when a new clause needs to be drafted for a transaction, or further customisation is required for an existing clause, those changes would be inserted into the base document and controlled through conditional logic (rather than deleting the irrelevant clause options and losing that intellectual property). Over time, base documents would become more complex (but also more powerful and capable of dealing with a broader range of transaction requirements).
Lists and datasets	Variables used in legal documents are often related to other variables. For example, in a shareholders agreement, the key variables associated with a particular shareholder might include data such as the name of the shareholder, any company number associated with that shareholder, the registered address of that shareholder, the type of entity that the shareholder is, the type of signing block required for that shareholder, the number of shares that they hold, the type of shares that they hold and the number of directors that the shareholder is entitled to appoint to the board of the relevant company. When combined with for loops (see below), this would significantly increase the versatility (and again re-usability) of the documents once they have been drafted.

Loops	Loops allow for code to be run and repeated multiple times. A simple legal drafting example would be: for each party stored in a list or dataset, include a signing block for that party. If the entity type associated with that party is also stored in the dataset, the type of signing block required for each party could also be automatically updated programmatically. Loops and the ability to iterate over datasets are critical requirements for reusability of documents (as the number of parties, assets, steps or other things in legal documents regularly vary from one transaction to the next).
Importing libraries and functions	Libraries and functions allow code (stored in other files and repositories) to be re-used. Most programming languages (if not all of them) have simple commands to import libraries and then run functions stored in those libraries. The same approach could be used with legal drafting. Text for specific clauses could be stored in separate repositories and then simply called when needed with a simple command (or included within the relevant document using a single line of code). Boilerplate clauses, party information blocks and signing blocks are all examples of document sections that rarely get modified (so would be ideally suited to being coded and stored as a legal function and allow for object oriented legal programming). Currently, this process is usually done manually by lawyers cutting and pasting text from other documents.

Comment blocks	Most programming languages include the ability to insert text comments to explain how sections of code work. A common method is to include a hash/pound sign (#) at the beginning of a line of code. Lines with a hash at the beginning are not processed by the interpreter or compiler and are ignored. In addition to using hash signs for comments, they can be used when writing code to experiment with different options. Rather than deleting a line of code (and loosing that work), the programmer may just put a hash at the beginning of a line so that it is temporarily ignored while another line of code is worked on and tested. Comments in legal documents (which are not then displayed in the output) would be similarly useful for lawyers sharing clauses and explaining factors that may need to be taken into account. But they would also be useful for quick and nasty updates to documents (when there are client time pressures). As mentioned above, in an ideal world, new and bespoke drafting would be added and controlled using conditional logic - but when there is insufficient time to do so, hash signs could be used to remove irrelevant clauses (without deleting them from the document and loosing that knowledge).

If a new legal IDE were to be developed, these programming concepts would ideally be provided for (whether through the IDE itself, or through the underlying file format or programming language that is used).

With a syntax designed for drafting and document generation, I believe that most lawyers who are experienced and skilled in drafting quality legal documents would find the process quite natural - as they already apply the core principles when manually drafting and updating documents in DOCX format.

Preview Windows

Each of the programming concepts above (with perhaps the exception of comment blocks) are used by good document automation platforms. Automation systems like Docassemble use specific syntax typed within a .DOCX template file. The result is a .DOCX file that will automatically update and change in form based on the variables and other data that is obtained from a user when completing an online interview. However, one of the most time consuming aspects associated with preparing a .DOCX formatted template for automation (whether using Docassemble or any other automation server

system) is debugging the template file. Problems with code syntax or formatting issues within the output are generally only identified after a template has been uploaded to the server and the document generated. Documents then need to be updated and tested multiple times to identify the bugs (and this process becomes exponentially more complicated as document length and complexity increases).

Systems like RMarkdown using the RStudio IDE[8] (and most other Markdown editors) provide for a code window and a preview window to be displayed side-by-side (showing the formatted output as code is typed - or quickly after refreshing the preview window). If a new legal IDE were to be developed that used a programmable file format, a key requirement would be to have a preview window to display the output as it is being developed (and that could be refreshed quickly).

This would be particularly useful when testing conditional logic and loops within documents (as the output could be reviewed and checked as different drafting is tested). Since lawyers are accustomed to using WYSIWYG editors (such is Microsoft Word) this feature would probably make the transition process more natural and increase adoption rates for the new legal IDE.

Styling and Formatting

Styling and formatting in Microsoft Word is overly complicated and there are more features than are necessary - and as a consequence of lawyers hacking together documents by cutting and pasting from multiple sources, there are often multiple formats and styles within documents that need to be corrected (some of which are hidden and cause file corruption issues). However, simplified formats like Markdown are lacking in many of the areas that lawyers would want if they were to even consider switching from Microsoft Word.

In my opinion, the core styling and formatting requirements would be:

	Style/formatting requirement
1.	Automatic heading and clause/paragraph numbering (including multiple levels and sub-levels of numbering).
2.	Fonts, font sizes and line spacing - These would need to be customisable as most firms and businesses have their own style guides and preferences.
3.	Tables and text indenting - These often help to improve the readability of documents.

4.	Clause cross-referencing - The ability to insert (and automatically update) cross-references is an important feature for many lawyers. Clauses could be drafted without the use of cross references (and arguably this may result in a clearer style of drafting). However, without cross references, many lawyers would need to substantially update the drafting of many of their existing documents - which could impact on the adoption rates of the new legal IDE.
5.	Page numbers, page breaks and "keep with next" paragraph settings - Legal documents will still be printed, so these features are useful to ensure that documents print as intended.

The new legal IDE would ideally have these core style and formatting features, and would control the look and feel of those formats and styles using something similar to CSS (used to style websites) to ensure that those style and formatting rules are consistently applied.

Syntax Highlighting

Most IDEs for software development display code using customisable themes that apply colour coding to the text to make it easier to read and to assist with identifying different elements within the code (such as variables, reserved words used by the relevant programming language and symbols that have a specific purpose). The new legal IDE would also ideally have these features. However, in addition to specific colour coding in code blocks, the colour coding would also (ideally) be used to make normal text sections more readable (and assist with identifying typographical and grammatical errors).

Auto-Complete

Many IDEs provide intelligent code completion tools that speed up the coding process and reduce errors. Microsoft's own Intellisense system used in the Visual Studio IDE is a good example. Intelligent code completion tools behave in a similar fashion to predictive text on mobile phones, but suggest things like variable names, permitted functions and methods and appropriate syntax through the use of pop-ups.

Context aware auto-completion tools would be equally useful when drafting legal documents. In an ideal world, the auto-complete for a legal IDE would provide for each of the following:

- handling both coding related syntax suggestions and natural language and grammatical suggestions; and

- the ability to use custom machine learning models - this could allow firms and legal departments to use their own clause libraries and precedent databases to train the system (encouraging the use of consistent language and similar drafting styles).

Export Formats

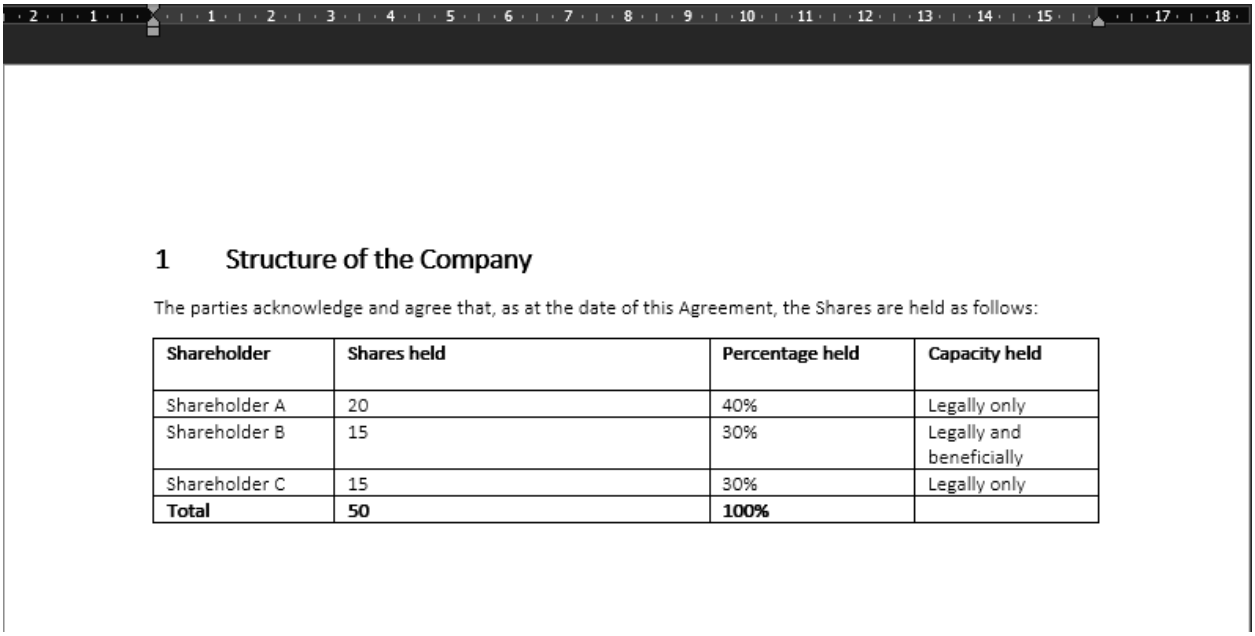
While much of this paper is advocating the benefits of file formats that allow the combination of text and code, in many instances it may not be appropriate for sharing all of the code and possible clause options with clients and counterparties.

For example, if the relevant legal document will be negotiated and the underlying base file (from which the relevant legal document was generated) contains other possible clauses or scenarios (and those clauses/scenarios are more advantageous to the other side), it would not be appropriate for the underlying code to be shared.

As a result, the new legal IDE would need to be able to export to other file formats (i.e. DOCX and PDF) for the negotiation and contract finalisation stages of the legal process.

Demonstration of Core Features Using Julia

The screenshot below shows a very simple clause from a shareholders agreement (in Microsoft Word format) which sets out the number of shares held by each shareholder.



1 Structure of the Company

The parties acknowledge and agree that, as at the date of this Agreement, the Shares are held as follows:

Shareholder	Shares held	Percentage held	Capacity held
Shareholder A	20	40%	Legally only
Shareholder B	15	30%	Legally and beneficially
Shareholder C	15	30%	Legally only
Total	**50**	**100%**	

While it is a simple clause, it is an example that incorporates most of the programming concepts listed above, and how they are applied in practice to legal drafting. The key variables include:

- whether the contract is an agreement or a deed;

- the names of each shareholder (and then related to each shareholder, the number of shares that they hold, shareholding percentages and whether or not they are held on trust); and
- the total number of shares issued by the relevant company.

Using the Julia programming language[9], the Julia IDE (Juno)[10], Julia Markdown[11] and the Weave.JL extension[12], it is possible to achieve many of these basic legal IDE requirements without any further development.

The sample clause could be generated using the code below:

````
````julia; echo = false
# Refer to note 1
deed = "No"
if deed == "Yes"
  contract_type = "Deed"
else
  contract_type = "Agreement"
end
# Refer to note 2
company_shares_number = 50
# Refer to note 3
shareholder_details = (
  name = ["Shareholder A", "Shareholder B", "Shareholder C"],
  shares_held = [20,15,15],
  capacity = ["Legally only", "Legally and beneficially", "Legally only"]
  )
# Refer to note 4
function company_share_structure_clause()
  println("#### Structure of the Company")
  println("The parties acknowledge and agree that, as at the date of this ", contract_type
  ")
  println("| Shareholder | Shares held | Percentage held | Capacity held |")
  println("| --- | --- | --- | --- |")
  x = 1
  for i in shareholder_details
    println("| ", shareholder_details.name[x]," | ", shareholder_details.shares_held[x], "
    x = x + 1
  end
  println("| **Total** |", company_shares_number, "| 100% |   |")
end;
```
`j company_share_structure_clause()`
````

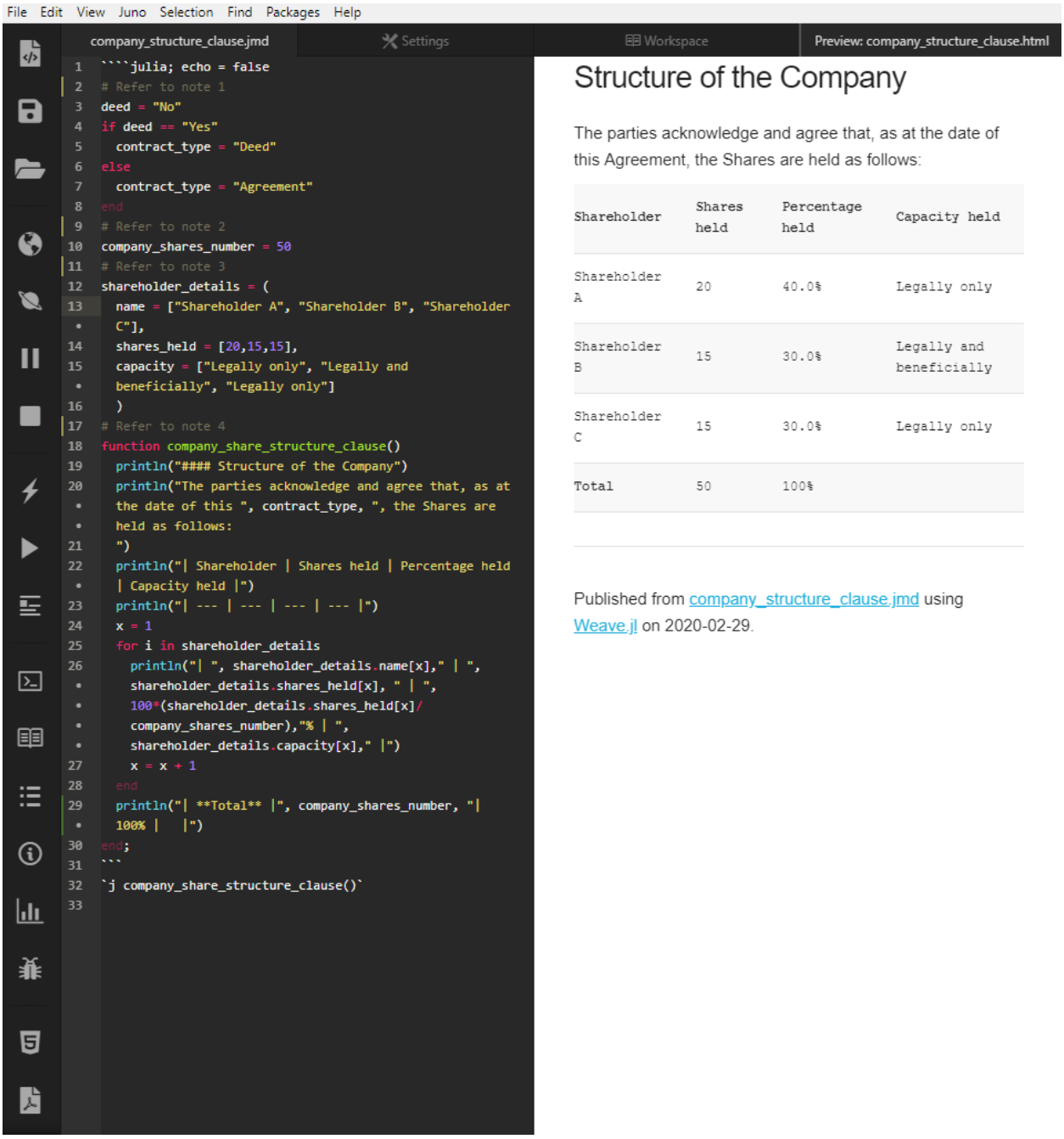
Note 1: This section is an example of conditional logic. The shareholders agreement could either be an agreement or a deed. Depending on the answer to this yes/no question, all references in the shareholders agreement to `contract_type` will display as either Agreement or Deed. There is only one reference in this short clause. However, in a full shareholders agreement, there are likely to be hundreds of instances that would need to change depending on whether the shareholders agreement is signed as an agreement or a deed.

Note 2: The variable `company_shares_number` is defined to be equal to 50. The variable is displayed in the bottom row of the table and is also used to calculate the shareholding percentages for each shareholder.

Note 3: This section of code creates a matrix of the data required for each shareholder. In a full shareholders agreement, there would be many more attributes required to be captured in relation to each shareholder and each other party.

Note 4: This section creates a function called `company_share_structure_clause`. The function is then called outside the code block in the line containing `j company_share_structure_clause()`. Ordinarily, when using formats such as Julia Markdown, RMarkdown and Jupyter notebooks, you can write text freely in standard Markdown format. It is possible to include variables within Markdown text sections. However, it is not possible to include conditionality or loops within standard text blocks. This function uses code to generate the required text and table in Markdown format.

The screenshot below shows what this code looks like when loaded within the Juno IDE. The left-hand side contains the code (with syntax highlighting) and the right-hand side displays the preview of the generated clause.

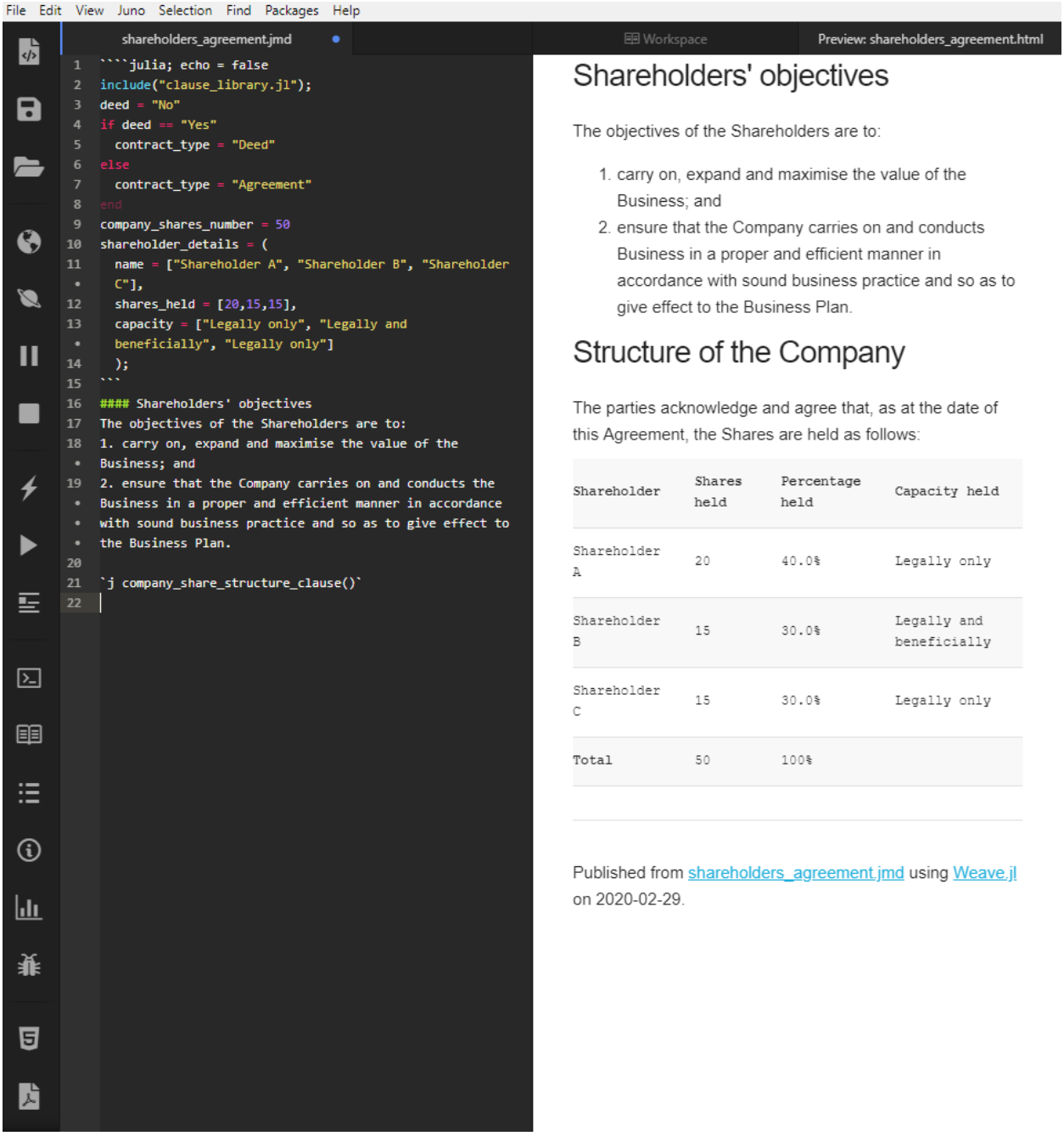


If the content of the function were instead stored within a separate library - perhaps a file called `clause_library.jl` the code could be significantly shortened (as set out below) and the output would be the same:

```
````julia; echo = false
include("clause_library.jl");
deed = "No"
if deed == "Yes"
  contract_type = "Deed"
```
```

```
  contract_type = "Agreement"
  end
  company_shares_number = 50
  shareholder_details = (
    name = ["Shareholder A", "Shareholder B", "Shareholder C"],
    shares_held = [20,15,15],
    capacity = ["Legally only", "Legally and beneficially", "Legally only"]
    );
  ```
  `j company_share_structure_clause()`
```

In my mind, this structure makes a lot of sense. There is an instruction at the top that specifies which clause library to use, with further lines of code to define the key variables and datasets. Below the code block, the lawyer could then write custom clauses or call pre-coded clauses stored within the clause library. In the further screenshot below, some custom drafting in Markdown has also been added to illustrate this point.
```



Note: When typing in the text sections, Markdown uses the hashtags to signify different levels of headings (not comments).

I am not advocating Julia Markdown as being the answer. It has not been designed for this process and there are clear limitations that are likely to dissuade lawyers from changing from their normal processes in Microsoft Word (particularly, the very limited formatting options associated with Markdown). However, it does illustrate (at a conceptual level) how it would be possible to change the typical legal drafting process to a method that combines both natural language and code.

With an IDE specifically designed for legal drafting (and probably a new file format - or even a new programming language - specifically designed with legal drafting in mind), it seems likely that significant efficiencies could be generated.

Additional Goals and Considerations

If this project went further than the theoretical concept outlined in this paper, improving the speed and efficiency of legal drafting would be the primary goal and measure that would need to be assessed. Being such a radical change to the current legal drafting process, it seems highly likely that this goal would **not** be achieved during the early stages of use (as it would require a substantial amount of time to train lawyers to use the new system and develop coded clause libraries). However, there could also be a number of other possible knock-on effects that would need to be evaluated when considering whether or not the use of the new legal IDE improves efficiency (and is otherwise beneficial to the legal professional).

From my perspective, the other key areas to consider would be:

- **Sharing** - Does the process encourage lawyers to start creating and sharing more open source contracts, clause banks and other content through platforms like Github, and does this raise or decrease the overall quality of legal work?
- **Standardised legal language** - Does this alternative method of drafting increase the use of standard legal language developed by organisations like SALI[13] and reduce the negotiation and clause re-drafting that currently goes on between lawyers?
- **Machine learning and automation** - If the chosen file format is primarily text and code (being a much cleaner and structured format than DOCX files), how much does this facilitate the use of other machine learning and automation systems within the law?
- **Error reduction** - Does the legal IDE and new drafting process decrease error rates in documents (or increase error rates as a result of lawyers becoming less thorough in their review process)?
- **Smart Contracts** - Can the relevant file format (and programming language) be used for drafting both traditional paper based contracts and smart contracts (reducing the number of new processes and systems that lawyers need to learn)?

Final thoughts

Law is an old and traditional profession - but unfortunately, so are many of the tools and processes that we currently use.

There are some really exciting projects in the legal tech space that are seeking to modernise aspects of the legal sector. However, with document generation being such a large part of the role that lawyers

play, I believe that we need to go back to basics and design the tools and systems that we need to perform this fundamental task more efficiently.

Footnotes

1. See Matthew Roach, *Towards a New Language of Legal Drafting*, Journal of High Technology Law, Vol XVII No. 1; Samuel Harden, *The Case for a Legal IDE*, Law Practice Today (https://www.lawpracticetoday.org/article/case-legal-ide/) ↩

2. Docassemble (https://docassemble.org/) ↩

3. Jupyter (https://jupyter.org/) ↩

4. RMarkdown (https://https://rmarkdown.rstudio.com/) ↩

5. Julia Markdown (https://docs.julialang.org/en/v1/stdlib/Markdown/) ↩

6. OpenLaw (https://www.openlaw.io/) ↩

7. Accord Project (https://www.accordproject.org/) ↩

8. RStudio (https://rstudio.com/) ↩

9. Julia (https://julialang.org/) ↩

10. Juno IDE (https://junolab.org/) ↩

11. Julia Markdown (https://docs.julialang.org/en/v1/stdlib/Markdown/) ↩

12. Weave for Julia (https://github.com/JunoLab/Weave.jl) ↩

13. SALI Alliance (https://www.sali.org/) ↩